

ASP.NET Core MVC 2

с примерами на C# для профессионалов

Разрабатывайте оптимизированные под облако веб-приложения с использованием новейшей версии ASP.NET Core MVC 2 производства Microsoft!

7-Е ИЗДАНИЕ

Адам Фримен

Диалектика
www.williamspublishing.com
Apress®

ASP.NET Core MVC 2

с примерами на C#

для профессионалов

7-е издание

Pro ASP.NET Core MVC 2

Seventh Edition

Adam Freeman

Apress®

ASP.NET Core MVC 2

с примерами на C#

для профессионалов

7-е издание

Адам Фримен

Диалектика

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Ф88
УДК 681.3.07

ООО "Диалектика"

Перевод с английского и редакция *Ю.Н. Артеменко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Фримен, Адам.
Ф88 ASP.NET Core MVC 2 с примерами на C# для профессионалов, 7-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 1008 с. : ил. — Парал. тит. англ.

ISBN 978-5-6041394-3-1 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства APress, Berkeley, CA.

Copyright © 2019 by Dialektika Computer Publishing.

Authorized translation from the English language edition published by APress, Inc., Copyright © 2017 by Adam Freeman.

This translation is published by arrangement with APress, Berkeley, CA.

All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Научно-популярное издание

Адам Фримен

ASP.NET Core MVC 2 с примерами на C# для профессионалов 7-е издание

Подписано в печать 28.09.2018. Формат 70×100/16
Гарнитура Times
Усл. печ. л. 81,27. Уч.-изд. л. 56,6
Тираж 400 экз. Заказ № 10168

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-6041394-3-1 (рус.)
ISBN 978-1-4842-3149-4 (англ.)

© ООО "Диалектика", 2019
© by Adam Freeman, 2017

Оглавление

Часть I. Введение в инфраструктуру ASP.NET Core MVC 2 21
Глава 1. Основы ASP.NET Core MVC 22
Глава 2. Ваше первое приложение MVC 31
Глава 3. Паттерн, проекты и соглашения MVC 72
Глава 4. Важные функциональные возможности языка C# 87
Глава 5. Работа с Razor 122
Глава 6. Работа с Visual Studio 143
Глава 7. Модульное тестирование приложений MVC 176
Глава 8. SportsStore: реальное приложение 205
Глава 9. SportsStore: навигация 249
Глава 10. SportsStore: завершение построения корзины для покупок 281
Глава 11. SportsStore: администрирование 303
Глава 12. SportsStore: защита и развертывание 330
Глава 13. Работа с Visual Studio Code 356

Часть II. Подробные сведения об инфраструктуре ASP.NET Core MVC 2 379
Глава 14. Конфигурирование приложений 380
Глава 15. Маршрутизация URL 435
Глава 16. Дополнительные возможности маршрутизации 477
Глава 17. Контроллеры и действия 513
Глава 18. Внедрение зависимостей 556
Глава 19. Фильтры 589
Глава 20. Контроллеры API 627
Глава 21. Представления 658
Глава 22. Компоненты представлений 693
Глава 23. Вспомогательные функции дескрипторов 723
Глава 24. Использование вспомогательных функций дескрипторов для форм 755
Глава 25. Использование других встроенных вспомогательных функций дескрипторов 779
Глава 26. Привязка моделей 809
Глава 27. Проверка достоверности моделей 844
Глава 28. Введение в ASP.NET Core Identity 877
Глава 29. Применение ASP.NET Core Identity 913
Глава 30. Расширенные средства ASP.NET Core Identity 941
Глава 31. Соглашения по модели и ограничения действий 973

Предметный указатель 1001

Содержание

Об авторе 19

Часть I. Введение в инфраструктуру ASP.NET Core MVC 2 21

Глава 1. Основы ASP.NET Core MVC 22

История развития ASP.NET Core MVC 22
- ASP.NET Web Forms 23
- Первоначальная инфраструктура MVC Framework 24

Обзор ASP.NET Core 25
- Что нового в ASP.NET Core MVC 2? 26
- Основные преимущества ASP.NET Core MVC 26

Что необходимо знать? 29
Какова структура книги? 30
- Часть I. Введение в инфраструктуру ASP.NET Core MVC 30
- Часть II. Подробные сведения об инфраструктуре ASP.NET Core MVC 30

Где можно получить код примеров? 30
Резюме 30

Глава 2. Ваше первое приложение MVC 31

Установка Visual Studio 31
Установка .NET Core 2.0 SDK 32
Создание нового проекта ASP.NET Core MVC 32
- Добавление контроллера 37
- Понятие маршрутов 39

Визуализация веб-страниц 39
- Создание и визуализация представления 39
- Добавление динамического вывода 43

Создание простого приложения для ввода данных 44
- Предварительная настройка 44
- Проектирование модели данных 46
- Создание второго действия и строго типизированного представления 46
- Ссылка на методы действий 48
- Построение формы 49
- Получение данных формы 51
- Добавление проверки достоверности 59
- Стилизация содержимого 65

Резюме 71

Глава 3. Паттерн, проекты и соглашения MVC 72

История создания MVC 72
Особенности паттерна MVC 72
- Понятие моделей 73
- Понятие контроллеров 74
- Понятие представлений 74
- Реализация паттерна MVC в ASP.NET Core 74
- Сравнение MVC с другими паттернами 76
- Паттерн "Интеллектуальный пользовательский интерфейс" 76

Проекты ASP.NET Core MVC 80
Создание проекта 80
Соглашения в проекте MVC 84
Резюме 86

Глава 4. Важные функциональные возможности языка C# 87

Подготовка проекта для примера 88
Включение ASP.NET Core MVC 89
Создание компонентов приложения MVC 89
Использование `null`-условной операции 91
Связывание в цепочки `null`-условных операций 92
Комбинирование `null`-условной операции и операции объединения с `null` 94
Использование автоматически реализуемых свойств 95
Использование инициализаторов автоматически реализуемых свойств 95
Создание автоматически реализуемых свойств только для чтения 96
Использование интерполяции строк 98
Использование инициализаторов объектов и коллекций 99
Использование инициализатора индексированной коллекции 100
Сопоставление с образцом 101
Сопоставление с образцом в операторах `switch` 102
Использование расширяющих методов 103
Применение расширяющих методов к интерфейсу 105
Создание фильтрующих расширяющих методов 106
Использование лямбда-выражений 108
Определение функций 109
Использование методов и свойств в форме лямбда-выражений 112
Использование автоматического выведения типа и анонимных типов 114
Использование анонимных типов 114
Использование асинхронных методов 116
Работа с задачами напрямую 117
Применение ключевых слов `async` и `await` 118
Получение имен 120
Резюме 121

Глава 5. Работа с Razor 122

Подготовка проекта для примера 123
Определение модели 124
Создание контроллера 124
Создание представления 125
Работа с объектом модели 126
Использование файла импортирования представлений 128
Работа с компоновками 129
Создание компоновки 130
Применение компоновки 132
Использование файла запуска представления 133
Использование выражений Razor 135
Вставка значений данных 136
Установка значений атрибутов 137
Использование условных операторов 138
Проход по содержимому массивов и коллекций 140
Резюме 142

Глава 6. Работа с Visual Studio 143

Подготовка проекта для примера 143
Создание модели 144
Создание контроллера и представления 145
Управление программными пакетами 146
Инструмент NuGet 147
Инструмент Bower 149
Итеративная разработка 153
Внесение изменений в представления Razor 153
Внесение изменений в классы C# 154
Использование средства Browser Link 162
Подготовка файлов JavaScript и CSS для развертывания 167
Включение доставки статического содержимого 168
Добавление в проект статического содержимого 168
Обновление представления 170
Пакетирование и минификация в приложениях MVC 172
Резюме 175

Глава 7. Модульное тестирование приложений MVC 176

Подготовка проекта для примера 177
Включение встроенных вспомогательных функций дескрипторов 177
Добавление действий к контроллеру 178
Создание формы для ввода данных 178
Обновление представления `Index` 179
Модульное тестирование приложений MVC 180
Создание проекта модульного тестирования 181
Создание ссылки на проект 182
Изолирование компонентов для модульного тестирования 187
Улучшение модульных тестов 195
Параметризация модульного теста 195
Улучшение фиктивных реализаций 199
Резюме 204

Глава 8. SportsStore: реальное приложение 205

Начало работы 206
Создание проекта MVC 206
Создание проекта модульного тестирования 209
Проверка и запуск приложения 211
Начало работы с моделью предметной области 212
Создание хранилища 212
Создание фиктивного хранилища 213
Регистрация службы хранилища 214
Отображение списка товаров 215
Добавление контроллера 216
Добавление и конфигурирование представления 217
Установка стандартного маршрута 219
Запуск приложения 220
Подготовка базы данных 221
Установка пакета инструментов командной строки Entity Framework Core 221
Создание классов базы данных 222
Создание класса хранилища 223

Определение строки подключения 223
Конфигурирование приложения 224
Создание миграции базы данных 227
Создание начальных данных 227
Добавление поддержки разбиения на страницы 231
Отображение ссылок на страницы 232
Улучшение URL 241
Стилизация содержимого 242
Установка пакета Bootstrap 243
Применение стилей Bootstrap к компоновке 243
Создание частичного представления 246
Резюме 248

Глава 9. SportsStore: навигация 249

Добавление навигационных элементов управления 249
Фильтрация списка товаров 249
Улучшение схемы URL 253
Построение меню навигации по категориям 257
Корректировка счетчика страниц 265
Построение корзины для покупок 267
Определение модели корзины 268
Создание кнопок добавления в корзину 271
Включение поддержки сеансов 273
Реализация контроллера для корзины 274
Отображение содержимого корзины 277
Резюме 280

Глава 10. SportsStore: завершение построения корзины для покупок 281

Усовершенствование модели корзины с помощью службы 281
Создание класса корзины, осведомленного о хранилище 281
Регистрация службы 282
Упрощение контроллера `Cart` 283
Завершение функциональности корзины 284
Удаление элементов из корзины 284
Добавление виджета с итоговой информацией по корзине 286
Отправка заказов 289
Создание класса модели 289
Добавление реализации процесса оплаты 290
Реализация обработки заказов 293
Завершение построения контроллера `Order` 297
Отображение сообщений об ошибках проверки достоверности 300
Отображение итоговой страницы 301
Резюме 302

Глава 11. SportsStore: администрирование 303

Управление заказами 303
Расширение модели 303
Добавление действий и представления 304
Добавление средств управления каталогом 307
Создание контроллера CRUD 308
Реализация представления списка 309

Редактирование сведений о товарах 311
Создание новых товаров 324
Удаление товаров 326
Резюме 329

Глава 12. SportsStore: защита и развертывание 330

Защита средств администрирования 330
Создание базы данных `Identity` 330
Применение базовой политики авторизации 335
Создание контроллера `Account` и представлений 337
Тестирование политики безопасности 341
Развертывание приложения 341
Создание баз данных 341
Подготовка приложения 343
Применение миграций к базам данных 347
Управление начальным заполнением баз данных 347
Развертывание приложения 351
Резюме 355

Глава 13. Работа с Visual Studio Code 356

Настройка среды разработки 356
Установка Node.js 356
Проверка установки Node 358
Установка Git 358
Проверка установки Git 359
Установка Bower 359
Установка .NET Core 359
Проверка установки .NET Core 360
Установка Visual Studio Code 360
Проверка установки Visual Studio Code 361
Установка расширения C# для Visual Studio Code 361
Создание проекта ASP.NET Core 362
Подготовка проекта с помощью Visual Studio Code 363
Управление пакетами клиентской стороны 364
Конфигурирование приложения 366
Построение и запуск проекта 366
Воссоздание приложения `PartyInvites` 367
Создание модели и хранилища 367
Создание базы данных 370
Создание контроллеров и представлений 372
Модульное тестирование в Visual Studio Code 376
Создание модульного теста 377
Прогон тестов 378
Резюме 378

Часть II. Подробные сведения об инфраструктуре ASP.NET Core MVC 2 379

Глава 14. Конфигурирование приложений 380

Подготовка проекта для примера 382
Конфигурирование проекта 383
Добавление пакетов в проект 384
Добавление пакетов инструментов в проект 386

Класс `Program` 387
Анализ деталей конфигурации 388
Класс `Startup` 391
Службы ASP.NET 394
Промежуточное программное обеспечение ASP.NET 397
Особенности вызова метода `Configure()` 407
Добавление оставшихся компонентов промежуточного программного обеспечения 411
Конфигурирование приложения 416
Создание конфигурационного файла JSON 418
Использование данных конфигурации 420
Конфигурирование регистрации в журнале 421
Конфигурирование внедрения зависимостей 426
Конфигурирование служб MVC 427
Работа со сложными конфигурациями 428
Создание разных внешних конфигурационных файлов 429
Создание разных методов конфигурирования 430
Создание разных классов конфигурирования 432
Резюме 434

Глава 15. Маршрутизация URL 435

Подготовка проекта для примера 437
Создание класса модели 437
Создание примеров контроллеров 438
Создание представления 439
Введение в шаблоны URL 440
Создание и регистрация простого маршрута 442
Определение стандартных значений 443
Определение встраиваемых стандартных значений 444
Использование статических сегментов URL 447
Определение специальных переменных сегментов 452
Использование специальных переменных в качестве параметров метода действия 454
Определение необязательных сегментов URL 455
Определение маршрутов переменной длины 458
Ограничение маршрутов 460
Ограничение маршрута с использованием регулярного выражения 464
Использование ограничений на основе типов и значений 465
Объединение ограничений 466
Определение специального ограничения 468
Использование маршрутизации с помощью атрибутов 471
Подготовка для маршрутизации с помощью атрибутов 471
Применение маршрутизации с помощью атрибутов 472
Применение ограничений к маршрутам 475
Резюме 476

Глава 16. Дополнительные возможности маршрутизации 477

Подготовка проекта для примера 478
Генерирование исходящих URL в представлениях 479
Генерирование исходящих ссылок 480
Генерирование URL (без ссылок) 490
Генерирование URL в методах действий 491

Настройка системы маршрутизации 492
Изменение конфигурации системы маршрутизации 492
Создание специального класса маршрута 493
Применение специального класса маршрута 496
Маршрутизация на контроллеры MVC 497
Работа с областями 504
Создание области 505
Создание маршрута для области 505
Заполнение области 506
Генерирование ссылок на действия в областях 509
Полезные советы относительно схемы URL 510
Делайте URL чистыми и понятными человеку 510
GET и POST: выбор правильного запроса 512
Резюме 512

Глава 17. Контроллеры и действия 513

Подготовка проекта для примера 514
Подготовка представлений 515
Понятие контроллеров 517
Создание контроллеров 518
Создание контроллеров POCO 518
Использование базового класса Controller 521
Получение данных контекста 522
Получение данных из объектов контекста 522
Использование параметров метода действия 527
Генерирование ответа 528
Генерирование ответа с использованием объекта контекста 529
Понятие результатов действий 530
Генерирование HTML-ответа 532
Передача данных из метода действия в представление 535
Выполнение перенаправления 541
Возвращение разных типов содержимого 548
Реагирование с помощью содержимого файлов 551
Возвращение ошибок и кодов HTTP 552
Другие классы результатов действий 555
Резюме 555

Глава 18. Внедрение зависимостей 556

Подготовка проекта для примера 557
Создание модели и хранилища 558
Создание контроллера и представления 559
Создание проекта модульного тестирования 561
Создание слабо связанных компонентов 561
Исследование сильно связанных компонентов 562
Введение в средство внедрения зависимостей ASP.NET 568
Подготовка к внедрению зависимостей 569
Конфигурирование поставщика служб 570
Модульное тестирование контроллера с зависимостью 572
Использование цепочек зависимостей 573
Использование внедрения зависимостей для конкретных типов 575

Жизненные циклы служб 577
Использование переходного жизненного цикла 579
Использование жизненного цикла, ограниченного областью действия 583
Использование жизненного цикла одиночки 585
Использование внедрения в действия 585
Использование атрибутов внедрения в свойства 587
Запрашивание объекта реализации вручную 587
Резюме 588

Глава 19. Фильтры 589

Подготовка проекта для примера 590
Включение SSL 591
Создание контроллера и представления 591
Использование фильтров 593
Понятие фильтров 596
Получение данных контекста 597
Использование фильтров авторизации 598
Создание фильтра авторизации 599
Использование фильтров действий 601
Создание фильтра действий 602
Создание асинхронного фильтра действий 604
Использование фильтров результатов 605
Создание фильтра результатов 606
Создание асинхронного фильтра результатов 608
Создание гибридного фильтра действий/результатов 609
Использование фильтров исключений 611
Создание фильтра исключений 612
Использование внедрения зависимостей для фильтров 614
Распознавание зависимостей в фильтрах 614
Управление жизненными циклами фильтров 618
Создание глобальных фильтров 621
Порядок применения фильтров и его изменение 624
Изменение порядка применения фильтров 626
Резюме 626

Глава 20. Контроллеры API 627

Подготовка проекта для примера 628
Создание модели и хранилища 628
Создание контроллера и представлений 630
Роль контроллеров REST 633
Проблема скорости 633
Проблема эффективности 634
Проблема открытости 635
Введение в REST и контроллеры API 635
Создание контроллера API 637
Тестирование контроллера API 641
Использование контроллера API в браузере 645
Форматирование содержимого 648
Стандартная политика содержимого 648
Согласование содержимого 649
Указание формата данных для действия 652

Получение формата данных из маршрута или строки запроса 653
Включение полного согласования содержимого 655
Получение разных форматов данных 656
Резюме 657

Глава 21. Представления 658

Подготовка проекта для примера 659
Создание специального механизма визуализации 660
Создание специальной реализации интерфейса `IView` 662
Создание реализации интерфейса `IViewEngine` 663
Регистрация специального механизма визуализации 664
Тестирование механизма визуализации 665
Работа с механизмом Razor 667
Подготовка проекта для примера 668
Прояснение представлений Razor 670
Добавление динамического содержимого к представлению Razor 674
Использование разделов компоновки 675
Использование частичных представлений 680
Добавление содержимого JSON в представления 683
Конфигурирование механизма Razor 685
Расширители местоположений представлений 686
Резюме 692

Глава 22. Компоненты представлений 693

Подготовка проекта для примера 694
Создание моделей и хранилищ 694
Создание контроллера и представлений 696
Конфигурирование приложения 699
Понятие компонентов представлений 700
Создание компонента представления 701
Создание компонентов представлений POCO 701
Наследование от базового класса `ViewComponent` 703
Понятие результатов компонентов представлений 704
Получение данных контекста 710
Создание асинхронных компонентов представлений 715
Создание гибридных классов контроллеров и компонентов представлений 718
Создание гибридных представлений 719
Применение гибридного класса 720
Резюме 722

Глава 23. Вспомогательные функции дескрипторов 723

Подготовка проекта для примера 724
Создание модели и хранилища 725
Создание контроллера, компоновки и представлений 725
Конфигурирование приложения 728
Создание вспомогательной функции дескриптора 729
Определение класса вспомогательной функции дескриптора 729
Регистрация вспомогательных функций дескрипторов 733
Использование вспомогательной функции дескриптора 734
Управление областью действия вспомогательной функции дескриптора 736

Усовершенствованные возможности вспомогательных функций дескрипторов 740
Создание сокращающих элементов 741
Вставка перед и после содержимого и элементов 743
Получение данных контекста представления и использование внедрения зависимостей 747
Работа с моделью представления 749
Согласование вспомогательных функций дескрипторов 751
Подавление выходного элемента 753
Резюме 754

Глава 24. Использование вспомогательных функций дескрипторов для форм 755

Подготовка проекта для примера 756
Переустановка представлений и компоновки 757
Работа с элементами `form` 759
Установка цели формы 759
Использование средства противодействия подделке 760
Работа с элементами `input` 762
Конфигурирование элементов `input` 763
Форматирование значений данных 765
Работа с элементами `label` 768
Работа с элементами `select` и `option` 770
Использование источника данных для заполнения элемента `select` 771
Генерирование элементов `option` из перечисления 772
Работа с элементами `textarea` 776
Вспомогательные функции дескрипторов для проверки достоверности форм 778
Резюме 778

Глава 25. Использование других встроенных вспомогательных функций дескрипторов 779

Подготовка проекта для примера 780
Использование вспомогательной функции дескриптора для среды размещения 781
Использование вспомогательных функций дескрипторов для JavaScript и CSS 782
Управление файлами JavaScript 782
Управление таблицами стилей CSS 792
Работа с якорными элементами 795
Работа с элементами `img` 797
Использование кеша данных 798
Установка времени истечения для кеша 801
Использование вариаций кеша 803
Использование URL, относительных к приложению 805
Резюме 808

Глава 26. Привязка моделей 809

Подготовка проекта для примера 810
Создание модели и хранилища 810
Создание контроллера и представления 812
Конфигурирование приложения 813
Понятие привязки моделей 814
Стандартные значения привязки 816
Привязка простых типов 818

Привязка сложных типов 819
Привязка массивов и коллекций 829
Привязка коллекций сложных типов 832
Указание источника данных привязки моделей 835
Выбор стандартного источника данных привязки 837
Использование заголовков в качестве источников данных привязки 838
Использование тел запросов в качестве источников данных привязки 841
Резюме 843

Глава 27. Проверка достоверности моделей 844

Подготовка проекта для примера 844
Создание модели 846
Создание контроллера 846
Создание компоновки и представлений 847
Необходимость в проверке достоверности модели 850
Явная проверка достоверности модели 850
Отображение пользователю ошибок проверки достоверности 853
Отображение сообщений об ошибках проверки достоверности 855
Отображение сообщений об ошибках проверки достоверности на уровне свойств 860
Отображение сообщений об ошибках проверки достоверности на уровне модели 861
Указание правил проверки достоверности с помощью метаданных 864
Создание специального атрибута проверки достоверности для свойства 867
Выполнение проверки достоверности на стороне клиента 870
Выполнение удаленной проверки достоверности 873
Резюме 876

Глава 28. Введение в ASP.NET Core Identity 877

Подготовка проекта для примера 879
Создание контроллера и представления 880
Настройка ASP.NET Core Identity 882
Создание класса пользователя 882
Создание класса контекста базы данных 883
Конфигурирование настройки строки подключения к базе данных 884
Создание базы данных Identity 886
Использование ASP.NET Core Identity 886
Перечисление пользовательских учетных записей 887
Создание пользователей 889
Проверка паролей 893
Проверка деталей, связанных с пользователем 900
Завершение построения средств администрирования 905
Реализация возможности удаления 906
Реализация возможности редактирования 907
Резюме 912

Глава 29. Применение ASP.NET Core Identity 913

Подготовка проекта для примера 913
Аутентификация пользователей 914
Подготовка к реализации аутентификации 916
Добавление аутентификации пользователей 919
Тестирование аутентификации 922

Авторизация пользователей с помощью ролей 922
Создание и удаление ролей 923
Управление членством в ролях 928
Использование ролей для авторизации 933
Помещение в базу данных начальной информации 937
Резюме 940

Глава 30. Расширенные средства ASP.NET Core Identity 941

Подготовка проекта для примера 942
Добавление специальных свойств в класс пользователя 942
Подготовка миграции базы данных 946
Тестирование специальных свойств 947
Работа с заявками и политиками 947
Понятие заявок 948
Создание заявок 952
Использование политик 955
Использование политик для авторизации доступа к ресурсам 962
Использование сторонней аутентификации 967
Регистрация приложения в Google 967
Включение аутентификации Google 968
Резюме 972

Глава 31. Соглашения по модели и ограничения действий 973

Подготовка проекта для примера 973
Создание модели представления, контроллера и представления 974
Использование модели приложения и соглашений по модели 976
Модель приложения 977
Роль соглашений по модели 981
Создание соглашения по модели 982
Порядок применения соглашений по модели 987
Создание глобальных соглашений по модели 988
Использование ограничений действий 989
Подготовка проекта для примера 990
Ограничения действий 992
Создание ограничения действия 993
Распознавание зависимостей в ограничениях действий 998
Резюме 1000

Предметный указатель 1001

Посвящается моей прекрасной жене Джеки Гриффитс.

Об авторе

Адам Фримен — опытный специалист в области информационных технологий, занимавший ведущие позиции во многих компаниях, последней из которых был глобальный банк, где он работал на должностях директора по внедрению технологий и руководителя административной службы. После ухода из банка Адам уделяет все свое время писательской деятельности и бегу на длинные дистанции.

О техническом рецензенте

Фабио Клаудио Ферраччати — ведущий консультант и главный аналитик/разработчик, использующий технологии Microsoft. Он работает в компании BluArancio (`www.bluarancio.com`). Фабио является сертифицированным Microsoft разработчиком решений для .NET (Microsoft Certified Solution Developer for .NET), сертифицированным Microsoft разработчиком приложений для .NET (Microsoft Certified Application Developer for .NET), сертифицированным Microsoft профессионалом (Microsoft Certified Professional), а также продуктивным автором и техническим рецензентом. За последние десять лет он написал множество статей для итальянских и международных журналов и выступал в качестве соавтора в более чем 10 книгах по разнообразным темам, связанным с компьютерами.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@dialektika.com

WWW: http://www.dialektika.com

ЧАСТЬ I

Введение в инфраструктуру ASP.NET Core MVC 2

Для разработчиков веб-приложений, использующих платформу Microsoft, инфраструктура ASP.NET Core MVC представляется как радикальное изменение. В ней особое значение придается чистой архитектуре, паттернам проектирования и удобству тестирования, а также не предпринимается попытка скрывать, каким образом работает веб-среда.

Первая часть книги поможет понять в общих чертах основополагающие идеи разработки приложений MVC, включая новые возможности ASP.NET Core MVC, и увидеть на деле, как применяется инфраструктура.

ГЛАВА 1

Основы ASP.NET Core MVC

ASP.NET Core MVC — это инфраструктура для разработки веб-приложений производства Microsoft, которая сочетает в себе эффективность и аккуратность архитектуры "модель-представление-контроллер" (model-view-controller — MVC), идеи и приемы гибкой разработки, а также лучшие части платформы .NET. В настоящей главе вы узнаете, почему компания Microsoft создала инфраструктуру ASP.NET Core MVC, увидите, как она соотносится со своими предшественниками и альтернативами, а также ознакомитесь с обзором нововведений ASP.NET Core MVC и с тем, что будет рассматриваться в книге.

История развития ASP.NET Core MVC

Первоначальная версия платформы ASP.NET появилась в 2002 году, в то время, когда компания Microsoft стремилась сохранить господствующее положение в области разработки традиционных настольных приложений и видела в Интернете опасность. На рис. 1.1 показано, как выглядел на то время стек технологий Microsoft.

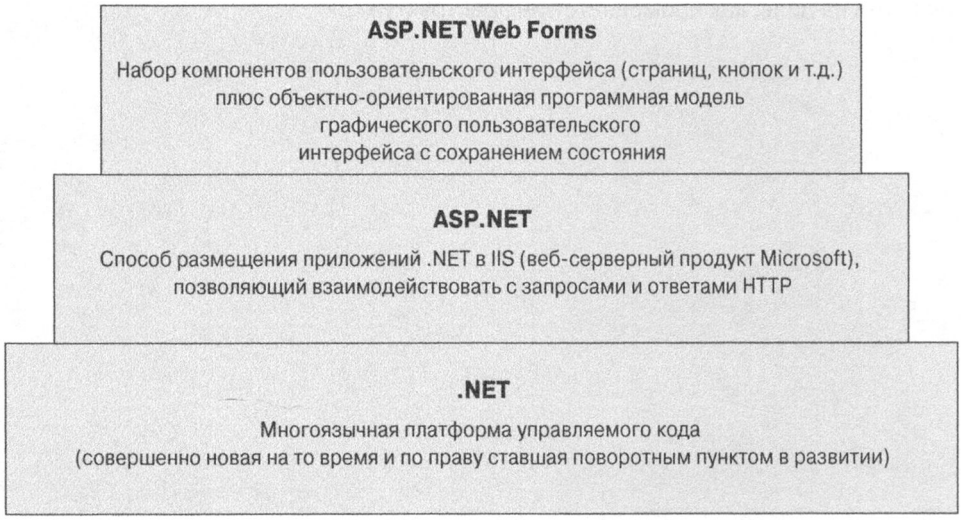


Рис. 1.1. Стек технологий ASP.NET Web Forms

ASP.NET Web Forms

В инфраструктуре Web Forms разработчики из Microsoft пытались скрыть и протокол HTTP (с присущим ему отсутствием состояния), и язык HTML (которым на тот момент не владели многие разработчики) за счет моделирования пользовательского интерфейса как иерархии объектов, представляющих серверные элементы управления. Каждый элемент управления отслеживал собственное состояние между запросами, по мере необходимости визуализируя себя в виде HTML-разметки и автоматически соединяя события клиентской стороны (например, щелчки на кнопках) с соответствующим кодом их обработки на стороне сервера. Фактически Web Forms является гигантским уровнем абстракции, предназначенным для доставки классического управляемого событиями графического пользовательского интерфейса через веб-среду.

Идея заключалась в том, чтобы разработка веб-приложений выглядела почти так же, как разработка настольных приложений. Разработчики могли оперировать понятиями пользовательского интерфейса, запоминающего состояние, и не иметь дело с последовательностями независимых запросов и ответов HTTP. У компании Microsoft появилась возможность плавно переместить армию разработчиков настольных Windows-приложений в новый мир веб-приложений.

Что было не так с ASP.NET Web Forms?

Разработка с использованием традиционной технологии ASP.NET Web Forms в принципе была хорошей, но реальность оказалась более сложной.

- *Тяжеловесность состояния представления (View State).* Действительный механизм поддержки состояния между запросами, известный как View State, вызвал необходимость передачи крупных блоков данных между клиентом и сервером. Объем таких данных мог достигать сотен килобайтов даже в скромных веб-приложениях. Эти данные путешествуют туда и обратно с каждым запросом, приводя к замедлению реакции и увеличивая требование к ширине полосы пропускания сервера.

- *Жизненный цикл страницы.* Механизм соединения событий клиентской стороны с кодом обработчиков событий на стороне сервера, являющийся частью жизненного цикла страницы, мог быть сложным и хрупким. Лишь немногим разработчикам удавалось успешно манипулировать иерархией элементов управления во время выполнения, не получая ошибки состояния представления или не сталкиваясь с ситуацией, когда некоторые обработчики событий загадочным образом отказывались запускаться.

- *Ложное ощущение разделения обязанностей.* Модель отделенного кода ASP.NET Web Forms предоставляла способ вынесения кода приложения из HTML-разметки в специальный файл отделенного кода. Это было сделано с целью разделения логики и представления, но в действительности попустительствовало смешиванию кода представления (например, манипуляций деревом элементов управления серверной стороны) с прикладной логикой (скажем, обработкой информации из базы данных) в гигантских классах отделенного кода. Конечный результат мог оказаться хрупким и непонятным.

- *Ограниченный контроль над HTML-разметкой.* Серверные элементы управления визуализировали себя в виде HTML-разметки, но она не обязательно была такой, как хотелось. В ранних версиях Web Forms выходная HTML-разметка не

соответствовала веб-стандартам или неэффективно применяла каскадные таблицы стилей (Cascading Style Sheets — CSS), а серверные элементы управления генерировали непредсказуемые и сложные атрибуты идентификаторов, к которым было трудно получать доступ с помощью JavaScript. Указанные проблемы были значительно смягчены в последних выпусках Web Forms, но получение ожидаемой HTML-разметки по-прежнему могло быть затруднительным.

- *Негерметичная абстракция.* Инфраструктура Web Forms, где только возможно, пыталась скрывать детали, связанные с HTML и HTTP. При попытке реализовать специальное поведение абстракция часто отбрасывалась, поэтому для генерирования желаемой HTML-разметки приходилось воссоздавать механизм событий обратной отправки или предпринимать другие сложные действия.
- *Низкая тестируемость.* Проектировщики Web Forms даже не предполагали, что автоматизированное тестирование станет неотъемлемой частью процесса разработки программного обеспечения. Построенная ими тесно связанная архитектура не была приспособлена для модульного тестирования. Интеграционное тестирование также могло превратиться в сложную задачу.

С технологией Web Forms не все было плохо, и со временем в Microsoft приложили немало усилий по улучшению степени соответствия стандартам, упрощению процесса разработки и даже заимствованию ряда возможностей из первоначальной инфраструктуры ASP.NET MVC Framework для их применения в Web Forms. Технология Web Forms превосходна, когда необходимы быстрые результаты, и с ее помощью вполне реально построить и запустить веб-приложение умеренной сложности всего за один день. Но если не проявлять осторожность во время разработки, то обнаружится, что созданное приложение трудно тестировать и сопровождать.

Первоначальная инфраструктура MVC Framework

В октябре 2007 года компания Microsoft объявила о выходе новой платформы разработки, построенной на основе существующей платформы ASP.NET, которая была задумана как прямая реакция на критику Web Forms и популярность конкурирующих платформ наподобие Ruby on Rails. Новая платформа получила название ASP.NET MVC Framework и отражала формирующиеся тенденции в разработке веб-приложений, такие как стандартизация HTML и CSS, веб-службы REST, эффективное модульное тестирование, а также идея о том, что разработчики должны принять факт отсутствия состояния у HTTP.

Концепции, которые подкрепляли первоначальную инфраструктуру MVC Framework, теперь кажутся естественными и очевидными, но в 2007 году мир разработки веб-приложений .NET был их лишен. Появление ASP.NET MVC Framework возвратило платформу разработки веб-приложений Microsoft в современную эпоху.

Инфраструктура MVC Framework также сигнализировала о важном изменении в отношении компании Microsoft, которая ранее пыталась контролировать каждый компонент в инструментальных средствах, необходимых для веб-приложений. Компания Microsoft встроила в инфраструктуру MVC Framework инструменты с открытым кодом, такие как jQuery, учла проектные соглашения и передовой опыт конкурирующих (и более успешных) платформ, а также выпустила в свет исходный код MVC Framework для изучения разработчиками.

Что было не так с первоначальной инфраструктурой MVC Framework?

На то время для Microsoft имело смысл создавать инфраструктуру MVC Framework поверх существующей платформы, содержащей большой объем монолитной низкоуровневой функциональности, которая обеспечила преимущество в начале процесса разработки и которую хорошо знали и понимали разработчики приложений ASP.NET.

Но компромиссы потребовали основать MVC Framework на платформе, которая изначально предназначалась для Web Forms. Разработчики MVC Framework привыкли использовать конфигурационные параметры и кодовые настройки, отключающие или реконфигурирующие средства, которые не имели ни малейшего отношения к их веб-приложениям, но требовались для того, чтобы все заработало.

С ростом популярности MVC Framework компания Microsoft приступила к добавлению ряда ее основных возможностей к Web Forms. Результат был все более и более странным, когда индивидуальные особенности проекта, требуемые для поддержки MVC Framework, расширялись с целью поддержки Web Forms, и предпринимались дополнительные проектные ухищрения, чтобы подогнать все друг к другу. Одновременно в Microsoft начали расширять ASP.NET новыми инфраструктурами для создания веб-служб (Web API) и организации коммуникаций в реальном времени (SignalR). Новые инфраструктуры добавили собственные соглашения по конфигурированию и разработке, каждое из которых обладало своими преимуществами и причудами, породив в результате фрагментированную смесь.

Обзор ASP.NET Core

В 2015 году Microsoft заявила о новом направлении для ASP.NET и MVC Framework, что в итоге привело к появлению инфраструктуры ASP.NET Core MVC, рассматриваемой в настоящей книге.

Платформа ASP.NET Core построена на основе .NET Core, которая представляет собой межплатформенную версию .NET Framework без интерфейсов программирования приложений (application programming interface — API), специфичных для Windows. Господствующей операционной системой по-прежнему является Windows, но веб-приложения все чаще размещаются в небольших и простых контейнерах на облачных платформах. За счет принятия межплатформенного подхода компания Microsoft расширила область охвата .NET, сделав возможным развертывание приложений ASP.NET Core на более широком наборе сред размещения, а в качестве бонуса предоставила разработчикам возможность создавать веб-приложения ASP.NET Core на машинах Linux и macOS.

ASP.NET Core — совершенно новая инфраструктура. Она проще, с нею легче работать, и она свободна от наследия, происходящего из Web Forms. Будучи основанной на .NET Core, она поддерживает разработку веб-приложений для ряда платформ и контейнеров.

Инфраструктура ASP.NET Core MVC предлагает функциональность первоначальной инфраструктуры ASP.NET MVC Framework, построенной поверх новой платформы ASP.NET Core. Она интегрирует функциональные средства, которые ранее предоставлялись Web API, поддерживает более естественный способ генерирования сложного содержимого и делает основные задачи разработки, такие как модульное тестирование, более простыми и предсказуемыми.

Что нового в ASP.NET Core MVC 2?

Выпуск ASP.NET Core MVC 2 концентрируется на консолидации за счет того, что задействует в своих интересах изменения в инструментах и платформе, которые были введены в более ранних версиях. Для работы ASP.NET Core MVC 2 требуется платформа .NET Core 2, которая имеет значительно расширенное покрытие API-интерфейсами и теперь поддерживается дополнительными дистрибутивами Linux. Полезные изменения включают новую систему мета-пакетов, упрощающую управление пакетами NuGet, новую систему конфигурации для ASP.NET Core и поддержку для Entity Framework Core 2. Крупнейшим новым средством стала инфраструктура Razor Pages, которая представляет собой попытку воссоздания стиля разработки, ассоциированного с инфраструктурой Web Pages, с применением более современной платформы, но Razor Pages не будет интересна разработчикам MVC (и в данной книге не рассматривается).

Основные преимущества ASP.NET Core MVC

В последующих разделах кратко описано, чем новая платформа MVC превосходит унаследованную инфраструктуру Web Forms и первоначальную инфраструктуру MVC Framework, и что именно позволит снова вывести ASP.NET на передний край.

Архитектура MVC

Инфраструктура ASP.NET Core MVC следует паттерну под названием *"модель-представление-контроллер"* (model-view-controller — MVC), который управляет формой веб-приложения ASP.NET и взаимодействиями между содержащимися в нем компонентами.

Важно различать архитектурный паттерн MVC и реализацию ASP.NET Core MVC. Паттерн MVC далеко не нов (его появление датируется 1978 годом и связано с проектом Smalltalk в Xerox PARC), но в наши дни он завоевал популярность в качестве паттерна для веб-приложений по перечисленным ниже причинам.

- Взаимодействие пользователя с приложением, которое придерживается паттерном MVC, следует естественному циклу: пользователь предпринимает действие, а приложение в ответ изменяет свою модель данных и доставляет обновленное представление пользователю. Затем цикл повторяется. Это удобно укладывается в схему веб-приложений, предоставляемых в виде последовательностей запросов и ответов HTTP.

- Веб-приложения, нуждающиеся в сочетании нескольких технологий (например, баз данных, HTML-разметки и исполняемого кода), обычно разделяются на набор слоев или уровней. Полученные в результате таких сочетаний комбинации естественным образом отображаются на концепции в паттерне MVC.

Инфраструктура ASP.NET Core MVC реализует паттерн MVC и вдобавок обеспечивает гораздо лучшее разделение обязанностей по сравнению с Web Forms. На самом деле в ASP.NET Core MVC внедрена разновидность паттерна MVC, которая особенно хорошо подходит для веб-приложений. Дополнительные сведения по теории и практике применения такой архитектуры приведены в главе 3.

Расширяемость

Инфраструктуры ASP.NET Core и ASP.NET Core MVC построены в виде последовательности независимых компонентов, которые имеют четко определенные характеристики, удовлетворяют интерфейсу .NET или созданы на основе абстрактного базового класса. Основные компоненты можно легко заменять другими компонентами с собственной реализацией. В общем случае для каждого компонента инфраструктура ASP.NET Core MVC предлагает три возможности.

- Использование стандартной реализации компонента в том виде, как есть (чего должно быть достаточно для большинства приложений).
- Создание подкласса стандартной реализации с целью корректировки существующего поведения.
- Полная замена компонента новой реализацией интерфейса или абстрактного базового класса.

Разнообразные компоненты, а также способы и причины их возможной настройки или замены будут рассматриваться, начиная с главы 14.

Жесткий контроль над HTML и HTTP

Инфраструктура ASP.NET Core MVC генерирует ясную и соответствующую стандартам разметку. Ее встроенные вспомогательные функции дескрипторов (tag helper) производят соответствующий стандартам вывод, но по сравнению с Web Forms имеется более значимое философское изменение. Вместо генерации громадного объема HTML-разметки, над которой вы имеете очень небольшой контроль, инфраструктура ASP.NET Core MVC поощряет создание простой и элегантной разметки, стилизованной с помощью CSS.

Конечно, если вы действительно хотите добавить готовые виджеты для таких сложных элементов пользовательского интерфейса, как окна выбора даты или каскадные меню, то подход "никаких специальных требований", принятый в ASP.NET Core MVC, позволяет легко использовать наилучшие клиентские библиотеки, подобные jQuery, Angular, React или Bootstrap CSS. Инфраструктура ASP.NET Core MVC настолько тесно сплетена с этими библиотеками, что в Microsoft предусмотрели шаблоны, которые включают их, чтобы придать ускорение процессу разработки новых проектов.

Инфраструктура ASP.NET Core MVC работает в гармонии с HTTP. Вы имеете контроль над запросами, передаваемыми между браузером и сервером, так что можете точно настраивать пользовательский интерфейс по своему усмотрению. Технология Ajax сделана легкой в применении, и создание веб-служб для получения браузерных HTTP-запросов выливается в простой процесс, который описан в главе 20.

Тестируемость

Естественное разнесение различных обязанностей приложения по независимым друг от друга частям, которое поддерживается архитектурой ASP.NET Core MVC, позволяет с самого начала делать приложение легко сопровождаемым и удобным для тестирования. Вдобавок каждый фрагмент платформы ASP.NET Core и инфраструктуры ASP.NET Core MVC может быть изолирован и заменен в целях модульного тестирования, которое допускается выполнять с использованием любой популярной инфраструктуры тестирования с открытым кодом, такой как xUnit, рассматриваемой в главе 7.

В настоящей книге вы увидите примеры написания ясных и простых модульных тестов для контроллеров и действий MVC, которые предоставляют фиктивные либо имитированные реализации компонентов инфраструктуры для эмуляции любого сценария с применением разнообразных стратегий тестирования и имитации. Даже если вам никогда ранее не приходилось создавать модульные тесты, у вас будет все необходимое для успешного старта.

Тестируемость имеет отношение не только к модульному тестированию. Приложения ASP.NET Core MVC успешно работают также с инструментами тестирования, встроенными в средства автоматизации пользовательского интерфейса. Можно создавать тестовые сценарии, которые имитируют взаимодействие с пользователем, не выдвигая догадки о том, какие структуры HTML-элементов, классы CSS или идентификаторы сгенерирует инфраструктура, и не испытывая беспокойства по поводу неожиданных изменений структуры.

Мощная система маршрутизации

По мере совершенствования технологии построения веб-приложений эволюционировал стиль унифицированных указателей ресурсов (uniform resource locator — URL). Адреса URL вроде приведенного ниже:

```
/App_v2/User/Page.aspx?action=show%20prop&prop_id=82742
```

встречаются все реже, а им на смену приходит более простой и понятный формат следующего вида:

```
/to-rent/chicago/2303-silver-street
```

Существует ряд веских причин для того, чтобы заботиться о структуре URL-адресов. Во-первых, поисковые механизмы придают вес ключевым словам, содержащимся в URL. Поиск по фразе "rent in Chicago" ("аренда в Чикаго") с большей вероятностью обнаружит более простой URL. Во-вторых, многие веб-пользователи достаточно сообразительны, чтобы понимать URL, и ценят возможность навигации путем ввода запроса в адресной строке своего браузера. В-третьих, когда структура URL-адреса понятна, люди с большей вероятностью пройдут по нему, поделятся им с другими или даже продиктуют его по телефону. В-четвертых, при таком подходе в Интернете не раскрываются технические детали, структура каталогов и имен файлов приложения; следовательно, вы вольны изменять лежащую в основе сайта реализацию, не нарушая работоспособности всех входящих ссылок.

В более ранних инфраструктурах понятные URL-адреса реализовать было трудно, но в ASP.NET Core MVC используется средство, известное как *маршрутизация URL*, которое обеспечивает предоставление понятных URL-адресов по умолчанию. В результате появляется контроль над схемой URL и ее взаимосвязью с приложением, что обеспечивает свободу создания понятного и удобного для пользователей шаблона URL без необходимости следования какому-то заранее определенному шаблону. И, разумеется, это означает также простоту определения современной схемы URL в стиле REST, если она нужна. Подробное описание маршрутизации URL приведено в главах 15 и 16.

Современный API-интерфейс

Платформа Microsoft .NET развивалась с каждым крупным выпуском, поддерживая — и даже определяя — многие передовые аспекты современного программирования. Инфраструктура ASP.NET Core MVC построена для платформы .NET Core, поэто-

му ее API-интерфейс может в полной мере задействовать последние новшества языка и исполняющей среды, знакомые программистам на C#, в том числе ключевое слово `await`, расширяющие методы, лямбда-выражения, анонимные и динамические типы, а также язык интегрированных запросов (Language Integrated Query — LINQ).

Многие методы и паттерны кодирования API-интерфейса ASP.NET Core MVC следуют более четкой и выразительной композиции, чем было возможно в ранних версиях инфраструктуры. Не переживайте, если вы пока не в курсе последних функциональных возможностей языка C#: в главе 4 представлена сводка по самым важным средствам C# для разработки приложений MVC.

Межплатформенная природа

Предшествующие версии ASP.NET были специфичными для Windows, требуя настольный компьютер с Windows при написании веб-приложений и сервер Windows при их развертывании и выполнении. Компания Microsoft сделала инфраструктуру ASP.NET Core межплатформенной, как в отношении разработки, так и в плане развертывания. Продукт .NET Core доступен для различных платформ, включая macOS и набор популярных дистрибутивов Linux. Межплатформенная поддержка облегчает развертывание приложений ASP.NET Core MVC, к тому же имеется хорошая поддержка работы с платформами контейнеров приложений, такими как Docker.

Вполне вероятно, большая часть разработки приложений ASP.NET Core MVC в ближайшем будущем будет выполняться с применением Visual Studio, но компания Microsoft также создала межплатформенный инструмент разработки под названием Visual Studio Code, появление которого означает, что разработка ASP.NET Core MVC больше не ограничена средой Windows.

Инфраструктура ASP.NET Core MVC имеет открытый код

В отличие от предшествующих платформ для разработки веб-приложений производства Microsoft вы можете загрузить исходный код ASP.NET Core и ASP.NET Core MVC и даже модифицировать и компилировать его с целью получения собственных версий инфраструктур. Это бесценно при отладке кода, обращающегося к системному компоненту, когда требуется пошагово выполнить его код (и даже почитать оригинальные комментарии программистов). Это также полезно, если вы создаете усовершенствованный компонент и хотите посмотреть, какие существуют возможности разработки, или узнать, как действительно работают встроенные компоненты.

Исходный код ASP.NET Core и ASP.NET Core MVC доступен для загрузки по адресу `https://github.com/aspnet`.

Что необходимо знать?

Чтобы получить от книги максимальную пользу, вы должны быть знакомы с основами разработки веб-приложений, понимать HTML и CSS, а также иметь практический опыт работы с языком C#. Не беспокойтесь, если детали разработки клиентской стороны, такие как JavaScript, для вас несколько туманны. Основной акцент в книге делается на разработке серверной стороны, и благодаря примерам вы сможете подобрать то, что вам нужно. В главе 4 приводится сводка по наиболее полезным средствам языка C# для разработки MVC, которую вы сочтете удобной, если переходите на последние версии .NET с более раннего выпуска.

Какова структура книги?

Книга разделена на две части, в каждой из которых раскрывается набор связанных тем.

Часть I. Введение в инфраструктуру ASP.NET Core MVC

Книга начинается с помещения ASP.NET Core MVC в контекст разработки. Здесь объяснены преимущества и практическое влияние паттерна MVC, рассмотрен способ, которым инфраструктура ASP.NET Core MVC вписывается в современную разработку веб-приложений, а также описаны инструменты и средства языка C#, необходимые каждому программисту ASP.NET Core MVC.

В главе 2 мы углубимся в детали и создадим простое веб-приложение, чтобы получить представление о том, каковы основные компоненты и строительные блоки, и каким образом они сочетаются друг с другом. Однако большинство материала этой части посвящено разработке проекта под названием SportsStore, посредством которого демонстрируется реалистичный процесс разработки, начиная с постановки задачи и заканчивая развертыванием, с привлечением основных функциональных возможностей ASP.NET Core MVC.

Часть II. Подробные сведения об инфраструктуре ASP.NET Core MVC

Во второй части объясняется внутренняя работа средств ASP.NET Core MVC, которые использовались при построении приложения SportsStore. Будет показано, как работает каждое средство, объяснена его роль и описаны доступные варианты конфигурирования и настройки. Широкий контекст, представленный в первой части, подробно раскрывается во второй части.

Где можно получить код примеров?

Все примеры, рассмотренные в книге, доступны для загрузки по адресу `https://github.com/apress/pro-asp.net-core-mvc-2` и на веб-сайте издательства. Загружаемый файл доступен бесплатно и включает все поддерживающие ресурсы, которые требуются для воссоздания примеров без необходимости в их ручном наборе. Загружать код необязательно, но это самый простой путь для экспериментирования с примерами и способ использования фрагментов кода в собственных проектах.

Резюме

В главе был объяснен контекст, в котором существует инфраструктура ASP.NET Core MVC, а также описано ее развитие от Web Forms и первоначальной инфраструктуры ASP.NET MVC Framework. Кроме того, рассматривались преимущества применения ASP.NET Core MVC и структура книги. В следующей главе вы увидите инфраструктуру ASP.NET Core MVC в действии благодаря простой демонстрации средств, которые обеспечивают все ее преимущества.

ГЛАВА 2

Ваше первое приложение MVC

Лучший способ оценки инфраструктуры, предназначенной для разработки программного обеспечения, заключается в том, чтобы приступить непосредственно к ее использованию. В настоящей главе мы создадим простое приложение ввода данных с применением ASP.NET Core MVC. Мы будем решать эту задачу пошагово, чтобы вы поняли, каким образом строится приложение MVC. Для простоты мы пока опускаем некоторые технические подробности. Но не беспокойтесь — если вы только начинаете знакомство с MVC, то узнаете много интересного. Когда что-либо используется без пояснения, то приводится ссылка на главу, в которой находятся все необходимые детали.

Установка Visual Studio

Настоящая книга опирается на продукт Visual Studio 2017, который предоставляет среду разработки для проектов ASP.NET Core MVC. Мы будем применять бесплатную редакцию *Visual Studio 2017 Community*, доступную для загрузки на веб-сайте `www.visualstudio.com`. При установке Visual Studio 2017 вы должны выбрать рабочую нагрузку .NET Core cross-platform development (Межплатформенная разработка .NET Core), как показано на рис. 2.1.

На заметку! Версия Visual Studio 2017 предшествовала выпуску ASP.NET Core MVC 2. В случае установки Visual Studio для более ранних версий ASP.NET Core MVC понадобится применить последние обновления. Применить обновления можно, запустив программу установки Visual Studio и щелкнув на кнопке Update (Обновить).

Совет. Среда Visual Studio поддерживает только Windows. Создавать приложения ASP.NET Core MVC на других платформах можно с использованием Visual Studio Code. Продукт Visual Studio Code не обладает всеми возможностями Visual Studio, но он предлагает великолепный редактор и любые средства, требующиеся для разработки приложений MVC. Детали ищите в главе 13.

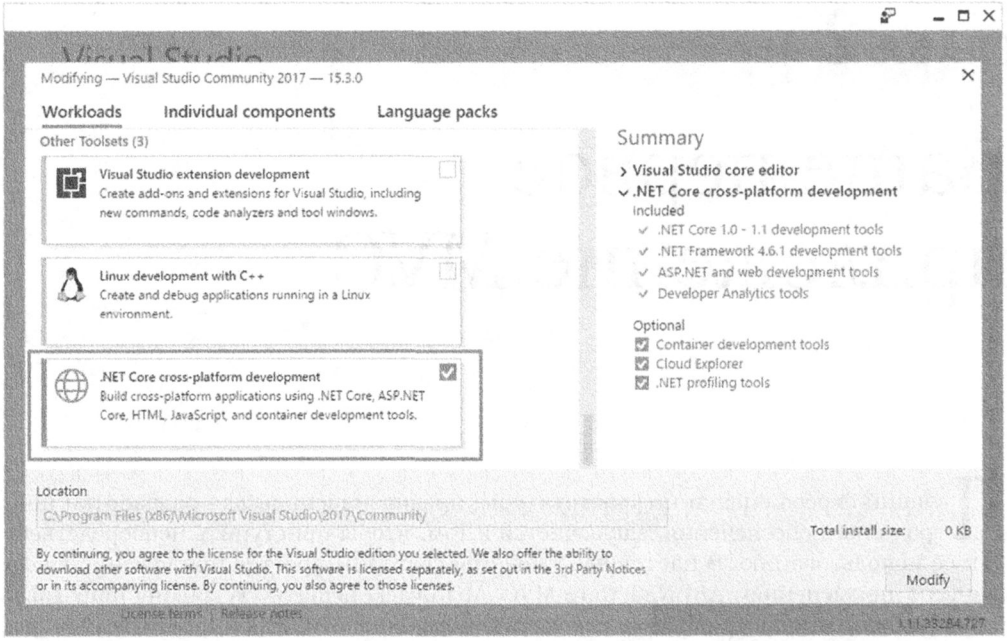


Рис. 2.1. Выбор рабочей нагрузки Visual Studio

Установка .NET Core 2.0 SDK

Установка Visual Studio содержит все средства, необходимые для разработки приложений ASP.NET Core MVC, но не включает комплект .NET Core 2.0 SDK, который должен быть загружен и установлен отдельно.

Перейдем по ссылке `https://www.microsoft.com/net/core`, загрузим программу установки .NET Core SDK для Windows и запустим ее. После завершения работы программы установки откроем окно командной строки или окно PowerShell и введем следующую команду для отображения установленной версии платформы .NET:

```
dotnet --version
```

Если установка прошла успешно, тогда результирующим выводом команды будет `2.0.0`.

Создание нового проекта ASP.NET Core MVC

Мы собираемся начать с создания нового проекта ASP.NET Core MVC в среде Visual Studio. Выберем в меню File (Файл) пункт New⇨Project (Создать⇨Проект), чтобы открыть диалоговое окно New Project (Новый проект). Перейдя в раздел Templates⇨Visual C#⇨Web (Шаблоны⇨Visual C#⇨Веб) в панели слева, можно заметить шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core). Выберем этот тип проекта (рис. 2.2).

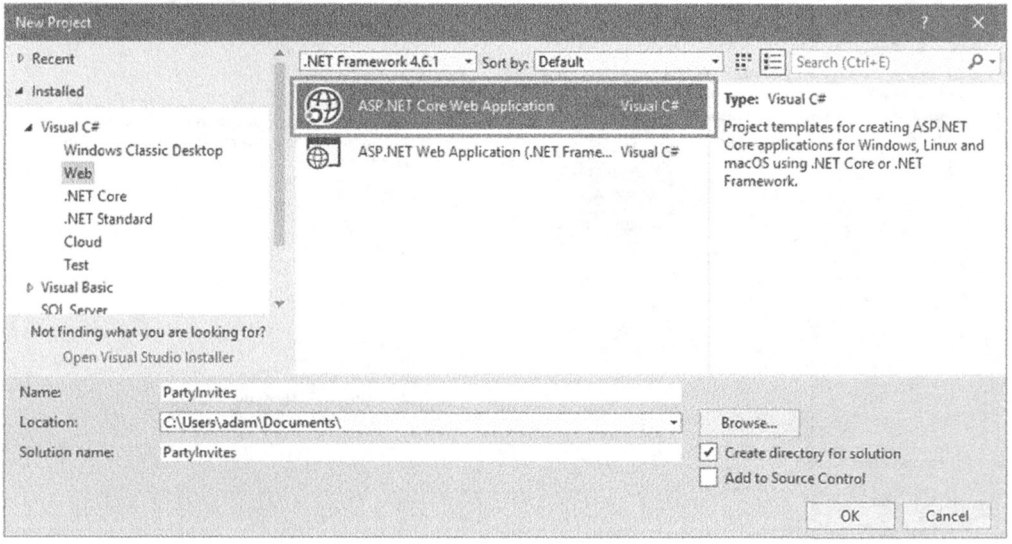


Рис. 2.2. Выбор шаблона проекта ASP.NET Core Web Application

Совет. При выборе шаблона проекта может возникать путаница из-за сильно похожих названий шаблонов. Шаблон ASP.NET Web Application (.NET Framework) (Веб-приложение ASP.NET (.NET Framework)) предназначен для создания проектов с использованием унаследованных версий ASP.NET и MVC Framework, которые предшествовали ASP.NET Core. Указанные два шаблона позволяют создавать приложения ASP.NET Core и отличаются применяемой исполняющей средой, предлагая на выбор .NET Framework или .NET Core. Разница между ними объясняется в главе 6, но в книге повсеместно используется вариант .NET Core, поэтому для получения идентичных результатов при работе с примерами приложений вы должны выбирать именно его.

В поле Name (Имя) для нового проекта введем `PartyInvites`. Для продолжения щелкнем на кнопке OK. Откроется еще одно диалоговое окно, которое предложит установить начальное содержимое проекта. Удостоверимся, что в раскрывающихся списках слева вверху выбраны элементы .NET Core и ASP.NET Core 2.0 (рис. 2.3).

Для шаблона ASP.NET Core Web Application доступно несколько вариантов, каждый из которых приводит к созданию проекта с отличающимся начальным содержимым. Для целей данной главы выберем вариант Web Application (Model-View-Controller) (Веб-приложение (модель-представление-контроллер)), который создаст приложение MVC с заранее определенным содержимым, чтобы немедленно приступить к разработке.

На заметку! Шаблон проекта Web Application (Model-View-Controller) применяется только в настоящей главе. Мне не нравится пользоваться заранее определенными шаблонами, поскольку они потворствуют интерпретации ряда важных средств наподобие аутентификации как черных ящиков. Моя цель здесь — предоставить вам достаточный объем знаний для понимания и управления каждым аспектом приложения MVC, так что в оставшихся материалах книги будет применяться шаблон Empty (Пустой). Текущая глава посвящена быстрому началу процесса разработки, для чего хорошо подходит шаблон Web Application (Model-View-Controller).

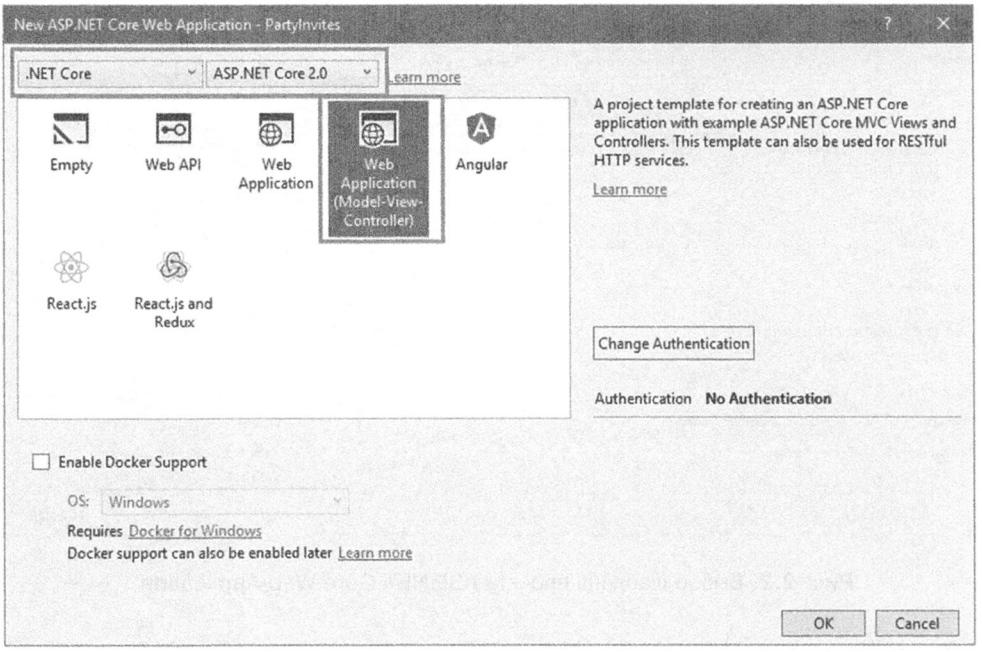


Рис. 2.3. Выбор начальной конфигурации проекта

Щелкнем на кнопке Change Authentication (Изменить аутентификацию) и в открывшемся диалоговом окне Change Authentication (Изменение аутентификации) проверим, что выбран переключатель No Authentication (Аутентификация отсутствует), как показано на рис. 2.4. Данный проект не требует какой-либо аутентификации, а в главах 28–30 будет объясняться, как защищать приложения ASP.NET.

Щелкнем на кнопке OK, чтобы закрыть диалоговое окно Change Authentication. Удостоверимся в том, что флажок Enable Docker Support (Включить поддержку Docker) не отмечен, и затем щелкнем на кнопке OK для создания проекта `PartyInvites`.

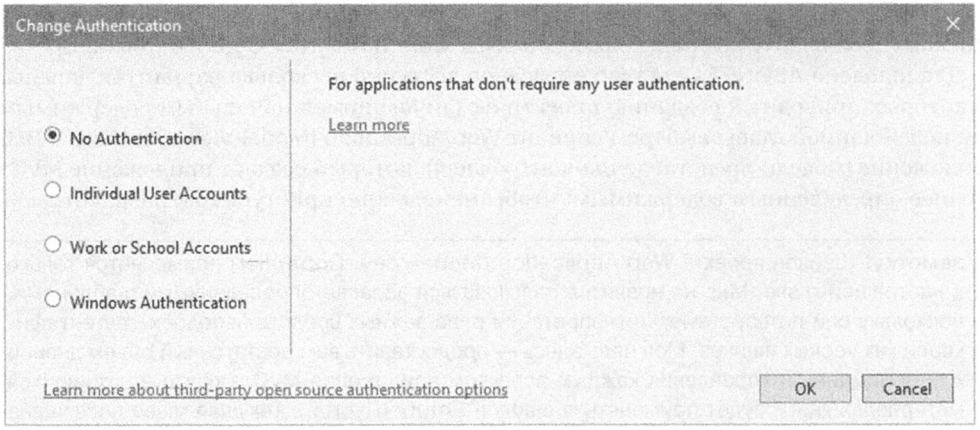


Рис. 2.4. Выбор настроек аутентификации

После того как среда Visual Studio создала проект, в окне Solution Explorer (Проводник решения) появится множество файлов и папок (рис. 2.5). Они представляют собой стандартную структуру для проекта MVC, созданного с использованием шаблона Web Application (Model-View-Controller), и вскоре вы узнаете назначение каждого файла и папки, которые были созданы Visual Studio.

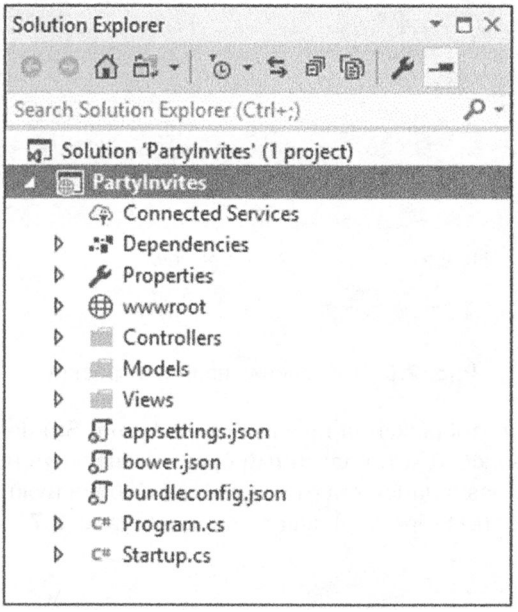


Рис. 2.5. Начальная структура файлов и папок проекта ASP.NET Core MVC

Совет. Если вместо папок `Controllers`, `Models` и `Views` вы видите папку `Pages`, то это означает, что вы выбрали шаблон Web Application (Веб-приложение), а не шаблон Web Application (Model-View-Controller). Я не имею ни малейшего понятия о том, почему в Microsoft решили, что настолько похожие названия шаблонов будут удачной идеей, но в подобной ситуации понадобится удалить созданный проект и начать все заново.

Теперь приложение можно запустить, выбрав в меню Debug (Отладка) пункт Start Debugging (Запустить отладку); если появится запрос на включение отладки, тогда нужно просто щелкнуть на кнопке OK. Среда Visual Studio скомпилирует приложение, с помощью сервера приложений IIS Express запустит его и откроет окно веб-браузера для запроса содержимого приложения. При запуске проекта в первый раз среде Visual Studio потребуется некоторое время; по завершении процесса будет получен результат, показанный на рис. 2.6.

Когда среда Visual Studio создает проект с применением шаблона Web Application (Model-View-Controller), она добавляет базовый код и содержимое, которое вы наблюдаете после запуска приложения. Далее в главе мы заменим такое содержимое, чтобы создать простое приложение MVC.

По завершении понадобится остановить отладку, для чего закрыть окно браузера или вернуться в Visual Studio и выбрать в меню Debug пункт Stop Debugging (Остановить отладку).

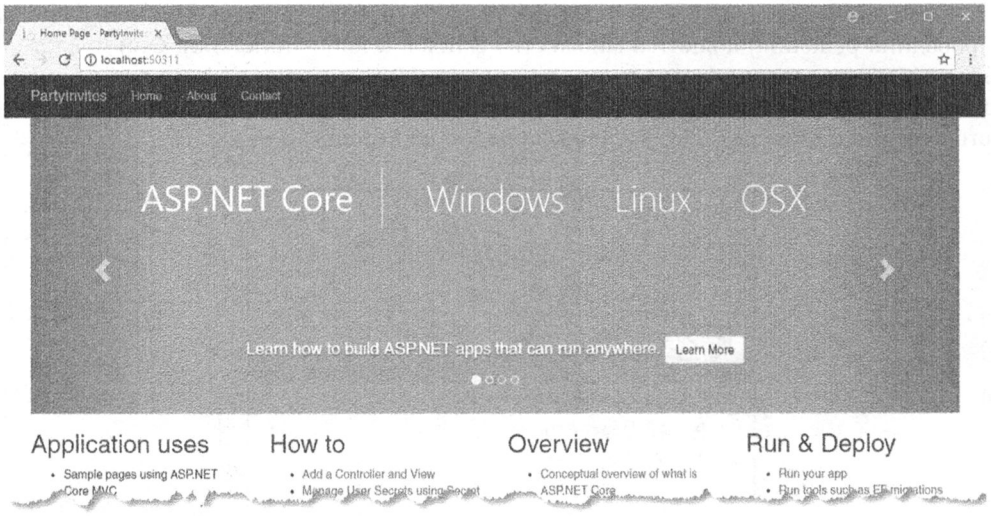


Рис. 2.6. Выполнение примера проекта

Вы видели, что для отображения проекта среда Visual Studio открывает окно браузера. Можно указать любой установленный браузер, щелкнув на кнопке со стрелкой правее кнопки IIS Express в панели инструментов и выбрав нужный вариант из списка в меню Web Browser (Веб-браузер), как показано на рис. 2.7.

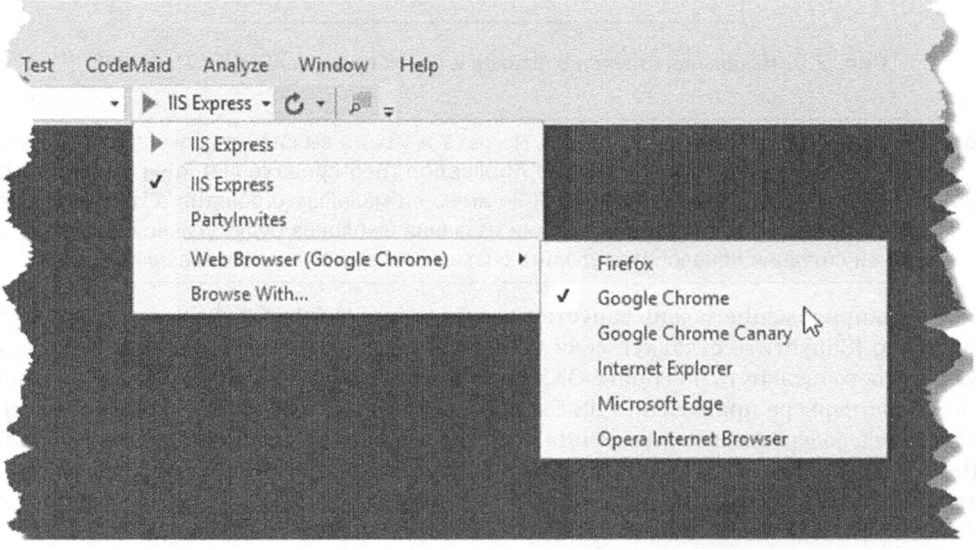


Рис. 2.7. Выбор браузера

В дальнейшем во всех примерах будет использоваться браузер Google Chrome или Google Chrome Canary, но вы можете применять любой современный браузер, включая Microsoft Edge.

Добавление контроллера

В рамках паттерна MVC входящие запросы обрабатываются *контроллерами*. В ASP.NET Core MVC контроллеры — это просто классы C# (обычно унаследованные от класса `Microsoft.AspNetCore.Mvc.Controller`, который является встроенным базовым классом контроллера MVC). Каждый открытый метод в контроллере называется *методом действия*, что означает возможность его вызова из веб-среды через некоторый URL для выполнения какого-то действия. В соответствии с соглашением MVC контроллеры помещаются в папку `Controllers`, автоматически создаваемую Visual Studio при настройке проекта.

Совет. Вы вовсе не обязаны соблюдать указанное или большинство других соглашений MVC, но рекомендуется его придерживаться — и не в последнюю очередь потому, что это поможет уяснить примеры, приведенные в настоящей книге.

Среда Visual Studio добавляет в проект класс стандартного контроллера, который можно увидеть, раскрыв папку `Controllers` в окне Solution Explorer. Файл называется `HomeController.cs`. Файлы классов контроллеров имеют имена, завершающиеся словом `Controller`, т.е. в файле `HomeController.cs` содержится код контроллера по имени `Home` — стандартного контроллера, используемого в приложениях MVC. Щелкнем на имени файла `HomeController.cs` в окне Solution Explorer, чтобы среда Visual Studio открыла его для редактирования. Отобразится код C#, приведенный в листинге 2.1.

Листинг 2.1. Первоначальное содержимое файла `HomeController.cs` из папки `Controllers`

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
  public class HomeController : Controller {
    public IActionResult Index() {
      return View();
    }
    public IActionResult About() {
      ViewData["Message"] = "Your application description page.";
      return View();
    }
    public IActionResult Contact() {
      ViewData["Message"] = "Your contact page.";
      return View();
    }
    public IActionResult Error() {
      return View(new ErrorViewModel { RequestId = Activity.Current?.Id
        ?? HttpContext.TraceIdentifier });
    }
  }
}
```

Заменим код в файле `HomeController.cs` кодом, показанным в листинге 2.2. Здесь были удалены все методы кроме одного, у которого изменен возвращаемый тип и его реализация, а также удалены операторы `using` для неиспользуемых пространств имен.

Листинг 2.2. Изменение содержимого файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
namespace PartyInvites.Controllers {
  public class HomeController : Controller {
    public string Index() {
      return "Hello World";
    }
  }
}
```

Изменения не приводят к особо впечатляющим результатам, но их вполне достаточно для хорошей демонстрации. Метод по имени `Index()` изменен так, что теперь он возвращает строку `"Hello World"`. Снова запустим проект, выбрав пункт Start Debugging из меню Debug в Visual Studio.

Совет. Если вы оставили в функционирующем состоянии приложение из предыдущего раздела, тогда выберите в меню Debug пункт Restart (Перезапустить) или при желании пункт Stop Debugging и затем Start Debugging.

Браузер сделает HTTP-запрос серверу. Стандартная конфигурация MVC предусматривает, что данный запрос будет обрабатываться с применением метода `Index()`, называемого *методом действия* или просто *действием*, а результат, полученный из этого метода, будет отправлен обратно браузеру (рис. 2.8).

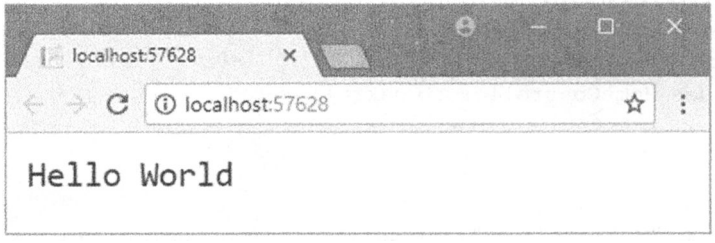


Рис. 2.8. Вывод из метода действия

Совет. Обратите внимание, что среда Visual Studio направляет браузер на порт 57628. Внутри URL, который будет запрашивать ваш браузер, почти наверняка будет присутствовать другой номер порта, т.к. Visual Studio при создании проекта выделяет произвольный порт. Если вы заглянете в область уведомлений панели задач Windows, то найдете там значок для IIS Express. Данный значок представляет усеченную версию полного сервера приложений IIS, которая входит в состав Visual Studio и используется для доставки содержимого и служб ASP.NET во время разработки. Развертывание проекта MVC в производственной среде будет описано в главе 12.

Понятие маршрутов

В дополнение к моделям, представлениям и контроллерам в приложениях MVC применяется *система маршрутизации* ASP.NET, которая определяет, как URL отображаются на контроллеры и действия. Маршрут — это правило, которое используется для решения о том, как обрабатывать запрос. Когда среда Visual Studio создает проект MVC, она добавляет ряд стандартных маршрутов, выступающих в качестве начальных. Можно запрашивать любой из следующих URL, и они будут направлены на действие `Index` класса `HomeController`:

- `/`
- `/Home`
- `/Home/Index`

Таким образом, когда браузер запрашивает `http://`*ваш-сайт*`/` или `http://`*ваш_сайт*`/Home`, он получает вывод из метода `Index()` класса `HomeController`. Можете проверить сказанное самостоятельно, изменив URL в браузере. В настоящий момент он будет выглядеть как `http://localhost:57628/`, но представляющая порт часть может быть другой. Если вы добавите к URL порцию `/Home` или `/Home/Index` и нажмете клавишу <Enter>, то получите от приложения MVC тот же самый результат — строку `"Hello World"`.

Это хороший пример получения выгоды от соблюдения соглашений, поддерживаемых ASP.NET Core MVC. В данном случае соглашение заключается в том, что существует класс контроллера по имени `HomeController`, который будет служить стартовой точкой приложения MVC. Стандартная конфигурация, создаваемая средой Visual Studio для нового проекта, предполагает, что мы будем следовать такому соглашению. И поскольку мы *действительно* соблюдаем соглашение, то автоматически получаем поддержку всех URL из приведенного выше списка. Если *не* следовать соглашению, тогда конфигурацию пришлось бы модифицировать для указания на контроллер, созданный взамен стандартного. В рассматриваемом простом примере стандартной конфигурации вполне достаточно.

Визуализация веб-страниц

Выводом предыдущего примера была не HTML-разметка, а просто строка `"Hello World"`. Чтобы сгенерировать HTML-ответ на запрос браузера, понадобится создать *представление*, которое сообщает MVC, каким образом генерировать ответ на запрос, поступивший из браузера.

Создание и визуализация представления

Прежде всего, необходимо модифицировать метод действия `Index()`, как показано в листинге 2.3. Для простоты восприятия в этом и во всех будущих листингах изменения выделяются полужирным.

Листинг 2.3. Визуализация представления в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {
```

```
public class HomeController : Controller {
  public ViewResult Index() {
    return View("MyView");
  }
}
```

Возвращая из метода действия объект `ViewResult`, мы инструктируем MVC о *визуализации* представления. Объект `ViewResult` создается посредством вызова метода `View()` с указанием имени представления, которое должно применяться, т.е. `MyView`. Запустив приложение, можно заметить, что инфраструктура MVC пытается найти представление, как отражено в сообщении об ошибке на рис. 2.9.

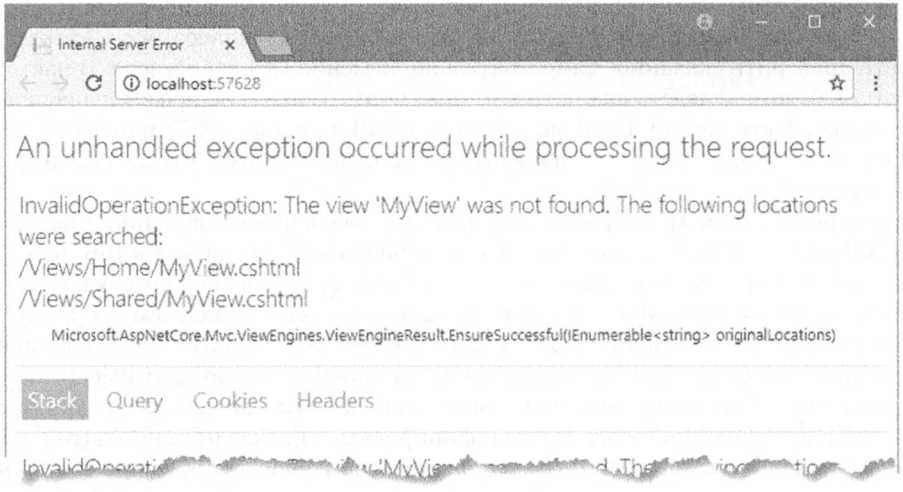


Рис. 2.9. Инфраструктура MVC пытается найти представление

Сообщение об ошибке очень полезно. Оно объясняет, что инфраструктура MVC не смогла найти представление, указанное для метода действия, и показывает, где производился поиск. Представления хранятся в подпапках внутри папки `Views`. Например, представления, которые связаны с контроллером `Home`, содержатся в папке по имени `Views/Home`. Представления, не являющиеся специфическими для отдельного контроллера, хранятся в папке под названием `Views/Shared`. Среда Visual Studio создает папки `Home` и `Shared` автоматически, когда используется шаблон Web Application (Model-View-Controller), и в рамках начальной подготовки проекта помещает в них несколько представлений-заполнителей.

Чтобы создать представление, необходимое для текущего примера, раскроем папку `Views` в окне Solution Explorer, щелкнем правой кнопкой мыши на папке `Home` и выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент). Среда Visual Studio предложит список шаблонов элементов. Добравшись до категории ASP.NET Core⇨Web⇨ASP.NET в панели слева, выберем элемент MVC View Page (Страница представления MVC) в центральной панели (рис. 2.10). (Не следует применять шаблон Razor Page (Страница Razor), который не относится к MVC Framework.)

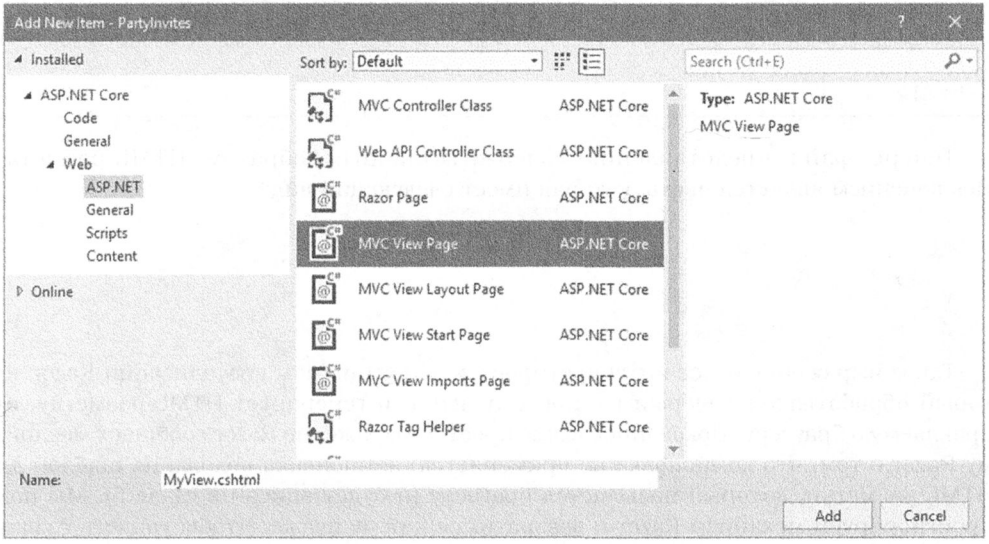


Рис. 2.10. Создание представления

Совет. В папке `Views` уже есть несколько файлов, которые были добавлены Visual Studio с целью предоставления начального содержимого, показанного на рис. 2.6. Такие файлы можно спокойно проигнорировать.

Введем в поле Name (Имя) имя `MyView.cshtml` и щелкнем на кнопке Add (Добавить) для создания представления. Среда Visual Studio создаст файл `Views/Home/MyView.cshtml` и откроет его для редактирования. Начальное содержимое файла представления — это просто ряд комментариев и заполнитель. Заменим его содержимым, приведенным в листинге 2.4.

Совет. Довольно легко создать файл представления не в той папке. Если в итоге вы не получили файл по имени `MyView.cshtml` в папке `Views/Home`, тогда удалите созданный файл и попробуйте создать заново.

Листинг 2.4. Замена содержимого файла `MyView.cshtml` из папки `Views/Home`

```
@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  <div>
```

```
    Hello World (from the view)
  </div>
</body>
</html>
```

Теперь файл представления содержит главным образом HTML-разметку. Исключением является часть, которая имеет следующий вид:

```
...
@{
  Layout = null;
}
...
```

Такое выражение будет интерпретироваться механизмом визуализации Razor, который обрабатывает содержимое представлений и генерирует HTML-разметку, отправляемую браузеру. Показанное выше простое выражение Razor сообщает механизму Razor о том, что компоновка не применяется; компоновка похожа на шаблон для HTML-разметки, который посылается браузеру (и будет описан в главе 5). Мы пока проигнорируем механизм Razor и возвратимся к нему позже. Чтобы увидеть созданное представление, выберем в меню Debug пункт Start Debugging для запуска приложения. Должен получиться результат, приведенный на рис. 2.11.

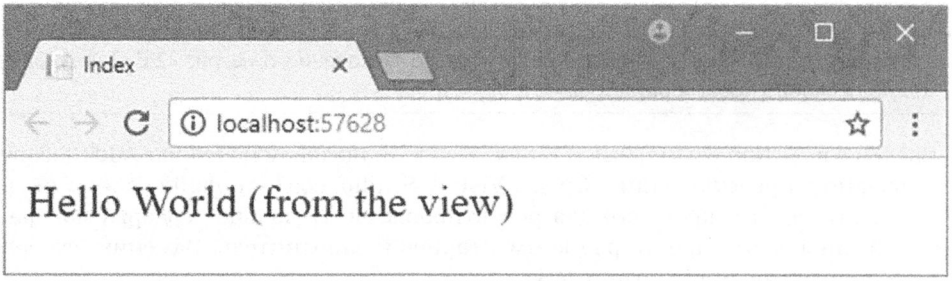


Рис. 2.11. Тестирование представления

При первом редактировании метод действия `Index()` возвращал строковое значение, так что инфраструктура MVC всего лишь передавала браузеру строковое значение в том виде, как есть. Теперь, когда метод `Index()` возвращает объект `ViewResult`, инфраструктура MVC визуализирует представление и возвращает сгенерированную HTML-разметку. Мы сообщили инфраструктуре MVC о том, какое представление должно использоваться, поэтому с помощью соглашения об именовании она автоматически выполнила его поиск. Соглашение предполагает, что имя файла представления совпадает с именем метода действия, а файл представления хранится в папке, названной по имени контроллера: `/Views/Home/MyView.cshtml`.

Кроме строк и объектов `ViewResult` методы действий могут возвращать другие результаты. Например, если мы возвращаем объект `RedirectResult`, то браузер будет перенаправлен на другой URL. Если мы возвращаем объект `HttpUnauthorizedResult`, то предлагаем пользователю войти в систему. Все вместе такие объекты называются *результатами действий*. Система результатов действий позволяет инкапсулировать и повторно использовать часто встречающиеся ответы в действиях. В главе 17 мы рассмотрим их подробнее и продемонстрируем разные способы их применения.

Добавление динамического вывода

Весь смысл платформы для разработки веб-приложений состоит в конструировании и отображении *динамического* вывода. В рамках MVC работа контроллера заключается в подготовке данных и передаче их представлению, которое отвечает за их визуализацию в виде HTML-разметки.

Один из способов передачи данных из контроллера в представление предусматривает использование объекта `ViewBag`, который является членом базового класса `Controller`. По существу `ViewBag` — это динамический объект, в котором можно устанавливать произвольные свойства, делая их значения доступными в любом визуализируемом далее представлении. В листинге 2.5 демонстрируется передача таким способом простых динамических данных в файле `HomeController.cs`.

Листинг 2.5. Установка данных представления в `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() {
      int hour = DateTime.Now.Hour;
      ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
      return View("MyView");
    }
  }
}
```

Данные для представления указываются во время присваивания значения свойству `ViewBag.Greeting`. Свойство `Greeting` не существует вплоть до момента, когда ему присваивается значение, что позволяет передавать данные из контроллера в представление в свободной и гибкой манере, без необходимости в предварительном определении классов. Чтобы получить значение данных, необходимо еще раз сослаться на свойство `ViewBag.Greeting`, но уже в представлении, как показано в листинге 2.6, содержащем изменение, которое было внесено в файл `MyView.cshtml`.

Листинг 2.6. Извлечение значения данных `ViewBag` в файле `MyView.cshtml` из папки `Views/Home`

```
@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  <div>
```

```
    @ViewBag.Greeting World (from the view)
  </div>
</body>
</html>
```

Добавленный в листинге фрагмент — это выражение Razor, которое оценивается, когда MVC применяет представление для генерации ответа. Вызов метода `View()` в методе `Index()` контроллера приводит к тому, что MVC находит файл представления `MyView.cshtml` и запрашивает у механизма визуализации Razor синтаксический анализ содержимого данного файла. Механизм Razor ищет выражения, подобные добавленному в листинге 2.6, и обрабатывает их. В рассматриваемом примере обработка выражения означает вставку в представление значения, которое было присвоено свойству `ViewBag.Greeting` внутри метода действия.

Выбор для свойства имени `Greeting` не диктуется какими-то особыми соображениями. Его можно было бы заменить любым другим именем, и все работало бы точно так же при условии, что имя, используемое в контроллере, совпадает с именем, которое применяется в представлении. Присваивая значения более чем одному свойству, можно передавать из контроллера в представление множество значений данных. После запуска проекта будет виден результат внесенных изменений (рис. 2.12).

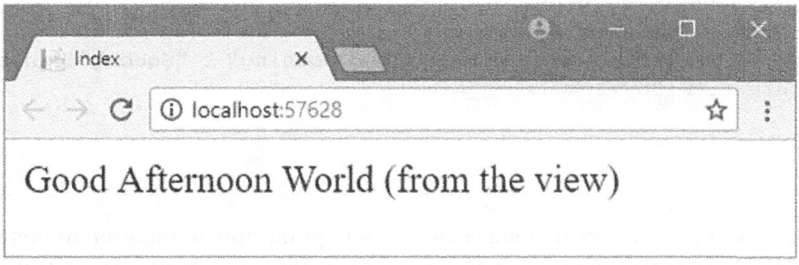


Рис. 2.12. Динамический ответ, сгенерированный MVC

Создание простого приложения для ввода данных

В оставшихся разделах главы будут исследованы другие базовые функциональные средства MVC за счет построения простого приложения для ввода данных. В этом разделе мы собираемся несколько увеличить темп изложения. Целью является демонстрация инфраструктуры MVC в действии, а потому некоторые объяснения того, что происходит "за кулисами", будут пропущены. Однако не беспокойтесь — мы вернемся к подробному обсуждению данных тем в последующих главах.

Предварительная настройка

Представьте себе, что ваша подруга решила организовать вечеринку в канун нового года и попросила создать веб-приложение, которое позволяет приглашенным ответить на приглашение по электронной почте. Она высказала пожелание относительно четырех основных средств, которые перечислены ниже:

- домашняя страница, отображающая информацию о вечеринке;
- форма, которая может использоваться для ответа на приглашение (répondez s'il vous plaît — RSVP);

- проверка достоверности для формы RSVP, которая будет отображать страницу с выражением благодарности за внимание;
- итоговая страница, которая показывает, кто собирается прийти на вечеринку.

В последующих разделах мы достроим проект MVC, созданный в начале главы, и добавим в него перечисленные выше средства. Первый пункт можно убрать из списка, применив то, что было показано ранее — добавить HTML-разметку с подробной информацией о вечеринке в существующее представление. В листинге 2.7 приведено содержимое файла `Views/Home/MyView.cshtml` с внесенными дополнениями.

Листинг 2.7. Отображение подробностей о вечеринке в файле `MyView.cshtml` из папки `Views/Home`

```
@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  <div>
    @ViewBag.Greeting World (from the view)
    <p>We're going to have an exciting party.<br />
    (To do: sell it better. Add pictures or something.)
    </p>
  </div>
</body>
</html>
```

Мы двигаемся в верном направлении. Если запустить приложение, выбрав в меню Debug пункт Start Debugging, то отобразятся подробности о вечеринке — точнее, заполнитель для подробностей, но сама идея должна быть понятной (рис. 2.13).

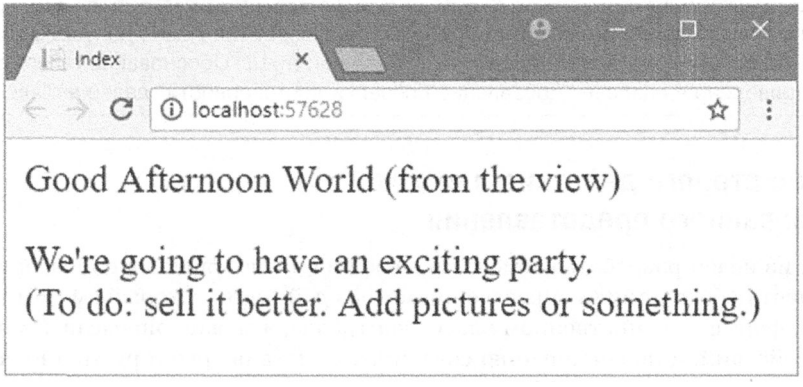


Рис. 2.13. Добавление информации о вечеринке в HTML-разметку представления

Проектирование модели данных

Буква "M" в аббревиатуре MVC обозначает *model* (модель), которая является самой важной частью приложения. Модель — это представление реальных объектов, процессов и правил, которые определяют сферу приложения, известную как *предметная область*. Модель, которую часто называют *моделью предметной области*, содержит объекты C# (или *объекты предметной области*), образующие "вселенную" приложения, и методы, позволяющие манипулировать ими. Представления и контроллеры открывают доступ клиентам к предметной области в согласованной манере, и любое корректно разработанное приложение MVC начинается с хорошо спроектированной модели, которая затем служит центральным узлом при добавлении контроллеров и представлений.

Для проекта `PartyInvites` сложная модель не требуется, поскольку приложение совсем простое, и нужно создать только один класс предметной области, который получит имя `GuestResponse`. Такой объект будет отвечать за хранение, проверку достоверности и подтверждение ответа на приглашение (RSVP).

По соглашению MVC классы, которые образуют модель, помещаются в папку по имени `Models`, которую среда Visual Studio создает автоматически в случае выбора шаблона Web Application (Model-View-Controller).

Чтобы создать файл класса, щелкнем правой кнопкой мыши на папке `Models` в окне Solution Explorer и выберем в контекстном меню пункт Add⇨Class (Добавить⇨Класс). Назначим новому классу имя `GuestResponse.cs` и щелкнем на кнопке Add (Добавить). Приведем содержимое нового файла класса к виду, показанному в листинге 2.8.

Листинг 2.8. Содержимое файла `GuestResponse.cs` из папки `Models`

```
namespace PartyInvites.Models {

  public class GuestResponse {

    public string Name { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public bool? WillAttend { get; set; }
  }
}
```

Совет. Вы могли заметить, что свойство `WillAttend` имеет тип `bool`, допускающий `null`, т.е. оно может принимать значение `true`, `false` или `null`. Обоснование такого решения будет приведено в разделе "Добавление проверки достоверности" далее в главе.

Создание второго действия и строго типизированного представления

Одной из целей разрабатываемого приложения является включение формы RSVP, что означает необходимость определения метода действия, который сможет получать запросы к форме. В единственном классе контроллера можно определять множество методов действий, а по соглашению связанные действия группируются вместе в одном контроллере. В листинге 2.9 иллюстрируется добавление нового метода действия к контроллеру `Home`.

Листинг 2.9. Добавление метода действия в файле `HomeController.cshtml` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() {
      int hour = DateTime.Now.Hour;
      ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
      return View("MyView");
    }

    public ViewResult RsvpForm() {
      return View();
    }
  }
}
```

Метод действия `RsvpForm()` вызывает метод `View()` без аргументов, что сообщает инфраструктуре MVC о необходимости визуализации стандартного представления, связанного с этим методом действия, которым будет представление с таким же именем, как у метода действия (`RsvpForm.cshtml` в данном случае).

Щелкнем правой кнопкой мыши на папке `Views/Home` и выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент). Выберем шаблон MVC View Page (Страница представления MVC), укажем `RsvpForm.cshtml` в качестве имени нового файла и щелкнем на кнопке Add (Добавить), чтобы создать файл. Приведем содержимое нового файла в соответствие с листингом 2.10.

Листинг 2.10. Содержимое файла `RsvpForm.cshtml` из папки `Views/Home`

```
@model PartyInvites.Models.GuestResponse

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RsvpForm</title>
</head>
<body>
  <div>
    This is the RsvpForm.cshtml View
  </div>
</body>
</html>
```

Содержимое состоит в основном из HTML-разметки, но с добавлением Razor-выражения `@model`, которое используется для создания *строго типизированного представления*. Строго типизированное представление предназначено для визуали-

зации специфического типа модели, и если указан желаемый тип (в данном случае класс `GuestResponse` из пространства имен `PartyInvites.Models`), то MVC может создать ряд удобных сокращений, чтобы сделать его проще. Вскоре мы задействуем преимущество характеристики строгой типизации.

Чтобы протестировать новый метод действия и его представление, запустим приложение, выбрав в меню Debug пункт Start Debugging, и с помощью браузера перейдем на URL вида `/Home/RsvpForm`.

Инфраструктура MVC применит описанное ранее соглашение об именовании для направления запроса методу действия `RsvpForm()`, определенному в контроллере `Home`. Данный метод действия указывает MVC о том, что должно визуализироваться стандартное представление, которое посредством еще одного применения того же соглашения об именовании визуализирует `RsvpForm.cshml` из папки `Views/Home`. Результат показан на рис. 2.14.



Рис. 2.14. Визуализация второго представления

Ссылка на методы действий

Нам необходимо создать в представлении `MyView` ссылку, чтобы гости могли видеть представление `RsvpForm` без обязательного знания URL, который указывает на специфический метод действия (листинг 2.11).

Листинг 2.11. Добавление ссылки на форму RSVP в файле `MyView.cshtml` из папки `Views/Home`

```
@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  <div>
    @ViewBag.Greeting World (from the view)
    <p>We're going to have an exciting party.<br />
    (To do: sell it better. Add pictures or something.)
    </p>
```

```
    <a asp-action="RsvpForm">RSVP Now</a>
  </div>
</body>
</html>
```

В листинге 2.11 добавлен элемент `a`, который имеет атрибут `asp-action`. Данный атрибут является примером атрибута *вспомогательной функции дескриптора*, т.е. инструкцией Razor, которая будет выполнена, когда представление визуализируется. Атрибут `asp-action` — это инструкция по добавлению к элементу `a` атрибута `href`, содержащего URL для метода действия. Работа вспомогательных функций дескрипторов объясняется в главах 24–26, а пока достаточно знать, что `asp-action` — простейший вид атрибута вспомогательной функции дескриптора для элементов `a`. Он указывает Razor на необходимость вставки URL для метода действия, определенного в том же контроллере, для которого визуализируется текущее представление. Запустив проект, можно увидеть ссылку, которую создала вспомогательная функция дескриптора (рис. 2.15).

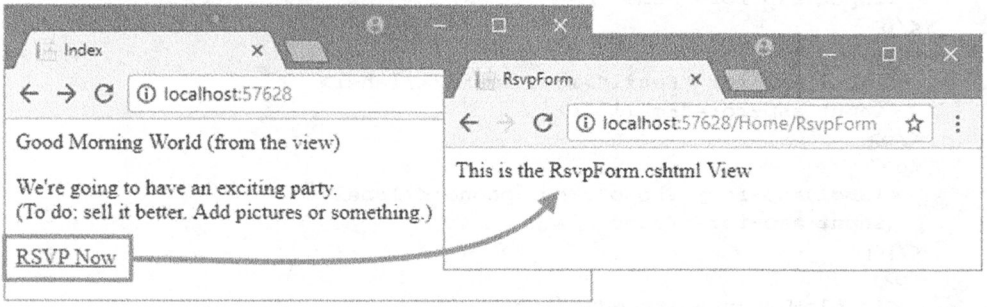


Рис. 2.15. Ссылка на метод действия

Если после запуска приложения навести курсор мыши на ссылку RSVP Now (Ответить на приглашение) в окне браузера, то можно заметить, что ссылка указывает на следующий URL (возможно, вашему проекту Visual Studio назначит другой номер порта):

```
http://localhost:57628/Home/RsvpForm
```

Здесь требуется соблюдать один важный принцип: вы должны использовать средства, предлагаемые MVC для генерации URL, а не жестко кодировать их в своих представлениях. Когда вспомогательная функция дескриптора создает атрибут `href` для элемента `a`, она инспектирует конфигурацию приложения, чтобы выяснить, каким должен быть URL. В итоге появляется возможность изменять конфигурацию приложения с целью поддержки разных форматов URL без необходимости в обновлении каких-либо представлений. Особенности работы такого механизма рассматриваются в главе 15.

Построение формы

Теперь, когда строго типизированное представление создано и достижимо из представления `Index`, займемся подгонкой содержимого файла `RsvpForm.cshtml`, чтобы превратить его в HTML-форму для редактирования объектов `GuestResponse` (листинг 2.12).

Листинг 2.12. Создание представления в виде формы в файле `RsvpForm.cshtml` из папки `Views/Home`

```
@model PartyInvites.Models.GuestResponse

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RsvpForm</title>
</head>
<body>
  <form asp-action="RsvpForm" method="post">
    <p>
      <label asp-for="Name">Your name:</label>
      <input asp-for="Name" />
    </p>
    <p>
      <label asp-for="Email">Your email:</label>
      <input asp-for="Email" />
    </p>
    <p>
      <label asp-for="Phone">Your phone:</label>
      <input asp-for="Phone" />
    </p>
    <p>
      <label>Will you attend?</label>
      <select asp-for="WillAttend">
        <option value="">Choose an option</option>
        <option value="true">Yes, I'll be there</option>
        <option value="false">No, I can't come</option>
      </select>
    </p>
    <button type="submit">Submit RSVP</button>
  </form>
</body>
</html>
```

Для каждого свойства класса модели `GuestResponse` определены элементы `label` и `input` (или элемент `select` в случае свойства `WillAttend`). Каждый элемент ассоциирован со свойством модели с применением еще одного атрибута вспомогательной функции дескриптора — `asp-for`. Атрибуты вспомогательных функций дескрипторов конфигурируют элементы, чтобы привязать их к объекту модели. Вот пример HTML-разметки, которую генерируют вспомогательные функции дескрипторов для отправки браузеру:

```
<p>
  <label for="Name">Your name:</label>
  <input type="text" id="Name" name="Name" value="">
</p>
```

Атрибут `asp-for` в элементе `label` устанавливает значение атрибута `for`. Атрибут `asp-for` в элементе `input` устанавливает атрибуты `id` и `name`. В данный момент это не выглядит особенно полезным, но по мере определения прикладной функциональности вы увидите, что ассоциирование элементов со свойством модели предлагает дополнительные преимущества.

Более непосредственный результат дает атрибут `asp-action` в элементе `form`, который использует конфигурацию маршрутизации URL приложения для установки атрибута `action` в URL, нацеленный на специфический метод действия, например:

```
<form method="post" action="/Home/RsvpForm">
```

Как и в случае атрибута вспомогательной функции дескриптора, примененного к элементу `a`, преимущество такого подхода заключается в том, что вы можете изменять систему URL, используемую приложением, и содержимое, которое генерируется вспомогательными функциями дескрипторов, автоматически отразит изменения.

Форму можно увидеть, запустив приложение и щелкнув на ссылке RSVP Now (рис. 2.16).

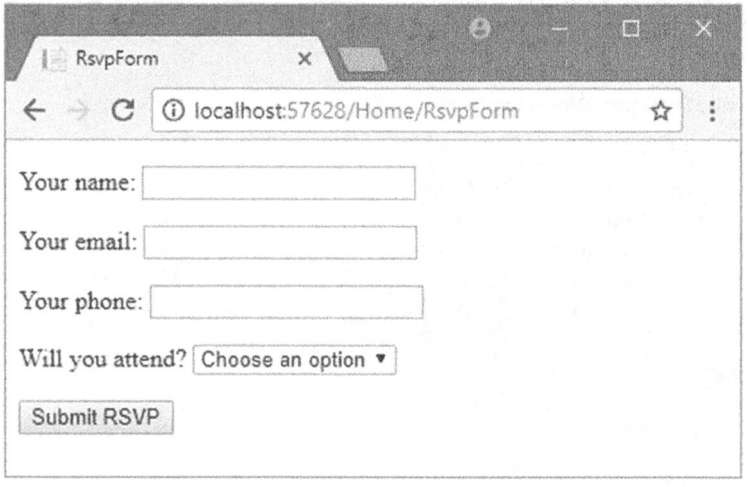


Рис. 2.16. Добавление HTML-формы к приложению

Получение данных формы

Мы пока еще не указали инфраструктуре MVC, что должно быть сделано, когда форма отправляется серверу. При нынешнем состоянии приложения щелчок на кнопке Submit RSVP (Отправить RSVP) лишь очищает любые значения, введенные внутри формы. Причина в том, что форма осуществляет обратную отправку методу действия `RsvpForm()` контроллера `Home`, который только сообщает MVC о необходимости повторной визуализации представления.

Чтобы получить и обработать отправленные данные формы, мы собираемся воспользоваться основной возможностью контроллера. Мы добавим второй метод действия `RsvpForm()`, чтобы получить в свое распоряжение следующие возможности.

- *Метод, который отвечает на HTTP-запросы GET.* Каждый раз, когда кто-то щелкает на ссылке, браузер обычно выдает запрос GET. Эта версия действия будет отвечать за отображение изначально пустой формы, когда кто-нибудь впервые посещает /Home/RsvpForm.
- *Метод, который отвечает на HTTP-запросы POST.* По умолчанию формы, визуализированные с помощью Html.BeginForm(), отправляются браузером как запросы POST. Эта версия действия будет отвечать за получение отправленных данных и принятие решения о том, что с ними делать.

Обработка запросов GET и POST в отдельных методах C# способствует обеспечению аккуратности кода контроллера, т.к. описанные выше два метода имеют разные обязанности. Оба метода действий вызываются через тот же самый URL, но в зависимости от вида запроса — GET или POST — инфраструктура MVC вызывает подходящий метод. В листинге 2.13 показаны изменения, которые необходимо внести в класс HomeController.

Листинг 2.13. Добавление метода в файле HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() {
      int hour = DateTime.Now.Hour;
      ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
      return View("MyView");
    }

    [HttpGet]
    public ViewResult RsvpForm() {
      return View();
    }
    [HttpPost]
    public ViewResult RsvpForm(GuestResponse guestResponse) {
      // Что сделать: сохранить ответ от гостя
      return View();
    }
  }
}
```

Существующий метод действия RsvpForm() был снабжен атрибутом HttpGet, который указывает MVC на то, что данный метод должен применяться только для запросов GET. Затем была добавлена перегруженная версия метода RsvpForm(), принимающая объект GuestResponse. К ней был применен атрибут HttpPost, который сообщает MVC о том, что новый метод будет иметь дело только с запросами POST. Произведенные добавления объясняются в последующих разделах. Кроме того, было импортировано пространство имен PartyInvites.Models. Это сделано для того, чтобы на тип модели GuestResponse можно было ссылаться без необходимости в указании полностью определенного имени класса.

Использование привязки модели

Первая перегруженная версия метода действия RsvpForm() визуализирует то же самое представление, что и ранее (файл RsvpForm.cshtml), для генерации формы, показанной на рис. 2.16. Вторая перегруженная версия более интересна из-за наличия параметра. Но с учетом того, что данный метод действия будет вызываться в ответ на HTTP-запрос POST, а тип GuestResponse является классом C#, каким образом они соединяются между собой?

Секрет кроется в *привязке модели* — удобной функциональной возможности MVC, посредством которой производится разбор входящих данных и применение пар "ключ/значение" в HTTP-запросе для заполнения свойств в типах моделей предметной области.

Привязка модели — мощное и настраиваемое средство, которое избавляет от кропотливого и тяжелого труда по взаимодействию с HTTP-запросами напрямую и позволяет работать с объектами C#, а не иметь дело с индивидуальными значениями данных, отправляемыми браузером. Объект GuestResponse, который передается методу действия RsvpForm() в качестве параметра, автоматически заполняется данными из полей формы. Привязка модели, включая ее настройку, подробно рассматривается в главе 26.

Одной из целей приложения является предоставление итоговой страницы с деталями о том, кто придет на вечеринку, что означает необходимость сохранения получаемых ответов. Мы собираемся делать это за счет создания в памяти коллекции объектов. В реальном приложении такой подход не подойдет, т.к. данные ответов будут утрачиваться в результате останова или перезапуска приложения, но он позволяет сосредоточить внимание на MVC и создать приложение, которое может быть легко сброшено в свое начальное состояние.

Совет. В главе 8 будет продемонстрировано использование MVC для постоянного хранения и доступа к данным как часть более реалистичного примера приложения под названием SportsStore.

Мы добавили в проект новый файл, щелкнув правой кнопкой мыши на папке Models и выбрав в контекстном меню пункт Add⇨Class (Добавить⇨Класс). Файл имеет имя Repository.cs и содержимое, показанное в листинге 2.14.

Листинг 2.14. Содержимое файла Repository.cs из папки Models

```
using System.Collections.Generic;

namespace PartyInvites.Models {

  public static class Repository {
    private static List<GuestResponse> responses =
       new List<GuestResponse>();
    public static IEnumerable<GuestResponse> Responses {
      get {
        return responses;
      }
    }
    public static void AddResponse(GuestResponse response) {
      responses.Add(response);
    }
  }
}
```

Класс Repository и его члены объявлены статическими, чтобы облегчить сохранение и извлечение данных в разных местах приложения. Инфраструктура MVC предлагает более сложный подход к определению общей функциональности, называемый *внедрением зависимостей*, который будет описан в главе 18, но для простого приложения вроде рассматриваемого вполне достаточно и статического класса.

Сохранение ответов

Теперь, когда есть куда сохранять данные, можно обновить метод действия, который получает HTTP-запросы POST (листинг 2.15).

Листинг 2.15. Обновление метода действия в файле HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() {
      int hour = DateTime.Now.Hour;
      ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
      return View("MyView");
    }

    [HttpGet]
    public ViewResult RsvpForm() {
      return View();
    }

    [HttpPost]
    public ViewResult RsvpForm(GuestResponse guestResponse) {
      Repository.AddResponse(guestResponse);
      return View("Thanks", guestResponse);
    }
  }
}
```

Все, что необходимо сделать с данными формы, отправленными в запросе — передать методу Repository.AddResponse() в качестве аргумента объект GuestResponse, который был передан методу действия, чтобы ответ мог быть сохранен.

Почему привязка модели не похожа на инфраструктуру Web Forms?

В главе 1 упоминалось, что одним из недостатков традиционной инфраструктуры ASP.NET Web Forms было сокрытие деталей HTTP и HTML от разработчиков. Вас может интересовать, делает ли то же самое привязка модели MVC, которая применялась для создания объекта GuestResponse из HTTP-запроса POST в листинге 2.15.

Нет, не делает. Привязка модели освобождает нас от решения утомительной и подверженной ошибкам задачи по инспектированию HTTP-запроса и извлечению всех требующихся значений данных, но (что самое важное) при желании мы могли бы обрабатывать запрос

вручную, поскольку MVC обеспечивает легкий доступ ко всем данным запроса. Ничто не скрыто от разработчика, но есть несколько удобных средств, которые упрощают работу с HTTP и HTML; тем не менее, использовать их вовсе не обязательно.

Это может показаться едва заметной разницей, но по мере углубления знаний инфраструктуры MVC вы увидите, что практика разработки в ней полностью отличается от традиционной инфраструктуры Web Forms. Вы всегда будете осведомлены относительно того, как обрабатываются получаемые приложением запросы.

Вызов метода `View()` внутри метода действия `RsvpForm()` сообщает MVC о том, что нужно визуализировать представление по имени `Thanks` и передать ему объект `GuestResponse`. Для создания упомянутого представления щелкнем правой кнопкой мыши на папке `Views/Home` в окне Solution Explorer и выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент). Укажем шаблон MVC View Page (Страница представления MVC) из категории ASP.NET, назначим ему имя `Thanks.cshtml` и щелкнем на кнопке Add (Добавить). Среда Visual Studio создаст файл `Views/Home/Thanks.cshtml` и откроет его для редактирования. Поместим в файл содержимое, приведенное в листинге 2.16.

Листинг 2.16. Содержимое файла `Thanks.cshtml` из папки `Views/Home`

```
@model PartyInvites.Models.GuestResponse

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Thanks</title>
</head>
<body>
  <p>
    <h1>Thank you, @Model.Name!</h1>
    @if (Model.WillAttend == true) {
      @:It's great that you're coming. The drinks are already in the fridge!
    } else {
      @:Sorry to hear that you can't make it, but thanks for letting us know.
    }
  </p>
  <p>Click <a asp-action="ListResponses">here</a> to see who is coming.</p>
</body>
</html>
```

Представление `Thanks.cshtml` применяет механизм визуализации Razor для отображения содержимого на основе значения свойства `GuestResponse`, которое передается методу `View()` внутри `RsvpForm()`. Выражение `@model` синтаксиса Razor указывает тип модели предметной области, с помощью которого представление строго типизировано.

Для доступа к значению свойства в объекте предметной области используется конструкция `Model.`*`ИмяСвойства`*. Например, чтобы получить значение свойства

Name, применяется Model.Name. Не беспокойтесь, если синтаксис Razor пока не понятен — он более подробно объясняется в главе 5.

Теперь, когда создано представление Thanks, мы получили базовый пример обработки формы посредством MVC. Запустим приложение в Visual Studio, выбрав в меню Debug пункт Start Debugging, щелкнем на ссылке RSVP Now, введем на форме какие-нибудь данные и щелкнем на кнопке Submit RSVP. Появится результат, показанный на рис. 2.17 (он может отличаться, если введено другое имя либо указано о невозможности посетить вечеринку).

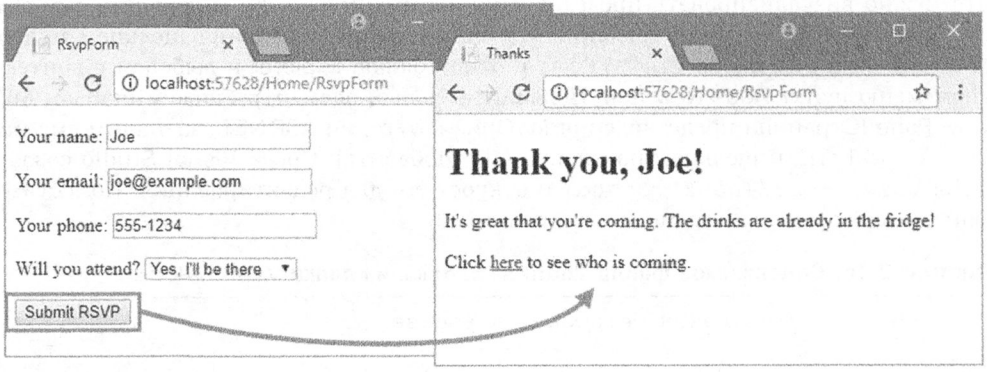


Рис. 2.17. Представление Thanks

Отображение ответов

В конце представления Thanks.cshtml мы добавили элемент a для создания ссылки, которая позволяет отобразить список людей, собирающихся посетить вечеринку. С применением атрибута вспомогательной функции дескриптора asp-action создается URL, который нацелен на метод действия по имени ListResponses():

```
...
<p>Click <a asp-action="ListResponses">here</a> to see who is coming.
</p>
...
```

Наведя курсор мыши на ссылку, которую отображает браузер, легко заметить, что она указывает на URL вида /Home/ListResponses. Это не соответствует ни одному методу действия в контроллере Home, и если щелкнуть на ссылке, то появится страница ошибки "404 — Not Found" (не найдено).

Мы устраним проблему путем создания в контроллере Home метода действия, на который нацелен URL (листинг 2.17).

Листинг 2.17. Добавление метода действия в файле HomeController.cs

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {

  public class HomeController : Controller {
```

```
    public ViewResult Index() {
      int hour = DateTime.Now.Hour;
      ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
      return View("MyView");
    }

    [HttpGet]
    public ViewResult RsvpForm() {
      return View();
    }

    [HttpPost]
    public ViewResult RsvpForm(GuestResponse guestResponse) {
      Repository.AddResponse(guestResponse);
      return View("Thanks", guestResponse);
    }

    public ViewResult ListResponses() {
      return View(Repository.Responses.Where(r => r.WillAttend == true));
    }
  }
}
```

Новый метод действия называется `ListResponses()`; он вызывает метод `View()`, используя свойство `Repository.Responses` в качестве аргумента. Именно так метод действия предоставляет данные строго типизированному представлению. Коллекция объектов `GuestResponse` фильтруется с применением LINQ, так что используются только ответы с положительным решением об участии в вечеринке.

Метод действия `ListResponses()` не указывает имя представления, которое должно применяться для отображения коллекции объектов `GuestResponse`, поэтому будет задействовано соглашение об именовании и MVC инициирует поиск представления по имени `ListResponses.cshtml` в папках `Views/Home` и `Views/Shared`. Чтобы создать представление, щелкнем правой кнопкой мыши на папке `Views/Home` в окне Solution Explorer и выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент). Выберем шаблон MVC View Page (Страница представления MVC) из категории ASP.NET, назначим ему имя `ListResponses.cshtml` и щелкнем на кнопке Add (Добавить). Приведем содержимое нового файла представления в соответствие с листингом 2.18.

Листинг 2.18. Отображение принятых приглашений в файле `ListResponses.cshtml` из папки `Views/Home`

```
@model IEnumerable<PartyInvites.Models.GuestResponse>

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Responses</title>
</head>
```

```
<body>
  <h2>Here is the list of people attending the party</h2>
  <table>
    <thead>
      <tr>
        <th>Name</th>
        <th>Email</th>
        <th>Phone</th>
      </tr>
    </thead>
    <tbody>
      @foreach (PartyInvites.Models.GuestResponse r in Model) {
        <tr>
          <td>@r.Name</td>
          <td>@r.Email</td>
          <td>@r.Phone</td>
        </tr>
      }
    </tbody>
  </table>
</body>
</html>
```

Файлы представлений Razor имеют расширение `.cshtml`, потому что содержат смесь кода C# и элементов HTML. Это можно заметить в листинге 2.18, где используется цикл `foreach` для обработки всех объектов `GuestResponse`, которые метод действия передает представлению с применением метода `View()`. В отличие от нормального цикла `foreach` языка C# тело цикла `foreach` из Razor содержит элементы HTML, добавляемые к ответу, который будет отправлен обратно браузеру. В данном представлении для каждого объекта `GuestResponse` генерируется элемент `tr`, который содержит элементы `td`, заполненные значениями свойств объекта.

Чтобы увидеть список в работе, запустим приложение, выбрав в меню Debug пункт Start Debugging, отправим какие-то данные формы и затем щелкнем на ссылке для просмотра списка ответов. Отобразится сводка по данным, введенным вами с момента запуска приложения (рис. 2.18). Представление не оформляет данные привлекательным образом, но пока этого вполне достаточно, а стилизацией мы займемся позже в главе.

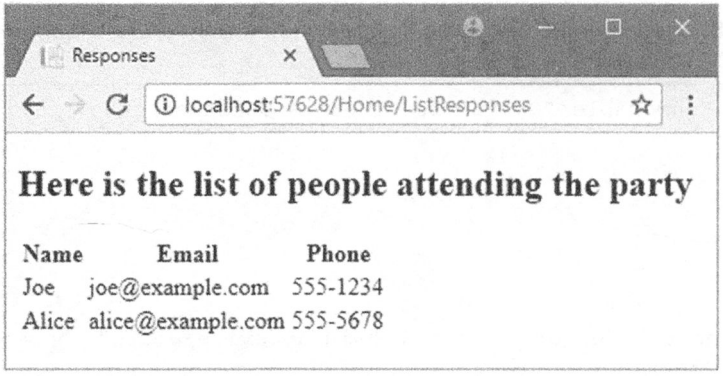


Рис. 2.18. Отображение списка участников вечеринки

Добавление проверки достоверности

Теперь мы готовы добавить в приложение проверку достоверности вводимых данных. В отсутствие проверки достоверности пользователи смогут вводить бессмысленные данные или даже отправлять пустую форму. В приложении MVC проверка достоверности обычно применяется к модели предметной области, а не производится в пользовательском интерфейсе. Это значит, что проверка достоверности определяется в одном месте, но оказывает воздействие в приложении везде, где используется класс модели. Инфраструктура MVC поддерживает *декларативные правила проверки достоверности*, определяемые с помощью атрибутов из пространства имен `System.ComponentModel.DataAnnotations`, т.е. ограничения проверки достоверности выражаются посредством стандартных атрибутов C#. В листинге 2.19 показано, как применить такие атрибуты к классу модели `GuestResponse`.

Листинг 2.19. Применение проверки достоверности в файле `GuestResponse.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {

  public class GuestResponse {
    [Required(ErrorMessage = "Please enter your name")]
                              // Пожалуйста, введите свое имя
    public string Name { get; set; }
    [Required(ErrorMessage = "Please enter your email address")]
    [RegularExpression(".+\\@.+\\..+",
     ErrorMessage = "Please enter a valid email address")]
                    // Пожалуйста, введите свой адрес электронной почты
    public string Email { get; set; }
    [Required(ErrorMessage = "Please enter your phone number")]
                              // Пожалуйста, введите свой номер телефона
    public string Phone { get; set; }
    [Required(ErrorMessage = "Please specify whether you'll attend")]
                              // Пожалуйста, укажите, примете ли участие
    public bool? WillAttend { get; set; }
  }
}
```

Инфраструктура MVC автоматически обнаруживает атрибуты проверки достоверности и использует их для проверки данных во время процесса привязки модели. Мы импортировали пространство имен, которое содержит атрибуты проверки достоверности, так что к ним можно обращаться, не указывая полные имена.

Совет. Как отмечалось ранее, для свойства `WillAttend` был выбран булевский тип, допускающий `null`. Мы поступили так для того, чтобы появилась возможность применять атрибут проверки достоверности `Required`. Если бы использовался обычный булевский тип, то значением, получаемым посредством привязки модели, могло быть только `true` или `false`, и отсутствовала бы возможность определить, выбрал ли пользователь значение. Булевский тип, допускающий `null`, имеет три разрешенных значения: `true`, `false` и `null`. Браузер отправляет `null`, если пользователь не выбрал значение, и тогда атрибут `Required` сообщит об ошибке проверки достоверности. Это хороший пример того, насколько элегантно инфраструктура MVC сочетает средства C# с HTML и HTTP.

Проверку на наличие проблемы с достоверностью данных можно выполнить с применением свойства `ModelState.IsValid` в классе контроллера. В листинге 2.20 показано, как она реализована в методе действия `RsvpForm()`, поддерживающем запросы `POST`, внутри класса контроллера `Home`.

Листинг 2.20. Проверка на наличие ошибок проверки достоверности в файле HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {

  public class HomeController : Controller {
    public ViewResult Index() {
      int hour = DateTime.Now.Hour;
      ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
      return View("MyView");
    }

    [HttpGet]
    public ViewResult RsvpForm() {
      return View();
    }
    [HttpPost]
    public ViewResult RsvpForm(GuestResponse guestResponse) {
      if (ModelState.IsValid) {
        Repository.AddResponse(guestResponse);
        return View("Thanks", guestResponse);
      } else {
        // Обнаружена ошибка проверки достоверности.
        return View();
      }
    }
    public ViewResult ListResponses() {
      return View(Repository.Responses.Where(r => r.WillAttend == true));
    }
  }
}
```

Базовый класс `Controller` предоставляет свойство по имени `ModelState`, которое сообщает информацию о преобразовании данных HTTP-запроса в объекты C#. Если свойство `ModelState.IsValid` возвращает `true`, то известно, что инфраструктура MVC сумела удовлетворить ограничения проверки достоверности, указанные через атрибуты для класса `GuestResponse`. Когда такое происходит, визуализируется представление `Thanks`, как делалось до того.

Если свойство `ModelState.IsValid` возвращает `false`, то известно, что есть ошибки проверки достоверности. Возвращаемый свойством `ModelState` объект предоставляет подробные сведения о каждой возникшей проблеме, но нам нет нужды погружаться на такой уровень деталей, поскольку мы можем полагаться на удобное средство, которое автоматизирует процесс указания пользователю на необходимость решения любых проблем, вызывая метод `View()` без параметров.

Когда MVC визуализирует представление, механизм Razor имеет доступ к деталям любых ошибок проверки достоверности, связанных с запросом, и вспомогательные функции дескрипторов могут обращаться к этим деталям, чтобы отображать сообщения об ошибках пользователю. В листинге 2.21 приведено содержимое файла представления `RsvpForm.cshtml` с добавленными атрибутами вспомогательных функций дескрипторов для проверки достоверности.

Листинг 2.21. Добавление сводки по проверке достоверности в файле `RsvpForm.cshtml` из папки `Views/Home`

```
@model PartyInvites.Models.GuestResponse

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RsvpForm</title>
</head>
<body>
  <form asp-action="RsvpForm" method="post">
    <div asp-validation-summary="All"></div>
    <p>
      <label asp-for="Name">Your name:</label>
      <input asp-for="Name" />
    </p>
    <p>
      <label asp-for="Email">Your email:</label>
      <input asp-for="Email" />
    </p>
    <p>
      <label asp-for="Phone">Your phone:</label>
      <input asp-for="Phone" />
    </p>
    <p>
      <label>Will you attend?</label>
      <select asp-for="WillAttend">
        <option value="">Choose an option</option>
        <option value="true">Yes, I'll be there</option>
        <option value="false">No, I can't come</option>
      </select>
    </p>
    <button type="submit">Submit RSVP</button>
  </form>
</body>
</html>
```

Атрибут `asp-validation-summary` применяется к элементу `div` и отображает список ошибок проверки достоверности при визуализации представления. Значение для атрибута `asp-validation-summary` берется из перечисления по име-

ни `ValidationSummary`, которое указывает типы ошибок проверки достоверности, помещаемые в сводку. Мы указали `All`, что является хорошей отправной точкой для большинства приложений, а в главе 27 будут описаны другие значения.

Чтобы взглянуть, как работает сводка по проверке достоверности, запустим приложение, заполним поле `Name` и отправим форму, не вводя другие данные. Появится сводка с ошибками (рис. 2.19).

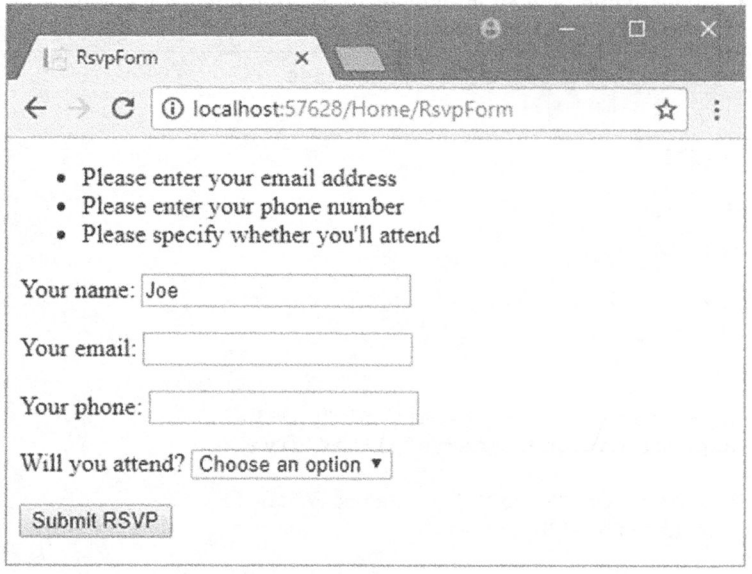


Рис. 2.19. Отображение ошибок проверки достоверности

Метод действия `RsvpForm()` не визуализирует представление `Thanks` до тех пор, пока не будут удовлетворены все ограничения проверки достоверности, примененные к классу `GuestResponse`. Обратите внимание, что введенные в поле `Name` данные были сохранены и отображены снова при визуализации механизмом Razor представления со сводкой проверки достоверности. Это еще одно преимущество привязки модели, и оно упрощает работу с данными формы.

На заметку! Если вы работали с ASP.NET Web Forms, то знаете, что в Web Forms поддерживается концепция *серверных элементов управления*, которые сохраняют состояние, сериализируя значения в скрытое поле формы по имени `__VIEWSTATE`. Привязка модели MVC не имеет никакого отношения к концепциям серверных элементов управления, обратным отправкам или состоянию представления, принятым в Web Forms. Инфраструктура MVC не внедряет скрытое поле `__VIEWSTATE` в визуализированные HTML-страницы. Взамен она включает данные, устанавливая атрибуты `value` элементов управления `input`.

Подсветка полей с недопустимыми значениями

Атрибуты вспомогательных функций дескрипторов, которые ассоциируют свойства модели с элементами, обладают удобным средством, которое можно использовать

в сочетании с привязкой модели. Когда свойство класса модели не проходит проверку достоверности, атрибуты вспомогательных функций дескрипторов будут генерировать несколько отличающуюся HTML-разметку. Вот элемент `input`, который генерируется для поля `Phone` при отсутствии ошибок проверки достоверности:

```
<input type="text" data-val="true"
       data-val-required="Please enter your phone number"
       id="Phone" name="Phone" value="">
```

Для сравнения ниже показан тот же HTML-элемент после того, как пользователь отправил форму, не введя данные в текстовое поле (что является ошибкой проверки достоверности, поскольку мы применили к свойству `Phone` класса `GuestResponse` атрибут проверки `Required`):

```
<input type="text" class="input-validation-error" data-val="true"
       data-val-required="Please enter your phone number" id="Phone"
       name="Phone" value="">
```

Отличие выделено полужирным: атрибут вспомогательной функции дескриптора `asp-for` добавил к элементу `input` класс по имени `input-validation-error`. Мы можем воспользоваться такой возможностью, создав таблицу стилей, которая содержит стили CSS для этого класса и другие стили, применяемые различными атрибутами вспомогательных функций дескрипторов.

По принятому в проектах MVC соглашению статическое содержимое, доставляемое клиентам, помещается в папку `wwwroot`, подпапки которой организованы по типу содержимого, так что таблицы стилей CSS находятся в папке `wwwroot/css`, файлы JavaScript — в папке `wwwroot/js` и т.д.

Для создания таблицы стилей щелкнем правой кнопкой мыши на папке `wwwroot/css` в окне Solution Explorer и выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент). В открывшемся диалоговом окне Add New Item (Добавление нового элемента) перейдем в раздел ASP.NET Core⇨Web⇨Content (ASP.NET Core⇨Веб⇨Содержимое) и выберем шаблон Style Sheet (Таблица стилей) из списка шаблонов (рис. 2.20).

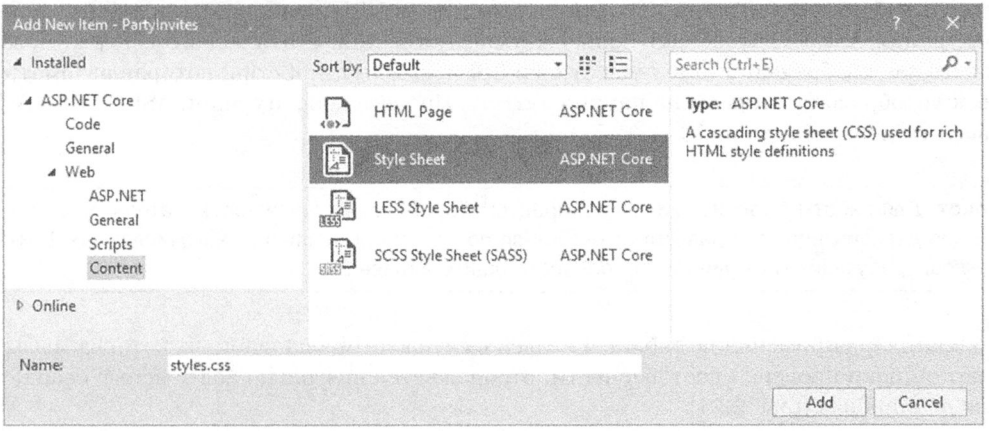


Рис. 2.20. Создание таблицы стилей CSS

Совет. Когда для проекта используется шаблон Web Application (Model-View-Controller), среда Visual Studio создает файл `site.css` в папке `wwwroot/css`. В текущей главе данный файл не задействован.

Назначим файлу имя `styles.css`, щелкнем на кнопке Add (Добавить), чтобы создать файл таблицы стилей, и приведем его содержимое к виду, показанному в листинге 2.22.

Листинг 2.22. Содержимое файла `styles.css` из папки `wwwroot/css`

```
.field-validation-error    { color: #f00; }
.field-validation-valid    { display: none; }
.input-validation-error    { border: 1px solid #f00;
                             background-color: #fee; }
.validation-summary-errors { font-weight: bold; color: #f00; }
.validation-summary-valid  { display: none; }
```

Чтобы применить эту таблицу стилей, мы добавили элемент `link` в раздел `head` представления `RsvpForm` (листинг 2.23).

Листинг 2.23. Применение таблицы стилей в файле `RsvpForm.cshtml`

```
...
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RsvpForm</title>
  <link rel="stylesheet" href="/css/styles.css" />
</head>
...
```

Элемент `link` использует атрибут `href` для указания местоположения таблицы стилей. Обратите внимание, что папка `wwwroot` в URL опущена. Стандартная конфигурация ASP.NET включает поддержку обслуживания статического содержимого, такого как изображения, таблицы стилей CSS и файлы JavaScript, которая автоматически отображает запросы на папку `wwwroot`. Процесс конфигурации ASP.NET и MVC рассматривается в главе 14.

Совет. Для работы с таблицами стилей предусмотрена специальная вспомогательная функция дескриптора, которая может оказаться полезной при наличии многочисленных файлов, требующих управления. Подробности ищите в главе 25.

Благодаря применению таблицы стилей в случае отправки данных, которые вызывают ошибки проверки достоверности, отображается визуально более ясные сообщения об ошибках (рис. 2.21).

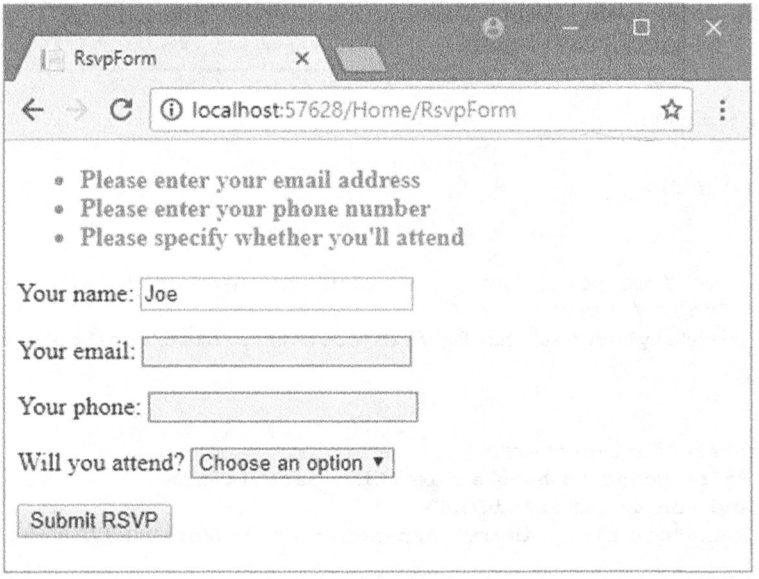


Рис. 2.21. Автоматическая подсветка полей с ошибками проверки достоверности

Стилизация содержимого

Все цели приложения, касающиеся функциональности, достигнуты, но его общий вид оставляет желать лучшего. Когда вы создаете проект с использованием шаблона Web Application (Model-View-Controller), как в рассматриваемом примере, среда Visual Studio устанавливает несколько распространенных пакетов для разработки на стороне клиента. Хотя я не являюсь сторонником применения шаблонов проектов, мне нравятся выбранные Microsoft библиотеки клиентской стороны. Одна из них называется Bootstrap и представляет собой удобную инфраструктуру CSS, первоначально разработанную в Twitter, которая постепенно превратилась в крупный проект с открытым кодом и стала главной опорой разработки веб-приложений.

На заметку! На момент написания книги текущей версией была Bootstrap 3, но версия Bootstrap 4 находилась на стадии разработки. В Microsoft могут принять решение обновить версию Bootstrap, используемую шаблоном Web Application, в последующих выпусках Visual Studio, что может привести к отображению содержимого по-другому. В других главах книги это не должно стать проблемой, т.к. там будет показано, каким образом явно указывать версию пакета, чтобы получить ожидаемые результаты.

Стилизация начального представления

Базовые средства Bootstrap работают за счет применения классов к элементам, которые соответствуют селекторам CSS, определенным внутри добавленных в папку `wwwroot/lib/bootstrap` файлов. Подробную информацию о классах, определенных в библиотеке Bootstrap, можно получить на веб-сайте `http://getbootstrap.com`, а в листинге 2.24 демонстрируется использование нескольких базовых стилей в представлении `MyView.cshtml`.

Листинг 2.24. Добавление классов Bootstrap в файле `MyView.cshtml` из папки `Views/Home`

```
@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css"
/>
</head>
<body>
  <div class="text-center">
    <h3>We're going to have an exciting party!</h3>
    <h4>And you are invited</h4>
    <a class="btn btn-primary" asp-action="RsvpForm">RSVP Now</a>
  </div>
</body>
</html>
```

В разметку был добавлен элемент `link`, атрибут `href` которого загружает файл `bootstrap.css` из папки `wwwroot/lib/bootstrap/dist/css`. По соглашению пакеты CSS и JavaScript от независимых поставщиков устанавливаются в папку `wwwroot/lib`, и в главе 6 будет описан инструмент, предназначенный для управления такими пакетами.

После импортирования таблиц стилей Bootstrap осталось стилизовать элементы. Рассматриваемый пример прост, поэтому требует использования лишь небольшого числа классов CSS из Bootstrap: `text-center`, `btn` и `btn-primary`.

Класс `text-center` центрирует содержимое элемента и его дочерних элементов. Класс `btn` стилизует элемент `button`, `input` или `a` в виде симпатичной кнопки, а класс `btn-primary` указывает диапазон цветов для кнопки. Запустив приложение, можно увидеть результат, показанный на рис. 2.22.

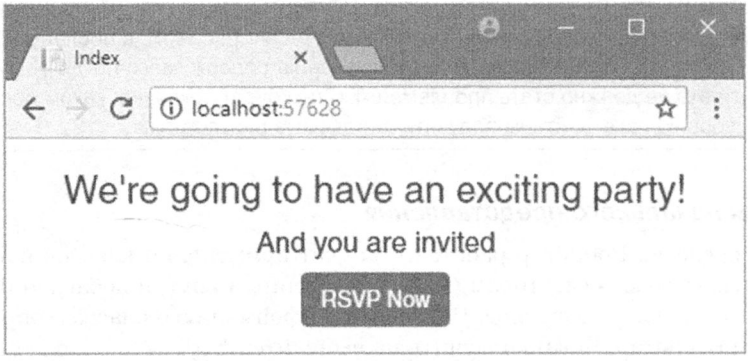


Рис. 2.22. Стилизация представления

Совершенно очевидно, что я — не веб-дизайнер. На самом деле, будучи еще ребенком, я был освобожден от уроков рисования по причине полного отсутствия таланта. Это произвело благоприятный эффект в виде того, что я стал уделять больше времени урокам математики, но вместе с тем мои художественные навыки не развивались примерно с десятилетнего возраста. Для реальных проектов я обратился бы к помощи профессионального дизайнера содержимого, но в настоящем примере я собираюсь делать все самостоятельно и применять Bootstrap с максимально возможной сдержанностью и согласованностью, на какую только способен.

Стилизация представления RsvpForm

В библиотеке Bootstrap определены классы, которые могут использоваться для стилизации форм. Я не планирую вдаваться в особые детали, но в листинге 2.25 показано, как были применены эти классы.

Листинг 2.25. Добавление классов Bootstrap в файле RsvpForm.cshtml из папки Views/Home

```
@model PartyInvites.Models.GuestResponse

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RsvpForm</title>
  <link rel="stylesheet" href="/css/styles.css" />
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css"
/>
</head>
<body>
  <div class="panel panel-success">
    <div class="panel-heading text-center"><h4>RSVP</h4></div>
    <div class="panel-body">
      <form class="p-a-1" asp-action="RsvpForm" method="post">
        <div asp-validation-summary="All"></div>
        <div class="form-group">
          <label asp-for="Name">Your name:</label>
          <input class="form-control" asp-for="Name" />
        </div>
        <div class="form-group">
          <label asp-for="Email">Your email:</label>
          <input class="form-control" asp-for="Email" />
        </div>
        <div class="form-group">
          <label asp-for="Phone">Your phone:</label>
          <input class="form-control" asp-for="Phone" />
        </div>
        <div class="form-group">
          <label>Will you attend?</label>
          <select class="form-control" asp-for="WillAttend">
```

```
            <option value="">Choose an option</option>
            <option value="true">Yes, I'll be there</option>
            <option value="false">No, I can't come</option>
          </select>
        </div>
        <div class="text-center">
          <button class="btn btn-primary" type="submit">
            Submit RSVP
          </button>
        </div>
      </form>
    </div>
  </div>
</body>
</html>
```

Классы Bootstrap в данном примере создают заголовок, просто чтобы придать компоновке структурированность. Для стилизации формы используется класс `form-group`, который стилизует элемент, содержащий `label` и связанный элемент `input` или `select`. Результаты стилизации можно видеть на рис. 2.23.

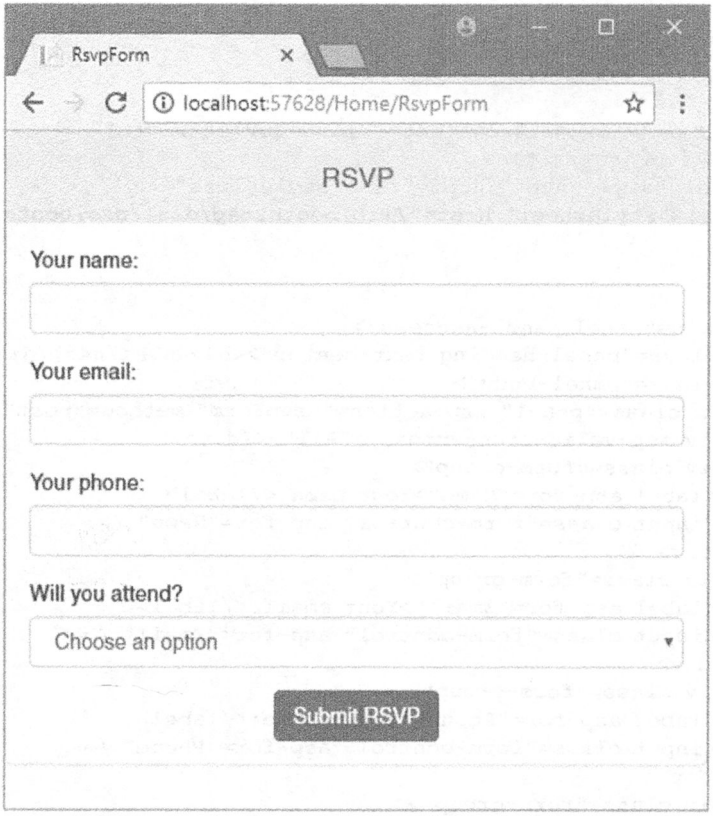


Рис. 2.23. Стилизация представления `RsvpForm`

Стилизация представления Thanks

Следующим стилизуемым представлением является `Thanks.cshtml`; в листинге 2.26 показано, как это делается с применением классов CSS, подобных тем, которые использовались для других представлений. Чтобы облегчить управление приложением, имеет смысл везде, где только возможно, избегать дублирования кода и разметки. Инфраструктура MVC предлагает несколько средств, помогающих сократить дублирование, которые рассматриваются в последующих главах. К таким средствам относятся компоновки Razor (глава 5), частичные представления (глава 21) и компоненты представлений (глава 22).

Листинг 2.26. Применение классов Bootstrap в файле `Thanks.cshtml` из папки `Views/Home`

```
@model PartyInvites.Models.GuestResponse
@{
  Layout = null;
}
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Thanks</title>
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body class="text-center">
  <p>
    <h1>Thank you, @Model.Name!</h1>
    @if (Model.WillAttend == true) {
      @:It's great that you're coming. The drinks are already in the fridge!
    } else {
      @:Sorry to hear that you can't make it, but thanks for letting us know.
    }
  </p>
  Click <a class="nav-link" asp-action="ListResponses">here</a>
  to see who is coming.
</body>
</html>
```

На рис. 2.24 показан результат стилизации.

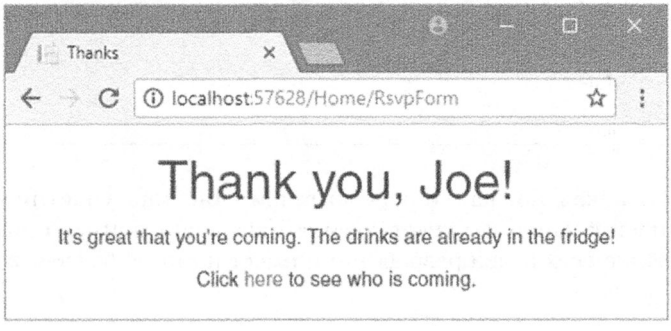


Рис. 2.24. Стилизация представления `Thanks`

Стилизация представления `ListResponses`

Последним мы стилизуем представление `ListResponses`, которое отображает список участников вечеринки. Стилизация содержимого следует тому же базовому подходу, который применялся в отношении всех стилей Bootstrap (листинг 2.27).

Листинг 2.27. Добавление классов Bootstrap в файле `ListResponses.cshtml` из папки `Views/Home`

```
@model IEnumerable<PartyInvites.Models.GuestResponse>

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
  <title>Responses</title>
</head>
<body>
  <div class="panel-body">
    <h2>Here is the list of people attending the party</h2>
    <table class="table table-sm table-striped table-bordered">
      <thead>
        <tr>
          <th>Name</th>
          <th>Email</th>
          <th>Phone</th>
        </tr>
      </thead>
      <tbody>
        @foreach (PartyInvites.Models.GuestResponse r in Model) {
          <tr>
            <td>@r.Name</td>
            <td>@r.Email</td>
            <td>@r.Phone</td>
          </tr>
        }
      </tbody>
    </table>
  </div>
</body>
</html>
```

На рис. 2.25 показано, как теперь выглядит таблица участников вечеринки. Добавление стилей к данному представлению завершает пример приложения, которое теперь достигло всех целей разработки и имеет намного более совершенный внешний вид.

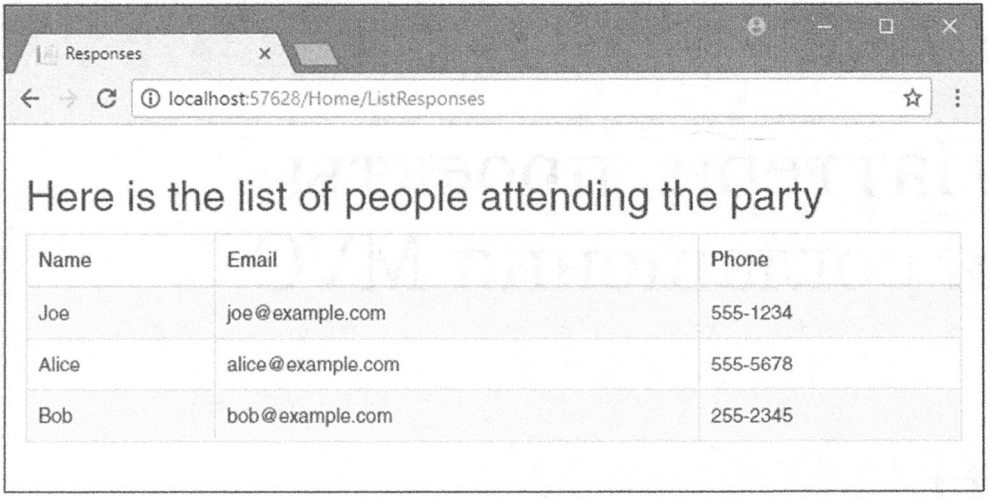


Рис. 2.25. Стилизация представления `ListResponses`

Резюме

В главе был создан новый проект MVC, который использовался для построения простого приложения ввода данных MVC, что позволило получить первое представление об архитектуре ASP.NET Core MVC и применяемом подходе. Некоторые основные средства (включая синтаксис Razor, маршрутизацию и тестирование) не рассматривались, но мы вернемся к данным темам в последующих главах. В следующей главе будет описаны паттерны проектирования MVC, которые формируют основу эффективной разработки с помощью ASP.NET Core MVC.

ГЛАВА 3

Паттерн, проекты и соглашения MVC

Прежде чем приступать к углубленному изучению деталей инфраструктуры ASP.NET Core MVC, необходимо освоить паттерн проектирования MVC, лежащие в его основе концепции и способ, которым они транслируются в проекты ASP.NET Core MVC. Возможно, вы уже знакомы с некоторыми идеями и соглашениями, обсуждаемыми в настоящей главе, особенно если вам приходилось заниматься разработкой сложных приложений ASP.NET или C#. Если это не так, тогда внимательно читайте главу. Хорошее понимание того, что положено в основу MVC, может помочь увязать функциональные возможности инфраструктуры с контекстом материала, излагаемого в оставшихся главах книги.

История создания MVC

Термин *модель-представление-контроллер* (model-view-controller — MVC) был в употреблении с конца 1970-х годов и происходит из проекта Smalltalk в Xerox PARC, где он был задуман как способ организации ряда ранних приложений с графическим пользовательским интерфейсом. Некоторые нюансы первоначального паттерна MVC были связаны с концепциями, специфичными для Smalltalk, такими как *экраны* и *инструменты*, но более широкие понятия по-прежнему применимы к приложениям — и особенно хорошо они подходят для веб-приложений.

Особенности паттерна MVC

Если оперировать высокоуровневыми понятиями, то паттерн MVC означает, что приложение MVC будет разделено, по крайней мере, на три части.

- Модели, содержащие или представляющие данные, с которыми работают пользователи.
- Представления, используемые для визуализации некоторой части модели в виде пользовательского интерфейса.
- Контроллеры, которые обрабатывают входящие запросы, выполняют операции с моделью и выбирают представления для визуализации пользователю.

Каждая порция архитектуры MVC четко определена и самодостаточна; такое положение вещей называют *разделением обязанностей*. Логика, которая манипулирует данными в модели, содержится *только* в модели. Логика, отображающая данные, присутствует *только* в представлении. Код, который обрабатывает пользовательские запросы и ввод, находится *только* в контроллере. Благодаря ясному разделению между частями приложение будет легче сопровождать и расширять на протяжении его времени существования вне зависимости от того, насколько большим оно станет.

Понятие моделей

Модели (*M* в *MVC*) содержат данные, с которыми работают пользователи. Существуют два обширных типа моделей: *модели представлений*, которые выражают сами данные, передаваемые из контроллера в представление, и *модели предметной области*, которые содержат данные в предметной области наряду с операциями, трансформациями и правилами для создания, хранения и манипулирования данными, вместе называемыми *логикой моделей*.

Модели — это определение "вселенной", в которой функционирует приложение. Например, в банковском приложении модель представляет все аспекты банковской деятельности, поддерживаемые приложением, такие как расчетные счета, главная бухгалтерская книга и кредитные лимиты для клиентов, равно как и операции, которые могут применяться для манипулирования данными в модели, подобные внесению денежных средств и списанию их со счетов. Модель отвечает также за сохранение общего состояния и целостности данных — например, удостоверяясь, что все транзакции внесены в главную книгу, а клиент не снимает со счета больше денежных средств, чем имеет на то право, или больше, чем находится в распоряжении самого банка.

Для каждого компонента в паттерне MVC будет описано, что он должен включать, а что не должен.

Модель в приложении, построенном с использованием паттерна MVC, *должна*:

- содержать данные предметной области;
- содержать логику для создания, управления и модификации данных предметной области;
- предоставлять чистый API-интерфейс, который открывает доступ к данным модели и операциям с ними.

Модель *не должна*:

- показывать детали того, как осуществляется получение или управление данными модели (другими словами, подробности механизма хранения данных не должны быть видны контроллерам и представлениям);
- содержать логику, которая трансформирует модель на основе взаимодействия с пользователем (поскольку это работа контроллера);
- содержать логику для отображения данных пользователю (т.к. это работа представления).

Преимущества обеспечения изоляции модели от контроллера и представлений заключаются в том, что вы можете гораздо легче тестировать логику (модульное тестирование описано в главе 7) и проще расширять и сопровождать приложение в целом.

Совет. Многих разработчиков, только приступивших к ознакомлению с паттерном MVC, приводит в замешательство идея помещения логики в модель данных из-за их уверенности в том, что целью паттерна MVC является отделение данных от логики. Это заблуждение: цель паттерна MVC — разделение приложения на три функциональных области, каждая из которых может содержать *и* логику, *и* данные. Цель не в том, чтобы устранить логику из модели. Наоборот, цель в том, чтобы гарантировать наличие в модели только логики, предназначенной для создания и управления данными модели.

Понятие контроллеров

Контроллеры являются "соединительной тканью" паттерна MVC, исполняя роль каналов между моделью данных и представлениями. Контроллеры определяют действия, предоставляющие бизнес-логику, которая оперирует на модели данных и обеспечивает представления данными, подлежащими отображению для пользователя.

Контроллер, построенный с применением паттерна MVC, *должен*:

- содержать действия, требующиеся для обновления модели на основе взаимодействия с пользователем.

Контроллер *не должен*:

- содержать логику, которая управляет внешним видом данных (это работа представления);
- содержать логику, которая управляет постоянством данных (это работа модели).

Понятие представлений

Представления содержат логику, которая требуется для отображения данных пользователю или для сбора данных от пользователя, так что они могут быть обработаны каким-то действием контроллера. Представления *должны*:

- содержать логику и разметку, необходимые для показа данных пользователю.

Представления *не должны*:

- содержать сложную логику (ее лучше поместить в контроллер);
- содержать логику, которая создает, сохраняет или манипулирует моделью предметной области.

Представления *могут* содержать логику, но она должна быть простой и использоваться умеренно. Помещение в представление чего угодно кроме вызовов простейших методов или несложных выражений затрудняет тестирование и сопровождение приложения в целом.

Реализация паттерна MVC в ASP.NET Core

Как подразумевает само название, реализация ASP.NET Core MVC адаптирует абстрактный паттерн MVC к миру разработки ASP.NET и C#. В инфраструктуре ASP.NET Core MVC контроллеры — это классы C#, обычно производные от класса `Microsoft.AspNetCore.Mvc.Controller`. Каждый открытый метод в производном от `Controller` классе является *методом действия*, который ассоциирован с каким-то URL. Когда запрос отправляется по URL, связанному с методом действия, операторы в данном методе действия выполняются, чтобы провести некоторую операцию над моделью предметной области и затем выбрать представление для отображения клиенту. Взаимодействия между контроллером, моделью и представлением проиллюстрированы на рис. 3.1.



Рис. 3.1. Взаимодействия в приложении MVC

В инфраструктуре ASP.NET Core MVC применяется *механизм визуализации*, известный как Razor, который является компонентом, ответственным за обработку представления для генерации ответа браузеру. Представления Razor — это HTML-шаблоны, содержащие логику C#, которая используется для обработки данных модели с целью генерации динамического содержимого, реагирующего на изменения в модели. Работа механизма Razor рассматривается в главе 5.

Инфраструктура ASP.NET MVC не налагает никаких ограничений на реализацию модели предметной области. Вы можете создать модель с применением обычных объектов C# и реализовать постоянство с использованием любых баз данных, инфраструктур объектно-реляционного отображения или других инструментов работы с данными, поддерживаемых в .NET.

Понятие одностраничных приложений

Исторически сложилось так, что при разработке веб-приложений браузеры обычно трактовались как простое устройство отображения для визуализации HTML-разметки и реагирования на щелчки кнопками мыши. Такой стиль веб-приложений называется *полным циклом обмена*. Каждый раз, когда пользователь щелкает на ссылке, приложению ASP.NET Core MVC посылается HTTP-запрос. Внутри приложения контроллер выбирает представление, которое визуализируется механизмом Razor и отправляется обратно браузеру, так что пользователю может быть отображена новая HTML-страница. Вся логика, данные и состояние находятся на сервере ASP.NET Core MVC, что упрощает разработку и означает отсутствие необходимости уделять много внимания браузеру, кроме как проверять, поддерживает ли он возможности HTML, реализованные в представлениях Razor.

По контрасту с ними *одностраничные приложения* включают браузер в платформу приложений. Сервер несет ответственность за управление данными приложения, в то время как код JavaScript, выполняющийся в браузере, запрашивает эти данные, отображает их пользователю и реагирует на пользовательское взаимодействие. В одностраничном приложении обязанности, касающиеся моделей, представлений и контроллеров, разделяются между браузером и сервером. Вместо отправки полных HTML-страниц браузеру часть ASP.NET Core MVC приложения обеспечивает доступ к данным приложения, которые запрашиваются и отображаются инфраструктурой JavaScript, такой как Angular или React.

Одностраничные приложения могут быть более отзывчивыми, чем приложения с полным циклом обмена, но их сложнее создавать, а эффективная разработка таких приложений требует навыков работы с языками C# и JavaScript, сложность которых не должна недооцениваться. В главе 20 будет показано, как можно использовать инфраструктуру ASP.NET Core MVC для предоставления данных в одностраничном приложении, но процесс разработки приложений подобного рода не демонстрируется, поскольку он сам по себе является отдельной темой. Среди инфраструктур JavaScript я отдаю предпочтение Angular, которая описана в моей книге *Pro Angular*. При желании задействовать Angular с ASP.NET Core MVC почитайте мою книгу *Essential Angular for ASP.NET Core MVC*.

Сравнение MVC с другими паттернами

Разумеется, MVC — далеко не единственный архитектурный паттерн программного обеспечения. Есть много других паттернов, и некоторые из них чрезвычайно популярны или, по крайней мере, были таковыми. О паттерне MVC можно узнать много интересного, сравнивая его с альтернативами. В последующих разделах кратко описаны разные подходы к структурированию приложения и приведено их сравнение с MVC. Одни паттерны будут близкими вариациями на тему MVC, тогда как другие совершенно отличаться.

Я не утверждаю, что MVC — идеальный паттерн во всех ситуациях. Я сторонник выбора такого подхода, который лучше всего решает имеющуюся задачу. Как вы вскоре увидите, есть ситуации, когда некоторые конкурирующие паттерны в равной степени полезны или лучше MVC. Я призываю делать *осознанный* и *взвешенный* выбор паттерна. Сам факт чтения вами этой книги наводит на мысль, что в определенной мере вы уже склонны применять паттерн MVC, но всегда полезно сохранять максимально широкую точку зрения.

Паттерн "Интеллектуальный пользовательский интерфейс"

Один из наиболее распространенных паттернов проектирования известен как *"Интеллектуальный пользовательский интерфейс"* (Smart UI). Большинству программистов приходилось создавать приложение с интеллектуальным пользовательским интерфейсом на том или ином этапе своей профессиональной деятельности — меня это определенно касается. Если вы использовали Windows Forms или ASP.NET Web Forms, то сказанное относится и к вам.

При построении приложения с интеллектуальным пользовательским интерфейсом разработчики конструируют интерфейс, часто перетаскивая набор *компонентов* либо *элементов управления* на поверхность проектирования или холст. Элементы управления сообщают о взаимодействии с пользователем, инициируя события для щелчков кнопками мыши, нажатий клавиш на клавиатуре, перемещений курсора и т.д. Разработчик добавляет код реакции на такие события в набор *обработчиков событий*, которые являются небольшими блоками кода, вызываемыми при выдаче специфического события в определенном компоненте. В конечном итоге получается монолитное приложение, показанное на рис. 3.2.

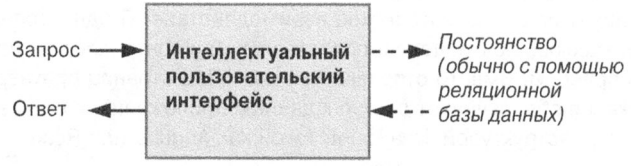


Рис. 3.2. Паттерн "Интеллектуальный пользовательский интерфейс"

Код, который поддерживает пользовательский интерфейс и реализует бизнес-логику, перемешан между собой без какого-либо разделения обязанностей. Код, который определяет приемлемые значения для вводимых данных и запрашивает данные или модифицирует, например, расчетный счет пользователя, оказывается размещенным в небольших фрагментах, связанных друг с другом в предполагаемом порядке поступления событий.

Интеллектуальные пользовательские интерфейсы идеальны для простых проектов, т.к. позволяют добиться неплохих результатов достаточно быстро (по сравнению с разработкой MVC, которая, как показано в главе 8, требует определенных начальных затрат, прежде чем будут доставлены результаты). Интеллектуальные пользовательские интерфейсы также подходят для построения прототипов пользовательских интерфейсов. Их инструменты визуального конструирования могут оказаться *по-настоящему* удобными, и если вы обсуждаете с заказчиком требования к внешнему виду и потоку пользовательского интерфейса, то инструмент интеллектуального пользовательского интерфейса может быть быстрым и удобным способом для генерации и проверки различных идей.

Самый крупный недостаток интеллектуальных пользовательских интерфейсов связан с тем, что их трудно сопровождать и расширять. Смешивание кода модели предметной области и бизнес-логики с кодом пользовательского интерфейса приводит к дублированию кода, когда один и тот же фрагмент бизнес-логики копируется и вставляется для поддержки вновь добавленного компонента. Нахождение всех дублированных фрагментов и применение к ним исправления может быть непростой задачей. Добавление новой функции может оказаться практически невозможным без нарушения работы каких-то существующих функций. Тестирование приложения с интеллектуальным пользовательским интерфейсом также может быть затруднено. Единственный способ тестирования — эмуляция взаимодействия с пользователем, что является далеким от идеала и трудно реализуемым фундаментом для обеспечения полного покрытия тестами.

В мире MVC интеллектуальный пользовательский интерфейс часто называют *антипаттерном*, т.е. чем-то таким, чего следует избегать любой ценой. Такая антипатия возникает (по крайней мере, частично) оттого, что к инфраструктуре MVC обращаются в поисках альтернативы после множества не слишком успешных попыток разработки и сопровождения приложений с интеллектуальным пользовательским интерфейсом, которые попросту вышли из-под контроля.

Тем не менее, полный отказ от паттерна "Интеллектуальный пользовательский интерфейс" будет заблуждением. Далеко не все в нем настолько плохо, и с данным подходом связаны также положительные аспекты. Приложения с интеллектуальным пользовательским интерфейсом быстро и легко разрабатывать. Производители компонентов и инструментов проектирования приложили немало усилий, чтобы сделать процесс разработки приятным занятием, поэтому даже совершенно неопытный программист за считанные часы может получить профессионально выглядящий и достаточно функциональный результат.

Наибольшая слабость приложений с интеллектуальным пользовательским интерфейсом — низкое удобство сопровождения — не проявляется при мелких объемах разработки. Если вы создаете несложный инструмент для небольшой аудитории, то приложение с интеллектуальным пользовательским интерфейсом может оказаться идеальным решением. Просто отсутствует дополнительная сложность, присущая разработке приложения MVC.

Архитектура "модель-представление"

В приложении с интеллектуальным пользовательским интерфейсом проблемы сопровождения обычно возникают в области бизнес-логики, которая настолько рассеяна по приложению, что внесение изменений или добавление новых функций становится мучительным процессом. Улучшить ситуацию помогает *архитектура "модель-представление"*, которая выносит бизнес-логику в отдельную модель предметной области. Все данные, процессы и правила концентрируются в одной части приложения (рис. 3.3).

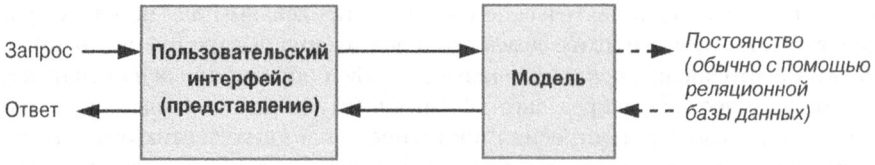


Рис. 3.3. Архитектура "модель-представление"

Архитектура "модель-представление" может быть значительным усовершенствованием по сравнению с монолитным интеллектуальным пользовательским интерфейсом (скажем, ее гораздо легче сопровождать), но возникают две проблемы. Первая: поскольку пользовательский интерфейс и модель предметной области тесно переплетены, модульное тестирование любой из упомянутых частей может оказаться затруднительным. Вторая проблема проистекает из практики, а не из определения паттерна. Обычно модель содержит большой объем кода доступа к данным (не обязательно, но в большинстве случаев), а это означает, что модель данных содержит не только бизнес-данные, операции и правила.

Классическая трехуровневая архитектура

Чтобы решить проблемы, присущие архитектуре "модель-представление", *трехзвенная* или *трехуровневая* архитектура отделяет код реализации постоянства от модели предметной области и помещает его в новый компонент, который называется *уровнем доступа к данным* (data access layer — DAL). Сказанное проиллюстрировано на рис. 3.4.

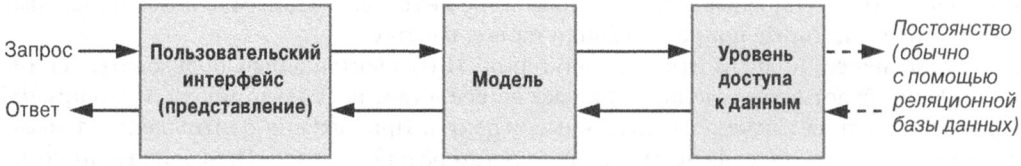


Рис. 3.4. Трехуровневая архитектура

Трехуровневая архитектура является наиболее широко используемым паттерном в бизнес-приложениях. Она не устанавливает никаких ограничений на способ реализации пользовательского интерфейса и обеспечивает хорошее разделение обязанностей, не будучи излишне сложной. Кроме того, ценой некоторых усилий уровень DAL может быть создан так, чтобы модульное тестирование проводилось относительно легко. Между классическим трехуровневым приложением и паттерном MVC можно заметить очевидные сходства. Разница в том, что когда уровень пользовательского интерфейса напрямую связан с инфраструктурой графического пользовательского интерфейса типа "щелчок-событие" (такой как Windows Forms или ASP.NET Web Forms), то выполнение автоматизированных модульных тестов становится практически невозможным. А поскольку часть пользовательского интерфейса трехуровневого приложения может быть сложной, останется много кода, не подвергавшегося серьезному тестированию.

В худшем сценарии отсутствие принудительных ограничений, накладываемых трехуровневой архитектурой на уровень пользовательского интерфейса, означает, что многие такие приложения в итоге оказываются тонко замаскированными при-

ложениями с интеллектуальным пользовательским интерфейсом без какого-либо реального разделения обязанностей. Это ведет к наихудшему конечному результату: не поддерживающему тестирование и трудному в сопровождении приложению, которое к тому же чрезмерно сложно.

Разновидности MVC

Основные принципы проектирования приложений MVC были уже описаны, особенно те из них, которые применимы к реализации ASP.NET Core MVC. Другие реализации интерпретируют аспекты паттерна MVC иначе, дополняя, подстраивая или как-то еще адаптируя его для соответствия области охвата и целям своих проектов. В последующих разделах мы кратко рассмотрим две наиболее распространенные вариации на тему MVC. Понимание этих разновидностей не имеет особого значения в случае работы с ASP.NET Core MVC; данная информация включена ради полноты картины, потому что такие термины будут употребляться в большинстве обсуждений паттернов проектирования ПО.

Паттерн "модель-представление-презентатор"

Паттерн *"модель-представление-презентатор"* (model-view-presenter — MVP) является разновидностью MVC и разработан для того, чтобы облегчить согласование с поддерживающими состояние платформами графического пользовательского интерфейса, такими как Windows Forms или ASP.NET Web Forms. Это достойная попытка извлечь лучшее из паттерна интеллектуального пользовательского интерфейса, избежав проблем, которые он обычно привносит.

В паттерне MVP презентатор имеет те же обязанности, что и контроллер MVC, но он также более непосредственно связан с представлением, сохраняющим информацию о состоянии, напрямую управляя значениями, которые отображаются в компонентах пользовательского интерфейса в соответствии с вводом и действиями пользователя. Существуют две реализации паттерна MVP.

- Реализация пассивного представления, в которой представление не содержит никакой логики. Такое представление служит контейнером для элементов управления пользовательского интерфейса, которыми напрямую управляет презентатор.
- Реализация координирующего контроллера, в которой представление может отвечать за определенные элементы логики презентации, такие как привязка данных, и получать ссылку на источник данных от моделей предметной области.

Отличие между указанными двумя реализациями касается уровня интеллектуальности представления. В любом случае презентатор отделен от инфраструктуры графического пользовательского интерфейса, что делает логику презентатора более простой и подходящей для модульного тестирования.

Паттерн "модель-представление-модель представления"

Паттерн *"модель-представление-модель представления"* (model-view-view model — MVVM) — недавно выпущенная Microsoft разновидность MVC, которая используется в инфраструктуре Windows Presentation Foundation (WPF). В рамках паттерна MVVM модели и представления играют те же самые роли, что и в MVC. Разница связана с присутствующей в MVVM концепцией *модели представления*, которая является абстрактным обозначением пользовательского интерфейса. Как правило, модель

представления — это класс C#, который открывает доступ к свойствам для данных, подлежащих отображению в пользовательском интерфейсе, и операциям с данными, инициируемым из пользовательского интерфейса. В отличие от контроллера MVC модель представления MVVM не имеет ни малейшего понятия о существовании представления (или любой конкретной технологии пользовательских интерфейсов). Представление MVVM применяет средство *привязки* WPF, чтобы установить двунаправленное соединение между свойствами, доступ к которым открывают элементы управления в представлении (вроде пунктов раскрывающегося меню или эффекта от щелчка на кнопке), и свойствами, доступ к которым открывает модель представления.

На заметку! В паттерне MVC также используется термин *модель представления*, но он относится к простому классу модели, предназначенному только для передачи данных из контроллера в представление, как противоположность *моделям предметной области*, которые являются сложными представлениями данных, операций и правил.

Проекты ASP.NET Core MVC

При создании нового проекта ASP.NET Core MVC среда Visual Studio предлагает на выбор несколько вариантов начального содержимого, которое желательно иметь в проекте. Идея заключается в том, чтобы облегчить разработчикам-новичкам процесс обучения и применить сберегающий время передовой опыт при описании распространенных средств и задач. Я не поклонник такого подхода в отношении заготовленных проектов или кода. Намерения обычно хороши, но исполнение никогда не приводит в восторг. Одной из характеристик, которая мне больше всего нравится в ASP.NET и MVC, является высочайшая гибкость в подгонке платформы под мой стиль разработки. Процессы, классы и представления, которые создает и наполняет среда Visual Studio, вызывают у меня чувство, что я вынужден работать в стиле кого-то другого. К тому же я нахожу содержимое и конфигурацию слишком обобщенными и примитивными, чтобы приносить хоть какую-нибудь пользу. Разработчики в Microsoft не могут знать, какой вид приложения необходим, поэтому они охватывают все основы, но настолько обобщенным путем, что в итоге я просто избавляюсь от всего стандартного содержимого.

Моя рекомендация (всем, кто по недоразумению спросил) — начинать с пустого проекта и добавлять нужные папки, файлы и пакеты. Вы не только узнаете больше о способе работы MVC, но и будете обладать полным контролем над тем, что содержит ваше приложение.

Но мои предпочтения не должны задавать тон вашей практике разработки. Вы можете счесть шаблоны более удобными, нежели я, особенно если являетесь новичком в разработке с помощью ASP.NET и еще не выработали стиль, который устраивает лично вас. Вы можете также признавать шаблоны проектов полезным ресурсом и источником идей, хотя должны проявлять осмотрительность и не добавлять функциональность к приложению до того, как полностью поймете, каким образом она работает.

Создание проекта

При создании нового проекта ASP.NET Core предлагается выбрать вариант из целого набора отправных точек, как показано на рис. 3.5.

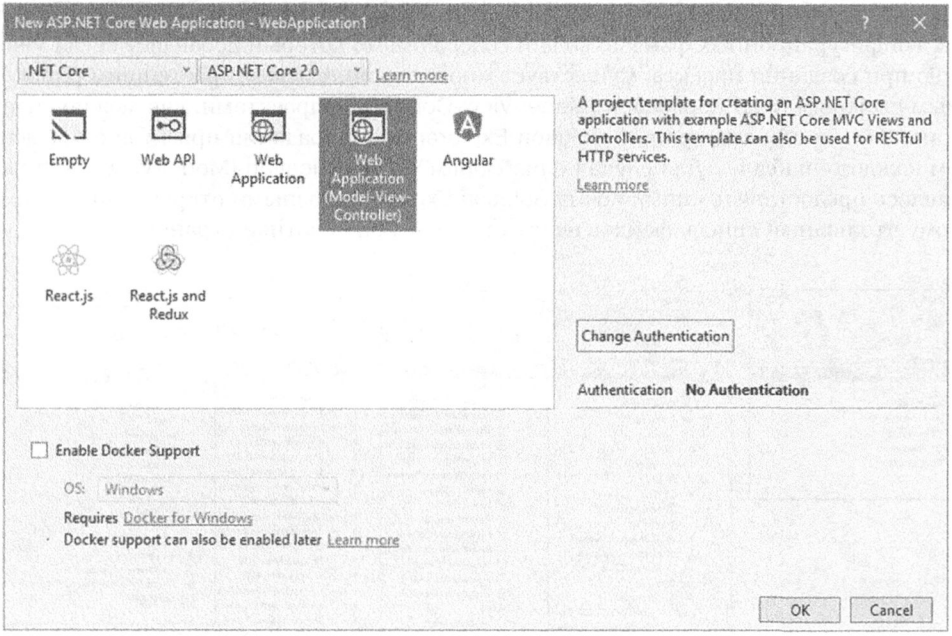


Рис. 3.5. Шаблоны проектов ASP.NET Core

Шаблон проекта Empty (Пустой) содержит связующие механизмы для инфраструктуры ASP.NET Core, но не включает библиотеки или конфигурацию, требующиеся приложению MVC. В состав шаблона проекта Web API входят инфраструктуры ASP.NET Core и MVC, а также пример приложения, который демонстрирует получение и обработку запросов данных от клиентов с использованием контроллера API, который будет описан в главе 20.

Шаблон проекта Web Application (Model-View-Controller) (Веб-приложение (модель-представление-контроллер)) содержит инфраструктуры ASP.NET Core и MVC с примером приложения, иллюстрирующим генерацию HTML-содержимого. Шаблоны Web API и Web Application (Model-View-Controller) могут конфигурироваться с различными схемами для аутентификации пользователей и авторизации их доступа к приложению.

Остальные шаблоны предоставляют начальное содержимое, подходящее для работы с инфраструктурами одностраничных приложений (Angular и React) и с инфраструктурой Razor Pages (которая разрешает смешивать в одном файле код и разметку, объединяя роли контроллеров и представлений, и ради простоты отказывается от ряда преимуществ модели MVC).

Шаблоны проектов могут производить впечатление, что для создания определенного вида приложения ASP.NET вы обязаны следовать специфическому пути, но дело обстоит вовсе не так. Шаблоны — это просто разные отправные точки для получения той же самой функциональности, и вы можете добавлять в проекты, созданные с помощью любого шаблона, любую желаемую функциональность. Скажем, в главе 20 объясняется, каким образом работать с HTTP-запросами данных, а в главах 28–30 — как поступать с аутентификацией и авторизацией, причем все примеры будут начинаться с шаблона проекта Empty.

Реальная разница между шаблонами проектов связана с начальным набором библиотек, конфигурационных файлов, кода и содержимого, который добавляет среда Visual Studio при создании проекта. Существует много отличий между простейшим (Empty) и самым сложным (Web Application (Model-View-Controller)) проектами, как можно видеть на рис. 3.6, где приведено окно Solution Explorer после создания проектов с применением каждого шаблона. Для случая с шаблоном Web Application (Model-View-Controller) пришлось предоставить снимки окна Solution Explorer с разными открытыми папками, потому что полный список файлов не уместился бы на печатной странице.

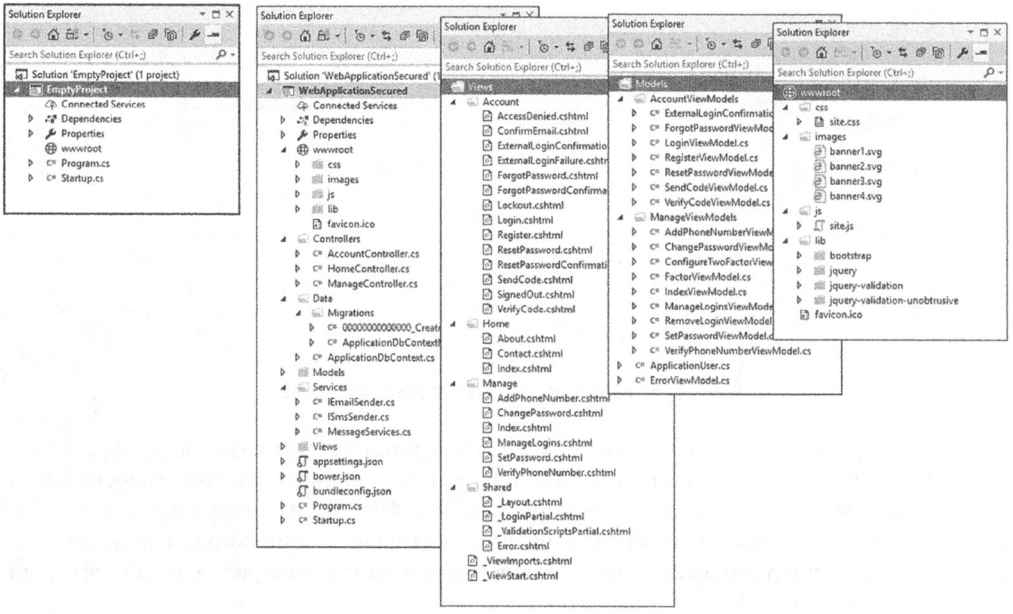


Рис. 3.6. Стандартное содержимое, добавляемое в проект шаблонами Empty и Web Application (Model-View-Controller)

Дополнительные файлы, которые шаблон Web Application (Model-View-Controller) добавляет в проект, выглядят устрашающе, но многие из них — всего лишь заполнители или реализации, иллюстрирующие общие функциональные возможности. Одни дополнительные файлы настраивают MVC или конфигурируют ASP.NET Core; другие являются библиотеками клиентской стороны, которые будут включаться в состав HTML-разметки, генерируемой приложением. Список файлов в настоящий момент может выглядеть огромным, но к концу книги вы будете понимать, что делает каждый из них.

Независимо от того, какой шаблон используется для создания проекта, существуют общие папки и файлы, имеющиеся во всех шаблонах. Одни элементы в проекте играют специальные роли, которые жестко закодированы в инфраструктуре ASP.NET Core или MVC либо каком-то инструменте из числа поддерживаемых Visual Studio. Другие связаны с соглашениями об именовании, которые используются в большинстве проектов ASP.NET Core или MVC. В табл. 3.1 описаны важные файлы и папки, с которыми вы столкнетесь в проекте ASP.NET Core MVC; часть из них по умолчанию не присутствует в проекте, но будет представлена в последующих главах.

Таблица 3.1. Сводка по элементам проекта MVC

Папка или файл	Описание
/Areas	Области — это способ разбиения крупных приложений на мелкие порции. Области рассматриваются в главе 16
/Dependencies	Зависимости предоставляют подробные сведения обо всех пакетах, от которых зависит проект. Диспетчеры пакетов, применяемые Visual Studio, описаны в главе 6
/Components	Здесь определены классы компонентов представлений, которые используются для отображения самодостаточных средств, таких как корзины для покупок. Компоненты представлений обсуждаются в главе 22
/Controllers	Сюда помещаются классы контроллеров. Это соглашение. Классы контроллеров могут размещаться где угодно, потому что все они компилируются в одну и ту же сборку. Контроллеры подробно рассматриваются в главе 17
/Data	Здесь определены классы контекста баз данных, хотя я предпочитаю игнорировать это соглашение и определять их в папке Models, как будет продемонстрировано в главе 8
/Data/Migrations	Здесь хранятся миграции Entity Framework Core, чтобы базы данных можно было подготовить для хранения данных приложения. Миграции будут применяться в главах 8–11 как часть проекта SportsStore
/Models	Сюда помещаются классы моделей представлений и моделей предметной области. Это соглашение. Классы моделей могут определяться в любом месте текущего проекта или даже в отдельном проекте
/Views	Здесь хранятся представления и частичные представления, обычно сгруппированные в папки с именами контроллеров, к которым они относятся. Представления будут подробно описаны в главе 21
/Views/Shared	Здесь хранятся компоновки и представления, которые не являются специфичными для каких-то контроллеров. Представления будут подробно описаны в главе 21
/Views/_ViewImports.cshtml	Этот файл используется для указания пространств имен, которые будут включены в файлы представлений Razor, как объясняется в главе 5. Он также применяется для настройки вспомогательных функций дескрипторов (глава 23)
/Views/_ViewStart.cshtml	Этот файл позволяет указать стандартную компоновку для механизма визуализации Razor, как описано в главе 5
/appsettings.json	Этот файл содержит настройки конфигурации, которые можно подгонять под разные среды, такие как среда разработки, среда тестирования и производственная среда. Чаще всего он используется для определения строк подключения к серверу баз данных, а также настроек ведения журналов и отладки (глава 14)

Окончание табл. 3.1

Папка или файл	Описание
/bower.json	Этот файл содержит список пакетов, управляемых диспетчером пакетов Bower (глава 6)
/<проект>.json	Этот файл содержит конфигурацию проекта, включая пакеты NuGet, которые требуются для приложения, как описано в главах 6 и 14. Данный файл скрыт и может быть отредактирован только путем щелчка правой кнопкой мыши на имени проекта в окне Solution Explorer и выбора в контекстном меню пункта Edit <проект>.csproj (Редактировать <проект>.csproj)
/Program.cs	Этот класс конфигурирует платформу, на которой размещено приложение (глава 14)
/Startup.cs	Этот класс конфигурирует само приложение (глава 14)
/wwwroot	Сюда помещается статическое содержимое, такое как файлы CSS и изображений. Кроме того, именно сюда диспетчер пакетов Bower устанавливает пакеты JavaScript и CSS (глава 6)

Соглашения в проекте MVC

Есть два вида соглашений в проекте MVC. Первый вид — просто предположения о том, как проект может быть структурирован. Например, пакеты JavaScript и CSS от независимых поставщиков, на которые вы полагаетесь, общепринято помещать в папку wwwroot/lib. Именно там ожидают их обнаружить другие разработчики MVC, и сюда их будет устанавливать диспетчер пакетов. Но вы вольны переименовать папку lib или вообще удалить ее и хранить пакеты в другом месте. Это не помешает инфраструктуре MVC выполнить ваше приложение при условии, что элементы script и link в представлениях ссылаются на местоположение, куда вы поместили пакеты.

Второй вид соглашений вытекает из принципа *соглашения по конфигурации* (или *соглашения над конфигурацией*, если делать акцент на преимуществе соглашения перед конфигурацией), который был одним из главных аспектов, обеспечивших популярность платформе Ruby on Rails. Соглашение по конфигурации означает, что вы не должны явно конфигурировать, скажем, ассоциации между контроллерами и их представлениями. Вы просто следуете определенному соглашению об именовании для своих файлов — и все работает. Но когда приходится иметь дело с соглашением такого вида, снижается гибкость в отношении изменения структуры проекта. В последующих разделах объясняются соглашения, которые используются вместо конфигурации.

Совет. Все соглашения могут быть изменены путем замены стандартных компонентов MVC собственными реализациями. В книге будут описаны различные способы изменения соглашений, что поможет объяснить, как работают приложения MVC, но с рассматриваемыми здесь соглашениями придется иметь дело в большинстве проектов.

Следование соглашениям для классов контроллеров

Классы контроллеров имеют имена, которые заканчиваются на слово `Controller`, например, `ProductController`, `AdminController` и `HomeController`. При ссылке на контроллер в другом месте проекта, скажем, в случае применения вспомогательного метода HTML, указывается первая часть имени (вроде `Product`), а инфраструктура MVC автоматически добавляет к такому имени слово `Controller` и начинает поиск класса контроллера.

Совет. Описанное поведение можно изменить, создав соглашение для моделей, как будет объясняться в главе 31.

Следование соглашениям для представлений

Представления помещаются в папку по имени `/Views/`*`ИмяКонтроллера`*. Например, представление, ассоциированное с классом `ProductController`, попадет в папку `/Views/Product`.

Совет. Обратите внимание, что часть `Controller` имени класса в имени папки внутри `Views` не указывается, т.е. используется имя `/Views/Product`, ***а не*** `/Views/ProductController`. Поначалу такой подход может показаться нелогичным, но он быстро войдет в привычку.

Инфраструктура MVC ожидает, что стандартное представление для метода действия должно иметь имя этого метода. Например, представление, ассоциированное с методом действия по имени `List()`, должно называться `List.cshtml`. Таким образом, ожидается, что для метода действия `List()` класса `ProductController` стандартным представлением будет `/Views/Product/List.cshtml`. Стандартное представление применяется при возвращении результата вызова метода `View()` в методе действия, примерно так:

```
...
return View();
...
```

Можно указать имя другого представления:

```
...
return View("MyOtherView");
...
```

Обратите внимание, что мы не включаем в представление расширение имени файла или путь. При поиске представления инфраструктура MVC просматривает папку, имеющую имя контроллера, а затем папку `/Views/Shared`. Это значит, что представления, которые будут использоваться более чем одним контроллером, можно поместить в папку `/Views/Shared` и MVC успешно найдет их.

Следование соглашениям для компоновок

Соглашение об именовании для компоновок предусматривает снабжение имени файла префиксом в виде символа подчеркивания (_) и размещение файлов компоновки внутри папки `/Views/Shared`. Стандартная компоновка применяется по умолчанию ко всем представлениям через файл `/Views/_ViewStart.cshtml`. Если вы не хотите, чтобы стандартная компоновка применялась к представлениям, тогда можете

изменить настройки в файле `_ViewStart.cshtml` (либо вообще удалить его), указав другую компоновку в представлении, например:

```
@{
  Layout = "~/_MyLayout.cshtml";
}
```

Или же можете отключить компоновку для заданного представления:

```
@{
  Layout = null;
}
```

Резюме

В главе был представлен архитектурный паттерн MVC и его сравнение с несколькими другими паттернами, с которыми вы могли сталкиваться или слышать о них ранее. Вы узнали о важности модели предметной области и ознакомились с концепцией разделения обязанностей, которая позволяет отвязывать компоненты друг от друга для обеспечения строгого отделения частей приложения. В следующей главе объясняется структура проектов MVC в Visual Studio и рассматриваются важнейшие средства языка C#, которые используются при разработке веб-приложений MVC.

ГЛАВА 4

Важные функциональные возможности языка C#

В этой главе будут описаны функциональные возможности C#, используемые при разработке веб-приложений, которые не очень хорошо понимают или часто путают. Однако настоящая книга не посвящена языку C#, так что для каждой возможности будет приводиться только краткий пример, поэтому вы можете проработать примеры из остальных глав книги и задействовать нужные возможности в своих проектах. В табл. 4.1 приведена сводка для данной главы.

Таблица 4.1. Сводка по главе

Задача	Решение	Листинг
Избегание обращения к свойствам по ссылкам `null`	Используйте `null`-условную операцию	4.5–4.8
Упрощение свойств C#	Используйте автоматически реализуемые свойства	4.9–4.11
Упрощение формирования строк	Используйте интерполяцию строк	4.12
Создание объекта и установка его свойств за один шаг	Используйте инициализатор объекта или коллекции	4.13–4.16
Проверка типа или характеристик объекта	Используйте сопоставление с образцом	4.17–4.18
Добавление функциональности в класс, который не может быть модифицирован	Используйте расширяющий метод	4.19–4.26
Упрощение использования делегатов и однострочных методов	Используйте лямбда-выражение	4.27–4.34
Применение неявной типизации	Используйте ключевое слово `var`	4.35
Создание объектов без определения типа	Используйте анонимный тип	4.36, 4.37
Упрощение использования асинхронных методов	Используйте ключевые слова `async` и `await`	4.38–4.41
Получение имени метода или свойства класса без определения статической строки	Используйте выражение `nameof`	4.42, 4.43

Подготовка проекта для примера

Создадим для текущей главы в Visual Studio новый проект по имени `LanguageFeatures`, используя шаблон ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core (рис. 4.1).

Когда отобразятся различные конфигурации проекта, выберем шаблон Empty (Пустой), как показано на рис. 4.2. Выберем варианты .NET Core и ASP.NET Core 2.0 в списках в верхней части окна и удостоверимся в том, что переключатель Authentication (Аутентификация) установлен в No Authentication (Аутентификация отсутствует), а флажок Enable Docker Support (Включить поддержку Docker) не отмечен. Щелкнем на кнопке OK, чтобы создать проект.

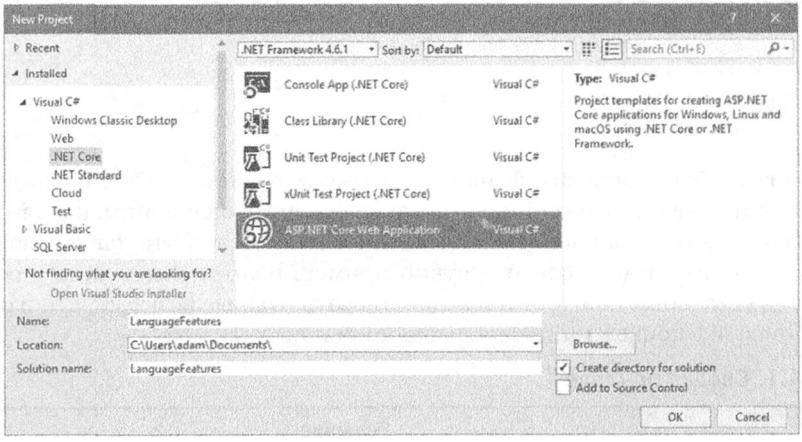


Рис. 4.1. Выбор типа проекта

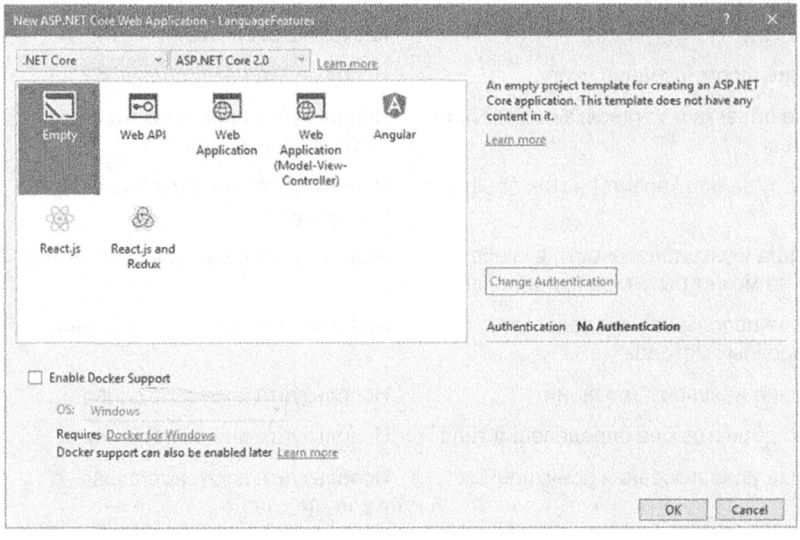


Рис. 4.2. Выбор шаблона проекта

Включение ASP.NET Core MVC

Шаблон проекта Empty создает проект, который содержит минимальную конфигурацию ASP.NET Core без какой-либо поддержки MVC. Это значит, что содержимое-заполнитель, добавляемое шаблоном Web Application (Model-View-Controller) (Веб-приложение (модель-представление-контроллер)), отсутствует, но также означает необходимость в выполнении ряда дополнительных шагов для включения MVC, чтобы заработали такие средства, как контроллеры и представления. В данном разделе мы внесем изменения, требуемые для включения MVC в проекте, но пока не будем вдаваться в детали каждого шага.

Чтобы включить инфраструктуру MVC, внесем в класс `Startup` изменения, приведенные в листинге 4.1.

Листинг 4.1. Включение MVC в файле `Startup.cs` из папки `LanguageFeatures`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace LanguageFeatures {

  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
      }

      // app.Run(async (context) => {
      //   await context.Response.WriteAsync("Hello World!");
      // });

      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Конфигурирование приложений ASP.NET Core MVC будет объясняться в главе 14, а пока достаточно знать, что два оператора, добавленные в листинге 4.1, обеспечивают базовую настройку MVC с применением стандартной конфигурации и соглашений.

Создание компонентов приложения MVC

Имея настроенную инфраструктуру MVC, можно заняться добавлением компонентов приложения MVC, которые будут использоваться для демонстрации важных языковых средств C#.

Создание модели

Начнем с создания простого класса модели, чтобы иметь какие-то данные, с которыми можно работать. Добавим папку по имени `Models` и создадим в ней файл класса `Product.cs` с определением, показанным в листинге 4.2.

Листинг 4.2. Содержимое файла `Product.cs` из папки `Models`

```
namespace LanguageFeatures.Models {
  public class Product {

    public string Name { get; set; }
    public decimal? Price { get; set; }

    public static Product[] GetProducts() {

      Product kayak = new Product {
        Name = "Kayak", Price = 275M
      };

      Product lifejacket = new Product {
        Name = "Lifejacket", Price = 48.95M
      };

      return new Product[] { kayak, lifejacket, null };
    }
  }
}
```

В классе `Products` определены свойства `Name` и `Price`, а также статический метод по имени `GetProducts()`, который возвращает массив элементов `Product`. Один из элементов, содержащихся в возвращаемом из метода `GetProducts()` массиве, установлен в `null`; он будет применяться для демонстрации ряда языковых средств позже в главе.

Создание контроллера и представления

В примерах настоящей главы для демонстрации различных языковых средств мы используем простой контроллер. Создадим папку `Controllers` и добавим в нее файл класса по имени `HomeController.cs`, содержимое которого приведено в листинге 4.3. В случае применения стандартной конфигурации инфраструктура MVC будет по умолчанию отправлять HTTP-запросы контроллеру `Home`.

Листинг 4.3. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      return View(new string[] { "C#", "Language", "Features" });
    }
  }
}
```

Метод действия `Index()` сообщает инфраструктуре MVC о необходимости визуализировать стандартное представление и передает ей массив строк, который дол-

жен быть включен в HTML-разметку, отправляемую клиенту. Чтобы создать соответствующее представление, добавим папку `Views/Home` (сначала создав папку `Views`, а затем внутри нее папку `Home`) и поместим в нее файл представления по имени `Index.cshtml` с содержимым, показанным в листинге 4.4.

Листинг 4.4. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<string>

@{ Layout = null; }

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Language Features</title>
</head>
<body>
  <ul>
    @foreach (string s in Model) {
      <li>@s</li>
    }
  </ul>
</body>
</html>
```

В результате запуска примера приложения за счет выбора пункта Start Debugging (Запустить отладку) в меню Debug (Отладка) появится вывод, представленный на рис. 4.3.

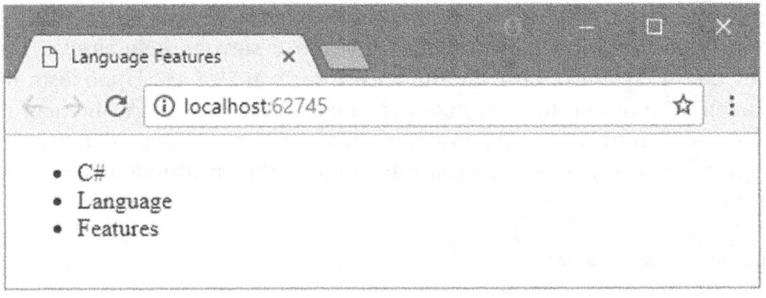


Рис. 4.3. Запуск пример приложения

Поскольку выводом во всех примерах данной главы является текст, отображаемые браузером сообщения в дальнейшем будут показаны так:

```
C#
Language
Features
```

Использование `null`-условной операции

С помощью `null`-условной операции можно более элегантно обнаруживать значения `null`. Во время разработки приложений MVC встречается много проверок на

предмет null, т.к. нужно выяснять, содержит ли запрос специфический заголовок или значение либо содержит ли модель определенный элемент данных. Традиционно работа со значениями null требовала явных проверок, которые могли становиться утомительными и подверженными ошибкам, когда требовалось инспектировать и объект, и его свойства. За счет применения null-условной операции такие проверки будут намного легче и компактнее (листинг 4.5).

Листинг 4.5. Обнаружение значений null в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {

      List<string> results = new List<string>();

      foreach (Product p in Product.GetProducts()) {
        string name = p?.Name;
        decimal? price = p?.Price;
        results.Add(string.Format("Name: {0}, Price: {1}", name, price));
      }
      return View(results);
    }
  }
}
```

Статический метод GetProducts(), определенный в классе Product, возвращает массив объектов, который инспектируется в методе действия Index() контроллера с целью получения списка значений Name и Price. Проблема в том, что и объект в массиве, и значения его свойств могут быть null, т.е. нельзя просто ссылаться на p.Name или p.Price внутри цикла foreach, не получив исключение NullReferenceException. Во избежание такой ситуации используется null-условная операция:

```
...
string name = p?.Name;
decimal? price = p?.Price;
...
```

null-условная операция обозначается знаком вопроса (?). Если значение p равно null, то переменная name также будет установлена в null. Если значение p не равно null, тогда переменной name будет присвоено значение свойства Person.Name. Аналогичной проверке подвергается и свойство Price. Обратите внимание, что переменная, которой выполняется присваивание с применением null-условной операции, должна быть в состоянии иметь дело со значениями null, поэтому переменная price объявлена с десятичным типом, допускающим null (decimal?).

Связывание в цепочки null-условных операций

Для навигации по иерархии объектов null-условные операции могут связываться в цепочки и превращаться в по-настоящему эффективный инструмент для упрощения

кода и обеспечения безопасной навигации. В листинге 4.6 к классу Product добавлено свойство с вложенными ссылками, что создает более сложную иерархию объектов.

Листинг 4.6. Добавление свойства в файле Product.cs из папки Models

```
namespace LanguageFeatures.Models {
  public class Product {

    public string Name { get; set; }
    public decimal? Price { get; set; }
    public Product Related { get; set; }

    public static Product[] GetProducts() {
      Product kayak = new Product {
        Name = "Kayak", Price = 275M
      };

      Product lifejacket = new Product {
        Name = "Lifejacket", Price = 48.95M
      };

      kayak.Related = lifejacket;

      return new Product[] { kayak, lifejacket, null };
    }
  }
}
```

Каждый объект Product имеет свойство Related, которое может ссылаться на другой объект Product. В методе GetProducts() мы устанавливаем свойство Related для объекта Product, представляющего каяк. В листинге 4.7 показано, как можно соединять вместе null-условные операции для навигации по свойствам объектов, не вызывая исключение.

Листинг 4.7. Обнаружение вложенных значений null в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      List<string> results = new List<string>();
      foreach (Product p in Product.GetProducts()) {
        string name = p?.Name;
        decimal? price = p?.Price;
        string relatedName = p?.Related?.Name;
        results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
            name, price, relatedName));
      }
      return View(results);
    }
  }
}
```

null-условную операцию можно применять к каждой части цепочки свойств, например:

```
...
string relatedName = p?.Related?.Name;
...
```

В таком случае переменная relatedName получит значение null, когда значение null имеет p или p.Related. Иначе relatedName будет присвоено значение свойства p.Related.Name. После запуска примера приложения в окне браузера появится следующий вывод:

```
Name: Kayak, Price: 275, Related: Lifejacket
Name: Lifejacket, Price: 48.95, Related:
Name: , Price: , Related:
```

Комбинирование null-условной операции и операции объединения с null

Для установки альтернативного значения, представляющего null, удобно комбинировать null-условную операцию (один знак вопроса) и операцию объединения с null (два знака вопроса), как демонстрируется в листинге 4.8.

Листинг 4.8. Сочетание операций работы с null в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      List<string> results = new List<string>();
      foreach (Product p in Product.GetProducts()) {
        string name = p?.Name ?? "<No Name>";
        decimal? price = p?.Price ?? 0;
        string relatedName = p?.Related?.Name ?? "<None>";
        results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
            name, price, relatedName));
      }
      return View(results);
    }
  }
}
```

null-условная операция гарантирует, что при навигации по свойствам объектов не возникнет исключение NullReferenceException, а операция объединения с null обеспечивает отсутствие значений null в результатах, отображаемых в браузере. В результате запуска примера приложения в окне браузера будет получен такой вывод:

```
Name: Kayak, Price: 275, Related: Lifejacket
Name: Lifejacket, Price: 48.95, Related: <None>
Name: <No Name>, Price: 0, Related: <None>
```

Использование автоматически реализуемых свойств

В языке C# поддерживаются автоматически реализуемые свойства, которые мы применяли при определении свойств класса `Person` в предыдущем разделе:

```
namespace LanguageFeatures.Models {
  public class Product {

    public string Name { get; set; }
    public decimal? Price { get; set; }
    public Product Related { get; set; }

    public static Product[] GetProducts() {

      Product kayak = new Product {
        Name = "Kayak", Price = 275M
      };
      Product lifejacket = new Product {
        Name = "Lifejacket", Price = 48.95M
      };

      kayak.Related = lifejacket;

      return new Product[] { kayak, lifejacket, null };
    }
  }
}
```

Данное средство дает возможность определять свойства, не реализуя блоки кода `get` и `set`. Средство автоматически реализуемых свойств позволяет трактовать следующее определение свойства:

```
...
public string Name { get; set; }
...
```

как эквивалентное приведенному ниже коду:

```
...
public string Name {
  get { return name; }
  set { name = value; }
}
...
```

Средства подобного типа известны как *"синтаксический сахар"*, который делает работу с C# более приятной (в этом случае за счет устранения избыточного кода, дублируемого в итоге для каждого свойства), однако существенно не изменяет поведение языка. Термин "сахар" может показаться умаляющим ценность средства, но любые улучшения, которые облегчают написание и сопровождение кода, приносят пользу — особенно в крупных и сложных проектах.

Использование инициализаторов автоматически реализуемых свойств

Автоматически реализуемые свойства поддерживаются, начиная с версии C# 3.0. В последней версии языка C# доступны инициализаторы для автоматически реали-

зуемых свойств, которые позволяют устанавливать начальные значения, не требуя применения конструкторов (листинг 4.9).

Листинг 4.9. Использование инициализатора автоматически реализуемого свойства в файле `Product.cs` из папки `Models`

```
namespace LanguageFeatures.Models {
  public class Product {

    public string Name { get; set; }
    public string Category { get; set; } = "Watersports";
    public decimal? Price { get; set; }
    public Product Related { get; set; }

    public static Product[] GetProducts() {
      Product kayak = new Product {
        Name = "Kayak",
        Category = "Water Craft",
        Price = 275M
      };
      Product lifejacket = new Product {
        Name = "Lifejacket", Price = 48.95M
      };
      kayak.Related = lifejacket;
      return new Product[] { kayak, lifejacket, null };
    }
  }
}
```

Присваивание значения автоматически реализуемому свойству не препятствует применению установщика для изменения свойства в более позднее время, а всего лишь приводит в порядок код для простых типов с конструкторами, которые содержат список присваиваний свойств, обеспечивающих наличие стандартных значений. В приведенном примере инициализатор присваивает свойству `Category` значение `"Watersports"`. Начальное значение может быть изменено, что и делается при создании объекта `kayak`, когда ему указывается значение `"Water Craft"`.

Создание автоматически реализуемых свойств только для чтения

Для создания свойства только для чтения в определении автоматически реализуемого свойства, которое имеет инициализатор, нужно опустить ключевое слово `set` (листинг 4.10).

Листинг 4.10. Создание свойства только для чтения в файле `Product.cs` из папки `Models`

```
namespace LanguageFeatures.Models {
  public class Product {

    public string Name { get; set; }
    public string Category { get; set; } = "Watersports";
    public decimal? Price { get; set; }
    public Product Related { get; set; }
    public bool InStock { get; } = true;
```

```
    public static Product[] GetProducts() {
      Product kayak = new Product {
        Name = "Kayak",
        Category = "Water Craft",
        Price = 275M
      };
      Product lifejacket = new Product {
        Name = "Lifejacket", Price = 48.95M
      };
      kayak.Related = lifejacket;
      return new Product[] { kayak, lifejacket, null };
    }
  }
}
```

Свойство `InStock` инициализируется значением `true` и не может быть изменено; тем не менее, ему можно присвоить значение внутри конструктора типа, как показано в листинге 4.11.

Листинг 4.11. Присваивание значения свойству только для чтения в файле `Product.cs` из папки `Models`

```
namespace LanguageFeatures.Models {
  public class Product {
    public Product(bool stock = true) {
      InStock = stock;
    }
    public string Name { get; set; }
    public string Category { get; set; } = "Watersports";
    public decimal? Price { get; set; }
    public Product Related { get; set; }
    public bool InStock { get; }
    public static Product[] GetProducts() {
      Product kayak = new Product {
        Name = "Kayak",
        Category = "Water Craft",
        Price = 275M
      };
      Product lifejacket = new Product(false) {
        Name = "Lifejacket",
        Price = 48.95M
      };
      kayak.Related = lifejacket;
      return new Product[] { kayak, lifejacket, null };
    }
  }
}
```

Конструктор позволяет указывать в качестве аргумента значение для свойства, допускающего только чтение, и если значение не предоставлено, тогда он устанавливает свойство в стандартное значение `true`. После установки конструктором значения свойства оно не может быть изменено.

Использование интерполяции строк

Традиционным инструментом C# для образования строк, содержащих значения данных, является метод `string.Format()`. Вот пример применения этого приема в контроллере `Home`:

```
...
results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
            name, price, relatedName));
...
```

Язык C# также поддерживает другой подход, называемый *интерполяцией строк*, который позволяет избежать необходимости гарантировать, что ссылки вида `{0}`, `{1}` и `{2}` в шаблоне строки соответствуют переменным, указанным в качестве аргументов. Взамен интерполяция строк использует имена переменных напрямую (листинг 4.12).

Листинг 4.12. Применение интерполяции строк в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {

      List<string> results = new List<string>();

      foreach (Product p in Product.GetProducts()) {
        string name = p?.Name ?? "<No Name>";
        decimal? price = p?.Price ?? 0;
        string relatedName = p?.Related?.Name ?? "<None>";
        results.Add($"Name: {name}, Price: {price}, Related: {relatedName}");
      }

      return View(results);
    }
  }
}
```

Интерполированная строка снабжается префиксом в виде символа `$` и содержит *"дыры"*, которые представляют собой ссылки на значения, содержащиеся внутри фигурных скобок (`{` и `}`). Когда строка оценивается, "дыры" заполняются текущими значениями указанных переменных и констант.

Среда Visual Studio обеспечивает поддержку средства IntelliSense для создания интерполированных строк и предлагает список доступных членов, когда набирается символ открывающей фигурной скобки, что помогает минимизировать количество опечаток, а результатом оказывается формат строки, который легче понять.

Совет. Интерполяция строк поддерживает все спецификаторы формата, которые доступны для метода `string.Format()`. Спецификаторы формата включаются в виде части "дыры", поэтому `$"Price: {price:C2}"` сформатирует значение `price` как денежное значение с двумя десятичными цифрами.

Использование инициализаторов объектов и коллекций

При создании объекта в статическом методе `GetProducts()` класса `Product` применялся *инициализатор объекта*, который позволяет создавать объект и указывать значения его свойств за один шаг, например:

```
...
Product kayak = new Product {
  Name = "Kayak",
  Category = "Water Craft",
  Price = 275M
};
...
```

Это еще одна форма "синтаксического сахара", делающая язык C# легче в использовании. Без такого средства пришлось бы вызывать конструктор `Product` и затем применять вновь созданный объект для установки всех его свойств:

```
...
Product kayak = new Product();
kayak.Name = "Kayak";
kayak.Category = "Water Craft";
kayak.Price = 275M;
...
```

Связанное с ним средство — *инициализатор коллекции* — позволяет создавать коллекцию и указывать ее содержимое за один шаг. Без такого инициализатора создание, скажем, массива потребовало бы указания размера и элементов массива по отдельности (листинг 4.13).

Листинг 4.13. Инициализация массива в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      string[] names = new string[3];
      names[0] = "Bob";
      names[1] = "Joe";
      names[2] = "Alice";
      return View("Index", names);
    }
  }
}
```

Инициализатор коллекции дает возможность указать содержимое массива как часть его конструирования, что неявно предоставляет компилятору размер массива (листинг 4.14).

Листинг 4.14. Использование инициализатора коллекции в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      return View("Index", new string[] { "Bob", "Joe", "Alice" });
    }
  }
}
```

Элементы массива задаются между символами { и }, что позволяет получить более краткое определение коллекции и делает возможным определять коллекцию внутри вызова метода. Код в листинге 4.14 дает тот же результат, что и код в листинге 4.13; если запустить пример приложения, то в окне браузера получится следующий вывод:

```
Bob
Joe
Alice
```

Использование инициализатора индексированной коллекции

В последних версиях C# улучшен способ инициализации коллекций с индексами, таких как словари. В листинге 4.15 приведен код метода действия Index(), переписанный для определения коллекции с помощью традиционного подхода C# к инициализации словаря.

Листинг 4.15. Инициализация словаря в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
     Dictionary<string, Product> products = new Dictionary<string, Product> {
        { "Kayak", new Product { Name = "Kayak", Price = 275M } },
        { "Lifejacket", new Product{ Name = "Lifejacket", Price = 48.95M } }
      };
      return View("Index", products.Keys);
    }
  }
}
```

Синтаксис для инициализации коллекции такого типа слишком сильно полагается на символы { и }, особенно когда значения коллекции создаются с применением инициализаторов объектов. В последних версиях C# поддерживается более естественный подход к инициализации индексированной коллекции, который согласован со способом извлечения или модификации значений после того, как коллекция была инициализирована (листинг 4.16).

Листинг 4.16. Использование синтаксиса инициализатора коллекции в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      Dictionary<string, Product> products = new Dictionary<string, Product> {
        ["Kayak"] = new Product { Name = "Kayak", Price = 275M },
        ["Lifejacket"] = new Product { Name = "Lifejacket", Price = 48.95M }
      };

      return View("Index", products.Keys);
    }
  }
}
```

Результат остается прежним, т.е. создание словаря, ключами которого являются Kayak и Lifejacket, а значениями — объекты Product, но элементы создаются с применением системы обозначений для индексов, используемой в других операциях с коллекциями. Вот вывод, получаемый в окне браузера после запуска примера приложения:

```
Kayak
Lifejacket
```

Сопоставление с образцом

Одним из наиболее полезных недавних добавлений в язык C# стала поддержка сопоставления с образцом, которую можно использовать для проверки, что объект относится к определенному типу или имеет специфические характеристики. Сопоставление с образцом — еще одна форма "синтаксического сахара", и она способна значительно упростить сложные блоки условных операторов. Ключевое слово is применяется для выполнения проверки типа, как демонстрируется в листинге 4.17.

Листинг 4.17. Выполнение проверки типа в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {
```

```
    public ViewResult Index() {
      object[] data = new object[] { 275M, 29.95M,
        "apple", "orange", 100, 10 };
      decimal total = 0;
      for (int i = 0; i < data.Length; i++) {
        if (data[i] is decimal d) {
          total += d;
        }
      }
      return View("Index", new string[] { $"Total: {total:C2}" });
    }
  }
}
```

Ключевое слово `is` выполняет проверку типа, и если значение принадлежит указанному типу, тогда оно присваивается новой переменной:

```
...
if (data[i] is decimal d) {
...
```

Выражение будет оценено как `true`, если значение, хранящееся в `data[i]`, имеет тип `decimal`. Значение `data[i]` будет присваиваться переменной `d`, что позволит его использовать в последующих операторах без необходимости в каких-либо преобразованиях типов. Ключевое слово `is` даст совпадение лишь с указанным типом, т.е. будут обработаны только два значения в массиве `data[]` (остальные элементы в массиве относятся к типам `string` и `int`). В результате запуска приложения в окне браузера появится следующий вывод:

```
Total: $304.95
```

Сопоставление с образцом в операторах `switch`

Сопоставление с образцом может также применяться в операторах `switch`, которые поддерживают ключевое слово `when` для ограничения случаев, когда значение сопоставляется в операторе `case` (листинг 4.18).

Листинг 4.18. Сопоставление с образцом в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      object[] data = new object[] { 275M, 29.95M,
        "apple", "orange", 100, 10 };
      decimal total = 0;
      for (int i = 0; i < data.Length; i++) {
        switch (data[i]) {
          case decimal decimalValue:
            total += decimalValue;
            break;
```

```
            case int intValue when intValue > 50:
              total += intValue;
              break;
          }
        }
        return View("Index", new string[] { $"Total: {total:C2}" });
      }
    }
}
```

Для сопоставления любого значения специфического типа в операторе `case` используются тип и имя переменной, например:

```
...
case decimal decimalValue:
...
```

Такой оператор `case` сопоставляется с любым значением `decimal` и присваивает его новой переменной по имени `decimalValue`. Чтобы достичь большей избирательности, можно добавить ключевое слово `when`:

```
...
case int intValue when intValue > 50:
...
```

Этот оператор `case` сопоставляет значения типа `int` и присваивает их переменной по имени `intValue`, но лишь когда значение больше 50. Запуск приложения приводит к следующему выводу в окне браузера:

```
Total: $404.95
```

Использование расширяющих методов

Расширяющие методы — удобный способ добавления методов в классы, владельцем которых вы не являетесь и не можете модифицировать напрямую. В листинге 4.19 приведено определение класса `ShoppingCart`, добавленного в папку `Models` в виде файла `ShoppingCart.cs`, который представляет коллекцию объектов `Product`.

Листинг 4.19. Содержимое файла `ShoppingCart.cs` из папки `Models`

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

  public class ShoppingCart {
    public IEnumerable<Product> Products { get; set; }
  }
}
```

Это простой класс, который действует как оболочка для последовательности объектов `Product` (в данном примере необходим лишь элементарный класс). Предположим, что нас интересует возможность определения общей стоимости объектов `Product` в классе `ShoppingCart`, но мы не можем изменить сам класс из-за того, что он поступил от третьей стороны и его исходный код отсутствует. Для добавления нужной функциональности можно применить расширяющий метод. В листинге 4.20 показан класс `MyExtensionMethods`, также добавленный в папку `Models` в виде файла `MyExtensionMethods.cs`.

Листинг 4.20. Содержимое файла MyExtensionMethods.cs из папки Models

```
namespace LanguageFeatures.Models {
  public static class MyExtensionMethods {
    public static decimal TotalPrices(this ShoppingCart cartParam) {
      decimal total = 0;
      foreach (Product prod in cartParam.Products) {
        total += prod?.Price ?? 0;
      }
      return total;
    }
  }
}
```

Ключевое слово `this`, расположенное перед первым параметром, помечает `TotalPrices()` как расширяющий метод. Первый параметр указывает .NET, к какому классу может применяться расширяющий метод — к `ShoppingCart` в данном случае. На экземпляр класса `ShoppingCart`, к которому применен расширяющий метод, можно ссылаться с использованием параметра `cartParam`. Расширяющий метод проходит по объектам `Product` в `ShoppingCart` и возвращает сумму значений их свойств `Product.Price`. В листинге 4.21 демонстрируется применение расширяющего метода в методе действия контроллера `Home`.

На заметку! Расширяющие методы не позволяют нарушать правила доступа, которые классы определяют для своих методов, полей и свойств. С помощью расширяющего метода функциональность класса можно расширить, но с использованием только тех членов, к которым в любом случае имеется доступ.

Листинг 4.21. Применение расширяющего метода в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {
    public ViewResult Index() {
      ShoppingCart cart
        = new ShoppingCart { Products = Product.GetProducts() };
      decimal cartTotal = cart.TotalPrices();
      return View("Index", new string[] { $"Total: {cartTotal:C2}" });
    }
  }
}
```

Вот ключевой оператор:

```
...
decimal cartTotal = cart.TotalPrices();
...
```

Метод TotalPrices() вызывается на объекте ShoppingCart, как если бы он был частью класса ShoppingCart, хотя он представляет собой расширяющий метод, который определен в совершенно другом классе. Среда .NET будет обнаруживать расширяющие классы, если они находятся в области действия текущего класса, т.е. являются частью того же самого пространства имен или пространства имен, которое указано в операторе using. В результате запуска примера приложения в окне браузера появится следующий вывод:

```
Total: $323.95
```

Применение расширяющих методов к интерфейсу

Можно также создавать расширяющие методы, которые применяются к интерфейсу, что позволит вызывать такие расширяющие методы для всех классов, реализующих данный интерфейс. В листинге 4.22 приведен код класса ShoppingCart, модифицированный с целью реализации интерфейса IEnumerable<Product>.

Листинг 4.22. Реализация интерфейса в файле ShoppingCart.cs из папки Models

```
using System.Collections;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

  public class ShoppingCart : IEnumerable<Product> {
    public IEnumerable<Product> Products { get; set; }

    public IEnumerator<Product> GetEnumerator() {
      return Products.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator() {
      return GetEnumerator();
    }
  }
}
```

Теперь расширяющий метод можно изменить так, чтобы он работал с интерфейсом IEnumerable<Product> (листинг 4.23).

Листинг 4.23. Модификация расширяющего метода в файле MyExtensionMethods.cs из папки Models

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

  public static class MyExtensionMethods {

    public static decimal TotalPrices(this IEnumerable<Product> products) {
      decimal total = 0;
      foreach (Product prod in products) {
        total += prod?.Price ?? 0;
      }
      return total;
    }
  }
}
```

Тип первого параметра был изменен на IEnumerable<Product>, а это значит, что цикл foreach в теле метода работает непосредственно с объектами Product. Переход на использование упомянутого интерфейса означает, что мы можем подсчитать общую стоимость объектов Product, перечисляемых посредством любого интерфейса IEnumerable<Product>, что включает не только экземпляры ShoppingCart, но также массивы объектов Product (листинг 4.24).

Листинг 4.24. Применение расширяющего метода к массиву в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {

      ShoppingCart cart
        = new ShoppingCart { Products = Product.GetProducts() };

      Product[] productArray = {
        new Product {Name = "Kayak", Price = 275M},
        new Product {Name = "Lifejacket", Price = 48.95M}
      };

      decimal cartTotal = cart.TotalPrices();
      decimal arrayTotal = productArray.TotalPrices();

      return View("Index", new string[] {
        $"Cart Total: {cartTotal:C2}",
        $"Array Total: {arrayTotal:C2}" });
    }
  }
}
```

После запуска проекта будут получены показанные ниже результаты, которые демонстрируют, что расширяющий метод возвращает один и тот же результат независимо от способа перебора объектов Product:

```
Cart Total: $323.95
Array Total: $323.95
```

Создание фильтрующих расширяющих методов

Последний аспект расширяющих методов, о котором необходимо упомянуть — возможность их использования для фильтрации коллекций объектов. Расширяющий метод, который оперирует на интерфейсе IEnumerable<T> и также возвращает IEnumerable<T>, может задействовать ключевое слово yield, чтобы применить критерий отбора к элементам в источнике данных с целью генерации сокращенного набора результатов. В листинге 4.25 представлен такой метод, который добавляется в класс MyExtensionMethods.

Листинг 4.25. Добавление фильтрующего расширяющего метода в файле MyExtensionMethods.cs из папки Controllers

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

  public static class MyExtensionMethods {

    public static decimal TotalPrices(this IEnumerable<Product> products) {
      decimal total = 0;
      foreach (Product prod in products) {
        total += prod?.Price ?? 0;
      }
      return total;
    }

    public static IEnumerable<Product> FilterByPrice(
       this IEnumerable<Product> productEnum, decimal minimumPrice) {
      foreach (Product prod in productEnum) {
        if ((prod?.Price ?? 0) >= minimumPrice) {
          yield return prod;
        }
      }
    }
  }
}
```

Расширяющий метод по имени `FilterByPrice()` принимает дополнительный параметр, который позволяет фильтровать товары, так что в результате возвращаются объекты `Product`, у которых значение свойства `Price` совпадает или превышает значение, указанное в параметре. Использование метода `FilterByPrice()` демонстрируется в листинге 4.26.

Листинг 4.26. Применение фильтрующего расширяющего метода в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      Product[] productArray = {
        new Product {Name = "Kayak", Price = 275M},
        new Product {Name = "Lifejacket", Price = 48.95M},
        new Product {Name = "Soccer ball", Price = 19.50M},
        new Product {Name = "Corner flag", Price = 34.95M}
      };

      decimal arrayTotal = productArray.FilterByPrice(20).TotalPrices();

      return View("Index", new string[] { $"Array Total: {arrayTotal:C2}" });
    }
  }
}
```

При вызове метода `FilterByPrice()` на массиве объектов `Product` метод `TotalPrices()` получает и использует для подсчета суммы только те из них, которые стоят больше $20. В результате запуска примера приложения в окне браузера отобразится следующий вывод:

```
Total: $358.90
```

Использование лямбда-выражений

Лямбда-выражения являются функциональной возможностью, ставшей причиной многочисленных заблуждений — не в последнюю очередь из-за того, что упрощаемое с их помощью средство само вызывает путаницу. Чтобы понять решаемую задачу, рассмотрим расширяющий метод `FilterByPrice()`, который был определен в предыдущем разделе. Метод реализован так, что он может фильтровать объекты `Product` по цене, а это значит, что если понадобится фильтровать объекты по названию, тогда придется создать второй метод вроде приведенного в листинге 4.27.

Листинг 4.27. Добавление фильтрующего метода в файле `MyExtensionMethods.cs` из папки `Models`

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

  public static class MyExtensionMethods {

    public static decimal TotalPrices(this IEnumerable<Product> products) {
      decimal total = 0;
      foreach (Product prod in products) {
        total += prod?.Price ?? 0;
      }
      return total;
    }

    public static IEnumerable<Product> FilterByPrice(
        this IEnumerable<Product> productEnum, decimal minimumPrice) {
      foreach (Product prod in productEnum) {
        if ((prod?.Price ?? 0) >= minimumPrice) {
          yield return prod;
        }
      }
    }

    public static IEnumerable<Product> FilterByName(
        this IEnumerable<Product> productEnum, char firstLetter) {
      foreach (Product prod in productEnum) {
        if (prod?.Name?[0] == firstLetter) {
          yield return prod;
        }
      }
    }
  }
}
```

В листинге 4.28 демонстрируется применение обоих фильтрующих методов внутри контроллера для создания двух разных итоговых сумм.

Листинг 4.28. Использование двух фильтрующих методов в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {

      Product[] productArray = {
        new Product {Name = "Kayak", Price = 275M},
        new Product {Name = "Lifejacket", Price = 48.95M},
        new Product {Name = "Soccer ball", Price = 19.50M},
        new Product {Name = "Corner flag", Price = 34.95M}
      };

      decimal priceFilterTotal = productArray.FilterByPrice(20).TotalPrices();
      decimal nameFilterTotal = productArray.FilterByName('S').TotalPrices();

      return View("Index", new string[] {
        $"Price Total: {priceFilterTotal:C2}",
        $"Name Total: {nameFilterTotal:C2}" });
    }
  }
}
```

Первый фильтр отбирает все товары с ценой $20 и выше, а второй фильтр — товары с названиями, начинающимися на букву S. После запуска примера приложения в окне браузера появится такой вывод:

```
Price Total: $358.90
Name Total: $19.50
```

Определение функций

Описанный выше процесс можно повторять до бесконечности и создавать фильтрующие методы для каждого интересующего свойства и сочетаний свойств. Более элегантный подход предусматривает отделение кода, обрабатывающего перечисление, от критерия отбора. Язык C# облегчает задачу, позволяя передавать функции как объекты. В листинге 4.29 показан единственный расширяющий метод, который фильтрует перечисление объектов Product, но делегирует отдельной функции решение о том, какие из них включать в результат.

Листинг 4.29. Создание универсального фильтрующего метода в файле MyExtensionMethods.cs из папки Models

```
using System.Collections.Generic;
using System;

namespace LanguageFeatures.Models {

  public static class MyExtensionMethods {

    public static decimal TotalPrices(this IEnumerable<Product> products) {
      decimal total = 0;
```

```
    foreach (Product prod in products) {
      total += prod?.Price ?? 0;
    }
    return total;
  }
  public static IEnumerable<Product> Filter(
      this IEnumerable<Product> productEnum,
      Func<Product, bool> selector) {
    foreach (Product prod in productEnum) {
      if (selector(prod)) {
        yield return prod;
      }
    }
  }
}
```

Вторым аргументом метода `Filter()` является функция, которая принимает объект `Product` и возвращает значение типа `bool`. Метод `Filter()` вызывает эту функцию для каждого объекта `Product` и включает его в результат, если функция возвращает `true`. Для применения метода `Filter()` можно указать метод или создать автономную функцию (листинг 4.30).

Листинг 4.30. Использование функции для фильтрации объектов `Product` в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    bool FilterByPrice(Product p) {
      return (p?.Price ?? 0) >= 20;
    }

    public ViewResult Index() {

      Product[] productArray = {
        new Product {Name = "Kayak", Price = 275M},
        new Product {Name = "Lifejacket", Price = 48.95M},
        new Product {Name = "Soccer ball", Price = 19.50M},
        new Product {Name = "Corner flag", Price = 34.95M}
      };

      Func<Product, bool> nameFilter = delegate (Product prod) {
        return prod?.Name?[0] == 'S';
      };

      decimal priceFilterTotal = productArray
        .Filter(FilterByPrice)
        .TotalPrices();
      decimal nameFilterTotal = productArray
        .Filter(nameFilter)
        .TotalPrices();
```

```
    return View("Index", new string[] {
      $"Price Total: {priceFilterTotal:C2}",
      $"Name Total: {nameFilterTotal:C2}" });
  }
 }
}
```

Оба подхода не идеальны. Определение методов, подобных `FilterByPrice()`, засоряет определение класса. Создание объекта `Func<Product, bool>` устраняет такую проблему, но сопряжено с неудобным синтаксисом, который труден для восприятия и сопровождения. Именно эту задачу решают лямбда-выражения, позволяя определять функции более элегантным и выразительным способом, как видно в листинге 4.31.

Листинг 4.31. Использование лямбда-выражений в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;

namespace LanguageFeatures.Controllers {
 public class HomeController : Controller {

  public ViewResult Index() {

    Product[] productArray = {
      new Product {Name = "Kayak", Price = 275M},
      new Product {Name = "Lifejacket", Price = 48.95M},
      new Product {Name = "Soccer ball", Price = 19.50M},
      new Product {Name = "Corner flag", Price = 34.95M}
    };

    decimal priceFilterTotal = productArray
      .Filter(p => (p?.Price ?? 0) >= 20)
      .TotalPrices();
    decimal nameFilterTotal = productArray
      .Filter(p => p?.Name?[0] == 'S')
      .TotalPrices();

    return View("Index", new string[] {
      $"Price Total: {priceFilterTotal:C2}",
      $"Name Total: {nameFilterTotal:C2}" });
  }
 }
}
```

Лямбда-выражения выделены полужирным. Параметры выражаются без указания типа, который будет выведен автоматически. Символы `=>` можно читать как "направляется в" и они связывают параметр с результатом лямбда-выражения. В рассматриваемом примере параметр `Product` по имени `p` направляется в результат `bool`, который будет равен `true`, если значение свойства `Price` эквивалентно или превышает `20` в первом выражении или если значение свойства `Name` начинается с буквы `S` во втором выражении. Такой код работает аналогично отдельному методу и делегату в виде функции, но он короче и для большинства людей легче в восприятии.

Другие формы лямбда-выражений

Представлять логику делегата посредством лямбда-выражения вовсе не обязательно. С тем же успехом можно вызвать метод:

```
prod => EvaluateProduct(prod)
```

Если требуется лямбда-выражение для делегата, который имеет несколько параметров, то параметры должны быть заключены в круглые скобки:

```
(prod, count) => prod.Price > 20 && count > 0
```

Наконец, если в лямбда-выражении необходима логика, которая требует более одного оператора, то ее можно реализовать, используя фигурные скобки ({}) и завершая блок оператором `return`:

```
(prod, count) => {
  // ...несколько операторов кода...
  return result;
}
```

Применять лямбда-выражения в коде не обязательно, но они служат изящным способом выражения сложных функций в читабельной и ясной манере. Вы будете часто встречать лямбда-выражения в примерах, приводимых далее в книге.

Использование методов и свойств в форме лямбда-выражений

Лямбда-выражения можно применять для реализации конструкторов, методов и свойств. Во время разработки приложений MVC, особенно при написании контроллеров, часто появляются методы, которые содержат единственный оператор, выбирающий данные для отображения и представление для визуализации. В листинге 4.32 приведен код метода действия `Index()`, переписанный в соответствии с таким общим шаблоном.

Листинг 4.32. Создание общего шаблона действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {
    public ViewResult Index() {
      return View(Product.GetProducts().Select(p => p?.Name));
    }
  }
}
```

Метод действия `Index()` получает от статического метода `Product.GetProducts()` коллекцию объектов `Product` и с помощью LINQ строит проекцию значений свойств `Name`. Затем проекция применяется в качестве модели представления для стандартного представления. В результате запуска примера приложения в окне браузера окажется следующий вывод:

```
Kayak
Lifejacket
```

В окне браузера также будет присутствовать пустой элемент списка, потому что метод `GetProducts()` включает в свой результат ссылку `null`, но в данном разделе главы это неважно.

Когда тело конструктора или метода состоит из единственного оператора, его можно переписать в виде лямбда-выражения (листинг 4.33).

Листинг 4.33. Метод действия, представленный как лямбда-выражение, в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() =>
      View(Product.GetProducts().Select(p => p?.Name));
  }
}
```

Лямбда-выражения для методов позволяют опустить ключевое слово `return` и используют символы => (направляется в) для ассоциирования сигнатуры метода (включая аргументы) с его реализацией. Метод `Index()`, показанный в листинге 4.33, работает точно так же, как метод `Index()` из листинга 4.32, но представлен более лаконично. Тот же самый базовый подход можно также применять для определения свойств. В листинге 4.34 демонстрируется добавление в класс `Product` свойства, которое использует лямбда-выражение.

Листинг 4.34. Представление свойства как лямбда-выражения в файле `Product.cs` из папки `Models`

```
namespace LanguageFeatures.Models {
  public class Product {

    public Product(bool stock = true) {
      InStock = stock;
    }

    public string Name { get; set; }
    public string Category { get; set; } = "Watersports";
    public decimal? Price { get; set; }
    public Product Related { get; set; }
    public bool InStock { get; }
    public bool NameBeginsWithS => Name?[0] == 'S';

    public static Product[] GetProducts() {
      Product kayak = new Product {
        Name = "Kayak",
        Category = "Water Craft",
        Price = 275M
      };
```

```
      Product lifejacket = new Product(false) {
        Name = "Lifejacket",
        Price = 48.95M
      };
      kayak.Related = lifejacket;
      return new Product[] { kayak, lifejacket, null };
    }
  }
}
```

Использование автоматического выведения типа и анонимных типов

Ключевое слово `var` позволяет определять локальную переменную без явного указания ее типа (листинг 4.35). Такой прием называется *выведением типа* или *неявной типизацией*.

Листинг 4.35. Использование выведения типа в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      var names = new [] { "Kayak", "Lifejacket", "Soccer ball" };
      return View(names);
    }
  }
}
```

Речь идет вовсе не о том, что переменная `myVariable` не имеет типа; мы всего лишь предложили компилятору самостоятельно вывести тип из кода. Компилятор исследует объявление массива и решает, что он является строковым. Выполнение примера дает следующий вывод:

```
Kayak
Lifejacket
Soccer ball
```

Использование анонимных типов

Комбинируя инициализаторы объектов и выведение типов, можно создавать простые объекты модели представления, которые удобны для передачи данных между контроллером и представлением, без необходимости в определении класса или структуры (листинг 4.36).

Листинг 4.36. Создание анонимного типа в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      var products = new [] {
        new { Name = "Kayak", Price = 275M },
        new { Name = "Lifejacket", Price = 48.95M },
        new { Name = "Soccer ball", Price = 19.50M },
        new { Name = "Corner flag", Price = 34.95M }
      };

      return View(products.Select(p => p.Name));
    }
  }
}
```

Каждый объект в массиве `products` является анонимно типизированным. Это не означает, что объекты имеют динамическую природу в том смысле, в каком считаются динамическими переменные JavaScript. Это просто значит, что определение типа будет создано автоматически компилятором. Строгая типизация по-прежнему обеспечивается. Скажем, вы можете получать и устанавливать только те свойства, которые были определены в инициализаторе. В результате запуска примера в окне браузера отобразится следующий вывод:

```
Kayak
Lifejacket
Soccer ball
Corner flag
```

Компилятор C# генерирует класс на основе имени и типов параметров в инициализаторе. Два анонимно типизированных объекта, которые имеют те же самые имена и типы свойств, будут относиться к одному и тому же автоматически сгенерированному классу. В результате все объекты в массиве `products` получат один и тот же тип, поскольку они определяют те же самые свойства.

Совет. Для определения массива анонимно типизированных объектов должно применяться ключевое слово `var`, т.к. тип не будет создан до тех пор, пока код не скомпилируется, и его имя не известно. Все элементы в массиве анонимно типизированных объектов обязаны определять те же самые свойства, иначе компилятор не сможет выяснить тип массива.

В целях демонстрации изменим вывод в листинге 4.37, чтобы отображать имя типа вместо значения свойства `Name`.

Листинг 4.37. Отображение имени анонимного типа в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      var products = new [] {
        new { Name = "Kayak", Price = 275M },
        new { Name = "Lifejacket", Price = 48.95M },
        new { Name = "Soccer ball", Price = 19.50M },
        new { Name = "Corner flag", Price = 34.95M }
      };

      return View(products.Select(p => p.GetType().Name));
    }
  }
}
```

Всем объектам в массиве был назначен один и тот же тип, в чем можно убедиться, запустив пример. Имя типа не выглядит дружественным к пользователю, но оно и не рассчитано на непосредственное использование, и в вашем случае может оказаться другим:

```
<>f__AnonymousType0`2
<>f__AnonymousType0`2
<>f__AnonymousType0`2
<>f__AnonymousType0`2
```

Использование асинхронных методов

Асинхронные методы осуществляют возврат и выполняют работу в фоновом режиме с уведомлением о ее завершении, позволяя коду в это время заниматься другими действиями. Асинхронные методы — важный инструмент при устранении узких мест в коде; они позволяют приложениям извлекать преимущества от наличия нескольких процессоров и процессорных ядер, чтобы выполнять работу в параллельном режиме.

В инфраструктуре MVC асинхронные методы могут применяться для увеличения общей производительности приложения, предоставляя серверу большую степень гибкости относительно того, как запросы планируются и выполняются. Для выполнения работы асинхронным образом используются два ключевых слова C# — `async` и `await`.

Преследуемые в настоящем разделе цели требуют добавления в пример проекта новой сборки .NET, чтобы можно было делать асинхронные HTTP-запросы. Щелкнем правой кнопкой мыши на имени проекта `LanguageFeatures` в окне Solution Explorer, выберем в контекстном меню пункт Edit LanguageFeatures.csproj (Редактировать `LanguageFeatures.csproj`) и добавим элемент, показанный в листинге 4.38.

Листинг 4.38. Добавление пакета в файле LanguageFeatures.csproj из папки LanguageFeatures

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version="4.3.2" />
  </ItemGroup>
</Project>
```

После сохранения файла LanguageFeatures.csproj среда Visual Studio загрузит сборку System.Net.Http и добавит ее в проект. Более подробно процесс будет описан в главе 6.

Работа с задачами напрямую

Язык C# и платформа .NET предлагают великолепную поддержку для асинхронных методов, но код быстро становится многословным, а разработчики, не привыкшие к параллельному программированию, зачастую не могут справиться с необычным синтаксисом. В качестве примера в листинге 4.39 приведен асинхронный метод по имени GetPageLength(), который определен в классе MyAsyncMethods, добавленном в папку Models в виде файла MyAsyncMethods.cs.

Листинг 4.39. Содержимое файла MyAsyncMethods.cs из папки Models

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {

  public class MyAsyncMethods {

    public static Task<long?> GetPageLength() {

      HttpClient client = new HttpClient();

      var httpTask = client.GetAsync("http://apress.com");

      return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) =>
      {
        return antecedent.Result.Content.Headers.ContentLength;
      });
    }
  }
}
```

Метод использует объект System.Net.Http.HttpClient для запрашивания содержимого домашней страницы издательства Apress и возвращает длину полученного содержимого. Работа, которая будет выполняться асинхронно, представлена в .NET

как объект Task. Объекты Task строго типизируются на основе результата, выдаваемого фоновой работой. Таким образом, при вызове метода HttpClient.GetAsync() получается объект Task<HttpResponseMessage>. Он сообщает о том, что запрос выполнится в фоновом режиме и его результатом будет объект HttpResponseMessage.

Совет. Когда мы употребляем понятия вроде *фоновый режим*, то опускаем массу деталей, чтобы подчеркнуть только ключевые аспекты, которые важны для мира MVC. Поддержка .NET для асинхронных методов и параллельного программирования в целом превосходна, и ее рекомендуется внимательно изучить, если вы хотите создавать по-настоящему высокопроизводительные приложения, которые способны извлекать преимущества от многопроцессорного или многоядерного оборудования. Вы увидите, как MVC облегчает создание асинхронных веб-приложений далее в книге по мере представления разнообразных функциональных средств.

Самой непонятной для большинства программистов частью является *продолжение*, представляющее собой механизм, с помощью которого указывается то, что должно произойти, когда фоновая задача завершится. В приведенном примере применяется метод ContinueWith() для обработки объекта HttpResponseMessage, получаемого из метода HttpClient.GetAsync(). Это делается с использованием лямбда-выражения, которое возвращает значение свойства, содержащего длину полученного от веб-сервера Apress содержимого. Вот код продолжения:

```
...
return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) =>
{
  return antecedent.Result.Content.Headers.ContentLength;
});
...
```

Обратите внимание, что ключевое слово return встречается два раза. Именно данная часть вызывает путаницу. Первое применение ключевого слова return указывает, что возвращается объект Task<HttpResponseMessage>, который при завершении задачи возвратит (второе ключевое слово return) длину заголовка ContentLength. Заголовок ContentLength возвращает значение long? (тип long, допускающий null), т.е. результатом метода GetPageLength() является Task<long?>:

```
...
public static Task<long?> GetPageLength() {
...
```

Не переживайте, если все сказанное выглядит для вас бессмысленным — в своем замешательстве вы не одиноки. Именно по этой причине в Microsoft и решили добавить в C# ключевые слова, упрощающие работу с асинхронными методами.

Применение ключевых слов async и await

Разработчики из Microsoft ввели в язык C# два ключевых слова, которые специально призваны облегчить использование асинхронных методов, подобных HttpClient.GetAsync(). Ими являются async и await; в листинге 4.40 показано, как с их помощью упростить наш пример метода.

Листинг 4.40. Применение ключевых слов async и await в файле MyAsyncMethods.cs из папки Models

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {
  public class MyAsyncMethods {
    public async static Task<long?> GetPageLength() {
      HttpClient client = new HttpClient();
      var httpMessage = await client.GetAsync("http://apress.com");
      return httpMessage.Content.Headers.ContentLength;
    }
  }
}
```

Ключевое слово await используется при вызове асинхронного метода. Оно сообщает компилятору C# о том, что необходимо подождать результата Task, который возвращается методом GetAsync(), и затем заняться выполнением остальных операторов в том же методе.

Применение ключевого слова await означает, что мы можем трактовать результат метода GetAsync(), как если бы он был обычным методом, и просто присвоить возвращаемый им объект HttpResponseMessage какой-нибудь переменной. Еще лучше то, что затем можно использовать ключевое слово return традиционным образом для выдачи результата из другого метода — значения свойства ContentLength в данном случае. Это намного более естественный подход, к тому же нам не придется беспокоиться по поводу метода ContinueWith() и многократного применения ключевого слова return.

При использовании ключевого слова await потребуется также добавить к сигнатуре метода ключевое слово async, как было сделано в рассмотренном выше примере. Тип результата метода не изменяется — метод GetPageLength() по-прежнему возвращает Task<long?>. Причина в том, что ключевые слова await и async реализованы с применением ряда искусных трюков компилятора, которые позволяют использовать более естественный синтаксис, но не изменяют того, что происходит внутри методов, к которым они применены. Программист, вызывающий метод GetPageLength(), по-прежнему имеет дело с результатом Task<long?>, т.к. все еще существует фоновая операция, которая выдает значение long, допускающее null; хотя, конечно же, программист может также предпочесть пользоваться ключевыми словами await и async.

Такой шаблон повсеместно соблюдается в контроллере MVC, что позволяет легко писать асинхронные методы действий (листинг 4.41).

Листинг 4.41. Определение асинхронных методов действий в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace LanguageFeatures.Controllers {
```

```
  public class HomeController : Controller {
    public async Task<ViewResult> Index() {
      long? length = await MyAsyncMethods.GetPageLength();
      return View(new string[] { $"Length: {length}" });
    }
  }
}
```

Тип результата метода действия `Index()` изменен на `Task<ViewResult>`. Подобным образом инфраструктуре MVC сообщается о том, что метод действия будет возвращать объект `Task`, который по завершении выдаст объект `ViewResult`, а тот предоставит детали представления, подлежащего визуализации, и требующиеся ему данные. К определению метода было добавлено ключевое слово `async`, что позволило использовать ключевое слово `await` при вызове метода `MyAsyncMethods.GetPathLength()`. О продолжениях позаботятся инфраструктура MVC и платформа .NET, а результатом будет код, который легко писать, читать и сопровождать. После запуска приложения появится вывод, похожий на показанный ниже (хотя наверняка с отличающейся длиной, т.к. содержимое веб-сайта Apress часто изменяется):

```
Length: 54576
```

Получение имен

Во время разработки веб-приложений есть много задач, в которых необходимо ссылаться на имя аргумента, переменной, метода или класса. Распространенными примерами может служить генерация исключения или создание ошибки проверки достоверности при обработке пользовательского ввода. Традиционный подход предполагал применение строкового значения с жестко закодированным именем (листинг 4.42).

Листинг 4.42. Использование жестко закодированного имени в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {
    public ViewResult Index() {
      var products = new [] {
        new { Name = "Kayak", Price = 275M },
        new { Name = "Lifejacket", Price = 48.95M },
        new { Name = "Soccer ball", Price = 19.50M },
        new { Name = "Corner flag", Price = 34.95M }
      };
      return View(products.Select(p => $"Name: {p.Name}, Price: {p.Price}"));
    }
  }
}
```

Вызов метода `Select()` из LINQ генерирует последовательность строк, каждая из которых содержит жестко закодированную ссылку на свойства `Name` и `Price`.

Запуск приложения дает следующий вывод в окне браузера:

```
Name: Kayak, Price: 275
Name: Lifejacket, Price: 48.95
Name: Soccer ball, Price: 19.50
Name: Corner flag, Price: 34.95
```

Проблема такого подхода в том, что он предрасположен к ошибкам, которые обусловлены либо неправильно набранным именем, либо некорректно обновленным именем в строке после рефакторинга. Результат может вводить в заблуждение, что особенно проблематично для сообщений, отображаемых пользователю. Язык C# поддерживает выражение `nameof`, благодаря которому ответственность за формирование строки имени возлагается на компилятор (листинг 4.43).

Листинг 4.43. Использование выражений `nameof` в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
namespace LanguageFeatures.Controllers {
  public class HomeController : Controller {
    public ViewResult Index() {
      var products = new [] {
        new { Name = "Kayak", Price = 275M },
        new { Name = "Lifejacket", Price = 48.95M },
        new { Name = "Soccer ball", Price = 19.50M },
        new { Name = "Corner flag", Price = 34.95M }
      };
      return View(products.Select(p =>
        $"{nameof(p.Name)}: {p.Name}, {nameof(p.Price)}: {p.Price}"));
    }
  }
}
```

Компилятор обрабатывает ссылку вроде `p.Name` таким образом, что в строку включается только последняя часть, порождая тот же самый вывод, как и в предыдущем примере. Среда Visual Studio располагает поддержкой средства IntelliSense для выражений `nameof`, поэтому вы будете снабжены подсказками при выборе ссылок, а выражения корректно обновятся в случае рефакторинга кода. Поскольку за работу с `nameof` отвечает компилятор, применение недопустимой ссылки приводит к ошибке на этапе компиляции, что не позволит некорректным или устаревшим ссылкам ускользнуть от глаз.

Резюме

В главе был дан обзор основных языковых средств C#, которые должен знать результативный программист приложений MVC. Язык C# достаточно гибкий для того, чтобы обычно существовало несколько способов решения любой задачи, но есть средства, с которыми вы будете чаще всего встречаться во время разработки веб-приложений и видеть повсюду в примерах, рассматриваемых в данной книге. В следующей главе будет представлен механизм визуализации Razor и приведены объяснения, как его использовать для генерации динамического содержимого в веб-приложениях MVC.

ГЛАВА 5

Работа с Razor

В приложении ASP.NET Core MVC для выпуска содержимого, отправляемого клиентам, используется компонент, который называется *механизмом визуализации*. Стандартным механизмом визуализации является Razor, и он обрабатывает аннотированные HTML-файлы, производя поиск инструкций, которые вставляют динамическое содержимое в вывод, отправляемый браузеру.

В настоящей главе дается краткое введение в синтаксис Razor, так что вы сможете опознавать выражения Razor, когда столкнетесь с ними. Мы не собираемся превращать главу в исчерпывающий справочник по Razor; считайте ее в большей степени ускоренным курсом по синтаксису. Особенности Razor будут раскрываться в последующих главах книги при рассмотрении других средств MVC. В табл. 5.1 приведена сводка, позволяющая поместить Razor в контекст.

Таблица 5.1. Помещение Razor в контекст

Вопрос	Ответ
Что это такое?	Razor — это механизм визуализации, отвечающий за встраивание данных в документы HTML
Чем он полезен?	Возможность динамической генерации содержимого является неотъемлемой частью процесса написания веб-приложения. Механизм визуализации Razor предоставляет средства, которые упрощают работу с остальной инфраструктурой ASP.NET Core MVC с применением операторов C#
Как он используется?	Выражения Razor добавляются к статической HTML-разметке в файлах представлений. Такие выражения оцениваются для генерации ответов на клиентские запросы
Существуют ли какие-то скрытые ловушки или ограничения?	Выражения Razor могут содержать почти любые операторы C#. При решении, должна логика принадлежать представлению или контроллеру, иногда возникают затруднения, и это способно разрушить принцип разделения обязанностей, который является центральным в паттерне MVC
Существуют ли альтернативы?	В главе 21 объясняется, как написать собственный механизм визуализации. Доступны механизмы визуализации от независимых поставщиков, но они обычно полезны только в ограниченных ситуациях и для них не гарантируется долгосрочная поддержка

В табл. 5.2 приведена сводка по главе.

Таблица 5.2. Сводка по главе

Задача	Решение	Листинг
Доступ к модели представления	Используйте выражение `@model` для определения типа модели и выражения `@Model` для доступа к объекту модели	5.5, 5.14, 5.17
Использование имен типов без их уточнения с помощью пространств имен	Создайте файл импортирования представлений	5.6, 5.7
Определение содержимого, которое будет использоваться множеством представлений	Применяйте компоновку	5.8–5.10
Указание стандартной компоновки	Используйте файл запуска представления	5.11–5.13
Передача данных из контроллера представлению за пределами модели представления	Применяйте объект `ViewBag`	5.15, 5.16
Выборочная генерация содержимого	Используйте условные выражения Razor	5.18, 5.19
Генерация содержимого для каждого элемента в массиве или коллекции	Применяйте выражение `@foreach` механизма Razor	5.20, 5.21

Подготовка проекта для примера

Для демонстрации работы механизма визуализации Razor создадим новый проект по имени `Razor` типа ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core, используя шаблон Empty (Пустой), как поступали в предыдущей главе. Включим инфраструктуру MVC, внеся в класс `Startup` изменения, которые показаны в листинге 5.1.

Листинг 5.1. Включение MVC в файле `Startup.cs` из папки `Razor`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Razor {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
      }
```

```
      // app.Run(async (context) => {
      //   await context.Response.WriteAsync("Hello World!");
      // });
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Определение модели

Создадим папку `Models` и добавим в нее файл класса по имени `Product.cs` с приведенным в листинге 5.2 определением простого класса модели.

Листинг 5.2. Содержимое файла `Product.cs` из папки `Models`

```
namespace Razor.Models {

  public class Product {

    public int ProductID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public string Category { set; get; }
  }
}
```

Создание контроллера

Стандартная конфигурация, установленная в файле `Startup.cs`, следует соглашению MVC относительно отправки запросов контроллеру с именем `Home` по умолчанию. Создадим папку `Controllers` и добавим в нее файл класса `HomeController.cs`, поместив в него простое определение контроллера (листинг 5.3).

Листинг 5.3. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      Product myProduct = new Product {
        ProductID = 1,
        Name = "Kayak",
        Description = "A boat for one person",
        Category = "Watersports",
        Price = 275M
      };

      return View(myProduct);
    }
  }
}
```

В контроллере определен метод действия по имени `Index()`, в котором создается объект `Product` с заполнением его свойств. Объект `Product` передается методу `View()`, так что он используется как модель во время визуализации представления. При вызове метода `View()` имя файла представления не указывается, поэтому будет применяться стандартное представление для метода действия.

Создание представления

Чтобы определить стандартное представление для метода действия `Index()`, создадим папку `Views/Home` и добавим в нее файл типа MVC View Page (Страница представления MVC) по имени `Index.cshtml` с содержимым из листинга 5.4.

Листинг 5.4. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model Razor.Models.Product

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  Content will go here
</body>
</html>
```

В последующих разделах мы рассмотрим различные части представления Razor и продемонстрируем разнообразные вещи, которые с ним можно делать. При изучении Razor полезно помнить, что представления существуют для выражения пользователю одной или более частей модели — и это означает генерацию HTML-разметки, которая отображает данные, извлеченные из одного или множества объектов. Если не забывать, что мы всегда пытаемся строить HTML-страницу, которая может быть отправлена клиенту, тогда вся активность механизма визуализации Razor обретает смысл. В результате запуска приложения отобразится простой вывод, приведенный на рис. 5.1.

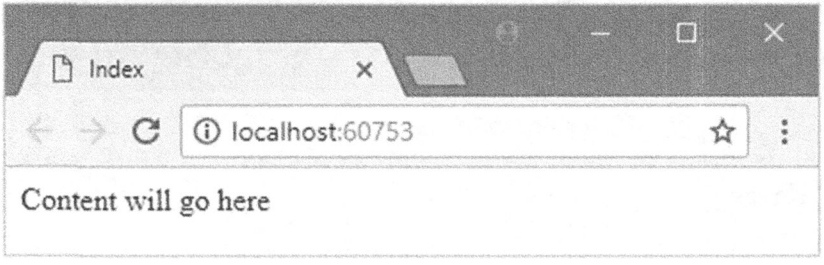


Рис. 5.1. Выполнение примера приложения

Работа с объектом модели

Давайте начнем с самой первой строки в файле представления `Index.cshtml`:

```
...
@model Razor.Models.Product
...
```

Выражения Razor начинаются с символа `@`. В данном случае выражение `@model` объявляет тип объекта модели, который будет передаваться представлению из метода действия. В итоге появляется возможность ссылки на методы, поля и свойства объекта модели представления посредством `@Model`, как показано в листинге 5.5, где отображается простое дополнение к представлению `Index`.

Листинг 5.5. Ссылка на свойство объекта модели представления в файле `Index.cshtml` из папки `Views/Home`

```
@model Razor.Models.Product

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  @Model.Name
</body>
</html>
```

На заметку! Обратите внимание, что тип объекта модели представления объявляется с использованием `@model` (со строчной буквой `m`), а доступ к свойству `Name` производится с применением `@Model` (с прописной буквой `M`). Поначалу такая особенность может немного запутывать, но со временем она станет вполне привычной.

После запуска приложения отобразится вывод, представленный на рис. 5.2.

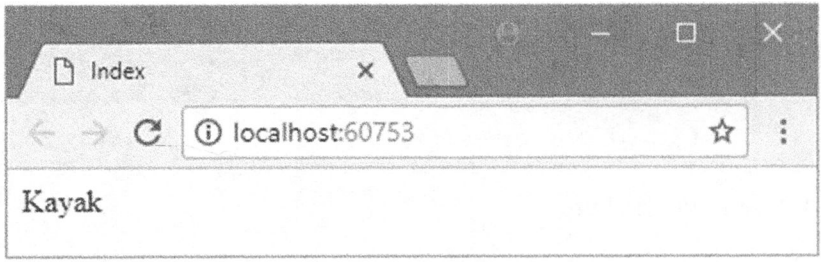


Рис. 5.2. Результат чтения значения свойства внутри представления

Представление, которое использует выражение `@model` для указания типа, называется *строго типизированным представлением*. Среда Visual Studio способна применять выражение `@model` для открытия окна со списком предполагаемых имен членов в случае ввода `@Model` с последующей точкой (рис. 5.3).

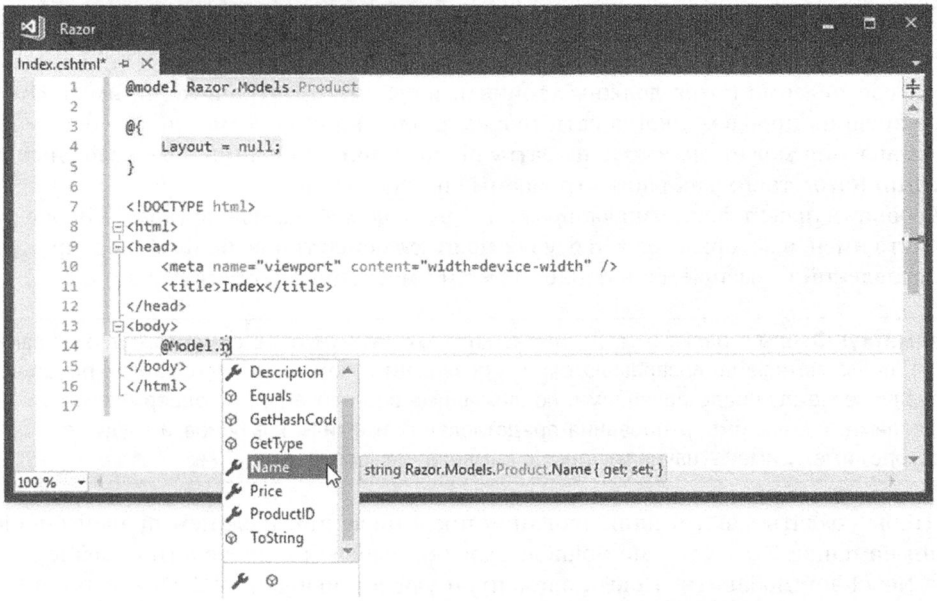


Рис. 5.3. Среда Visual Studio предлагает список предполагаемых имен членов на основе выражения `@Model`

Список предполагаемых имен членов, отображаемый Visual Studio, помогает избегать ошибок в представлениях Razor. При желании такие предположения можно игнорировать, и тогда Visual Studio будет подсвечивать проблемные имена членов, чтобы можно было внести исправления, как поступает с обычными файлами классов C#. Пример подсветки проблемы можно видеть на рис. 5.4, где мы пытается сослаться на свойство `@Model.NotARealProperty`. Среда Visual Studio обнаруживает, что класс `Product`, указанный в качестве типа модели, не содержит данное свойство, и подсвечивает ошибку в редакторе.

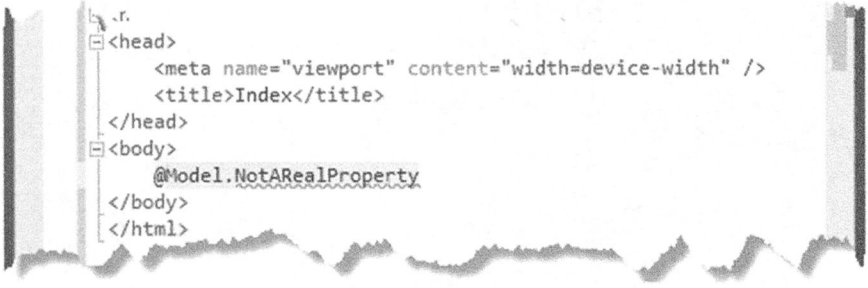


Рис. 5.4. Среда Visual Studio сообщает о проблеме с выражением `@Model`

Использование файла импортирования представлений

При определении объекта модели в начале файла `Index.cshtml` необходимо было включать пространство имен, которое содержит класс модели, например:

```
...
@model Razor.Models.Product
...
```

По умолчанию все типы, на которые производится ссылка в строго типизированном представлении Razor, должны уточняться своими пространствами имен. Это не особо крупная проблема, когда есть только ссылка на объект модели, но понимание представления может значительно затрудниться при написании более сложных выражений Razor, таких как рассматриваемые позже в главе.

Добавив в проект файл *импортирования представлений*, можно указать набор пространств имен, в которых должен осуществляться поиск типов. Файл импортирования представлений размещается в папке `Views` и имеет имя `_ViewImports.cshtml`.

На заметку! Файлы в папке `Views`, имена которых начинаются с символа подчеркивания (_), пользователю не возвращаются, что позволяет с помощью имен файлов проводить различие между представлениями, подлежащими визуализации, и поддерживающими их файлами. Файлы импортирования представлений и файлы компоновки (будут описаны вскоре) имеют имена, начинающиеся с символа подчеркивания.

Чтобы создать файл импортирования представлений, щелкнем правой кнопкой мыши на папке `Views` в окне Solution Explorer, выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажем шаблон MVC View Imports Page (Страница импортирования представлений MVC) из категории ASP.NET Core⇨Web (рис. 5.5).

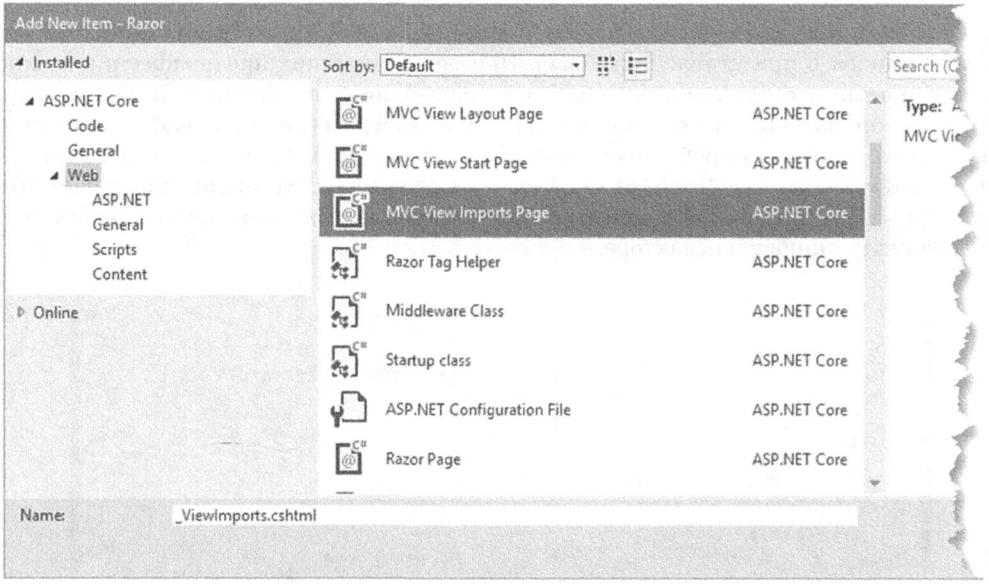


Рис. 5.5. Создание файла импортирования представлений

Среда Visual Studio автоматически назначит файлу имя `_ViewImports.cshtml`, а щелчок на кнопке Add (Добавить) приведет к созданию файла, который будет пустым. Поместим в файл выражение, показанное в листинге 5.6.

Листинг 5.6. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@using Razor.Models
```

Пространства имен, где должен проводиться поиск классов, применяемых в представлениях Razor, указываются с использованием выражения `@using`, за которым следует пространство имен. В листинге 5.6 добавлена запись для пространства имен `Razor.Models`, содержащего класс модели в примере приложения.

Теперь, когда пространство имен `Razor.Models` включено в файл импортирования представлений, можно удалить пространство имен из файла `Index.cshtml` (листинг 5.7).

Листинг 5.7. Пропуск пространства имен с классом модели в файле `Index.cshtml` из папки `Views/Home`

```
@model Product

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  @Model.Name
</body>
</html>
```

Совет. Выражение `@using` можно также добавлять в отдельные файлы представлений, что позволит применять внутри них типы без указания пространства имен.

Работа с компоновками

Ниже приведено еще одно важное выражение Razor из файла представления `Index.cshtml`:

```
...
@{
  Layout = null;
}
...
```

Это пример *блока кода* Razor, который позволяет включать в представление операторы C#. Блок кода открывается посредством `@{` и закрывается с помощью `}`, а содержащиеся в нем операторы оцениваются при визуализации представления.

Показанный выше блок кода устанавливает значение свойства `Layout` в `null`. Представления Razor компилируются в классы C# внутри приложения MVC, а в базовом классе, который они используют, определено свойство `Layout`. В главе 21 объясняется, как все это работает, а пока достаточно знать, что результатом установки свойства `Layout` в `null` является сообщение инфраструктуре MVC о том, что наше представление является самодостаточным и будет визуализировать все свое содержимое, которое необходимо возвратить клиенту.

Самодостаточные представления хороши для простых примеров приложений, но реальный проект может включать десятки представлений, и некоторые из них будут иметь общее содержимое. Дублированное общее содержимое в представлениях становится трудным в управлении, особенно если нужно внести изменение и отследить все представления, подлежащие модификации.

Более эффективный подход предусматривает использование компоновки Razor, являющейся шаблоном, который хранит общее содержимое и может быть применен к одному или большему числу представлений. Когда вы вносите изменение в компоновку, оно автоматически воздействует на все представления, которые ее используют.

Создание компоновки

Компоновки обычно совместно используются представлениями, применяемыми множеством контроллеров, и хранятся в папке по имени `Views/Shared`, которая входит в перечень местоположений, просматриваемых Razor в попытках найти файл. Чтобы создать компоновку, создадим папку `Views/Shared`, щелкнем на ней правой кнопкой мыши и выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент). Укажем шаблон MVC View Layout Page (Страница компоновки представлений MVC) в категории ASP.NET Core⇨Web и введем `_BasicLayout.cshtml` в качестве имени файла (рис. 5.6). Щелкнем на кнопке Add (Добавить) для создания файла. (Подобно файлам импортирования представлений имена файлов компоновок начинаются с символа подчеркивания.)

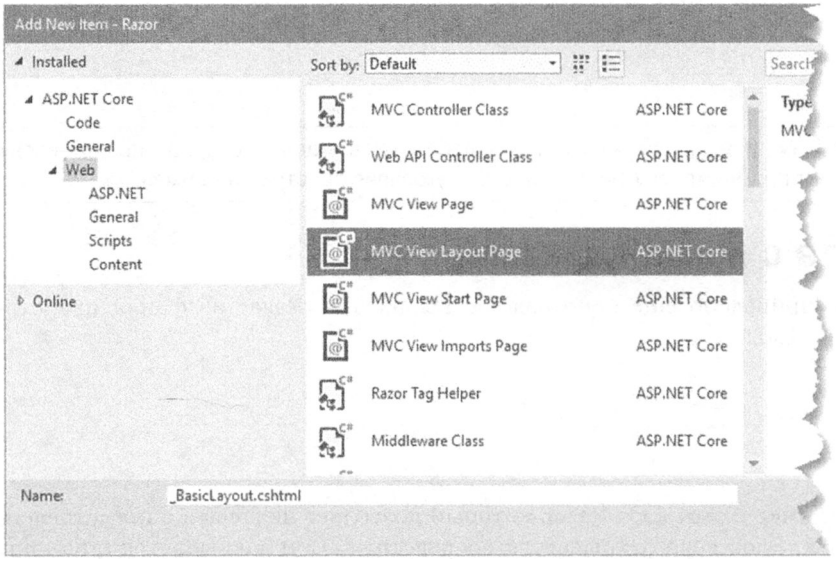


Рис. 5.6. Создание компоновки

В листинге 5.8 показано начальное содержимое файла _BasicLayout.cshtml, добавленное средой Visual Studio при создании файла.

Листинг 5.8. Начальное содержимое файла _BasicLayout.cshtml из папки Views/Shared

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

Компоновки — это специализированная форма представлений; в листинге 5.8 присутствуют два выражения @. Вызов метода @RenderBody() вставляет в разметку компоновки содержимое представления, указанное с помощью метода действия:

```
...
<div>
  @RenderBody()
</div>
...
```

Второе выражение Razor в компоновке обращается к свойству по имени ViewBag.Title для установки содержимого элемента title:

```
...
<title>@ViewBag.Title</title>
...
```

Объект ViewBag — удобное средство, которое позволяет передавать значения данных внутри приложения (в рассматриваемом случае между представлением и его компоновкой). Вы увидите, как он работает, когда компоновка будет применена к представлению.

Элементы HTML в компоновке будут применены к любому представлению, которое использует компоновку, предоставляя шаблон для определения общего содержимого. В листинге 5.9 к компоновке добавлена простая разметка, чтобы ее эффект как шаблона был очевидным.

Листинг 5.9. Добавление содержимого в файл _BasicLayout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <style>
```

```
    #mainDiv {
      padding: 20px;
      border: solid medium black;
      font-size: 20pt
    }
  </style>
</head>
<body>
  <h1>Product Information</h1>
  <div id="mainDiv">
    @RenderBody()
  </div>
</body>
</html>
```

Здесь был добавлен элемент заголовка, а также стиль CSS для стилизации содержимого элемента `div`, в котором находится выражение `@RenderBody()`, просто ради прояснения того, какое содержимое поступает из компоновки, а какое — из представления.

Применение компоновки

Чтобы применить компоновку к представлению, понадобится установить значение свойства `Layout` и удалить HTML-разметку, которая теперь будет предоставляться компоновкой, такую как элементы `html`, `head` и `body` (листинг 5.10).

Листинг 5.10. Применение компоновки в файле `Index.cshtml` из папки `Views/Home`

```
@model Product

@{
  Layout = "_BasicLayout";
  ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

Свойство `Layout` указывает имя файла компоновки, который будет использоваться для представления, без расширения `.cshtml`. Механизм Razor будет искать указанный файл компоновки в папках `/Views/Home` и `Views/Shared`.

Кроме того, установлено свойство `ViewBag.Title`, которое будет применяться компоновкой для определения содержимого элемента `title` при визуализации представления.

Трансформация представления значительна даже для такого простого приложения. Компоновка располагает всей структурой, требуемой для любого HTML-ответа, оставляя представлению только заботу о динамическом содержимом, которое отображает данные пользователю. Когда инфраструктура MVC обрабатывает файл `Index.cshtml`, она применяет компоновку для создания унифицированного HTML-ответа (рис. 5.7).

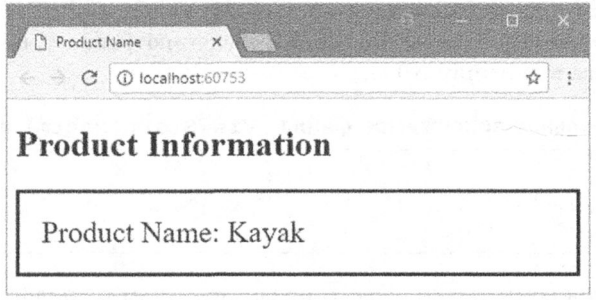


Рис. 5.7. Результат применения компоновки к представлению

Использование файла запуска представления

Осталась еще одна мелкая шероховатость, которую необходимо устранить — мы должны указывать файл компоновки для применения в каждом представлении. Следовательно, если файл компоновки переименовывается, тогда придется отыскать все ссылающиеся на него представления и внести необходимое изменение, что будет процессом, чреватым ошибками, и противоречит лейтмотиву, который пронизывает всю разработку приложений MVC — легкости сопровождения.

Решить упомянутую проблему можно с использованием *файла запуска представления*. При визуализации представления инфраструктура MVC ищет файл по имени `_ViewStart.cshtml`. Содержимое этого файла будет трактоваться так, как если бы оно присутствовало внутри самого файла представления, и данное средство можно применять для автоматической установки свойства `Layout`.

Чтобы создать файл запуска представления, щелкнем правой кнопкой мыши на папке `Views`, выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажем шаблон MVC View Start Page (Файл запуска представления MVC) в категории ASP.NET Core⇨Web (рис. 5.8).

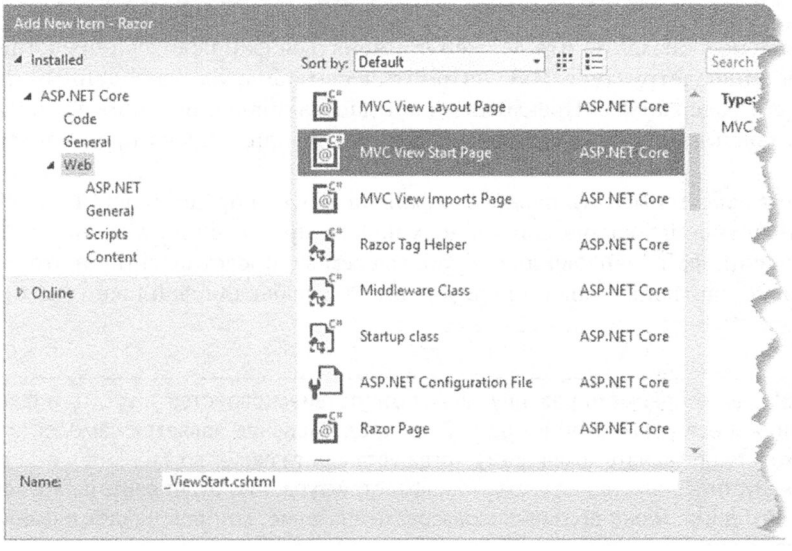


Рис. 5.8. Создание файла запуска представления

Среда Visual Studio автоматически назначит файлу имя _ViewStart.cshtml; щелчок на кнопке Add (Добавить) приведет к созданию файла с начальным содержимым, приведенным в листинге 5.11.

Листинг 5.11. Начальное содержимое файла _ViewStart.cshtml из папки Views

```
@{
  Layout = "_Layout";
}
```

Для применения компоновки ко всем представлениям в приложении понадобится изменить значение, присваиваемое свойству Layout (листинг 5.12).

Листинг 5.12. Применение стандартного представления в файле _ViewStart.cshtml из папки Views

```
@{
  Layout = "_BasicLayout";
}
```

Поскольку файл запуска представления содержит значение для свойства Layout, можно удалить соответствующее выражение из файла Index.cshtml (листинг 5.13).

Листинг 5.13. Применение файла запуска представления в файле Index.cshtml из папки Views/Home

```
@model Product

@{
  ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

Специально указывать, что должен применяться файл запуска представления, не требуется. Инфраструктура MVC автоматически обнаруживает данный файл и использует его содержимое. Преимущество отдается значениям, определяемым в файле представления, что делает переопределение файла запуска представления очень простым.

Можно также применять несколько файлов запуска представлений, чтобы установить стандартное поведение для разных частей приложения. Механизм Razor ищет самый близкий файл запуска для обрабатываемого представления, а это значит, что стандартные настройки можно переопределять, добавляя файл запуска представления в папку Views/Home или Views/Shared.

Внимание! Важно понимать разницу между отсутствием свойства Layout в файле представления и его установкой в null. Если представление является самодостаточным, и вы не хотите применять компоновку, тогда установите свойство Layout в null. Если же просто опустить свойство Layout, то инфраструктура MVC будет считать, что компоновка вам необходима, и она должна использовать значение, которое найдет в файле запуска представления.

Использование выражений Razor

Теперь, когда вы ознакомились с основами представлений и компоновок, давайте переключимся на другие виды выражений, поддерживаемые Razor, и посмотрим, как их применять для создания содержимого представлений. В хорошем приложении MVC имеется четкое разделение между ролями метода действия и представления. Роли просты и кратко описаны в табл. 5.3.

Таблица 5.3. Роли, исполняемые методом действия и представлением

Компонент	Что делает	Чего не делает
Метод действия	Передает представлению объект модели представления	Не передает представлению сформатированные данные
Представление	Использует объект модели представления для отображения содержимого пользователю	Не изменяет ни одного аспекта в объекте модели представления

Далее в книге мы будем неоднократно возвращаться к данной теме. Чтобы извлечь максимум из MVC, необходимо соблюдать и обеспечивать разделение между разными частями приложения. Как будет показано, механизм Razor позволяет делать очень многое, включая применение операторов C#, но вы не должны использовать Razor для выполнения бизнес-логики или манипулирования объектами моделей предметной области. В листинге 5.14 демонстрируется добавление нового выражения в представление `Index`.

Листинг 5.14. Добавление выражения в файле `Index.cshtml` из папки `Views/Home`

```
@model Product
@{
  ViewBag.Title = "Product Name";
}
<p>Product Name: @Model.Name</p>
<p>Product Price: @($"{Model.Price:C2}")</p>
```

Значение свойства `Price` можно было бы сформатировать в методе действия и передать его представлению. Такой подход работает, но разрушает преимущество шаблона MVC и сокращает возможность реагировать на изменения в будущем. Как уже упоминалось, позже мы еще возвратимся к этой теме, но вы должны запомнить, что инфраструктура ASP.NET Core MVC не принуждает надлежаще применять паттерн MVC и нужно учитывать влияние решений, принимаемых во время проектирования и написания кода.

Обработка или форматирование данных

Важно отличать *обработку* данных от их *форматирования*. Представления *форматируют* данные, поэтому в предыдущем разделе представлению передается объект `Product`, а не отображаемая строка с форматированными свойствами объекта. За обработку данных, включая выбор объектов данных для отображения, отвечает контроллер, который будет обращаться к модели с целью получения и модификации требующихся данных. Иногда трудно понять, где проходит черта между обработкой и форматированием, но в качестве эмпирического правила рекомендуется занять осторожную позицию и выносить из представления в контроллер все кроме простейших выражений.

Вставка значений данных

Самое простое, что можно делать с помощью выражения Razor — вставлять значения данных в разметку. Наиболее распространенный способ предусматривает использование выражения `@Model`. Представление `Index` уже содержит примеры применения такого подхода вроде показанного ниже:

```
...
<p>Product Name: @Model.Name</p>
...
```

Вставлять значения можно также с использованием объекта `ViewBag`, который применялся для установки содержимого элемента `title` в компоновке. Объект `ViewBag` можно использовать для передачи данных из контроллера в представление, дополняющее модель (листинг 5.15).

Листинг 5.15. Применение `ViewBag` в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
  public class HomeController : Controller {

    public ViewResult Index() {
      Product myProduct = new Product {
        ProductID = 1,
        Name = "Kayak",
        Description = "A boat for one person",
        Category = "Watersports",
        Price = 275M
      };

      ViewBag.StockLevel = 2;

      return View(myProduct);
    }
  }
}
```

Свойство `ViewBag` возвращает динамический объект, который может использоваться для определения произвольных свойств. В листинге 5.15 было определено и установлено в 2 свойство по имени `StockLevel`. Так как свойство `ViewBag` является динамическим, объявлять имена свойств заранее необязательно, но это означает, что для свойств `ViewBag` среда Visual Studio не в состоянии предоставлять предполагаемые варианты автозавершения.

Знание того, когда применять `ViewBag`, а когда расширять модель — вопрос опыта и сложившихся предпочтений. Мой персональный стиль заключается в том, чтобы использовать объект `ViewBag` только для предоставления визуальных подсказок о способе визуализации данных и не применять его для значений данных, которые отображаются пользователю. Но это просто то, что подходит лично мне. Если вы хотите использовать `ViewBag` для данных, отображаемых пользователю, тогда обращайтесь к значениям с помощью выражения `@ViewBag` (листинг 5.16).

Листинг 5.16. Отображение значения ViewBag в файле Index.cshtml из папки Views/Home

```
@model Product

@{
  ViewBag.Title = "Product Name";
}

<p>Product Name: @Model.Name</p>
<p>Product Price: @($"{Model.Price:C2}")</p>
<p>Stock Level: @ViewBag.StockLevel</p>
```

Результат можно видеть на рис. 5.9.

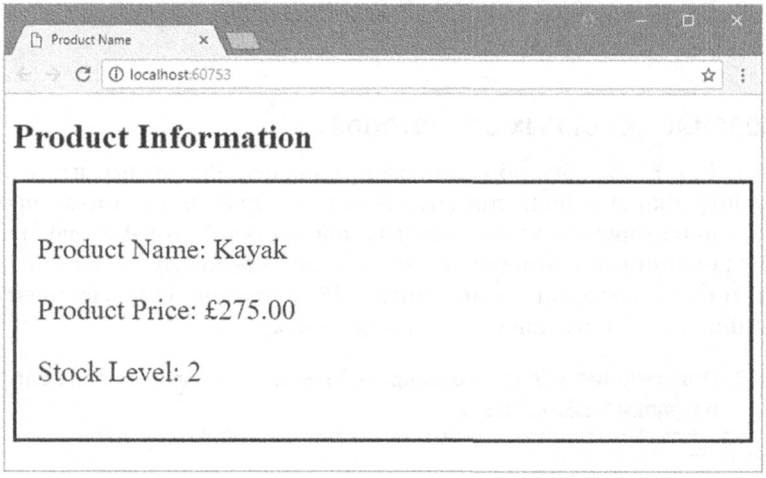


Рис. 5.9. Применение выражений Razor для вставки значений данных

Установка значений атрибутов

Во всех рассмотренных до сих пор примерах устанавливалось содержимое элементов, но выражения Razor можно использовать также для установки значений *атрибутов* элемента. В листинге 5.17 демонстрируется применение выражений `@Model` и `@ViewBag` для установки содержимого атрибутов в элементах представления `Index`.

Листинг 5.17. Установка значений атрибутов в файле Index.cshtml из папки Views/Home

```
@model Product

@{
  ViewBag.Title = "Product Name";
}

<div data-productid="@Model.ProductID" data-stocklevel="@ViewBag.StockLevel">
  <p>Product Name: @Model.Name</p>
  <p>Product Price: @($"{Model.Price:C2}")</p>
  <p>Stock Level: @ViewBag.StockLevel</p>
</div>
```

Выражения Razor применяются для установки значений некоторых атрибутов данных в элементе `div`.

Совет. Атрибуты данных, которые представляют собой атрибуты с именами, начинающимися на `data-`, в течение многих лет были неформальным способом создания специальных атрибутов и затем сделаны частью формального стандарта HTML5. Чаще всего они используются так, что код JavaScript может находить специфические элементы, или так, что стили CSS могут применяться более ограниченным образом.

Если запустить пример приложения и просмотреть HTML-разметку, отправленную браузеру, то легко увидеть, как механизм Razor установил значения атрибутов:

```
<div data-productid="1" data-stocklevel="2">
  <p>Product Name: Kayak</p>
  <p>Product Price: $275.00</p>
  <p>Stock Level: 120</p>
</div>
```

Использование условных операторов

Механизм Razor способен обрабатывать условные операторы, что означает возможность настройки вывода из представления на основе значений данных представления. Такой прием является наиболее важной частью Razor и позволяет создавать сложные и изменчивые компоновки, которые по-прежнему остаются простыми в понимании и сопровождении. В листинге 5.18 приведено обновленное содержимое представления `Index`, включающее условный оператор.

Листинг 5.18. Применение условного оператора Razor в файле `Index.cshtml` из папки `Views/Home`

```
@model Product

@{
  ViewBag.Title = "Product Name";
}
<div data-productid="@Model.ProductID" data-stocklevel="@ViewBag.StockLevel">
  <p>Product Name: @Model.Name</p>
  <p>Product Price: @($"{Model.Price:C2}")</p>
  <p>Stock Level:
    @switch (ViewBag.StockLevel) {
      case 0:
        @: Out of Stock
        break;
      case 1:
      case 2:
      case 3:
        <b>Low Stock (@ViewBag.StockLevel)</b>
        break;
      default:
        @: @ViewBag.StockLevel in Stock
        break;
    }
  </p>
</div>
```

Чтобы начать условный оператор, нужно поместить символ @ перед условным ключевым словом C# (switch в текущем примере). Блок кода завершается закрывающей фигурной скобкой (}) подобно обычному блоку кода C#.

Внутри блока кода Razor в вывод представления можно включать HTML-элементы и значения данных, просто определяя HTML-разметку и выражения Razor:

```
...
<b>Low Stock (@ViewBag.StockLevel)</b>
...
```

Помещать элементы или выражения в кавычки либо отмечать их каким-то специальным образом не требуется — механизм Razor будет интерпретировать это как вывод, подлежащий обработке. Тем не менее, если нужно вставить в представление литеральный текст, когда он не содержится в HTML-элементе, тогда о данном факте понадобится сообщить Razor, добавив к строке префикс:

```
...
@: Out of Stock
...
```

Символы @: предотвращают обработку механизмом Razor строки как оператора C#, что является стандартным поведением в отношении текста. Результат выполнения такого условного оператора показан на рис. 5.10.

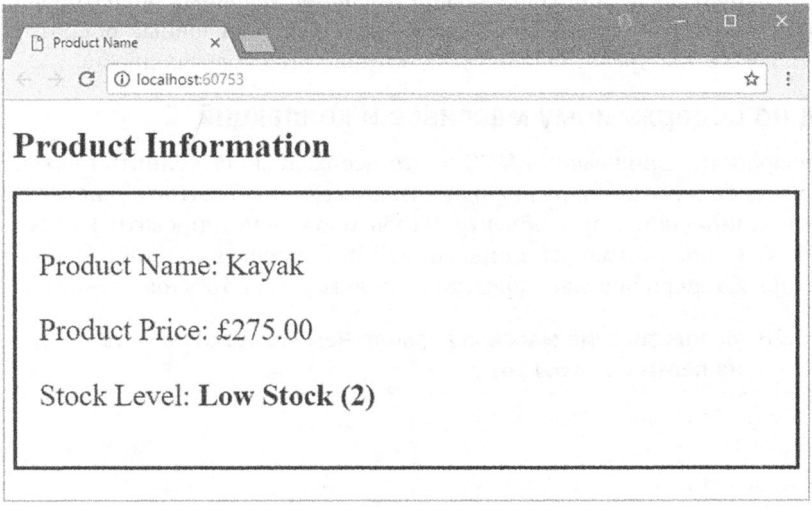


Рис. 5.10. Применение оператора switch в представлении Razor

Условные операторы играют важную роль в представлениях Razor, поскольку они позволяют варьировать содержимое на основе значений данных, которые представление получает от метода действия. В качестве дополнительной демонстрации в листинге 5.19 приведено представление Index.cshtml с добавленным оператором if — еще одним распространенным условным оператором.

Листинг 5.19. Использование оператора `if` в файле `Index.cshtml` из папки `Views/Home`

```
@model Product

@{
  ViewBag.Title = "Product Name";
}

<div data-productid="@Model.ProductID"
     data-stocklevel="@ViewBag.StockLevel">
  <p>Product Name: @Model.Name</p>
  <p>Product Price: @($"{Model.Price:C2}")</p>
  <p>Stock Level:
    @if (ViewBag.StockLevel == 0) {
      @: Out of Stock
    } else if (ViewBag.StockLevel > 0 && ViewBag.StockLevel <= 3) {
      <b>Low Stock (@ViewBag.StockLevel)</b>
    } else {
      @: @ViewBag.StockLevel in Stock
    }
  </p>
</div>
```

Условный оператор `if` выдает те же самые результаты, что и оператор `switch`, но мы просто хотели продемонстрировать применение условных операторов C# в представлениях Razor. Дополнительные сведения о работе условных операторов можно найти в главе 21, где представления рассматриваются более подробно.

Проход по содержимому массивов и коллекций

При разработке приложения MVC часто необходимо выполнять проход по содержимому массива или другой разновидности коллекции объектов с генерацией подробной информации для каждого объекта. Чтобы продемонстрировать, как это делается, в листинге 5.20 показан модифицированный метод действия `Index()` в контроллере `Home`, который теперь передает представлению массив объектов `Product`.

Листинг 5.20. Использование массива в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Razor.Models;

namespace Razor.Controllers {
  public class HomeController : Controller {

    public IActionResult Index() {
      Product[] array = {
        new Product {Name = "Kayak", Price = 275M},
        new Product {Name = "Lifejacket", Price = 48.95M},
        new Product {Name = "Soccer ball", Price = 19.50M},
        new Product {Name = "Corner flag", Price = 34.95M}
      };
      return View(array);
    }
  }
}
```

Метод действия `Index()` создает объект `Product[]`, который содержит простые значения данных, и передает его методу `View()` для визуализации с применением стандартного представления. В листинге 5.21 изменен тип модели для представления `Index` и с помощью цикла `foreach` производится проход по объектам в массиве.

Листинг 5.21. Проход по массиву в файле `Index.cshtml` из папки `Views/Home`

```
@model Product[]

@{
  ViewBag.Title = "Product Name";
}

<table>
  <thead>
    <tr><th>Name</th><th>Price</th></tr>
  </thead>
  <tbody>
    @foreach (Product p in Model) {
      <tr>
        <td>@p.Name</td>
        <td>@($"{p.Price:C2}")</td>
      </tr>
    }
  </tbody>
</table>
```

Оператор `@foreach` выполняет проход по содержимому массива объектов модели и генерирует для каждого элемента массива строку в таблице. Как видите, в цикле `foreach` создается локальная переменная по имени `p`, на свойства которой производится ссылка с использованием выражений Razor вида `@p.Name` и `@p.Price`. Результат приведен на рис. 5.11.

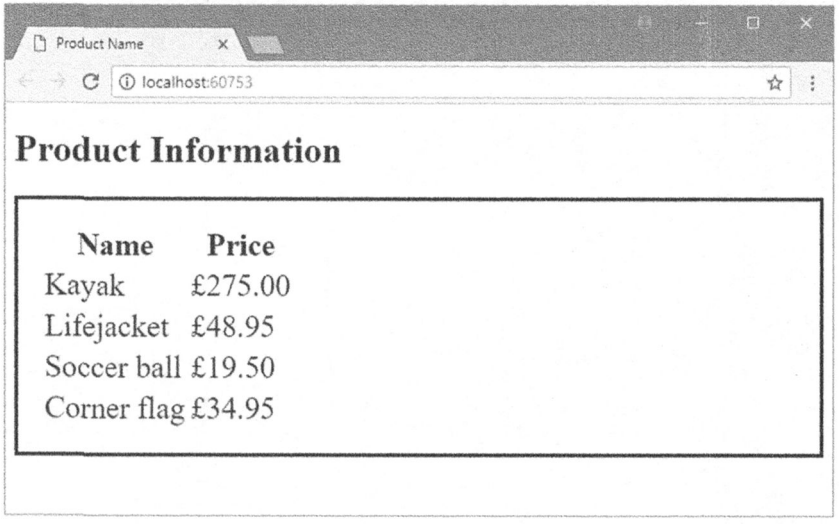


Рис. 5.11. Применение Razor для прохода по массиву

Резюме

В главе был предложен обзор механизма визуализации Razor и показано, как его использовать для генерации HTML-разметки. Мы выяснили, каким образом ссылаться на данные, передаваемые из контроллера, через объект модели представления и объект `ViewBag`, а также продемонстрировали применение выражений Razor для подстройки ответов пользователю на основе значений данных. В оставшихся главах книги вы увидите много разных примеров использования Razor, а в главе 21 найдете подробное обсуждение функционирования механизма визуализации MVC. В следующей главе будут описаны некоторые средства, предлагаемые Visual Studio для работы с проектами ASP.NET Core MVC.

ГЛАВА 6

Работа с Visual Studio

В настоящей главе рассматриваются основные средства, предоставляемые Visual Studio для разработки проектов ASP.NET Core MVC. В табл. 6.1 приведена сводка по главе.

Таблица 6.1. Сводка по главе

Задача	Решение	Листинг
Добавление пакетов к проекту	Используйте инструмент NuGet для пакетов .NET и Bower для пакетов клиентской стороны	6.6–6.8
Просмотр результатов изменения представления или класса	Применяйте модель итеративной разработки	6.9–6.11
Отображение подробных сообщений в браузере	Используйте страницы исключений разработчика	6.12
Получение детальной информации и контроля над выполнением приложения	Применяйте отладчик	6.13
Повторная загрузка одного или большего числа браузеров с применением Visual Studio	Используйте средство Browser Link (Ссылка на браузер)	6.14, 6.15
Сокращение количества HTTP-запросов и ширины полосы пропускания, требуемой для файлов JavaScript и CSS	Применяйте расширение Bundler & Minifier (Упаковщик и минификатор)	6.16–6.23

Подготовка проекта для примера

Для целей текущей главы создадим новый проект по имени `WorkingWithVisualStudio` типа ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core, используя шаблон Empty (Пустой). Включим инфраструктуру MVC с ее стандартной конфигурацией в классе `Startup` (листинг 6.1).

Листинг 6.1. Включение инфраструктуры MVC в файле `Startup.cs` из папки `WorkingWithVisualStudio`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace WorkingWithVisualStudio {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание модели

Создадим папку `Models` и добавим в нее файл класса по имени `Product.cs` с определением, показанным в листинге 6.2.

Листинг 6.2. Содержимое файла `Product.cs` из папки `Models`

```
namespace WorkingWithVisualStudio.Models {

  public class Product {
    public string Name { get; set; }
    public decimal Price { get; set; }
  }
}
```

Чтобы создать простое хранилище объектов `Product`, добавим в папку `Models` файл класса по имени `SimpleRepository.cs` и поместим в него определение, приведенное в листинге 6.3.

Листинг 6.3. Содержимое файла `SimpleRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Models {
  public class SimpleRepository {
    private static SimpleRepository sharedRepository
      = new SimpleRepository();
    private Dictionary<string, Product> products
      = new Dictionary<string, Product>();

    public static SimpleRepository SharedRepository => sharedRepository;

    public SimpleRepository() {
      var initialItems = new[] {
        new Product { Name = "Kayak", Price = 275M },
        new Product { Name = "Lifejacket", Price = 48.95M },
        new Product { Name = "Soccer ball", Price = 19.50M },
        new Product { Name = "Corner flag", Price = 34.95M }
      };
```

```
    foreach (var p in initialItems) {
      AddProduct(p);
    }
  }
  public IEnumerable<Product> Products => products.Values;
  public void AddProduct(Product p) => products.Add(p.Name, p);
 }
}
```

Класс `SimpleRepository` хранит объекты модели в памяти, т.е. любые внесенные в модель изменения утрачиваются, когда приложение останавливается или перезапускается. Для примеров, приводимых в настоящей главе, непостоянного хранилища вполне достаточно, но такой подход не может применяться во многих реальных проектах; в главе 8 рассматривается пример создания хранилища, которое запоминает объекты модели на постоянной основе, используя реляционную базу данных.

На заметку! В листинге 6.3 определено статическое свойство по имени `SharedRepository`, предоставляющее доступ к единственному объекту `SimpleRepository`, который может применяться повсюду в приложении. Это не является установившейся практикой, но я хочу продемонстрировать распространенную проблему, с которой вы столкнетесь при разработке приложений MVC; в главе 18 будет описан более эффективный способ работы с разделяемыми компонентами.

Создание контроллера и представления

Добавим в проект папку `Controllers` и поместим в нее файл класса по имени `HomeController.cs`, в котором определен контроллер, показанный в листинге 6.4.

Листинг 6.4. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;

namespace WorkingWithVisualStudio.Controllers {
 public class HomeController : Controller {
  public IActionResult Index()
   => View(SimpleRepository.SharedRepository.Products);
 }
}
```

Здесь имеется единственный метод действия `Index()`, который получает все объекты модели и передает их методу `View()` для визуализации стандартного представления. Чтобы добавить стандартное представление, создадим папку `Views/Home` и поместим в нее файл представления по имени `Index.cshtml`, содержимое которого приведено в листинге 6.5.

Листинг 6.5. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<WorkingWithVisualStudio.Models.Product>

@{ Layout = null; }

<!DOCTYPE html>
```

```
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Working with Visual Studio</title>
</head>
<body>
  <table>
    <thead>
      <tr><td>Name</td><td>Price</td></tr>
    </thead>
    <tbody>
      @foreach (var p in Model) {
        <tr>
          <td>@p.Name</td>
          <td>@p.Price</td>
        </tr>
      }
    </tbody>
  </table>
</body>
</html>
```

Представление включает таблицу, в которой с помощью Razor-цикла `foreach` создаются строки для всех объектов модели, причем каждая строка содержит ячейки для значений свойств `Name` и `Price`. После запуска примера приложения отобразятся результаты, показанные на рис. 6.1.

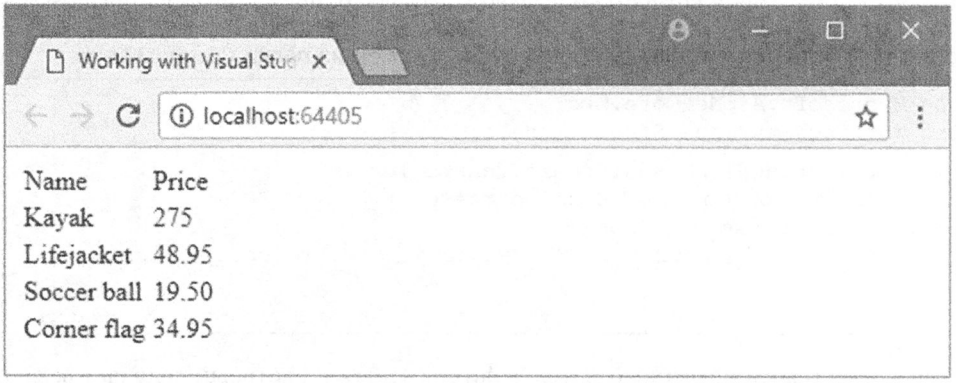


Рис. 6.1. Выполнение примера приложения

Управление программными пакетами

Проектам ASP.NET Core MVC требуются два разных вида программных пакетов. Такие типы пакетов будут описаны в последующих разделах вместе с инструментами, которые среда Visual Studio предлагает для управления ими.

Инструмент NuGet

Среда Visual Studio предоставляет графический инструмент для управления пакетами .NET, включаемыми в проект. Чтобы открыть инструмент NuGet, выберем в меню Tools⇨NuGet Package Manager (Сервис⇨Диспетчер пакетов NuGet) пункт Manage NuGet Packages for Solution (Управление пакетами NuGet для решения). Инструмент NuGet откроется и отобразит список пакетов, которые уже установлены (рис. 6.2).

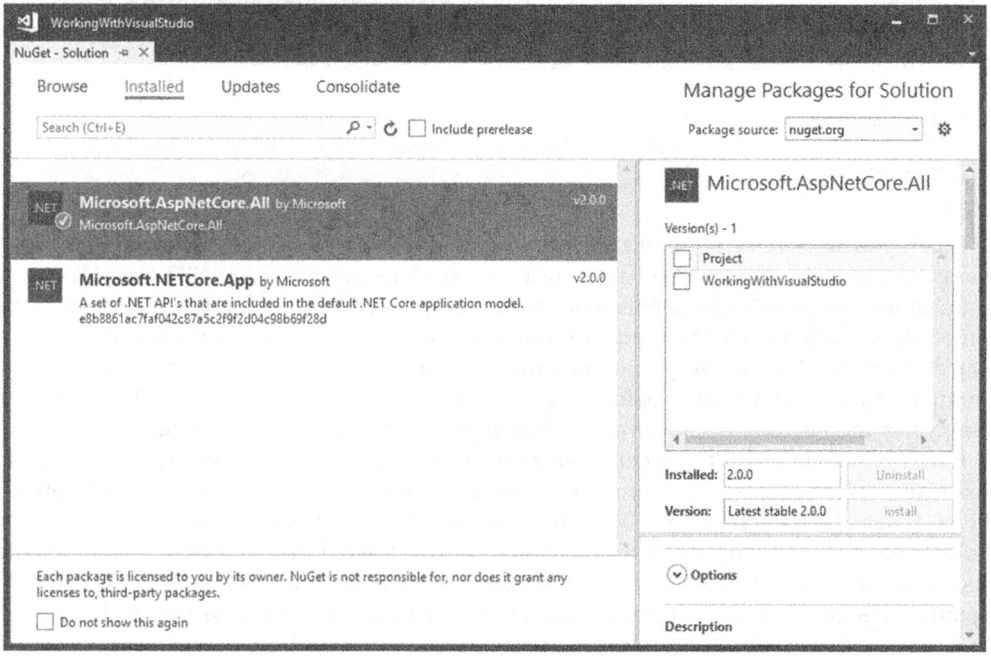


Рис. 6.2. Использование диспетчера пакетов NuGet

На вкладке Installed (Установленные) приводится сводка по пакетам, уже установленным в проекте. Вкладку Browse (Обзор) можно применять для нахождения и установки новых пакетов, а вкладку Updates (Обновления) — чтобы получить список пакетов, для которых были выпущены свежие версии.

Пакет `Microsoft.AspNetCore.All`

Если вы работали с более ранними версиями ASP.NET Core, тогда вам известно о необходимости добавления в новый проект длинного списка пакетов NuGet. В версии ASP.NET Core 2 принят другой подход, который опирается на единственный пакет по имени `Microsoft.AspNetCore.All`.

Пакет `Microsoft.AspNetCore.All` представляет собой мета-пакет, содержащий все индивидуальные пакеты NuGet, которые требуются платформой ASP.NET Core и инфраструктурой MVC, т.е. добавлять пакеты по одному не придется. При опубликовании приложения любые пакеты, которые входят в состав мета-пакета, но в приложении не используются, будут удалены, гарантируя тем самым, что лишние пакеты развертываться не будут.

Список и местоположение пакетов NuGet

Инструмент NuGet отслеживает пакеты проекта в файле `<имя_проекта>.csproj`. В текущем примере приложения это означает, что детали, касающиеся пакетов NuGet, хранятся в файле по имени `WorkingWithVisualStudio.csproj`. Среда Visual Studio не отображает файл `.csproj` в окне Solution Explorer. Чтобы отредактировать данный файл, щелкнем правой кнопкой мыши на имени проекта в окне Solution Explorer и выберем в контекстном меню пункт Edit WorkingWithVisualStudio.csproj (Редактировать `WorkingWithVisualStudio.csproj`). Среда Visual Studio откроет файл для редактирования. Файл `.csproj` имеет формат XML и содержит элемент вроде показанного ниже, который добавляет к проекту мета-пакет ASP.NET Core:

```
...
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
</ItemGroup>
...
```

Пакет указывается с помощью своего имени и номера требуемой версии. Хотя мета-пакет включает все функциональные средства, требующиеся для ASP.NET Core MVC, вам по-прежнему понадобится добавлять пакеты в проект, чтобы появилась возможность применения дополнительных средств. Пакеты можно добавлять с использованием графического интерфейса, показанного на рис. 6.2, либо с применением инструментов командной строки. Кроме того, файл `.csproj` можно редактировать и среда Visual Studio обнаружит изменения, после чего загрузит и установит добавленные пакеты.

Когда для добавления в проект какого-то пакета используется инструмент NuGet, пакет автоматически устанавливается вместе со всеми пакетами, от которых он зависит. Исследовать пакеты NuGet и их зависимости можно, раскрыв в окне Solution Explorer узел Dependencies⇨NuGet (Зависимости⇨NuGet), что приведет к отображению всех пакетов в файле `.csproj` и их зависимостей. Мета-пакет ASP.NET Core содержит большое количество зависимостей, часть которых видна на рис. 6.3.

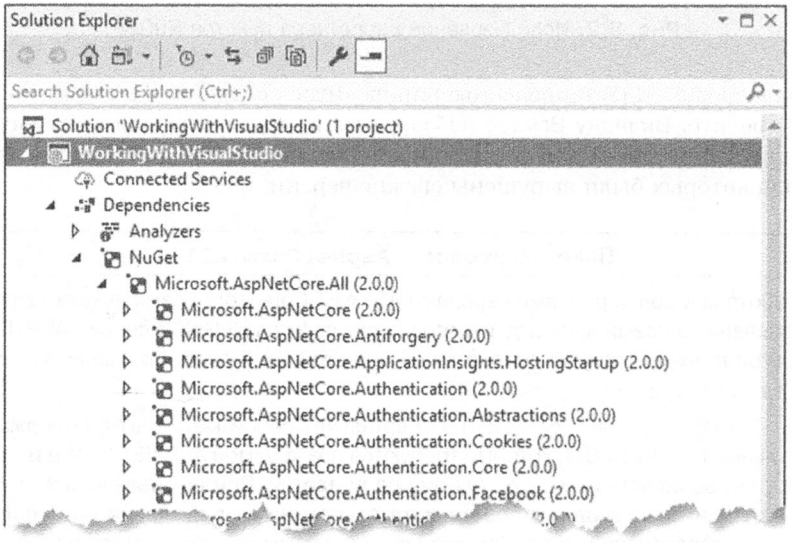


Рис. 6.3. Узел Dependencies⇨NuGet в окне Solution Explorer

Инструмент Bower

Пакет клиентской стороны включает содержимое, отправляемое клиенту, в числе которого файлы JavaScript, таблицы стилей CSS либо изображения. Для управления пакетами подобного рода также можно задействовать NuGet, но инфраструктура ASP.NET Core MVC теперь полагается на инструмент под названием Bower. Инструмент с открытым кодом Bower был разработан за пределами Microsoft и мира .NET и широко применялся при разработке веб-приложений, отличных от ASP.NET.

На заметку! Недавно инструмент Bower был объявлен устаревшим. Вы можете видеть предупреждения, рекомендующие альтернативные инструменты; тем не менее, Bower все еще активно сопровождается, а поддержка Bower интегрирована в Visual Studio. Можно ожидать, что в какой-то момент в Microsoft решат поддерживать другой инструмент для управления пакетами клиентской стороны, но пока этого не случилось, вы должны продолжить пользоваться Bower.

Список пакетов Bower

Пакеты Bower указываются в файле `bower.json`. Чтобы создать файл `bower.json`, щелкнем правой кнопкой мыши на имени проекта `WorkingWithVisualStudio` в окне Solution Explorer, выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажем шаблон элемента Bower Configuration File (Файл конфигурации Bower) из категории ASP.NET Core⇨Web⇨General (ASP.NET Core⇨Веб⇨Общие), как показано на рис. 6.4.

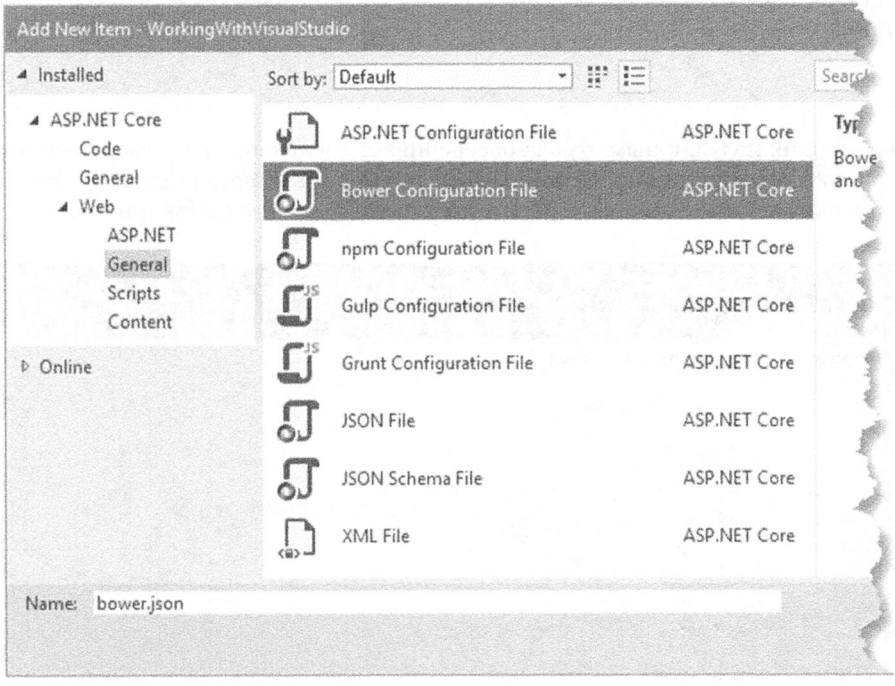


Рис. 6.4. Создание файла конфигурации Bower

Среда Visual Studio установит `bower.json` в качестве имени и после щелчка на кнопке Add (Добавить) добавит в проект файл со стандартным содержимым (листинг 6.6).

Листинг 6.6. Стандартное содержимое файла `bower.json` из папки `WorkingWithVisualStudio`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
  }
}
```

В листинге 6.7 демонстрируется добавление пакета клиентской стороны в файле `bower.json`, что делается включением записи в раздел `dependencies`.

Совет. По адресу `http://bower.io/search` находится хранилище пакетов Bower, где можно искать пакеты для добавления в свои проекты.

Листинг 6.7. Добавление пакетов в файле `bower.json` из папки `WorkingWithVisualStudio`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.7"
  }
}
```

Выделенная полужирным строка обеспечивает добавление в пример проекта CSS-пакета Bootstrap. При редактировании файла `bower.json` среда Visual Studio предложит список имен пакетов и перечислит доступные версии пакетов (рис. 6.5).

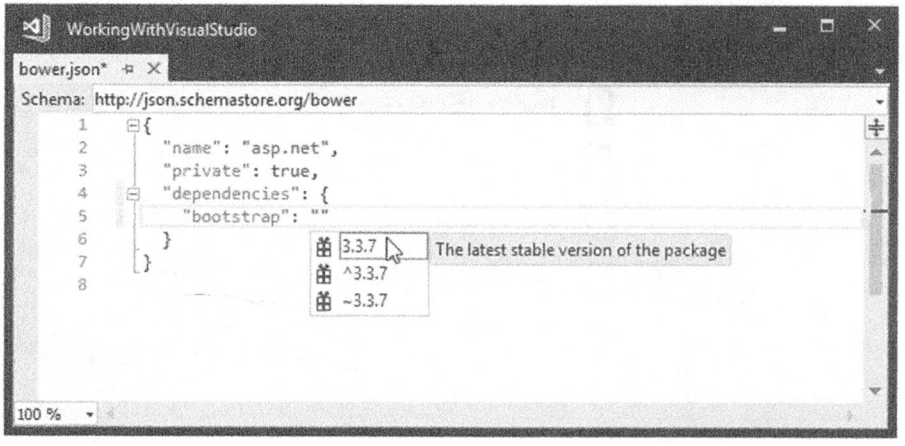


Рис. 6.5. Перечисление доступных версий пакетов клиентской стороны

На момент написания главы последней версией пакета bootstrap была 3.3.7. Однако обратите внимание, что Visual Studio предлагает три варианта: 3.3.7, ^3.3.7 и ~3.3.7. Номера версий в файле bower.json могут указываться в виде диапазона разнообразными способами, наиболее полезные из которых описаны в табл. 6.2. Самый безопасный способ указания пакета предусматривает применение явного номера версии. Это гарантирует, что вы всегда будете работать с определенной версией до тех пор, пока преднамеренно не обновите файл bower.json для запроса другой версии.

Таблица 6.2. Распространенные форматы для номеров версий в файле bower.json

Формат	Описание
3.3.7	Выражение номера версии напрямую приведет к установке пакета с точно совпадающим номером версии, например, 3.3.7
*	Использование символа звездочки позволяет инструменту Bower загружать и устанавливать любую версию пакета
>3.3.7 >=3.3.7	Снабжение номера версии префиксом > или >= позволяет инструменту Bower устанавливать любую версию пакета, которая больше либо больше или равна заданной версии
<3.3.7 <=3.3.7	Снабжение номера версии префиксом < или <= позволяет инструменту Bower устанавливать любую версию пакета, которая меньше либо меньше или равна заданной версии
~3.3.7	Снабжение номера версии префиксом в виде символа ~ позволяет инструменту Bower устанавливать версии, даже если номер уровня исправлений (последнее число в номере версии) не совпадает. Например, указание ~3.3.7 позволяет инструменту Bower устанавливать версию 3.3.8 или 3.3.9 (которая будет исправлена до версии 3.3.7), но не версию 3.4.0 (которая будет новым младшим выпуском)
^3.3.7	Снабжение номера версии префиксом в виде символа ^ позволяет инструменту Bower устанавливать версии, даже если номер младшего выпуска (второе число в номере версии) или номер исправления не совпадает. Например, указание ^3.3.0 позволит Bower устанавливать версии 3.3.1, 3.4.0 и 3.5.0, но не версию 4.0.0

Совет. Для примеров в настоящей книге файл bower.json создавался и редактировался напрямую. Редактировать файл очень просто, к тому же это способствует получению предсказуемых результатов при проработке примеров. Среда Visual Studio также предоставляет графический инструмент для управления пакетами Bower, который можно открыть, щелкнув правой кнопкой мыши на имени файла bower.json и выбрав в контекстном меню пункт Manage Bower Packages (Управление пакетами Bower).

Среда Visual Studio отслеживает изменения в файле bower.json и автоматически задействует инструмент Bower для загрузки и установки пакетов. После сохранения изменений, внесенных в файл согласно листингу 6.7, среда Visual Studio загрузит пакет Bootstrap и установит его в папку wwwroot/lib (рис. 6.6).

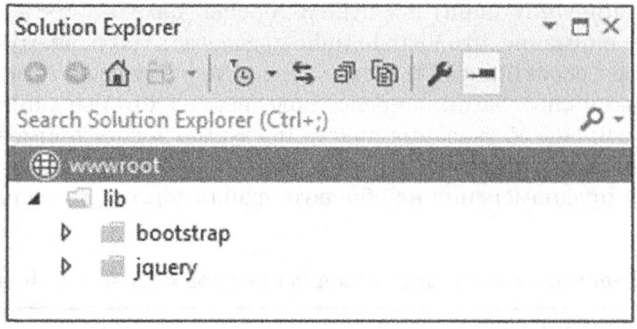


Рис. 6.6. Добавление в проект пакетов клиентской стороны

Подобно NuGet инструмент Bower управляет зависимостями пакетов, добавляемых в проект. Некоторые расширенные функциональные средства пакета Bootstrap зависят от JavaScript-библиотеки jQuery, из-за чего на рис. 6.6 видны два пакета. Для просмотра списка пакетов и их зависимостей необходимо развернуть узел Dependencies (Зависимости) в окне Solution Explorer (рис. 6.7).

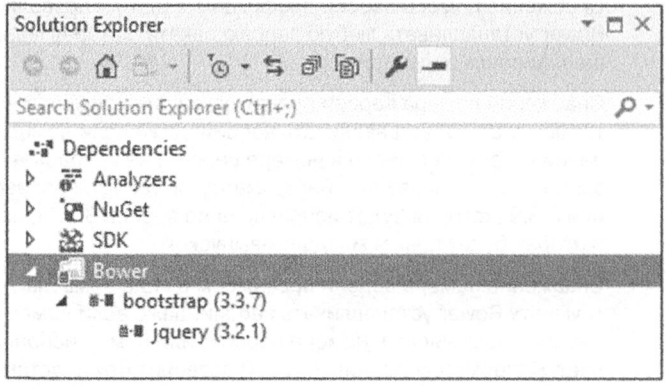


Рис. 6.7. Исследование пакетов клиентской стороны и их зависимостей

Обновление пакета Bootstrap

В оставшемся материале книге применяется предварительный выпуск инфраструктуры Bootstrap CSS. Во время написания книги команда разработчиков Bootstrap занималась построением версии Bootstrap 4 и произвела несколько ранних выпусков. Выпуски были помечены словом `alpha`, но обладали высоким качеством, достаточным для их использования в примерах данной книги. Находясь перед выбором, применять версию Bootstrap 3, которая скоро выйдет из употребления, или предварительный выпуск Bootstrap 4, я решил использовать новую версию, несмотря на то, что имена классов, применяемых для стилизации элементов HTML, в финальном выпуске вполне могут измениться. Это означает, что для получения ожидаемых результатов от примеров вы должны использовать ту же самую версию Bootstrap.

Чтобы обновить пакет Bootstrap, изменим номер версии в файле `bower.json` (листинг 6.8).

Листинг 6.8. Изменение версии пакета Bootstrap в файле `bower.json` из папки `WorkingWithVisualStudio`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

После сохранения изменений, внесенных в файл `bower.json`, среда Visual Studio загрузит новую версию пакета Bootstrap.

Итеративная разработка

Разработка веб-приложений часто может быть итеративным процессом, когда вы вносите небольшие изменения в представления или классы и запускаете приложение, чтобы протестировать результат. Инфраструктура MVC и среда Visual Studio на пару поддерживают такой итеративный подход, делая наблюдение за влиянием изменений быстрым и легким.

Внесение изменений в представления Razor

Во время разработки изменения, внесенные в представления Razor, вступают в силу, как только поступает HTTP-запрос от браузера. Чтобы продемонстрировать, каким образом это работает, запустим приложение, выбрав пункт Start Debugging (Запустить отладку) в меню Debug (Отладка), и после открытия вкладки браузера, отображающей данные, внесем в файл `Index.cshtml` изменения, приведенные в листинге 6.9.

Листинг 6.9. Внесение изменений в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<WorkingWithVisualStudio.Models.Product>
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Working with Visual Studio</title>
</head>
<body>
  <h3>Products</h3>
  <table>
    <thead>
      <tr><td>Name</td><td>Price</td></tr>
    </thead>
    <tbody>
      @foreach (var p in Model) {
        <tr>
          <td>@p.Name</td>
          <td>@($"{p.Price:C2}")</td>
        </tr>
      }
```

```
    </tbody>
  </table>
</body>
</html>
```

Сохраним изменения в представлении `Index` и перезагрузим текущую веб-страницу с помощью кнопки обновления в браузере. Изменения в представлении (добавление заголовка и форматирование свойства `Price` объекта модели как денежного значения) оказывают воздействие и отображаются в браузере (рис. 6.8).

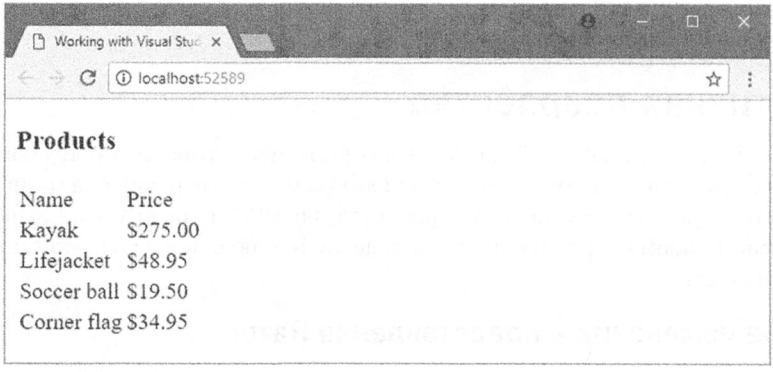


Рис. 6.8. Результаты внесения изменений в представление

Совет. Процесс подготовки к использованию представлений Razor рассматривается в главе 21.

Внесение изменений в классы C#

Для классов C#, включая контроллеры и модели, способ обработки изменений зависит от того, как запускается приложение. В последующих разделах будут показаны два доступных подхода, которые инициируются посредством разных пунктов в меню Debug и кратко описаны в табл. 6.3.

Таблица 6.3. Пункты меню Debug

Пункт меню	Описание
Start Without Debugging (Запустить без отладки)	Классы в проекте компилируются автоматически, когда поступает HTTP-запрос, делая возможной более динамичную практику разработки. Приложение запускается без отладчика, что не позволяет взять под контроль выполнение кода
Start Debugging (Запустить отладку)	При таком стиле разработки потребуется явно компилировать проект и перезапускать приложение, чтобы изменения вступили в силу. Во время выполнения к приложению присоединен отладчик, позволяя инспектировать его состояние и анализировать причины возникновения любых проблем

Автоматическая компиляция классов

В течение нормальной стадии разработки быстрый итеративный цикл позволяет видеть результаты изменений немедленно независимо от того, связаны они с добавлением нового действия или с изменением способа, которым выбираются данные модели представления. Для разработки такого вида среда Visual Studio поддерживает обнаружение изменений, как только от браузера получен HTTP-запрос, и автоматическую перекомпиляцию классов. Чтобы посмотреть, как это работает, выберем пункт Start Without Debugging (Запустить без отладки) в меню Debug (Отладка) среды Visual Studio. После отображения данных приложения в браузере внесем в контроллер `Home` изменения, показанные в листинге 6.10.

Листинг 6.10. Фильтрация данных модели в файле `HomeController.cs` из папки `Views/Home`

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;

namespace WorkingWithVisualStudio.Controllers {
  public class HomeController : Controller {

    public IActionResult Index()
      => View(SimpleRepository.SharedRepository.Products
            .Where(p => p.Price < 50));
  }
}
```

Изменения заключаются в применении LINQ для фильтрации объектов `Product`, чтобы представлению передавались только те из них, свойство `Price` которых имеет значение меньше `50`. Сохраним изменения, произведенные в файле класса контроллера, и обновим окно браузера, не останавливая или не перезапуская приложение в Visual Studio. Отправленный браузером HTTP-запрос инициирует процесс компиляции, и приложение будет запущено повторно с использованием модифицированного класса контроллера, генерируя результаты, в которых из таблицы исчез товар `Kayak` (рис. 6.9).

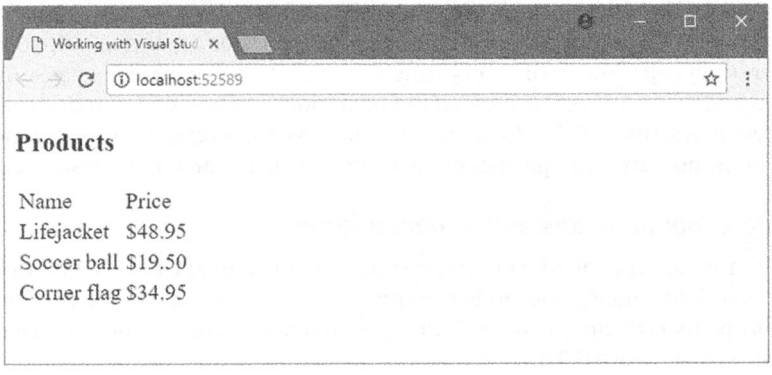


Рис. 6.9. Автоматическая компиляция классов

Средство автоматической компиляции удобно, когда все идет по плану. Его недостаток в том, что ошибки этапа компиляции и времени выполнения отображаются в браузере, а не в Visual Studio, затрудняя выяснение причины возникновения проблемы. В качестве примера в листинге 6.11 демонстрируется добавление ссылки `null` в коллекцию объектов моделей внутри хранилища.

Листинг 6.11. Добавление ссылки `null` в файле `SimpleRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Models {
  public class SimpleRepository {
   private static SimpleRepository sharedRepository = new SimpleRepository();
   private Dictionary<string, Product> products
      = new Dictionary<string, Product>();

   public static SimpleRepository SharedRepository => sharedRepository;

   public SimpleRepository() {
      var initialItems = new[] {
        new Product { Name = "Kayak", Price = 275M },
        new Product { Name = "Lifejacket", Price = 48.95M },
        new Product { Name = "Soccer ball", Price = 19.50M },
        new Product { Name = "Corner flag", Price = 34.95M }
      };
      foreach (var p in initialItems) {
        AddProduct(p);
      }
      products.Add("Error", null);
    }

    public IEnumerable<Product> Products => products.Values;

    public void AddProduct(Product p) => products.Add(p.Name, p);
  }
}
```

Проблема вроде ссылки `null` не будет видна до тех пор, пока приложение не начнет выполняться. Перезагрузка страницы в браузере приведет к тому, что класс `SimpleRepository` скомпилируется, а приложение запустится заново. Когда инфраструктура MVC создает экземпляр класса контроллера для обработки HTTP-запроса от браузера, конструктор `HomeController` создаст экземпляр класса `SimpleRepository`, который в свою очередь попытается обработать ссылку `null`, добавленную в листинге 6.11. Ссылка `null` вызовет проблему, но ее сущность не будет очевидной, потому что браузер не выводит сколько-нибудь полезное сообщение.

Включение страниц исключений разработчика

Во время процесса разработки при возникновении проблемы удобно отображать в окне браузера более полезную информацию. Это можно делать, разрешив страницы исключений разработчика, что требует изменения конфигурации в классе `Startup`, как показано в листинге 6.12.

Роль класса `Startup` подробно объясняется в главе 14, а пока достаточно знать, что вызов расширяющего метода `UseDeveloperExceptionPage()` устанавливает описательные страницы ошибок.

Листинг 6.12. Разрешение страниц исключений разработчика в файле `Startup.cs` из папки `WorkingWithVisualStudio`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace WorkingWithVisualStudio {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Если перезагрузить окно браузера, то процесс автоматической компиляции перестроит приложение и выдаст более полезное сообщение об ошибке (рис. 6.10).

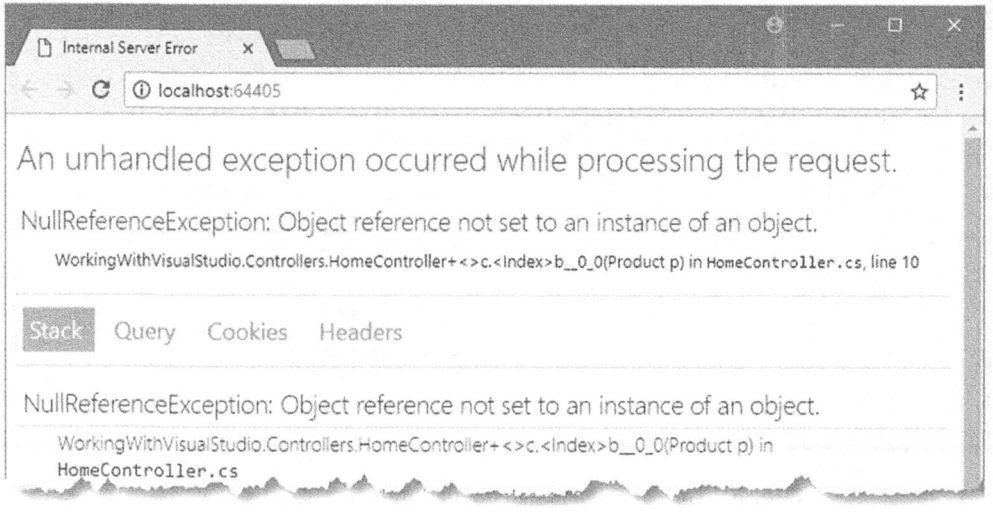


Рис. 6.10. Страница исключения разработчика

Отображаемого в браузере сообщения об ошибке может быть достаточно для выявления простых проблем, особенно из-за того, что итеративный стиль разработки означает, что причиной, вероятно, стали самые последние изменения. Но для устранения более сложных проблем — и проблем, которые не становятся очевидными немедленно — требуется отладчик Visual Studio.

Использование отладчика

Среда Visual Studio также поддерживает выполнение приложения MVC с применением отладчика, который позволяет останавливать приложение для инспектирования его состояния и пути прохождения запроса по коду. Это требует другого стиля разработки, поскольку модификации классов C# не будут применены до тех пор, пока приложение не запустится заново (хотя изменения, вносимые в представления Razor, по-прежнему вступают в силу автоматически).

Такой стиль разработки не настолько динамичен, как использование средства автоматической компиляции, но отладчик Visual Studio великолепен и может предоставлять намного более глубокое проникновение в суть способа работы приложения, нежели то, что возможно с помощью сообщений, отображаемых в окне браузера.

Чтобы запустить приложение с применением отладчика, выберем пункт Start Debugging (Запустить отладку) в меню Debug (Отладка) среды Visual Studio. Перед запуском приложения среда Visual Studio скомпилирует классы C# в проекте, но код можно также скомпилировать вручную, используя пункты меню Build (Построение).

Пример приложения по-прежнему содержит ссылку `null`, т.е. необработанное исключение `NullReferenceException`, которое генерируется в классе `SimpleRepository`, прервет приложение и передаст управление выполнением отладчику (рис. 6.11).

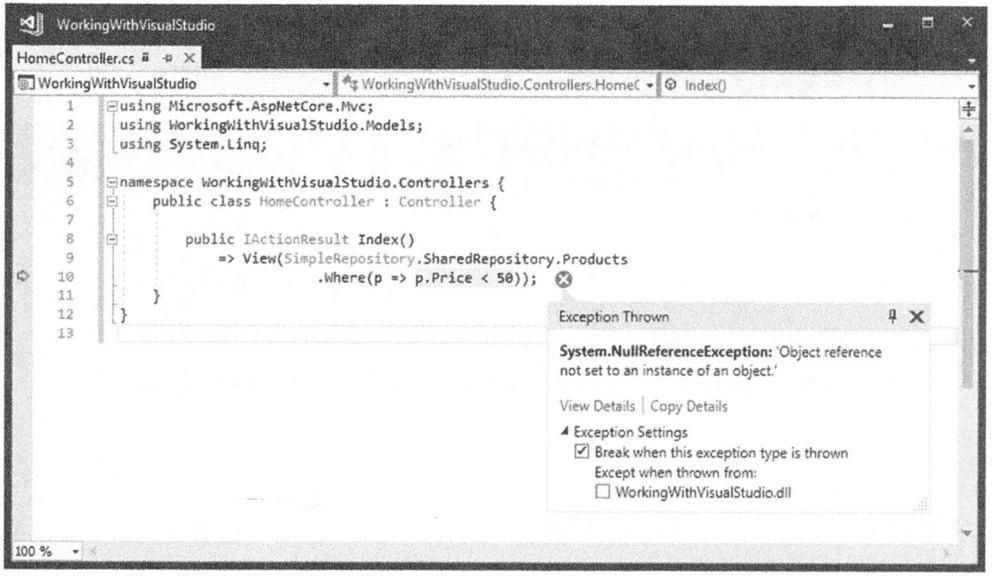


Рис. 6.11. Возникновение необработанного исключения

Совет. Если отладчик не перехватывает исключение, тогда понадобится выбрать пункт Windows⇨Exception Settings (Окна⇨Настройки исключений) в меню Debug (Отладка) среды Visual Studio и удостовериться, что в списке Common Language Runtime Exceptions (Общие исключения языка во время выполнения) отмечены флажки для всех типов исключений.

Настройка точки останова

Отладчик вовсе не показывает основную причину проблемы, а только место, где она проявилась. Подсвечиваемый Visual Studio оператор указывает на то, что проблема случилась при фильтрации объектов с применением LINQ, но для получения деталей и выявления лежащей в основе причины потребуется проделать небольшую работу.

Точка останова — это инструкция, которая сообщает отладчику о необходимости остановить выполнение приложения и передать управление программисту. Вы можете исследовать состояние приложения, посмотреть, что произошло, и при желании возобновить выполнение.

Чтобы создать точку останова, щелкнем правой кнопкой мыши на операторе кода и выберем в контекстном меню пункт Breakpoint⇨Insert Breakpoint (Точка останова⇨Вставить точку останова). В целях демонстрации поместим точку останова на метод `AddProduct()` в классе `SimpleRepository` (рис. 6.12).

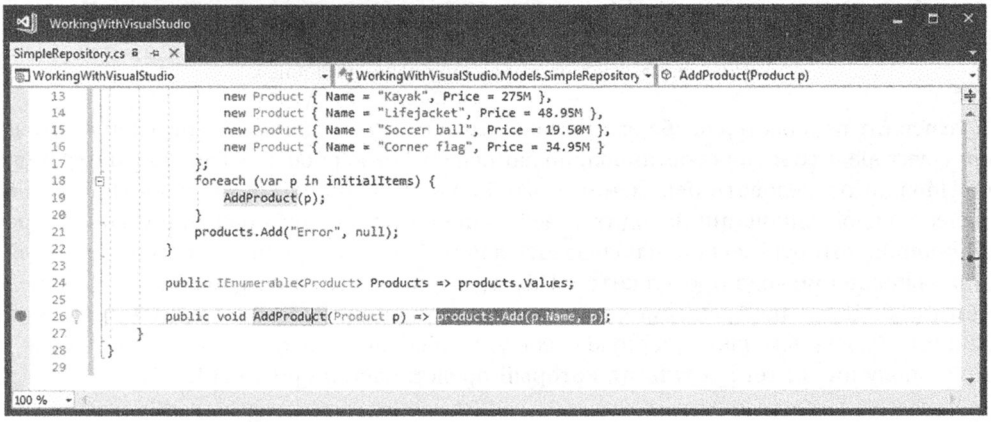


Рис. 6.12. Создание точки останова

Выберем пункт меню Debug⇨Start Debugging (Отладка⇨Запустить отладку), чтобы запустить приложение, используя отладчик, или Debug⇨Restart (Отладка⇨Перезапустить), если приложение уже выполняется. Во время обработки начального HTTP-запроса от браузера будет создан экземпляр класса `SimpleRepository`, а выполнение кода достигнет точки останова, где произойдет пауза.

Во время паузы можно применять пункты меню Debug среды Visual Studio или элементы управления в верхней части окна для управления выполнением приложения либо использовать разнообразные представления отладчика, доступные через меню Debug⇨Windows (Отладка⇨Окна), чтобы инспектировать состояние приложения.

Просмотр значений данных в редакторе кода

Наиболее распространенное применение точек останова связано с отслеживанием дефектов в коде. Прежде чем можно будет исправить дефект, потребуется выяснить, что происходит, и одним из самых полезных средств, предлагаемых Visual Studio, является возможность просмотра и наблюдения за значениями переменных прямо в редакторе кода. Если навести курсор мыши на аргумент `p` в методе `AddProduct()`, подсвеченном отладчиком, то появится всплывающее окно, которое показывает текущее значение `p` (рис. 6.13). Рассмотреть всплывающее окно может быть затруднительно, поэтому на рис. 6.13 показана также его увеличенная версия.

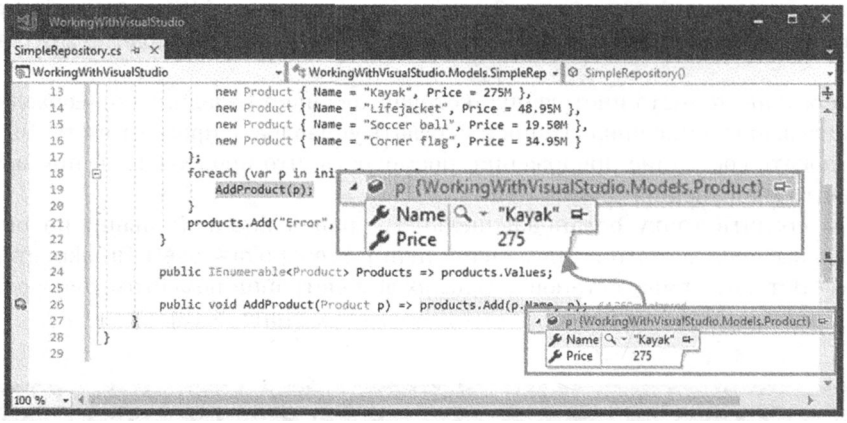


Рис. 6.13. Инспектирование значения данных

Результат не кажется особо впечатляющим, т.к. объект данных определен в том же конструкторе, что и точка останова, но данное средство работает для любой переменной. Можно исследовать переменные, чтобы увидеть значения их свойств и полей. Правее каждого значения находится небольшая кнопка с изображением канцелярской кнопки, которую можно использовать для наблюдения за значением переменной, когда выполнение кода продолжится. Наведем курсор мыши на переменную `p` и щелкнем на кнопке с изображением канцелярской кнопки, чтобы закрепить ссылку на `Product`. Развернем закрепленную ссылку, чтобы также закрепить свойства `Name` и `Price`, получив в итоге результат, который представлен на рис. 6.14.

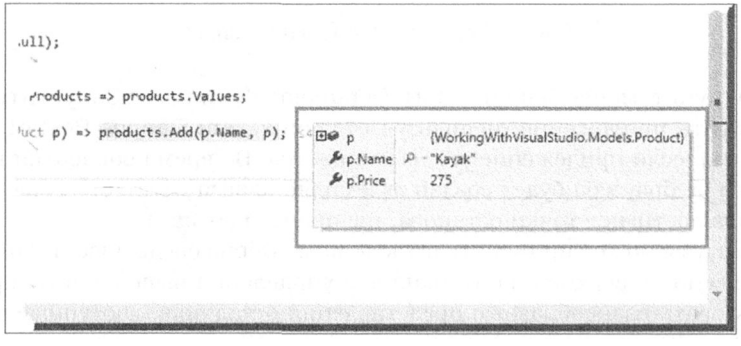


Рис. 6.14. Закрепление значений в редакторе кода

Выберем пункт Continue (Продолжить) в меню Debug (Отладка) среды Visual Studio, чтобы продолжить выполнение приложения. Поскольку приложение выполняет цикл `foreach`, когда точка останова встретится опять, то снова возникнет пауза. Закрепленные значения показывают, как изменяется объект, присвоенный переменной `p`, и его свойства (рис. 6.15).

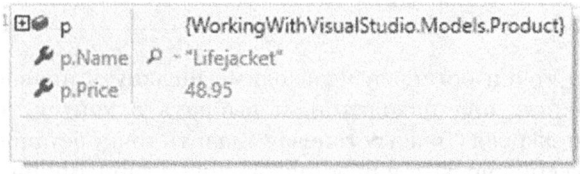


Рис. 6.15. Наблюдение за изменением состояния с применением закрепленных значений

Использование окна Locals

Связанным средством является окно Locals (Локальные), которое открывается путем выбора пункта меню Debug⇨Windows⇨Locals (Отладка⇨Окна⇨Локальные). В окне Locals отображаются значения данных похожим на закрепленные значения образом, но здесь присутствуют все локальные объекты, относящиеся к точке останова (рис. 6.16).

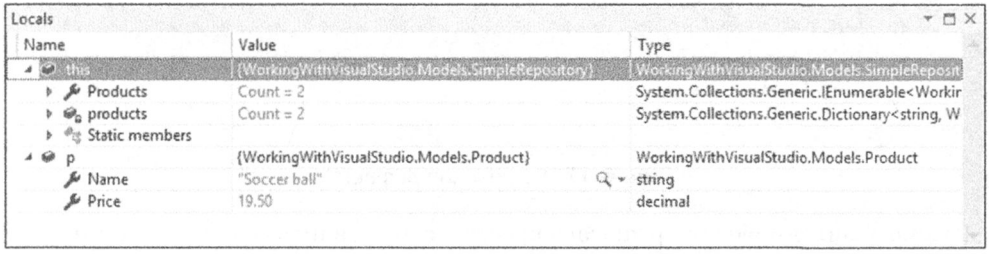


Рис. 6.16. Окно Locals

Каждый раз, когда выбирается пункт Continue, выполнение приложения возобновляется и в цикле `foreach` обрабатывается новый объект. Продолжая поступать так, можно будет увидеть ссылку `null` в окне Locals и в закрепленных значениях, отображаемых внутри редактора кода. За счет применения отладчика для управления выполнением приложения появляется возможность отслеживать его продвижение по коду и получить представление о том, что происходит.

Устранить проблему со ссылкой `null` можно было бы путем очистки коллекции объектов `Product`, но альтернативный подход предусматривает повышение надежности контроллера, как показано в листинге 6.13, где для проверки на предмет значений `null` используется `null`-условная операция (описанная в главе 4).

Листинг 6.13. Исправление проблемы со ссылкой `null` в файле `HomeController.cs` из папки `Views/Home`

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;
```

```
namespace WorkingWithVisualStudio.Controllers {
  public class HomeController : Controller {

    public IActionResult Index()
      => View(SimpleRepository.SharedRepository.Products
            .Where(p => p?.Price < 50));
  }
}
```

Для отключения точки останова необходимо щелкнуть правой кнопкой мыши на операторе кода, где она находится, и выбрать в контекстном меню пункт Breakpoint⇨Delete Breakpoint (Точка останова⇨Удалить точку останова). После повторного запуска приложения появится простая таблица с данными (рис. 6.17).



Рис. 6.17. Устранение дефекта

Рассмотренная проблема решалась просто, если сравнить ее с проблемами, которые требуют реальной охоты за ошибками, но отладчик Visual Studio действительно великолепен, и с помощью многочисленных доступных представлений приложения и управления выполнением можно легко углубляться в детали с целью выяснения, что пошло не так.

Использование средства Browser Link

Средство Browser Link (Ссылка на браузер) позволяет упростить процесс разработки за счет помещения одного или большего числа браузеров под контроль Visual Studio. Оно особенно полезно, если нужно видеть влияние изменений в некотором диапазоне браузеров. Средство Browser Link работает с отладчиком или без него, но оно наиболее полезно в случае применения средства автоматической компиляции классов, потому что появляется возможность модифицировать любой файл в проекте и видеть влияние изменения, не переходя в окно браузера и не перезагружая страницу вручную.

Настройка средства Browser Link

Включение средства Browser Link требует изменения конфигурации в классе `Startup` (листинг 6.14).

Листинг 6.14. Включение средства Browser Link в файле `Startup.cs` из папки `WorkingWithVisualStudio`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace WorkingWithVisualStudio {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseBrowserLink();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Использование средства Browser Link

Чтобы выяснить, как работает средство Browser Link, выберем пункт Start Without Debugging (Запустить без отладки) в меню Debug (Отладка) среды Visual Studio. Среда Visual Studio запустит приложение и откроет новую вкладку в окне браузера для отображения результатов. Просмотрев HTML-разметку, отправленную браузеру, можно заметить, что она содержит дополнительный раздел, подобный приведенному ниже:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Working with Visual Studio</title>
</head>
<body>
  <h3>Products</h3>
  <table>
    <thead>
      <tr><td>Name</td><td>Price</td></tr>
    </thead>
    <tbody>
      <tr><td>Lifejacket</td><td>&#xA3;48.95</td></tr>
      <tr><td>Soccer ball</td><td>&#xA3;19.50</td></tr>
      <tr><td>Corner flag</td><td>&#xA3;34.95</td></tr>
    </tbody>
  </table>
  <!-- Visual Studio Browser Link -->
  <script type="application/json"
```

```
    id="__browserLink_initializationData">
        {"requestId":"968949d8affc47c4a9c6326de21dfa03","requestMappingFro
    mServer":false}
      </script>
      <script type="text/javascript"
        src="http://localhost:55356/d1a038413c804e178ef009a3be07b262/
    browserLink"
        async="async">
      </script>
      <!-- Конец Browser Link -->
    </body>
    </html>
```

Совет. Если дополнительный раздел отсутствует, тогда нужно выбрать пункт Enable Browser Link (Включить Browser Link) в меню, показанном на рис. 6.18, и перезагрузить страницу в браузере.

Среда Visual Studio добавляет в HTML-разметку, посылаемую браузеру, пару элементов `script`, которые применяются для открытия долговечного HTTP-подключения к серверу приложений, чтобы среда Visual Studio могла обеспечить принудительную перезагрузку страницы браузером. (Если вы не видите элементы `script`, тогда удостоверьтесь в том, что отмечен пункт Enable Browser Link в меню, показанном на рис. 6.18.) В листинге 6.15 приведено изменение, внесенное в представление `Index`, которое проиллюстрирует результат использования средства Browser Link.

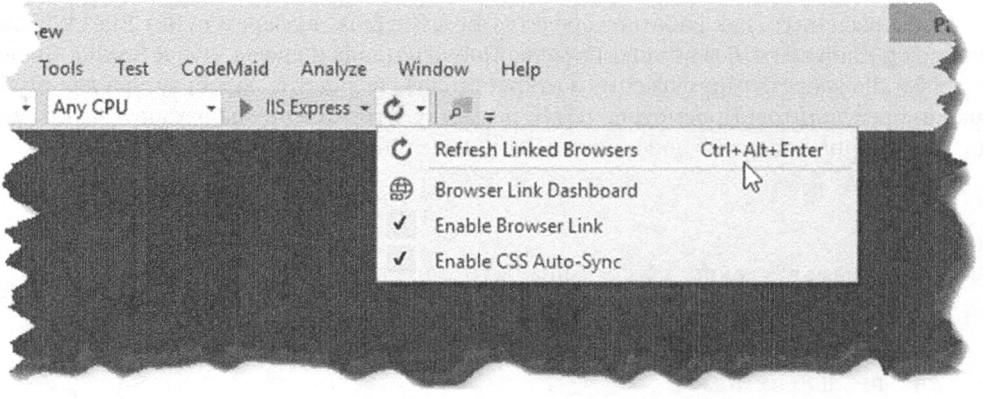


Рис. 6.18. Применение средства Browser Link для перезагрузки страницы в браузере

Листинг 6.15. Добавление временной отметки в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<WorkingWithVisualStudio.Models.Product>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
```

```
  <title>Working with Visual Studio</title>
</head>
<body>
  <h3>Products</h3>
  <p>Request Time: @DateTime.Now.ToString("HH:mm:ss")</p>
  <table>
    <thead>
      <tr><td>Name</td><td>Price</td></tr>
    </thead>
    <tbody>
      @foreach (var p in Model) {
        <tr>
          <td>@p.Name</td>
          <td>@($"{p.Price:C2}")</td>
        </tr>
      }
    </tbody>
  </table>
</body>
</html>
```

Сохраним изменение в файле представления и выберем пункт Refresh Linked Browsers (Обновить связанные браузеры) в меню средства Browser Link внутри панели инструментов Visual Studio (см. рис. 6.18). (Если средство Browser Link не работает, тогда можно попробовать перезагрузить страницу в браузере или перезапустить Visual Studio и повторить попытку.)

Код JavaScript, встроенный в отправляемую браузеру HTML-разметку, будет перезагружать страницу, демонстрируя результат добавления, которым является появление простой временной отметки. Каждый раз, когда выбирается пункт меню Refresh Linked Browsers, браузер будет делать новый запрос к серверу. Результатом запроса окажется визуализация представления `Index` и генерирование новой HTML-страницы с обновленной временной отметкой.

На заметку! Элементы `script`, относящиеся к средству Browser Link, встраиваются только в успешные ответы. Это значит, что если во время компиляции класса, визуализации представления Razor или обработки запроса возникает исключение, то подключение между браузером и Visual Studio утрачивается и страницу придется перезагружать, используя сам браузер, пока проблема не будет устранена.

Использование нескольких браузеров

Средство Browser Link может применяться для отображения приложения в нескольких браузерах одновременно, что удобно, когда нужно уладить вопросы несходства реализаций между браузерами (особенно при реализации специальных таблиц стилей CSS) или увидеть, как приложение визуализируется набором настольных и мобильных браузеров.

Чтобы отобрать браузеры, которые будут использоваться, выберем пункт Browse With (Просмотреть с помощью) в меню, связанном с кнопкой IIS Express в панели инструментов Visual Studio (рис. 6.19).

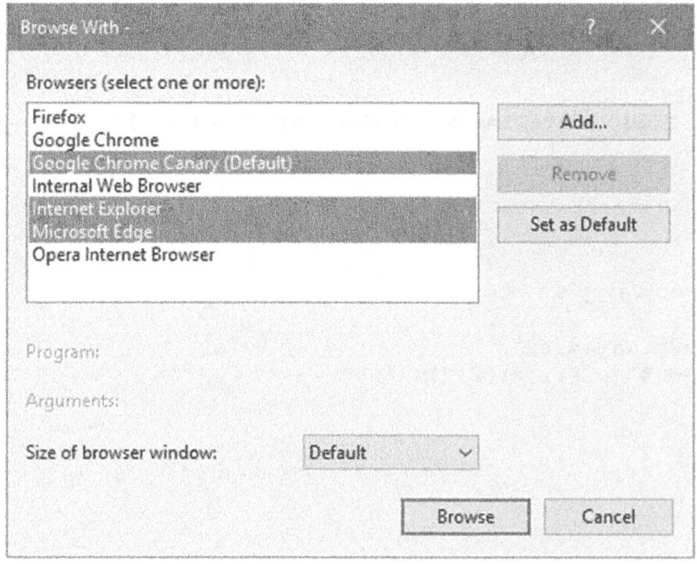


Рис. 6.19. Выбор множества браузеров

Среда Visual Studio отобразит список известных ей браузеров. На рис. 6.20 показаны браузеры, установленные в моей системе; часть из них установлена вместе с Windows (Internet Explorer и Edge), а часть — лично мною по причине их широкого распространения.

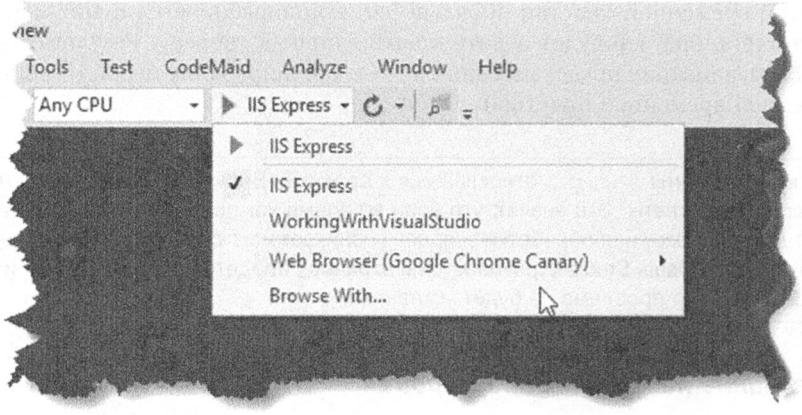


Рис. 6.20. Отбор браузеров из списка

Среда Visual Studio ищет общие браузеры во время процесса установки, но с помощью кнопки Add (Добавить) можно указывать браузеры, которые не были обнаружены автоматически. Кроме того, можно настраивать инструменты тестирования от независимых поставщиков наподобие Browser Stack, которые запускают браузеры на расположенных в облаке виртуальных машинах, так что во время тестирования не придется управлять крупной матрицей операционных систем и браузеров.

На рис 6.20 выбраны три браузера: Chrome, Internet Explorer и Edge. Щелчок на кнопке Browse (Обзор) приводит к запуску всех трех браузеров и загрузки в них URL примера приложения (рис. 6.21).

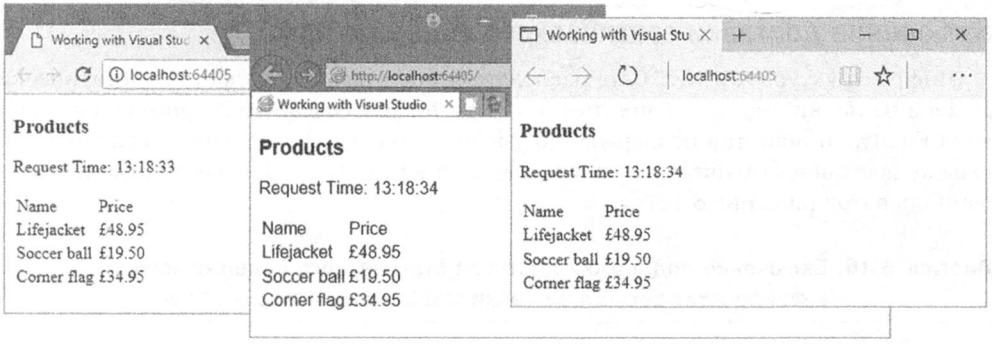


Рис. 6.21. Работа с несколькими браузерами

Чтобы посмотреть, какими браузерами управляет средство Browser Link, выберем пункт Browser Link Dashboard (Инструментальная панель Browser Link) в меню средства Browser Link; откроется окно Browser Link Dashboard (Инструментальная панель Browser Link), представленное на рис. 6.22. Инструментальная панель показывает URL, отображаемый каждым браузером, и позволяет обновлять страницу в каждом браузере по отдельности.

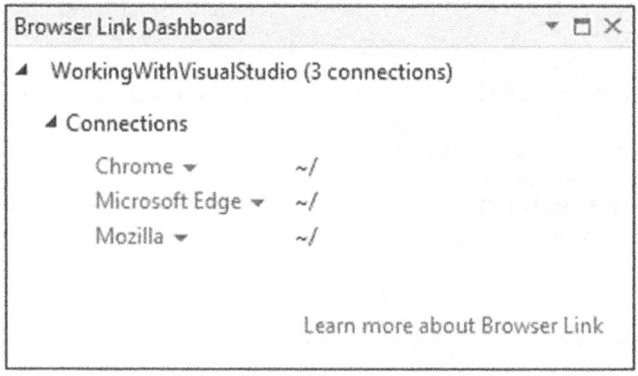


Рис. 6.22. Окно Browser Link Dashboard

Подготовка файлов JavaScript и CSS для развертывания

При создании клиентской части веб-приложения обычно будет создаваться ряд специальных файлов JavaScript и CSS, которые дополняют аналогичные файлы в пакетах, установленных инструментом Bower. Такие файлы требуют специальной обработки для оптимизации их доставки в производственную среду. Цель в том, чтобы ми-

нимизировать количество HTTP-запросов и долю полосы пропускания, необходимую для доставки файлов клиенту. Процесс известен как *пакетирование* и *минификация*. В этом разделе объясняется, как включить доставку статического содержимого и каким образом подготовить такое содержимое для развертывания.

Включение доставки статического содержимого

Инфраструктура ASP.NET Core располагает поддержкой для доставки статических файлов из папки `wwwroot` клиентам, но когда проект создается с применением шаблона Empty, упомянутая поддержка по умолчанию отключена. Чтобы включить поддержку доставки статического содержимого, в класс `Startup` необходимо добавить один оператор (листинг 6.16).

Листинг 6.16. Включение поддержки доставки статического содержимого в файле `Startup.cs` из папки `WorkingWithVisualStudio`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace WorkingWithVisualStudio {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseBrowserLink();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Добавление в проект статического содержимого

Чтобы продемонстрировать процесс пакетирования и минификации, понадобится добавить в проект какое-нибудь статическое содержимое и внедрить его в пример приложения. Для начала создадим папку `wwwroot/css`, в которой по соглашению хранятся специальные файлы CSS. Затем добавим файл по имени `first.css`, используя шаблон элемента Style Sheet (Таблица стилей), как показано на рис. 6.23. Шаблон Style Sheet находится в разделе ASP.NET Core⇨Web⇨Content (ASP.NET Core⇨Веб⇨Содержимое).

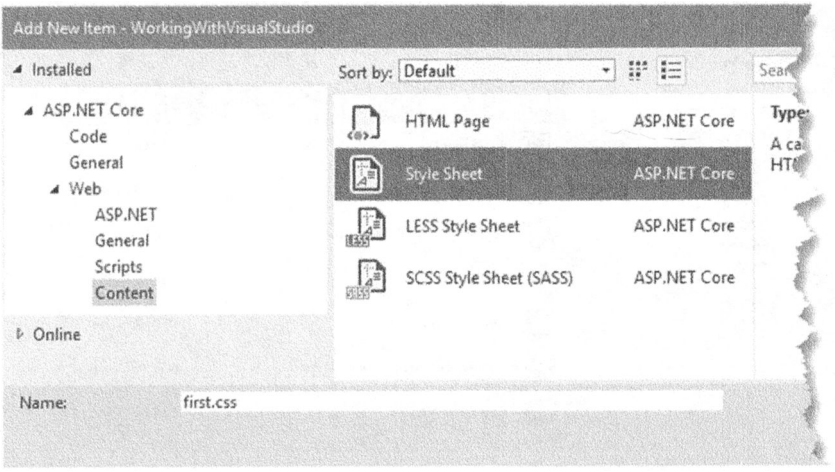


Рис. 6.23. Создание таблицы стилей CSS

Поместим в файл `first.css` стили CSS из листинга 6.17.

Листинг 6.17. Содержимое файла `first.css` из папки `wwwroot/css`

```
h3 {
  font-size: 18pt;
  font-family: sans-serif;
}
table, td {
  border: 2px solid black;
  border-collapse:collapse;
  padding: 5px;
  font-family: sans-serif;
}
```

Повторим процесс для создания в папке `wwwroot/css` еще одной таблицы стилей по имени `second.css` с содержимым, приведенным в листинге 6.18.

Листинг 6.18. Содержимое файла `second.css` из папки `wwwroot/css`

```
p {
  font-family: sans-serif;
  font-size: 10pt;
  color: darkgreen;
  background-color:antiquewhite;
  border: 1px solid black;
  padding: 2px;
}
```

По соглашению специальные файлы JavaScript хранятся в папке `wwwroot/js`. Создадим такую папку и с применением шаблона элемента JavaScript File (Файл JavaScript) добавим в нее файл `third.js` (рис. 6.24). Шаблон JavaScript File находится в разделе ASP.NET Core⇨Web⇨Scripts (ASP.NET Core⇨Веб⇨Сценарии).

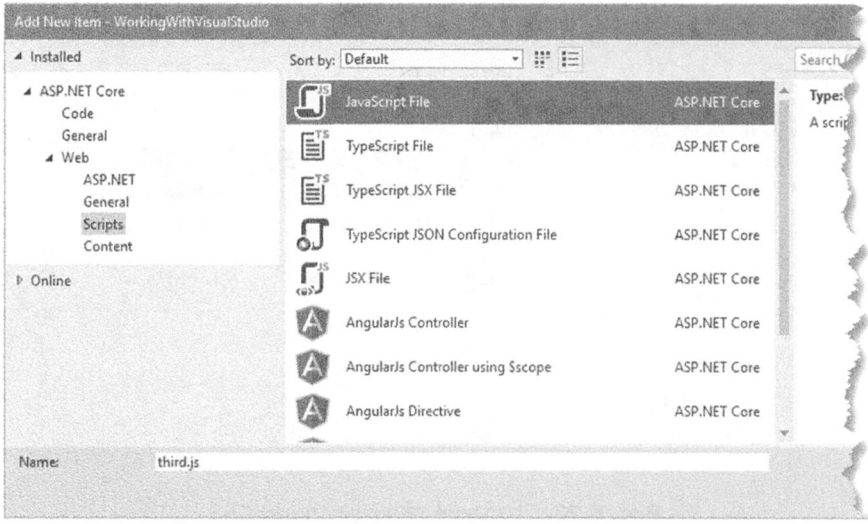


Рис. 6.24. Создание файла JavaScript

Поместим в новый файл простой код JavaScript, представленный в листинге 6.19.

Листинг 6.19. Содержимое файла `third.js` из папки `wwwroot/js`

```
document.addEventListener("DOMContentLoaded", function () {
  var element = document.createElement("p");
  element.textContent = "This is the element from the third.js file";
  document.querySelector("body").appendChild(element);
});
```

Нам необходим еще один файл JavaScript. Создадим в папке `wwroot/js` файл по имени `fourth.js` с содержимым, показанным в листинге 6.20.

Листинг 6.20. Содержимое файла `fourth.js` из папки `wwwroot/js`

```
document.addEventListener("DOMContentLoaded", function () {
  var element = document.createElement("p");
  element.textContent = "This is the element from the fourth.js file";
  document.querySelector("body").appendChild(element);
});
```

Обновление представления

Последний подготовительный шаг связан с обновлением представления `Index.cshtml` для использования новых таблиц стилей CSS и файлов JavaScript (листинг 6.21).

Листинг 6.21. Добавление элементов `script` и `link` в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<WorkingWithVisualStudio.Models.Product>
@{ Layout = null; }
```

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Working with Visual Studio</title>
  <link rel="stylesheet" href="css/first.css" />
  <link rel="stylesheet" href="css/second.css" />
  <script src="js/third.js"></script>
  <script src="js/fourth.js"></script>
</head>
<body>
  <h3>Products</h3>
  <p>Request Time: @DateTime.Now.ToString("HH:mm:ss")</p>
  <table>
    <thead>
      <tr><td>Name</td><td>Price</td></tr>
    </thead>
    <tbody>
      @foreach (var p in Model) {
        <tr>
          <td>@p.Name</td>
          <td>@($"{p.Price:C2}")</td>
        </tr>
      }
    </tbody>
  </table>
</body>
</html>
```

В результате запуска примера приложения появится содержимое, приведенное на рис. 6.25. Существующее содержимое было стилизовано посредством таблиц стилей CSS, а код JavaScript добавил новое содержимое.

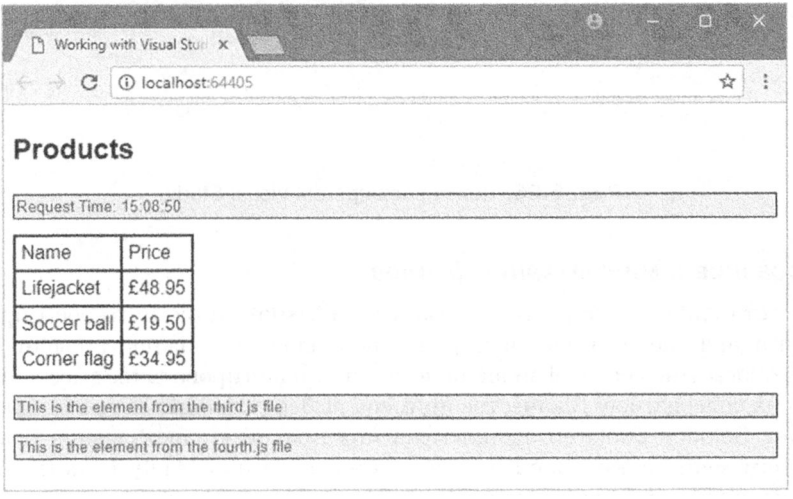


Рис. 6.25. Выполнение примера приложения

Пакетирование и минификация в приложениях MVC

В настоящий момент есть четыре статических файла, и браузер должен делать четыре запроса, чтобы получить эти статические файлы. К тому же каждый такой файл при доставке клиенту отнимает больше полосы пропускания, чем должен, поскольку содержит пробельные символы и имена переменных, которые являются содержательными для разработчика, но не имеют никакого значения для браузера.

Объединение файлов одного и того же типа называется *пакетированием*. Уменьшение размеров файлов называется *минификацией*. Обе задачи выполняются в приложениях ASP.NET Core MVC с помощью расширения Bundler & Minifier (Упаковщик и минификатор) для Visual Studio.

Установка расширения Visual Studio

Первый шаг заключается в установке расширения. Выберем пункт меню Tools⇨Extensions and Updates (Сервис⇨Расширения и обновления) и щелкнем на категории Online (Онлайновые), чтобы отобразить галерею доступных расширений Visual Studio. В поле поиска справа вверху введем слово `Bundler` (рис. 6.26). Отыщем расширение Bundler & Minifier и щелкнем на кнопке Download (Загрузить), чтобы добавить его в Visual Studio. Перезапустим Visual Studio для завершения процесса установки.

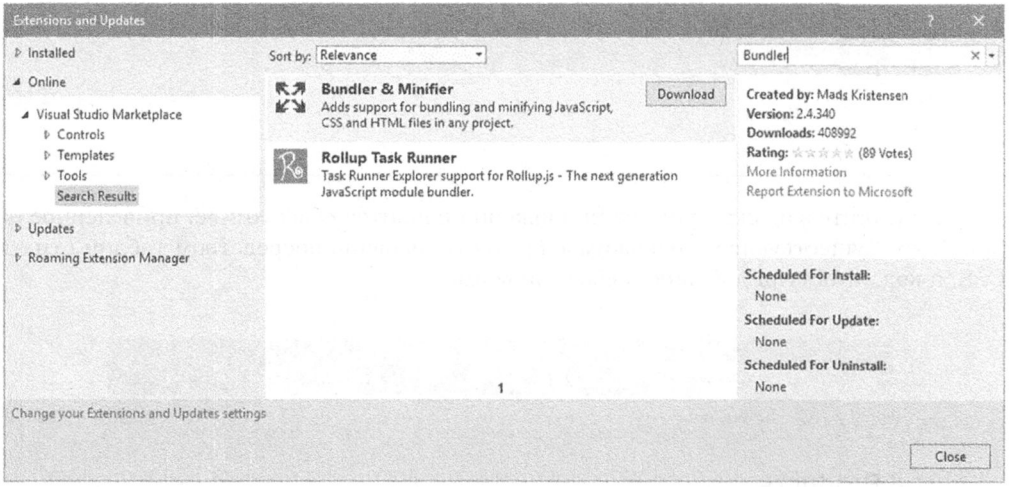


Рис. 6.26. Поиск расширения Visual Studio

Пакетирование и минификация файлов

После установки расширения перезапустим Visual Studio и откроем проект примера. Благодаря добавлению расширения появилась возможность выбирать в окне Solution Explorer множество файлов одного типа, пакетировать их вместе и минифицировать их содержимое. В качестве примера выберем файлы `first.css` и `second.css` в окне Solution Explorer, щелкнем на них правой кнопкой мыши и выберем в контекстном меню пункт Bundler & Minifier⇨Bundle and Minify Files (Упаковщик и минификатор⇨Пакетировать и минифицировать файлы), как показано на рис. 6.27.

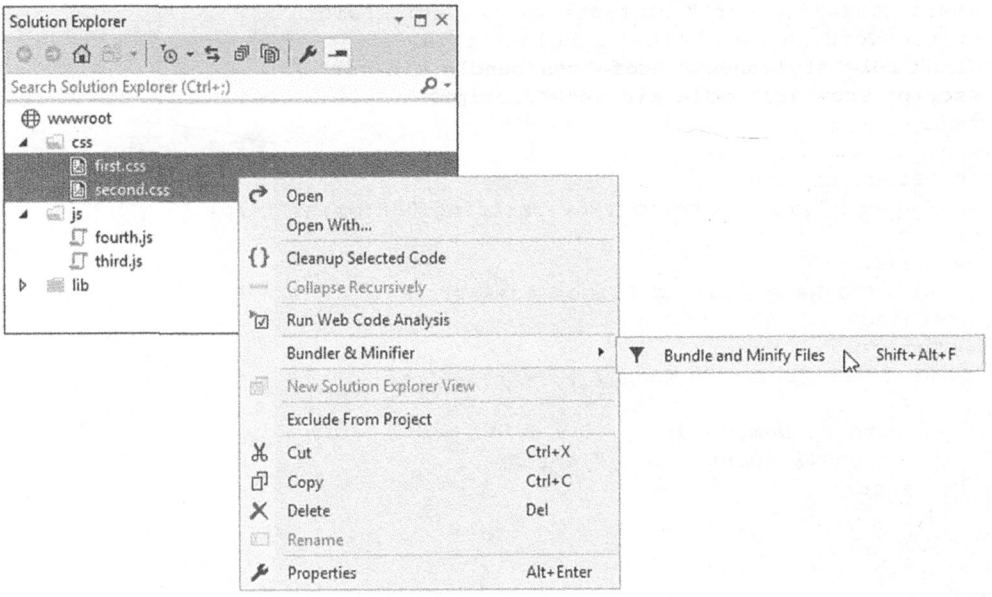


Рис. 6.27. Пакетирование и минификация файлов CSS

Сохраним выходной файл под именем `bundle.css`, в результате чего расширение обработает файлы CSS. В окне Solution Explorer отобразится новый элемент `bundle.css`, который можно раскрыть и увидеть минифицированный файл с именем `bundle.min.css`. Открыв минифицированный файл, вы заметите, что содержимое обоих отдельных файлов CSS было объединено, а все пробельные символы удалены. Работать с таким файлом напрямую вы вряд ли захотите, но он меньше по размеру и требует только одного HTTP-подключения для доставки стилей CSS клиенту.

Повторим процесс с файлами `third.js` и `fourth.js`, чтобы создать новые файлы `bundle.js` и `bundle.min.js` в папке `wwwroot/js`.

Внимание! Удостоверьтесь, что выбираете файлы в порядке их загрузки браузером, чтобы предохранить порядок следования стилей или операторов кода в выходном файле. Таким образом, например, выбирайте файл `third.js` перед файлом `fourth.js`, чтобы обеспечить выполнение кода в правильном порядке.

В листинге 6.22 показано представление `Index.cshtml`, в котором элементы `link` для отдельных файлов заменены одним таким элементом, запрашивающим пакетированные и минифицированные файлы.

Листинг 6.22. Применение пакетированных и минифицированных файлов в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<WorkingWithVisualStudio.Models.Product>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
```

```
  <meta name="viewport" content="width=device-width" />
  <title>Working with Visual Studio</title>
  <link rel="stylesheet" href="css/bundle.min.css" />
  <script src="js/bundle.min.js"></script>
</head>
<body>
  <h3>Products</h3>
  <p>Request Time: @DateTime.Now.ToString("HH:mm:ss")</p>
  <table>
    <thead>
      <tr><td>Name</td><td>Price</td></tr>
    </thead>
    <tbody>
      @foreach (var p in Model) {
        <tr>
          <td>@p.Name</td>
          <td>@($"{p.Price:C2}")</td>
        </tr>
      }
    </tbody>
  </table>
</body>
</html>
```

После запуска приложения никаких визуальных отличий не будет заметно, но приложение использует пакетированные и минифицированные файлы, предоставляя браузеру все стили и код, которые были определены в отдельных файлах.

По мере выполнения операций пакетирования и минификации расширение ведет учет обработанных файлов в файле по имени `bundleconfig.json`, находящемся в корневой папке проекта. Вот конфигурация, которая была сгенерирована для файлов в примере приложения:

```
[
  {
    "outputFileName": "wwwroot/css/bundle.css",
    "inputFiles": [
      "wwwroot/css/first.css",
      "wwwroot/css/second.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/bundle.js",
    "inputFiles": [
      "wwwroot/js/third.js",
      "wwwroot/js/fourth.js"
    ]
  }
]
```

Расширение автоматически отслеживает входные файлы на предмет изменений и заново генерирует выходные файлы, когда происходят изменения, гарантируя отражение в пакетированных и минифицированных файлах любых модификаций. Для демонстрации в листинге 6.23 приведено изменение, внесенное в файл `third.js`.

Листинг 6.23. Внесение изменения в файл `third.js` из папки `wwwroot/js`

```
document.addEventListener("DOMContentLoaded", function () {
  var element = document.createElement("p");
  element.textContent = "This is the element from the (modified) third.js
file";
  document.querySelector("body").appendChild(element);
});
```

После сохранения файла расширение заново сгенерирует файл `bundle.min.js`. Перезагрузив страницу в браузере, можно увидеть изменение (рис. 6.28).

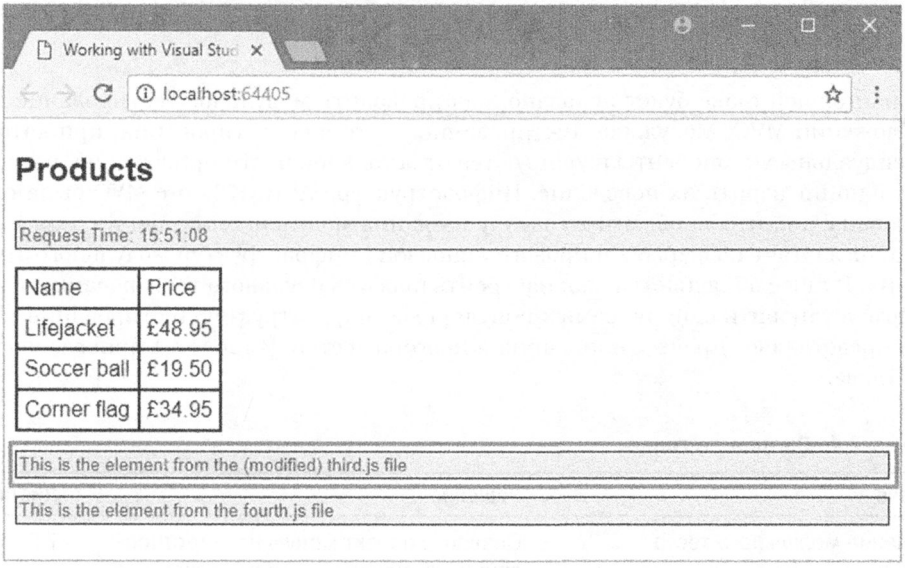


Рис. 6.28. Обнаружение изменений в пакетированных и минифицированных файлах

Резюме

В главе были описаны возможности, которые Visual Studio предлагает для разработки веб-приложений, включая автоматическую компиляцию классов, средство Browser Link, а также пакетирование и минификацию. В следующей главе объясняется, как подвергать модульному тестированию проекты ASP.NET Core MVC.

ГЛАВА 7

Модульное тестирование приложений MVC

В настоящей главе будет показано, как проводить модульное тестирование приложений MVC. Модульное тестирование — это вид тестирования, при котором индивидуальные компоненты изолируются от остальной части приложения, позволяя тщательно проверить их поведение. Инфраструктура ASP.NET Core MVC была спроектирована так, чтобы облегчить задачу создания модульных тестов, и среда Visual Studio предлагает поддержку широкого диапазона инфраструктур модульного тестирования. В главе объясняется, как настроить проект модульного тестирования, каким образом установить одну из самых популярных инфраструктур тестирования и что собой представляет процесс написания и прогона тестов. В табл. 7.1 приведена сводка по главе.

Таблица 7.1. Сводка по главе

Задача	Решение	Листинг
Создание модульного теста	Создайте проект модульного тестирования, установите тестовый пакет и добавьте классы, которые содержат тесты	7.5, 7.6
Изоляция компонентов для модульного тестирования	Используйте интерфейсы для разделения компонентов приложения и применяйте в модульных тестах фиктивные реализации с ограниченными тестовыми данными	7.7–7.14
Прогон тех же самых тестов xUnit.net с разными значениями данных	Используйте параметризованный модульный тест либо получайте тестовые данные из метода или свойства	7.15–7.17
Упрощение процесса создания фиктивных тестовых объектов	Применяйте инфраструктуру имитации	7.18, 7.19

Проводить ли модульное тестирование?

Наличие возможности легко выполнять модульное тестирование является одним из преимуществ использования инфраструктуры ASP.NET Core MVC, но такая процедура предназначена не для всех, и я не намерен притворяться, что дело обстоит иначе.

Мне нравится модульное тестирование, и я применяю его в своих проектах, но далеко не во всех и не настолько согласованно, как вы могли ожидать. Я предпочитаю концентрироваться на написании модульных тестов для средств и функций, о которых знаю, что они будут трудны в реализации и вполне вероятно станут источником ошибок после развертывания. В таких ситуациях модульное тестирование помогает структурировать мои мысли о том, как лучше всего реализовать то, что необходимо. Я считаю, что одно лишь размышление о том, *что* нужно тестировать, способствует появлению идей относительно потенциальных проблем, причем до того, как доведется иметь дело с действительными ошибками и дефектами.

Тем не менее, модульное тестирование — инструмент, а не догма, и только лично вы знаете, в каком объеме оно должно проводиться. Если вы не находите модульное тестирование полезным или располагаете другой методологией, которая вам больше подходит, то не должны думать, что обязаны использовать модульное тестирование просто потому, что это модно. (Однако если вы *не* располагаете более эффективной методологией и не тестируете вообще, то вероятно даете возможность пользователям обнаруживать имеющиеся ошибки, что редко оказывается идеальным решением. Вы *не* обязаны выполнять модульное тестирование, но на самом деле должны проводить тестирование *какого-то* вида.)

Если вы не сталкивались с модульным тестированием ранее, тогда я предлагаю опробовать его и посмотреть, как оно работает. Если вы не поклонник модульного тестирования, то можете пропустить данную главу и перейти к чтению главы 8, где начнется построение более реалистичного приложения MVC.

Подготовка проекта для примера

В главе мы продолжим пользоваться проектом `WorkingWithVisualStudio`, который был создан в главе 6. Здесь мы добавим в него поддержку для создания новых объектов `Product` в хранилище.

Включение встроенных вспомогательных функций дескрипторов

В данной главе применяется одна из встроенных вспомогательных функций дескрипторов для установки атрибута `href` элемента `a`. Работа вспомогательных функций дескрипторов подробно объясняется в главах 23–25, но пока мы просто должны включить их. Создадим файл импортирования представлений, щелкнув правой кнопкой мыши на папке `Views`, выбрав в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и указав шаблон элемента MVC View Imports Page (Страница импортирования представлений MVC) из категории ASP.NET. Среда Visual Studio автоматически назначит файлу имя `_ViewImports.cshtml`, а щелчок на кнопке Add (Добавить) приведет к его созданию. Поместим в файл содержимое, показанное в листинге 7.1.

Листинг 7.1. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Оператор в листинге 7.1 включает встроенные вспомогательные функции дескрипторов, в том числе ту, которая вскоре будет использоваться в представлении Index. Можно было бы добавить операторы using для импортирования пространств имен из проектов, но представления не являются важной частью рассматриваемого здесь примера приложения, так что ссылка на типы моделей с их пространствами имен проблем не вызовет.

Добавление действий к контроллеру

Первый шаг связан с добавлением действий к контроллеру Home, который будет визуализировать представление для ввода данных и для получения этих данных от браузера (листинг 7.2). Действия следуют тому же шаблону, который применялся в главе 2 и подробно обсуждается в главе 17.

Листинг 7.2. Добавление методов действий в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;

namespace WorkingWithVisualStudio.Controllers {
  public class HomeController : Controller {

    SimpleRepository Repository = SimpleRepository.SharedRepository;

    public IActionResult Index() => View(Repository.Products
              .Where(p => p?.Price < 50));

    [HttpGet]
    public IActionResult AddProduct() => View(new Product());

    [HttpPost]
    public IActionResult AddProduct(Product p) {
      Repository.AddProduct(p);
      return RedirectToAction("Index");
    }
  }
}
```

Создание формы для ввода данных

Чтобы снабдить пользователя возможностью создания нового товара, мы создадим представление Razor по имени AddProduct.cshtml в папке Views/Home. Соглашения относительно имени файла и его местоположения соответствуют стандартному представлению, которое визуализирует метод AddProduct() контроллера Home. В листинге 7.3 приведено содержимое нового представления, которое полагается на пакет Boostrap, добавленный к проекту с использованием Bower в главе 6.

Листинг 7.3. Содержимое файла AddProduct.cshtml из папки Views/Home

```
@model WorkingWithVisualStudio.Models.Product
@{ Layout = null; }

<!DOCTYPE html>
<html>
```

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Working with Visual Studio</title>
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.min.css" />
</head>
<body class="p-2">
  <h3 class="text-center">Create Product</h3>
  <form asp-action="AddProduct" method="post">
    <div class="form-group">
      <label asp-for="Name">Name:</label>
      <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
      <label asp-for="Price">Price:</label>
      <input asp-for="Price" class="form-control" />
    </div>
    <div class="text-center">
      <button type="submit" class="btn btn-primary">Add</button>
      <a asp-action="Index" class="btn btn-secondary">Cancel</a>
    </div>
  </form>
</body>
</html>
```

Представление содержит HTML-форму, которая применяет HTTP-запрос `POST` для отправки значений `Name` и `Price` действию `AddProduct` контроллера `Home`. Содержимое стилизовано с использованием пакета CSS из Bootstrap.

Обновление представления `Index`

Последний подготовительный шаг предусматривает обновление представления `Index`, чтобы включить в него ссылку на новую форму (листинг 7.4). Кроме того, появилась возможность удалить файлы JavaScript, используемые в предыдущей главе, и заменить специальные таблицы стилей CSS стилями Bootstrap, которые применяются к элементам HTML в представлении.

Листинг 7.4. Обновление содержимого в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<WorkingWithVisualStudio.Models.Product>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Working with Visual Studio</title>
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.min.css" />
</head>
<body class="p-1">
  <h3 class="text-center">Products</h3>
  <table class="table table-bordered table-striped">
    <thead>
      <tr><td>Name</td><td>Price</td></tr>
    </thead>
```

```
    <tbody>
      @foreach (var p in Model) {
        <tr>
          <td>@p.Name</td>
          <td>@($"{p.Price:C2}")</td>
        </tr>
      }
    </tbody>
  </table>
  <div class="text-center">
    <a class="btn btn-primary" asp-action="AddProduct">
      Add New Product
    </a>
  </div>
</body>
</html>
```

Запустив пример приложения, можно увидеть по-новому стилизованное содержимое и кнопку Add New Product (Добавить новый товар), щелчок на которой приводит к открытию формы для ввода данных. Отправка формы добавит в хранилище новый объект `Product` и перенаправит браузер для отображения первоначального представления приложения (рис. 7.1).

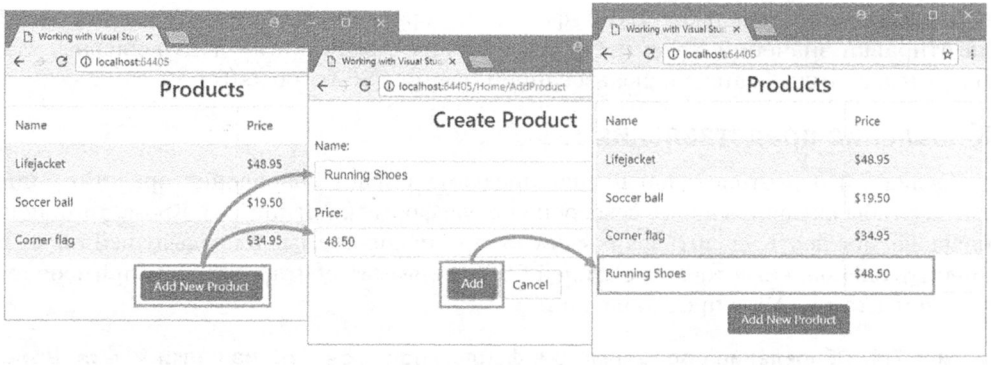


Рис. 7.1. Выполнение примера приложения

Совет. Не забывайте, что в этом примере объекты хранятся только в памяти, т.е. любые создаваемые товары при перезапуске приложения будут утеряны.

Модульное тестирование приложений MVC

Модульные тесты используются для проверки поведения отдельных компонентов и функциональных средств в приложении, а платформа ASP.NET Core и инфраструктура MVC спроектированы так, чтобы максимально облегчить настройку и запуск модульных тестов для веб-приложений. В последующих разделах объясняется, как настроить модульное тестирование в Visual Studio, и демонстрируется написание

модульных тестов для приложений MVC. Кроме того, будут представлены полезные инструменты, которые делают модульное тестирование проще и надежнее.

Доступен целый ряд разных пакетов для модульного тестирования. В книге применяется один из них, который называется xUnit.net; он выбран из-за хорошей интеграции с Visual Studio, к тому же данный пакет использовался командой разработчиков Microsoft при написании модульных тестов для ASP.NET Core. В табл. 7.2 приведена сводка, позволяющая поместить xUnit.net в контекст.

Таблица 7.2. Помещение xUnit.net в контекст

Вопрос	Ответ
Что это такое?	xUnit.net — это инфраструктура модульного тестирования, которая может применяться для тестирования приложений ASP.NET Core MVC
Чем она полезна?	xUnit.net — хорошо написанная инфраструктура модульного тестирования, которая легко интегрируется со средой Visual Studio
Как она используется?	Тесты определяются как методы, аннотированные с помощью атрибута `Fact` или `Theory`. Внутри тела метода используются методы класса `Assert` для сравнения ожидаемого результата теста с тем, что получилось в действительности
Существуют ли какие-то скрытые ловушки или ограничения?	Главный просчет при модульном тестировании связан с недостаточной изоляцией тестируемого компонента. За дополнительными сведениями обращайтесь в раздел "Изолирование компонентов для модульного тестирования" далее в главе. Самой крупной проблемой, специфичной для xUnit.net, является нехватка документации. Кое-какая базовая информация доступна на веб-сайте `http://xunit.github.io`, но расширенное применение требует метода проб и ошибок
Существуют ли альтернативы?	Доступно множество инфраструктур модульного тестирования. Двумя популярными альтернативами являются MSTest (производства Microsoft) и NUnit

На заметку! Практически все в области модульного тестирования является предметом персональных предпочтений и вопросом шумных разногласий. Некоторым разработчикам не нравится отделять модульные тесты от кода приложения, и они предпочитают определять тесты в том же самом проекте или даже в том же самом файле класса. Я описываю здесь распространенный подход, которому следую сам, но если он не кажется вам правильным, тогда вы должны поэкспериментировать с различными стилями тестирования, пока не подберете для себя подходящий.

Создание проекта модульного тестирования

Для приложения ASP.NET Core обычно создается отдельный проект Visual Studio, содержащий модульные тесты, каждый из которых определяется как метод в классе C#. Применение отдельного проекта означает, что приложение можно развернуть, не развертывая одновременно тесты.

Чтобы создать проект тестирования, щелкнем правой кнопкой мыши на элементе решения `WorkingWithVisualStudio` в окне Solution Explorer и выберем в контекстном меню пункт Add⇨New Project (Добавить⇨Новый проект).

В категории Visual C#⇨.NET Core выберем шаблон xUnit Test Project (.NET Core) (Проект тестирования xUnit (.NET Core)), как показано на рис. 7.2.

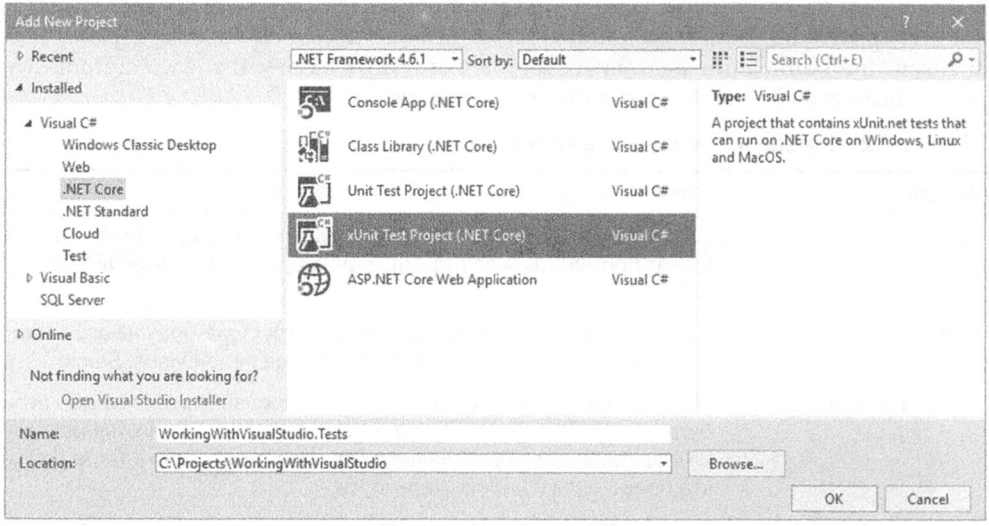


Рис. 7.2. Выбор шаблона xUnit Test Project (.NET Core)

Внимание! Удостоверьтесь в том, что выбираете правильный шаблон проекта. В Visual Studio предлагается несколько шаблонов для проектов модульного тестирования, которые имеют похожие названия.

По соглашению проекту модульного тестирования назначается имя *<ИмяПриложения>*`.Tests`.

Установим имя нового проекта в `WorkingWithVisualStudio.Tests` и щелкнем на кнопке OK. Среда Visual Studio создаст проект и установит пакеты NuGet для инфраструктуры xUnit.net и ее зависимостей.

Удаление стандартных тестовых классов

Среда Visual Studio добавляет в проект тестирования файл класса C#, который приведет в беспорядок результаты более поздних примеров. Щелкнем правой кнопкой мыши на файле `UnitTest1.cs` в проекте `WorkingWithVisualStudio.Tests` и выберем в контекстном меню пункт Delete (Удалить). В открывшемся диалоговом окне щелкнем на кнопке OK, и Visual Studio удалит указанный файл класса.

Создание ссылки на проект

Чтобы сделать классы в главном проекте доступными для тестирования, щелкнем правой кнопкой мыши на элементе `WorkingWithVisualStudio.Tests` в окне Solution Explorer и выберем в контекстном меню пункт Add⇨Reference (Добавить⇨Ссылка).

Отметим флажок для элемента `WorkingWithVisualStudio` в разделе Solution (Решение), как иллюстрируется на рис. 7.3.

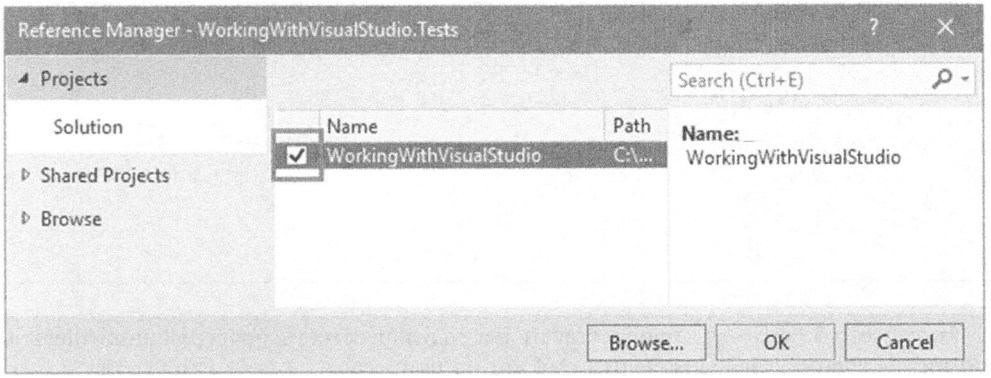


Рис. 7.3. Создание ссылки на проект приложения

Щелкнем на кнопке OK для создания ссылки на проект приложения. В окне Solution Explorer может быть виден работающий значок, отображающийся на элементе Dependencies (Зависимости) для проекта тестирования, но после построения проектов он исчезнет.

Написание и выполнение модульных тестов

Теперь, когда все подготовительные шаги завершены, можно приступать к написанию некоторых тестов.

Первым делом добавим в проект `WorkingWithVisualStudio.Tests` файл класса по имени `ProductTests.cs` с определением, представленным в листинге 7.5. Несмотря на простоту, класс содержит все, что требуется для начала модульного тестирования.

На заметку! В методе `CanChangeProductPrice()` умышленно допущена ошибка, которая позже в разделе будет исправлена.

Листинг 7.5. Содержимое файла `ProductTests.cs` из папки `WorkingWithVisualStudio.Tests`

```
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
  public class ProductTests {
    [Fact]
    public void CanChangeProductName() {
      // Организация
      var p = new Product { Name = "Test", Price = 100M };
      // Действие
      p.Name = "New Name";
      // Утверждение
      Assert.Equal("New Name", p.Name);
    }

    [Fact]
    public void CanChangeProductPrice() {
```

```
        // Организация
        var p = new Product { Name = "Test", Price = 100M };

        // Действие
        p.Price = 200M;

        // Утверждение
        Assert.Equal(100M, p.Price);
    }
  }
}
```

В классе `ProductTests` присутствуют два модульных теста, проверяющих отдельные линии поведения класса модели `Product` из проекта `WorkingWithVisualStudio`. Проект тестирования может содержать много классов, которые способны включать много модульных тестов.

По соглашению имя тестового метода описывает то, что делает тест, а имя класса — то, что подвергается тестированию. Это облегчает структурирование тестов в проекте и упрощает понимание того, какими будут результаты всех тестов, когда они прогоняются средой Visual Studio. Имя `ProductTests` указывает, что класс содержит тесты для класса `Product`, а имена методов говорят о том, что они проверяют возможность изменения названия и цены объекта `Product`.

Атрибут `Fact` применяется к каждому методу, указывая на то, что метод является тестом. Внутри тела метода модульный тест следует шаблону, который называется *организация/действие/утверждение* (arrange/act/assert — A/A/A). *Организация* относится к настройке условий для теста, *действие* — к выполнению теста, а *утверждение* — к проверке того, что результат оказался тем, который ожидали.

Разделы организации и действия этих тестов представляют собой обычный код C#, но раздел утверждения обрабатывается инфраструктурой xUnit.net, которая предоставляет класс по имени `Assert`, чьи методы используются для проверки, является ли результат действия тем, что ожидался.

Совет. Атрибут `Fact` и класс `Assert` определены в пространстве имен `Xunit`, для которого должен быть предусмотрен оператор `using` в каждом тестовом классе.

Методы класса `Assert` определены как статические и применяются для выполнения разных видов сравнений между ожидаемыми и действительными результатами. В табл. 7.3 описаны распространенные методы класса `Assert`.

Каждый метод класса `Assert` позволяет выполнять разные виды сравнений и генерирует исключение, если результат оказывается не тем, который ожидался. Исключение применяется для указания на то, что тест не прошел. В тестах из листинга 7.5 метод `Equal()` использовался для определения, корректно ли изменилось значение свойства:

```
...
Assert.Equal("New Name", p.Name);
...
```

Таблица 7.3. Часто используемые методы класса `Assert` из инфраструктуры xUnit.net

Метод	Описание
`Equal(expected, result)`	Добавляет утверждение о том, что результат равен ожидаемому исходу. Существуют перегруженные версии этого метода для сравнения различных типов и для сравнения коллекций. Имеется также версия, которая принимает дополнительный аргумент в форме объекта, реализующего интерфейс `IEqualityComparer<T>` для сравнения объектов
`NotEqual(expected, result)`	Добавляет утверждение о том, что результат не равен ожидаемому исходу
`True(result)`	Добавляет утверждение о том, что результат равен `true`
`False(result)`	Добавляет утверждение о том, что результат равен `false`
`IsType(expected, result)`	Добавляет утверждение о том, что результат принадлежит указанному типу
`IsNotType(expected, result)`	Добавляет утверждение о том, что результат не принадлежит указанному типу
`IsNull(result)`	Добавляет утверждение о том, что результат равен `null`
`IsNotNull(result)`	Добавляет утверждение о том, что результат не равен `null`
`InRange(result, low, high)`	Добавляет утверждение о том, что результат находится между `low` и `high`
`NotInRange(result, low, high)`	Добавляет утверждение о том, что результат не находится между `low` и `high`
`Throws(exception, expression)`	Добавляет утверждение о том, что указанное выражение генерирует исключение заданного типа

Прогон тестов с помощью окна Test Explorer

Среда Visual Studio предлагает поддержку для поиска и прогона модульных тестов посредством окна Test Explorer (Проводник тестов), которое доступно через пункт меню Test⇨Windows⇨Test Explorer (Тест⇨Окна⇨Проводник тестов) и показано на рис. 7.4.

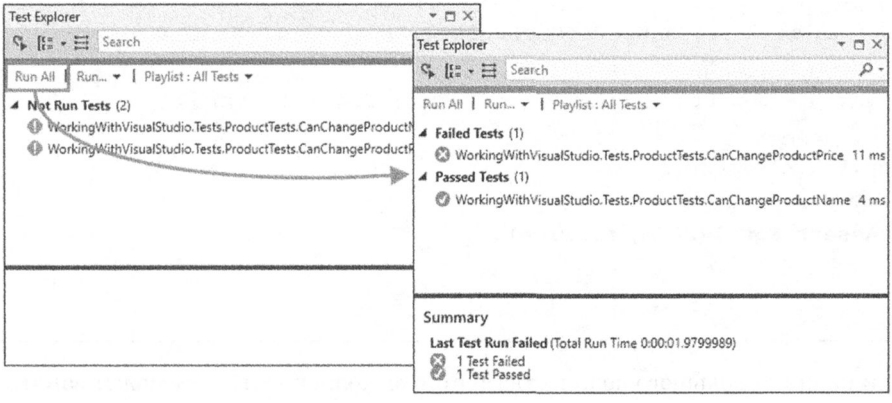


Рис. 7.4. Окно Test Explorer в Visual Studio

Совет. Если вы не видите модульные тесты в окне Test Explorer, тогда постройте решение. Компиляция инициирует процесс, с помощью которого обнаруживаются модульные тесты.

Выполним тесты, щелкнув на пункте Run All (Выполнить все) в окне Test Explorer. Среда Visual Studio применит xUnit.net для прогона тестов в проекте и отобразит результаты. Как отмечалось, тест `CanChangeProductPrice()` содержит ошибку, которая приводит к тому, что он не проходит. Проблема связана с аргументом метода `Assert.Equal()`, из-за чего происходит сравнение с исходным значением свойства `Price`, а не со значением, на которое оно изменилось. В листинге 7.6 проблема устранена.

Совет. Когда тест не проходит, то прежде чем просматривать компонент, на который он нацелен, всегда полезно проверить правильность самого теста, особенно если тест только что написан или недавно модифицировался.

Листинг 7.6. Исправление теста в файле `ProductTests.cs` из папки `WorkingWithVisualStudio.Tests`

```
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {

  public class ProductTests {

    [Fact]
    public void CanChangeProductName() {

      // Организация
      var p = new Product { Name = "Test", Price = 100M };

      // Действие
      p.Name = "New Name";

      // Утверждение
      Assert.Equal("New Name", p.Name);
    }

    [Fact]
    public void CanChangeProductPrice() {

      // Организация
      var p = new Product { Name = "Test", Price = 100M };

      // Действие
      p.Price = 200M;

      // Утверждение
      Assert.Equal(200M, p.Price);
    }
  }
}
```

При наличии большого количества тестов выполнение их всех может занять некоторое время. Таким образом, чтобы можно было работать быстро и итеративно, в окне

Test Explorer предлагаются различные варианты для выбора подмножества тестов, подлежащих прогону. Самым полезным подмножеством является набор тестов, которые не прошли (рис. 7.5). Запустим исправленный тест снова, и в окне Test Explorer будет показано, что не прошедшие тесты отсутствуют.

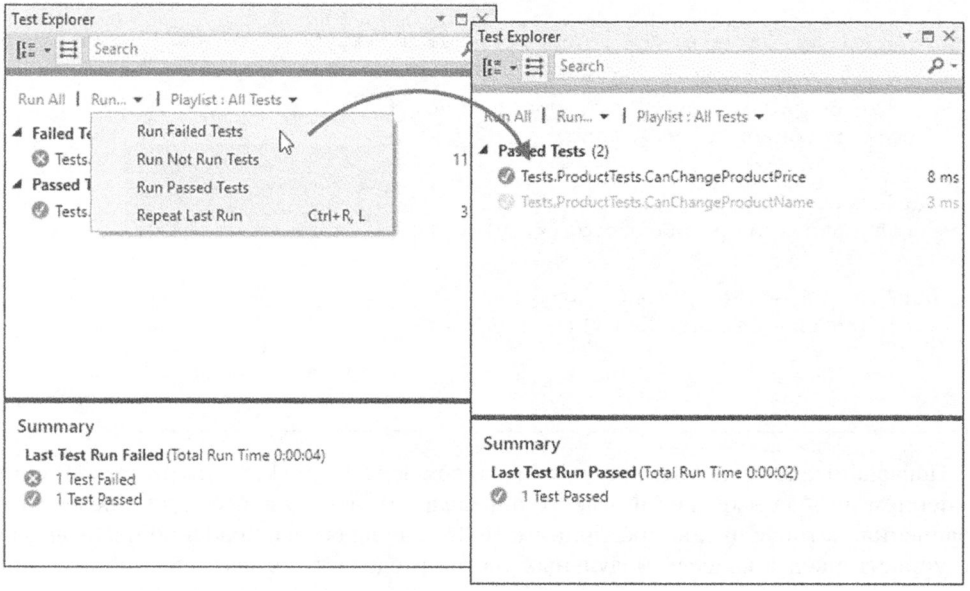


Рис. 7.5. Избирательный прогон тестов

Изолирование компонентов для модульного тестирования

Написание модульных тестов для классов моделей вроде `Product` особой сложностью не отличается. Класс `Product` не только прост, он также самодостаточен, т.е. при выполнении какого-то действия над объектом `Product` можно иметь уверенность в том, что тестируется функциональность, предоставляемая классом `Product`.

С другими компонентами в приложении MVC ситуация сложнее, потому что между ними есть зависимости. Следующий набор определяемых тестов будет оперировать на контроллере, исследуя последовательность объектов `Product`, которая передается между контроллером и представлением.

При сравнении объектов, являющихся экземплярами специальных классов, понадобится использовать метод `Assert.Equal()` из xUnit.net, который принимает аргумент, реализующий интерфейс `IEqualityComparer<T>`, так что объекты можно сравнивать. Сначала необходимо добавить в проект модульного тестирования файл класса по имени `Comparer.cs` и поместить в него определения вспомогательных классов, приведенные в листинге 7.7.

Листинг 7.7. Содержимое файла `Comparer.cs` из папки `WorkingWithVisualStudio.Tests`

```
using System;
using System.Collections.Generic;
```

```
namespace WorkingWithVisualStudio.Tests {
  public class Comparer {
    public static Comparer<U> Get<U>(Func<U, U, bool> func) {
      return new Comparer<U>(func);
    }
  }
  public class Comparer<T> : Comparer, IEqualityComparer<T> {
    private Func<T, T, bool> comparisonFunction;
    public Comparer(Func<T, T, bool> func) {
      comparisonFunction = func;
    }
    public bool Equals(T x, T y) {
      return comparisonFunction(x, y);
    }
    public int GetHashCode(T obj) {
      return obj.GetHashCode();
    }
  }
}
```

Показанные классы позволят создавать объекты `IEqualityComparer<T>` с применением лямбда-выражений вместо определения нового класса для каждого типа сравнения, которое нужно предпринимать. Это не является жизненно необходимым, но упростит код в классах модульных тестов и сделает его легче для понимания и сопровождения.

Теперь, когда можно легко делать сравнения, давайте рассмотрим проблему зависимостей между компонентами в приложении.

Добавим в проект `WorkingWithVisualStudio.Tests` новый файл класса по имени `HomeControllerTests.cs` и поместим в него определение модульного теста, представленное в листинге 7.8.

Листинг 7.8. Содержимое файла `HomeControllerTests.cs` из папки `WorkingWithVisualStudio.Tests`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
  public class HomeControllerTests {
    [Fact]
    public void IndexActionModelIsComplete() {
      // Организация
      var controller = new HomeController();
      // Действие
      var model = (controller.Index() as ViewResult)?.ViewData.Model
                as IEnumerable<Product>;
```

```
        // Утверждение
        Assert.Equal(SimpleRepository.SharedRepository.Products, model,
          Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
            && p1.Price == p2.Price));
    }
  }
}
```

Модульный тест в листинге 7.8 проверяет, что метод действия `Index()` передает представлению все объекты в хранилище. (На раздел действия можно пока не обращать внимания; класс `ViewResult` и роль, которую он играет в приложениях MVC, будут объясняться в главе 17. В настоящий момент достаточно знать, что здесь получаются данные модели, возвращаемые методом действия `Index()`.)

Запустив тест, можно заметить, что он не проходит, указывая на отличие между набором объектов в хранилище и набором объектов, которые возвратил метод `Index()`. Однако когда дело доходит до выявления причины, из-за чего тест не прошел, возникает проблема: предполагается, что тест действует на контроллере `Home`, но класс контроллера зависит от класса `SimpleRepository`. Данный факт затрудняет выяснение, отразил ли тест проблему с классом, на который он нацелен, или же проблему в другой части приложения.

Пример приложения достаточно прост, чтобы можно было легко выявить проблему, всего лишь взглянув на код классов `HomeController` и `SimpleRepository`. В реальном приложении визуальный осмотр не настолько прост, т.к. цепочка зависимостей способна затруднить понимание того, что привело к отказу в прохождении теста. Обычно хранилище будет полагаться на какую-то разновидность системы постоянного хранения, подобную базе данных, а также библиотеку, которая предоставляет к ней доступ. Модульный тест может взаимодействовать со всей цепочкой сложных компонентов, любой из которых может вызвать проблему.

Модульные тесты эффективны, когда они нацелены на небольшие части приложения, такие как отдельный метод или класс. Нам необходима возможность изоляции контроллера `Home` от остальной части приложения, чтобы можно было ограничить область действия теста и исключить влияние со стороны хранилища.

Изолирование компонента

Ключом к изолированию компонентов является использование интерфейсов C#. Чтобы отделить контроллер от хранилища, добавим в папку `Models` новый файл по имени `IRepository.cs` с определением интерфейса, которое показано в листинге 7.9.

Листинг 7.9. Содержимое файла `IRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Models {

  public interface IRepository {
    IEnumerable<Product> Products { get; }
    void AddProduct(Product p);
  }
}
```

С интерфейсом `IRepository` не связано ничего примечательного (за исключением того, что в нем не определен полный набор операций, который обычно будет нужен в веб-приложении; более реалистичный и завершенный пример можно найти в главе 8). Тем не менее, добавление интерфейса вроде `IRepository` позволяет легко изолировать компонент для тестирования. Первым делом нужно обновить класс `SimpleRepository`, чтобы он реализовывал новый интерфейс (листинг 7.10).

Листинг 7.10. Реализация интерфейса в файле `SimpleRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace WorkingWithVisualStudio.Models {
  public class SimpleRepository : IRepository {
    private static SimpleRepository sharedRepository = new
SimpleRepository();
    private Dictionary<string, Product> products
      = new Dictionary<string, Product>();

    public static SimpleRepository SharedRepository => sharedRepository;

    public SimpleRepository() {
      var initialItems = new[] {
        new Product { Name = "Kayak",       Price = 275M   },
        new Product { Name = "Lifejacket",  Price = 48.95M },
        new Product { Name = "Soccer ball", Price = 19.50M },
        new Product { Name = "Corner flag", Price = 34.95M }
      };
      foreach (var p in initialItems) {
        AddProduct(p);
      }
      products.Add("Error", null);
    }

    public IEnumerable<Product> Products => products.Values;

    public void AddProduct(Product p) => products.Add(p.Name, p);
  }
}
```

Следующий шаг предусматривает модификацию контроллера, чтобы свойство, применяемое для ссылки на хранилище, использовало интерфейс, а не класс (листинг 7.11).

Совет. Инфраструктура ASP.NET Core MVC поддерживает более элегантный подход к решению такой проблемы, который называется *внедрением зависимостей* и описан в главе 18. Внедрение зависимостей зачастую вызывает путаницу, поэтому в настоящей главе компоненты изолируются более простым и ручным способом.

Листинг 7.11. Добавление свойства `Repository` в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;
```

```
namespace WorkingWithVisualStudio.Controllers {
  public class HomeController : Controller {
    public IRepository Repository = SimpleRepository.SharedRepository;

    public IActionResult Index() => View(Repository.Products
              .Where(p => p?.Price < 50));

    [HttpGet]
    public IActionResult AddProduct() => View();

    [HttpPost]
    public IActionResult AddProduct(Product p) {
      Repository.AddProduct(p);
      return RedirectToAction("Index");
    }
  }
}
```

Модификация может выглядеть незначительной, но она позволяет изменять хранилище, которое контроллер применяет во время тестирования, что демонстрирует способ изоляции контроллера. В листинге 7.12 модульные тесты контроллера обновлены, так что они используют специальную версию хранилища.

Листинг 7.12. Изоляция контроллера в модульном тесте внутри файла HomeControllerTests.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
  public class HomeControllerTests {

    class ModelCompleteFakeRepository : IRepository {

      public IEnumerable<Product> Products { get; } = new Product[] {
        new Product { Name = "P1", Price = 275M   },
        new Product { Name = "P2", Price = 48.95M },
        new Product { Name = "P3", Price = 19.50M },
        new Product { Name = "P3", Price = 34.95M }
      };

      public void AddProduct(Product p) {
        // Ничего не делать - для теста не требуется
      }
    }

    [Fact]
    public void IndexActionModelIsComplete() {
      // Организация
      var controller = new HomeController();
      controller.Repository = new ModelCompleteFakeRepository();

      // Действие
      var model = (controller.Index() as ViewResult)?.ViewData.Model
                as IEnumerable<Product>;
```

```
        // Утверждение
        Assert.Equal(controller.Repository.Products, model,
          Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
            && p1.Price == p2.Price));
    }
  }
}
```

Мы определили фиктивную реализацию интерфейса `IRepository`, в которой присутствует только свойство, необходимое для теста, и всегда применяются согласованные данные (чего может не быть при работе с реальной базой данных, особенно в случае ее совместного использования с другими разработчиками, вносящими собственные изменения).

Исправленный модульный тест по-прежнему не проходит, указывая на то, что причиной проблемы является метод действия `Index()` в классе `HomeController`, а не компоненты, от которых он зависит. Метод действия, вызываемый модульным тестом, в достаточной степени прост для того, чтобы проблема стала очевидной в результате его осмотра:

```
...
public IActionResult Index() => View(Repository.Products.Where(p =>
p.Price < 50));
...
```

Проблема связана с применением метода `Where()` из LINQ, который используется для фильтрации объектов `Product` со значением свойства `Price`, равным `50` или больше. Здесь уже есть серьезное указание на причину проблемы, но передовой опыт предполагает создание теста, который подтвердит наличие проблемы, прежде чем вносить корректирующую правку (листинг 7.13).

Совет. В показанных тестах присутствует много дублированного кода. В следующем разделе объясняется, как упростить тесты.

Листинг 7.13. Добавление теста в файле `HomeControllerTests.cs` из папки `WorkingWithVisualStudio.Tests`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
  public class HomeControllerTests {

    class ModelCompleteFakeRepository : IRepository {

      public IEnumerable<Product> Products { get; } = new Product[] {
        new Product { Name = "P1", Price = 275M },
        new Product { Name = "P2", Price = 48.95M },
        new Product { Name = "P3", Price = 19.50M },
        new Product { Name = "P3", Price = 34.95M }
      };
```

```
    public void AddProduct(Product p) {
      // Ничего не делать - для теста не требуется
    }
  }

  [Fact]
  public void IndexActionModelIsComplete() {
    // Организация
    var controller = new HomeController();
    controller.Repository = new ModelCompleteFakeRepository();

    // Действие
    var model = (controller.Index() as ViewResult)?.ViewData.Model
              as IEnumerable<Product>;

    // Утверждение
    Assert.Equal(controller.Repository.Products, model,
      Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
        && p1.Price == p2.Price));
  }

  class ModelCompleteFakeRepositoryPricesUnder50 : IRepository {

    public IEnumerable<Product> Products { get; } = new Product[] {
      new Product { Name = "P1", Price = 5M },
      new Product { Name = "P2", Price = 48.95M },
      new Product { Name = "P3", Price = 19.50M },
      new Product { Name = "P3", Price = 34.95M }
    };

    public void AddProduct(Product p) {
      // Ничего не делать - для теста не требуется
    }
  }

  [Fact]
  public void IndexActionModelIsCompletePricesUnder50() {
    // Организация
    var controller = new HomeController();
    controller.Repository = new
ModelCompleteFakeRepositoryPricesUnder50();

    // Действие
    var model = (controller.Index() as ViewResult)?.ViewData.Model
              as IEnumerable<Product>;

    // Утверждение
    Assert.Equal(controller.Repository.Products, model,
      Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
        && p1.Price == p2.Price));
  }
 }
}
```

Мы определили фиктивное хранилище, которое содержит только объекты `Product` со значениями свойства `Price` меньше `50`, и применили его в новом тесте. Запустив тест, можно заметить, что он проходит, подкрепляя идею о том, что проблема связана с применением метода `Where()` в методе действия `Index()`.

В реальном проекте выяснение причины отказа теста означает необходимость согласования цели теста со спецификацией для приложения. Вполне может оказаться, что метод `Index()` обязан фильтровать объекты `Product` по свойству `Price`, в случае чего тест нуждается в пересмотре. Это распространенный итог, и тест, который не прошел, вовсе не всегда указывает на присутствие реальной проблемы в приложении. С другой стороны, если метод действия `Index()` не должен фильтровать объекты модели, тогда потребуется внести корректирующую правку (листинг 7.14).

Понятие разработки через тестирование

В настоящей главе я придерживаюсь наиболее часто используемого стиля модульного тестирования, при котором мы пишем функцию приложения и затем тестируем ее, чтобы удостовериться в том, что она работает требуемым образом. Такой стиль популярен, поскольку большинство разработчиков думают сначала о прикладном коде, а затем о его тестировании (я определенно подпадаю под эту категорию).

Проблема такого подхода в том, что он склоняет к созданию модульных тестов только для того прикладного кода, который было трудно писать или сложно отлаживать, оставляя ряд аспектов каких-то функций лишь частично протестированными или вообще без тестирования.

Альтернативным подходом является *разработка через тестирование* (Test-Driven Development — TDD). Существует много вариаций TDD, но основная идея в том, что тесты для функции пишутся до того, как она реализуется. Написание тестов первыми заставляет более тщательно думать о реализуемой спецификации и о том, как узнать, что функция была реализована корректно. Вместо погружения в детали реализации подход TDD заставляет заранее продумывать, чем будет измеряться успех или неудача.

Все создаваемые тесты изначально не будут проходить, т.к. новая функция еще не реализована. Однако по мере добавления кода в приложение тесты постепенно будут переходить из красного состояния в зеленое, и ко времени завершения функции все тесты окажутся пройденными. Подход TDD требует дисциплины, но приводит к получению более полного набора тестов и может дать в результате более надежный и устойчивый код.

Листинг 7.14. Удаление фильтрации посредством LINQ в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using WorkingWithVisualStudio.Models;
using System.Linq;

namespace WorkingWithVisualStudio.Controllers {
  public class HomeController : Controller {
    public IRepository Repository = SimpleRepository.SharedRepository;

    public IActionResult Index() => View(Repository.Products);

    [HttpGet]
    public IActionResult AddProduct() => View(new Product());

    [HttpPost]
    public IActionResult AddProduct(Product p) {
      Repository.AddProduct(p);
      return RedirectToAction("Index");
    }
  }
}
```

Запустив тесты снова, легко заметить, что все они проходят (рис. 7.6).

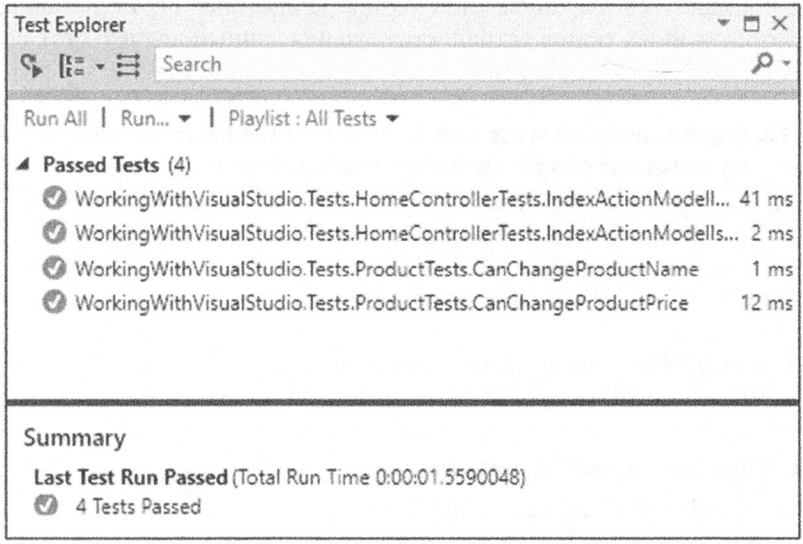


Рис. 7.6. Прохождение всех тестов

Может показаться, что мы проделали слишком много работы для устранения такой простой проблемы, однако возможность протестировать отдельный компонент жизненно важна при построении реального приложения. Достичь точки, где проблема идентифицирована и написаны тесты для проверки исправления, удастся лишь тогда, когда вы способны эффективно изолировать компоненты.

Улучшение модульных тестов

В предыдущем разделе был представлен базовый подход к написанию модульных тестов и прогону тестов в Visual Studio, а также акцентировано внимание на важности изоляции тестируемого компонента. В этом разделе рассматриваются расширенные инструменты и средства, которые можно применять для написания тестов более согласованным и выразительным образом. Если вы сильно увлечетесь модульным тестированием, то можете в итоге получить настолько много тестового кода, что его ясность становится очень важной, особенно с учетом необходимости пересмотра тестов для отражения изменений в приложении, вносимых во время разработки и сопровождения.

Параметризация модульного теста

Тесты, написанные для класса `HomeController`, выявили проблему, которая присутствовала только для определенных значений данных. Чтобы проверить такое условие, были созданы два похожих теста, каждый из которых содержал собственное фиктивное хранилище. При таком подходе появляется дублированный код, тем более с учетом того, что единственное отличие между двумя тестами связано с набором значений `decimal`, которые указываются для свойства `Price` объектов `Product` в фиктивных хранилищах.

Инфраструктура xUnit.net предлагает поддержку *параметризованных тестов*, когда данные, используемые в тесте, из него удаляются, так что единственный метод может применяться для множества тестов. В листинге 7.15 с помощью средства параметризованных тестов устраняется дублированный код в тестах для класса HomeController.

Листинг 7.15. Параметризация модульного теста в файле HomeControllerTests.cs из папки WorkingWithVisualStudio.Tests

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;

namespace WorkingWithVisualStudio.Tests {
  public class HomeControllerTests {

    class ModelCompleteFakeRepository : IRepository {

      public IEnumerable<Product> Products { get; set; }

      public void AddProduct(Product p) {
        // Ничего не делать - для теста не требуется
      }
    }

    [Theory]
    [InlineData(275, 48.95, 19.50, 24.95)]
    [InlineData(5, 48.95, 19.50, 24.95)]
    public void IndexActionModelIsComplete(decimal price1, decimal price2,
                decimal price3, decimal price4)
    {
      // Организация
      var controller = new HomeController();
      controller.Repository = new ModelCompleteFakeRepository {
        Products = new Product[] {
          new Product {Name = "P1", Price = price1 },
          new Product {Name = "P2", Price = price2 },
          new Product {Name = "P3", Price = price3 },
          new Product {Name = "P4", Price = price4 },
        }
      };

      // Действие
      var model = (controller.Index() as ViewResult)?.ViewData.Model
                as IEnumerable<Product>;

      // Утверждение
      Assert.Equal(controller.Repository.Products, model,
        Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
          && p1.Price == p2.Price));
    }
  }
}
```

Параметризованные модульные тесты помечаются атрибутом `Theory` вместо атрибута `Fact`, который используется для стандартных тестов. Здесь также применяется атрибут `InlineData`, который позволяет указывать значения для аргументов, определяемых методом модульного теста. Язык C# ограничивает способ выражения значений данных в атрибутах, поэтому мы определили четыре аргумента `decimal` тестового метода и посредством атрибута `InlineData` предоставили для них значения. Значения `decimal` внутри тестового метода используются для генерации массива объектов `Product`, который применяется для установки свойства `Products` объекта фиктивного хранилища.

Каждый атрибут `InlineData` определяет отдельный модульный тест, который показан как индивидуальный элемент в окне Test Explorer среды Visual Studio (рис. 7.7). Запись в окне Test Explorer отображает значения, которые будут использоваться для аргументов метода модульного теста.

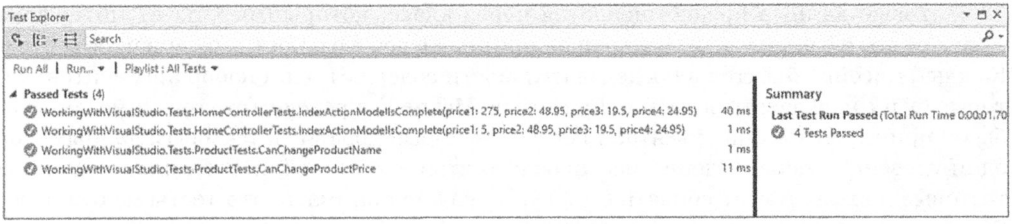


Рис. 7.7. Параметризованные тесты в окне Test Explorer среды Visual Studio

Получение тестовых данных из метода или свойства

Ограничения, налагаемые на выражение данных в атрибутах, сокращают полезность атрибута `InlineData`. Альтернативный подход предусматривает создание статического метода или свойства, возвращающего объект, который требуется для тестирования. В такой ситуации нет никаких ограничений относительно того, как определяются данные, и можно создавать более широкий диапазон тестовых значений. Чтобы продемонстрировать подход в работе, добавим в проект модульного тестирования файл класса по имени `ProductTestData.cs` с определением, показанным в листинге 7.16.

Листинг 7.16. Содержимое файла `ProductTestData.cs` из папки `WorkingWithVisualStudio.Tests`

```
using System.Collections;
using System.Collections.Generic;
using WorkingWithVisualStudio.Models;

namespace WorkingWithVisualStudio.Tests {
  public class ProductTestData : IEnumerable<object[]> {
    public IEnumerator<object[]> GetEnumerator() {
      yield return new object[] { GetPricesUnder50() };
      yield return new object[] { GetPricesOver50 };
    }
    IEnumerator IEnumerable.GetEnumerator() {
      return this.GetEnumerator();
    }
```

```
    private IEnumerable<Product> GetPricesUnder50() {
      decimal[] prices = new decimal[] { 275, 49.95M, 19.50M, 24.95M };
      for (int i = 0; i < prices.Length; i++) {
        yield return new Product { Name = $"P{i + 1}", Price = prices[i] };
      }
    }

    private Product[] GetPricesOver50 => new Product[] {
      new Product { Name = "P1", Price = 5 },
      new Product { Name = "P2", Price = 48.95M },
      new Product { Name = "P3", Price = 19.50M },
      new Product { Name = "P4", Price = 24.95M }
    };
  }
}
```

Тестовые данные предоставляются через класс, который реализует интерфейс `IEnumerable<object[]>`, возвращающий последовательность массивов объектов. Каждый массив объектов в последовательности содержит один набор аргументов, которые будут передаваться тестовому методу. Мы переопределим тестовый метод, чтобы он принимал массив объектов `Product`, который добавит к тестовым данным еще один уровень — перечисление массивов объектов `Product`. Такая глубина структуры тестовых данных может сбивать с толку, но важно понимать, что тесты не будут работать, если количество аргументов, которые xUnit.net пытается передать тестовому методу, не соответствует сигнатуре метода.

Мне нравится структурировать классы тестовых данных так, чтобы закрытые методы или свойства определяли индивидуальные наборы тестовых данных, которые затем с помощью метода `GetEnumerator()` объединялись в последовательности массивов объектов. Для иллюстрации различных приемов массивы объектов `Product` были созданы с применением и метода, и свойства, но я предпочитаю использовать в своих проектах один подход (на его выбор влияет вид данных, с которыми проводится тестирование). В листинге 7.17 показано, как можно применять класс тестовых данных с атрибутом `Theory` для настройки тестов.

Совет. Если вы хотите включить тестовые данные в тот же самый класс, который определяет модульные тесты, тогда можете использовать атрибут `MemberData` вместо `ClassData`. Атрибут `MemberData` конфигурируется с применением строки, указывающей имя статического метода, который будет предоставлять реализацию `IEnumerable<object[]>`, где каждый массив объектов в последовательности является набором аргументов для тестового метода.

Листинг 7.17. Использование класса тестовых данных в файле HomeControllerTests.cs из папки WorkingWithVisualStudio.Tests

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;
```

```
namespace WorkingWithVisualStudio.Tests {
  public class HomeControllerTests {
    class ModelCompleteFakeRepository : IRepository {
      public IEnumerable<Product> Products { get; set; }
      public void AddProduct(Product p) {
        // Ничего не делать - для теста не требуется
      }
    }
    [Theory]
    [ClassData(typeof(ProductTestData))]
    public void IndexActionModelIsComplete(Product[] products) {
      // Организация
      var controller = new HomeController();
      controller.Repository = new ModelCompleteFakeRepository {
        Products = products
      };
      // Действие
      var model = (controller.Index() as ViewResult)?.ViewData.Model
                as IEnumerable<Product>;
      // Утверждение
      Assert.Equal(controller.Repository.Products, model,
        Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
          && p1.Price == p2.Price));
    }
  }
}
```

Атрибут `ClassData` конфигурируется с типом класса тестовых данных, которым в рассматриваемом случае является `ProductTestData`. Во время выполнения тестов инфраструктура xUnit.net создаст новый экземпляр класса `ProductTestData`, после чего будет применять его для получения последовательности тестовых данных для теста.

На заметку! Если вы посмотрите на список тестов в окне Test Explorer, то увидите, что для тестов `IndexActionModelIsComplete()` предусмотрена единственная запись, хотя класс `ProductTestData` предоставляет два набора тестовых данных. Так происходит в ситуации, когда объекты тестовых данных не удается сериализировать, и проблему можно решить, применив к тестовым объектам атрибут `Serializable`.

Улучшение фиктивных реализаций

Эффективная изоляция компонентов требует фиктивных реализаций классов, чтобы предоставить тестовые данные или проверить, ведет ли себя компонент так, как должен. В предшествующих примерах создавался класс, реализующий интерфейс `IRepository`. Такой подход может быть эффективным, но он приводит к созданию классов реализаций для каждого вида теста, который желательно прогнать. В качестве примера в листинге 7.18 демонстрируется добавление теста, который проверяет, что метод действия `Index()` вызывает метод `Products()` в хранилище только один

раз. (Тест подобного вида распространен, когда имеется опасение, что компонент делает дублирующиеся запросы к хранилищу, становясь причиной множества запросов к базе данных.)

Листинг 7.18. Добавление модульного теста в файле `HomeControllerTests.cs` из папки `WorkingWithVisualStudio.Tests`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;
using System;

namespace WorkingWithVisualStudio.Tests {
  public class HomeControllerTests {

    class ModelCompleteFakeRepository : IRepository {

      public IEnumerable<Product> Products { get; set; }

      public void AddProduct(Product p) {
        // Ничего не делать - для теста не требуется
      }
    }

    [Theory]
    [ClassData(typeof(ProductTestData))]
    public void IndexActionModelIsComplete(Product[] products ) {
      // Организация
      var controller = new HomeController();
      controller.Repository = new ModelCompleteFakeRepository {
        Products = products
      };

      // Действие
      var model = (controller.Index() as ViewResult)?.ViewData.Model
                  as IEnumerable<Product>;

      // Утверждение
      Assert.Equal(controller.Repository.Products, model,
        Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
          && p1.Price == p2.Price));
    }

    class PropertyOnceFakeRepository : IRepository {
      public int PropertyCounter { get; set; } = 0;

      public IEnumerable<Product> Products {
        get {
          PropertyCounter++;
          return new[] { new Product { Name = "P1", Price = 100 } };
        }
      }

      public void AddProduct(Product p) {
        // Ничего не делать - для теста не требуется
      }
    }
```

```
    [Fact]
    public void RepositoryPropertyCalledOnce() {
      // Организация
      var repo = new PropertyOnceFakeRepository();
      var controller = new HomeController { Repository = repo };

      // Действие
      var result = controller.Index();

      // Утверждение
      Assert.Equal(1, repo.PropertyCounter);
    }
  }
}
```

Фиктивные реализации — не всегда простые источники данных; они также могут использоваться для оценки способа, которым компоненты выполняют свою работу. В этом случае было добавлено простое свойство счетчика, которое увеличивается каждый раз, когда читается свойство `Products` фиктивного хранилища, и с помощью метода `Assert.Equal()` выполнена проверка, что обращение к свойству производилось только один раз.

Добавление инфраструктуры имитации

Создание фиктивных объектов такого рода выходит из-под контроля, и лучший способ вернуть себе контроль — применить инфраструктуру фиктивных объектов, также называемую *инфраструктурой имитации*. (Существует формальная разница между фиктивными и имитированными объектами, но для облегчения использования современные инструменты тестирования размывают ее, а потому данные термины будут применяться взаимозаменяемо.) В главе используется инфраструктура Moq, которая описана в табл. 7.4.

Таблица 7.4. Помещение Moq в контекст

Вопрос	Ответ
Что это такое?	Moq — это программный пакет для создания фиктивных реализаций компонентов в приложении
Чем она полезна?	Инфраструктура имитации облегчает создание фиктивных компонентов с целью изоляции частей приложения для модульного тестирования
Как она используется?	Moq применяет лямбда-выражения для определения функциональности фиктивных компонентов и требует определения только тех средств, которые используются при тестировании
Существуют ли какие-то скрытые ловушки или ограничения?	Привыкание к синтаксису может потребовать некоторых усилий. Документация и примеры находятся по адресу `https://github.com/Moq/moq4`
Существуют ли альтернативы?	Доступно несколько альтернативных инфраструктур, в том числе NSubstitute (`https://nsubstitute.github.io`) и FakeItEasy (`https://fakeiteasy.github.io/`). Все они предлагают похожие возможности, и выбор между ними связан только с предпочитаемым вами синтаксисом

Чтобы установить Moq, щелкнем правой кнопкой мыши на проекте `WorkingWithVisualStudio.Tests` в окне Solution Explorer и выберем в контекстном меню пункт Manage NuGet Packages for Solution (Управление пакетами NuGet для решения). Перейдем на вкладку Browse (Обзор) и введем `Moq` в поле поиска. Выберем `Moq` из списка пакетов (рис. 7.8) и щелкнем на кнопке Install (Установить) для добавления пакета в проект.

На заметку! Пакет `Moq` добавляется в проект модульного тестирования, но не в проект приложения.

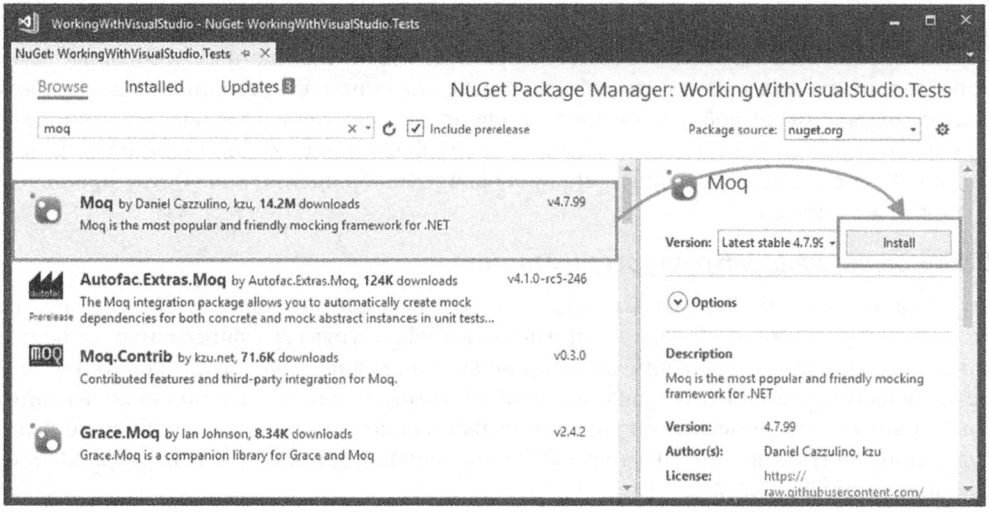


Рис. 7.8. Добавление пакета в проект модульного тестирования

После установки пакета `Moq` средой Visual Studio закроем окно управления пакетами NuGet.

Создание имитированного объекта

Создание имитированного объекта означает необходимость сообщения Moq о том, какого вида объект интересует, конфигурирование его поведения и применение этого объекта к тестируемой сущности. В листинге 7.19 инфраструктура Moq используется для замены двух фиктивных хранилищ в тестах, относящихся к классу `HomeController`.

Листинг 7.19. Применение имитированных объектов в файле `HomeControllerTests.cs` из папки `WorkingWithVisualStudio.Tests`

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WorkingWithVisualStudio.Controllers;
using WorkingWithVisualStudio.Models;
using Xunit;
using System;
```

```
using Moq;

namespace WorkingWithVisualStudio.Tests {
  public class HomeControllerTests {

    [Theory]
    [ClassData(typeof(ProductTestData))]
    public void IndexActionModelIsComplete(Product[] products ) {
      // Организация
      var mock = new Mock<IRepository>();
      mock.SetupGet(m => m.Products).Returns(products);
      var controller = new HomeController { Repository = mock.Object };

      // Действие
      var model = (controller.Index() as ViewResult)?.ViewData.Model
                as IEnumerable<Product>;

      // Утверждение
      Assert.Equal(controller.Repository.Products, model,
        Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
          && p1.Price == p2.Price));
    }

    [Fact]
    public void RepositoryPropertyCalledOnce() {
      // Организация
      var mock = new Mock<IRepository>();
      mock.SetupGet(m => m.Products)
          .Returns(new[] { new Product { Name = "P1", Price = 100 } });
      var controller = new HomeController { Repository = mock.Object };

      // Действие
      var result = controller.Index();

      // Утверждение
      mock.VerifyGet(m => m.Products, Times.Once);
    }
  }
}
```

Использование Moq позволило удалить фиктивные реализации интерфейса `IRepository` и заменить их всего лишь несколькими строками кода. Здесь не приводятся детальные сведения о разных поддерживаемых Moq средствах, но на примерах объясняется способ применения Moq. Примеры и документация по Moq доступны по адресу `https://github.com/Moq/moq4`. (При обсуждении модульного тестирования различных типов компонентов MVC в оставшихся главах книги также будут приводиться примеры.)

Первым делом создается новый объект `Mock` с указанием интерфейса, который должен быть реализован:

```
...
var mock = new Mock<IRepository>();
...
```

Созданный объект `Mock` будет имитировать интерфейс `IRepository`. Далее определяется функциональность, которая требуется для теста. В отличие от обычной реализации интерфейса классом для имитированного объекта конфигурируется только поведение, нужное для теста. В первом имитированном хранилище понадобится

реализовать свойство Products, чтобы оно возвращало набор объектов Product, который передается тестовому методу через атрибут ClassData:

```
...
mock.SetupGet(m => m.Products).Returns(products);
...
```

Метод SetupGet() используется при реализации средства извлечения для свойства. Аргументом этого метода является лямбда-выражение, которое указывает подлежащее реализации свойство (Products в данном примере). Метод Returns() вызывается на результате, полученном из метода SetupGet(), чтобы указать результат, который будет возвращаться, когда читается значение свойства. Для второго имитированного хранилища применяется тот же самый подход, но указывается фиксированное значение:

```
...
mock.SetupGet(m => m.Products)
    .Returns(new[] { new Product { Name = "P1", Price = 100 } });
...
```

В классе Mock определено свойство Object, возвращающее объект, который реализует указанный интерфейс и обладает ранее определенным поведением. В обоих модульных тестах свойство Object используется, чтобы получить хранилище для конфигурирования контроллера:

```
...
var controller = new HomeController { Repository = mock.Object };
...
```

Последним примененным средством Moq была проверка того, что к свойству Products осуществлялось только одно обращение:

```
...
mock.VerifyGet(m => m.Products, Times.Once);
...
```

Метод VerifyGet() относится к тем методам класса Mock, которые инспектируют состояние имитированного объекта, когда тест завершен. В данном случае метод VerifyGet() позволяет проверить, сколько раз читалось свойство Products. Значение Times.Once указывает, что метод VerifyGet() должен генерировать исключение, если свойство читалось не в точности один раз, и это приведет к тому, что тест не пройдет. (Методы класса Assert, обычно используемые в тестах, генерируют исключение, когда тест не проходит, и потому при работе с имитированными объектами метод VerifyGet() можно применять для замены метода класса Assert.)

Резюме

Глава была сконцентрирована на модульном тестировании, которое способно оказаться мощным инструментом, направленным на повышение качества кода. Модульное тестирование не подойдет абсолютно каждому разработчику, но с ним полезно поэкспериментировать, и оно может быть полезным, даже если используется только для сложных функций или обнаружения проблем. Было описано применение инфраструктуры тестирования xUnit.net, объяснена важность изоляции компонентов для целей тестирования и продемонстрированы некоторые инструменты и приемы, позволяющие упростить код модульных тестов. В следующей главе начнется разработка более реалистичного приложения MVC под названием SportsStore.

ГЛАВА 8

SportsStore: реальное приложение

В предшествующих главах мы создавали очень простое приложение MVC. Был описан паттерн MVC, основные средства языка C#, а также инструменты, необходимые профессиональным разработчикам приложений MVC. Наступило время собрать все вместе и построить несложное, но реалистичное приложение электронной коммерции.

Наше приложение под названием SportsStore будет следовать классическому подходу, который повсеместно используется в онлайновых магазинах. Мы создадим онлайновый каталог товаров, который потребители смогут просматривать по категориям и страницам, корзину для покупок, куда пользователи смогут добавлять и удалять товары, и форму оплаты, где потребители смогут вводить сведения, связанные с доставкой. Кроме того, мы создадим административную область, которая включает в себя средства создания, чтения, обновления и удаления (create, read, update, delete — CRUD) для управления каталогом товаров, и защитим ее так, чтобы изменения могли вносить только зарегистрированные администраторы.

Цель настоящей и последующих глав — дать вам возможность увидеть, на что похожа реальная разработка приложений MVC, за счет создания примера приложения, который максимально приближен к реальности. Разумеется, мы будем ориентироваться на ASP.NET Core MVC, поэтому интеграция с внешними системами, такими как база данных, предельно упрощена, а определенные части приложения, например, обработка платежей, вообще отброшены.

Построение всех уровней необходимой инфраструктуры может показаться несколько медленным, но первоначальные трудозатраты при разработке приложения MVC окупаются, обеспечивая удобный в сопровождении, расширяемый и хорошо структурированный код с великолепной поддержкой модульного тестирования.

Модульное тестирование

Я уже достаточно много говорил о легкости проведения модульного тестирования в MVC, а также о том, что модульное тестирование может быть важной и полезной частью процесса разработки. Это будет демонстрироваться на протяжении данной части книги, поскольку я описываю нюансы и приемы модульного тестирования в их взаимосвязи с основными средствами MVC.

Я понимаю, что мое мнение не является единственно верным. Если вы не хотите прибегать к модульному тестированию, то меня это вполне устроит. Таким образом, когда что-то относится исключительно к тестированию, оно будет помещаться во врезку, подобную настоящей. Если модульное тестирование вас не интересует, тогда можете смело пропускать такие врезки, и приложение SportsStore будет работать не менее успешно. Чтобы воспользоваться преимуществами технологии ASP.NET Core MVC, вовсе не обязательно проводить какое-либо модульное тестирование, хотя поддержка тестирования, конечно же, является основной причиной перехода на ASP.NET Core MVC.

Большинству средств MVC, используемых в приложении SportsStore, посвящены отдельные главы далее в книге. Вместо того чтобы повторять весь материал, я сообщаю ровно столько, сколько требуется для понимания этого примера приложения, и указываю главу, в которой можно почерпнуть более подробные сведения.

Каждый шаг, необходимый для построения приложения, будет выделен, что позволит понять, каким образом средства MVC сочетаются друг с другом. Вы должны уделить особое внимание созданию представлений. Если вы не будете в точности следовать примерам, то получите несколько странные результаты.

Начало работы

Если вы планируете писать код приложения SportsStore на своем компьютере во время изучения материала текущей части книги, тогда вам придется установить Visual Studio и удостовериться в том, что установлен вариант LocalDB, который требуется для постоянного хранения данных. Вариант LocalDB устанавливается автоматически в случае следования инструкциям из главы 2.

На заметку! Если вы просто хотите работать с проектом, не воссоздавая его, то можете загрузить готовый проект SportsStore из хранилища GitHub для настоящей книги (`https://github.com/apress/pro-asp.net-core-mvc-2`). Разумеется, вы вовсе не обязаны повторять все действия. Я старался делать снимки экрана и листинги кода максимально простыми в отслеживании на тот случай, если вы читаете книгу в поезде, кафе или где-то еще.

Создание проекта MVC

Мы будем следовать тому же самому базовому подходу, который использовался в предшествующих главах и заключается в том, чтобы начать с пустого проекта и добавлять в него все необходимые конфигурационные файлы и компоненты. Выберем в меню File (Файл) среды Visual Studio пункт New⇨Project (Создать⇨Проект) и укажем шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core), как показано на рис. 8.1. Установим имя проекта в `SportsStore` и щелкнем на кнопке OK.

Выберем шаблон Empty (Пустой), как проиллюстрировано на рис. 8.2. Прежде чем щелкать на кнопке OK для создания проекта `SportsStore`, удостоверимся в том, что в списках в верхней части окна выбраны варианты .NET Core и ASP.NET Core 2.0, а флажок Enable Docker Support (Включить поддержку Docker) не отмечен.

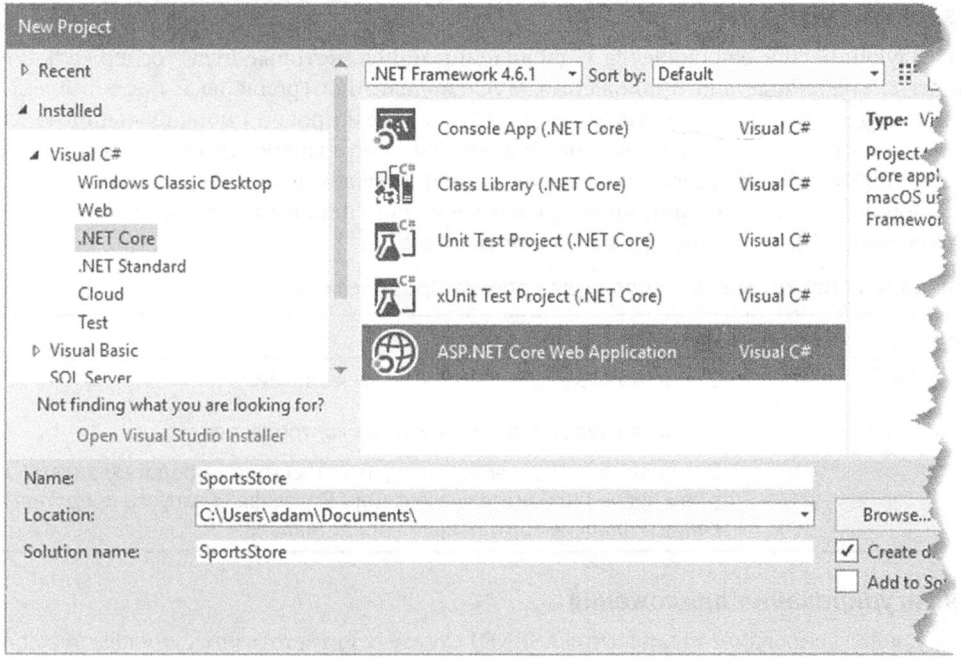


Рис. 8.1. Выбор типа проекта

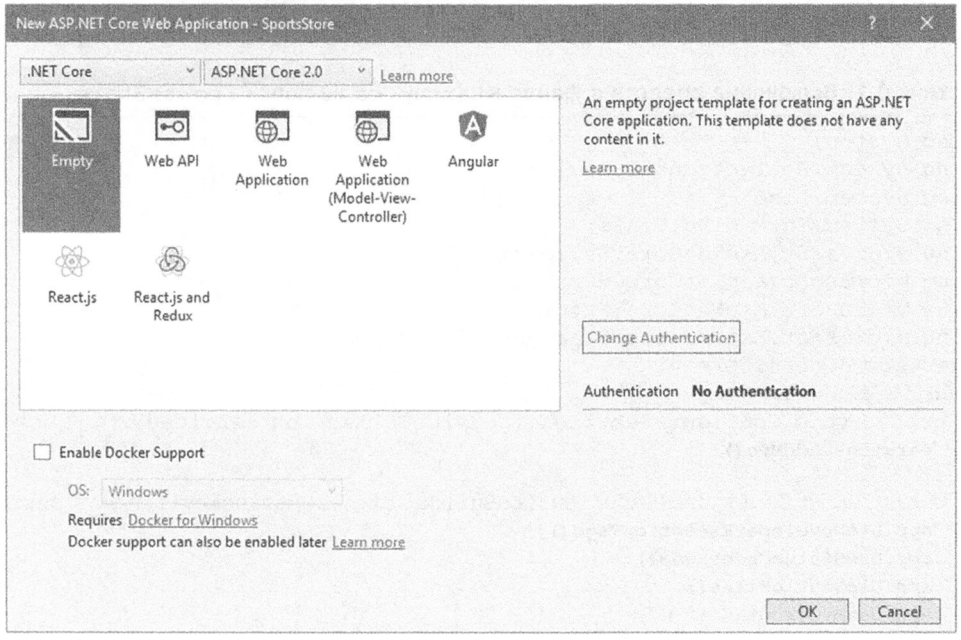


Рис. 8.2. Выбор шаблона проекта

Создание структуры папок

Следующий шаг заключается в добавлении папок, которые будут содержать компоненты, требуемые для приложения MVC: модели, контроллеры и представления. Чтобы создать папки, описанные в табл. 8.1, щелкнем правой кнопкой мыши на элементе проекта `SportsStore` в окне Solution Explorer, выберем в контекстном меню пункт Add⇨New Folder (Добавить⇨Новая папка) и введем имя папки. Позже потребуются дополнительные папки, но создаваемые сейчас папки отражают главные части приложения MVC и для начала их вполне достаточно.

Таблица 8.1. Папки, требующиеся для проекта `SportsStore`

Имя	Описание
`Models`	Эта папка будет содержать классы моделей
`Controllers`	Эта папка будет содержать классы контроллеров
`Views`	Эта папка будет содержать все, что относится к представлениям, в том числе индивидуальные файлы Razor, файл запуска представления и файл импортирования представлений

Конфигурирование приложения

За конфигурирование приложения ASP.NET Core несет ответственность класс `Startup`. В листинге 8.1 приведены изменения, внесенные в класс `Startup` для включения инфраструктуры MVC и нескольких связанных средств, которые полезны при разработке.

На заметку! Класс `Startup` является важным функциональным средством ASP.NET Core и будет подробно описан в главе 14.

Листинг 8.1. Включение средств в файле `Startup.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
namespace SportsStore {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
      });
    }
  }
}
```

Метод `ConfigureServices()` применяется для настройки разделяемых объектов, которые могут использоваться повсеместно в приложении через средство внедрения зависимостей, которое рассматривается в главе 18. Метод `AddMvc()`, вызываемый внутри `ConfigureServices()`, является расширяющим методом, который настраивает разделяемые объекты, применяемые в приложении MVC.

Метод `Configure()` используется для настройки средств, которые получают и обрабатывают HTTP-запросы. Каждый метод, вызываемый в методе `Configure()`, представляет собой расширяющий метод, который настраивает средство обработки HTTP-запросов (табл. 8.2).

Таблица 8.2. Методы настройки первоначальных средств, вызываемые в классе `Startup`

Метод	Описание
`UseDeveloperExceptionPage()`	Этот расширяющий метод отображает детали исключения, которое произошло в приложении, что полезно во время процесса разработки. Он не должен быть включен в развернутых приложениях и при развертывании приложения в главе 12 будет показано, как отключить данное средство
`UseStatusCodePages()`	Этот расширяющий метод добавляет простое сообщение в HTTP-ответы, которые иначе не имели бы тела, такие как ответы `404 - Not Found` (404 — не найдено)
`UseStaticFiles()`	Этот расширяющий метод включает поддержку для обслуживания статического содержимого из папки `wwwroot`
`UseMvc()`	Этот расширяющий метод включает инфраструктуру ASP.NET Core MVC

Далее понадобится подготовить приложение для представлений Razor. Щелкнем правой кнопкой мыши на папке `Views`, выберем в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажем шаблон MVC View Imports Page (Страница импортирования представлений MVC) из категории ASP.NET (рис. 8.3).

Щелкнем на кнопке Add (Добавить), чтобы создать файл `_ViewImports.cshtml`, и приведем его содержимое в соответствие с листингом 8.2.

Листинг 8.2. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@using SportsStore.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Оператор `@using` позволит применять типы из пространства имен `SportsStore.Models` в представлениях, не ссылаясь на это пространство имен. Оператор `@addTagHelper` включает встроенные вспомогательные функции дескрипторов, которые будут использоваться позже для создания элементов HTML, отражающих конфигурацию приложения SportsStore.

Создание проекта модульного тестирования

Создание проекта модульного тестирования требует выполнения того же самого процесса, который был описан в главе 7. Щелкнем правой кнопкой мыши на элементе решения `SportsStore` в окне Solution Explorer и выберем в контекстном меню пункт Add⇨New Project (Добавить⇨Новый проект).

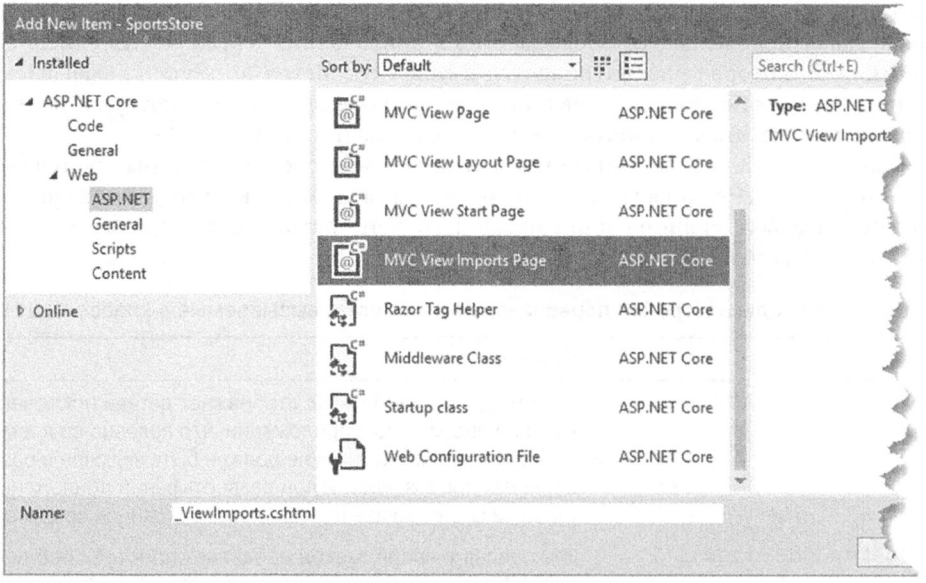


Рис. 8.3. Создание файла импортирования представлений

Укажем шаблон xUnit Test Project (.NET Core) (Проект тестирования xUnit (.NET Core)), как показано на рис. 8.4, и установим имя проекта в `SportsStore.Tests`. Щелкнем на кнопке OK, чтобы создать проект модульного тестирования.

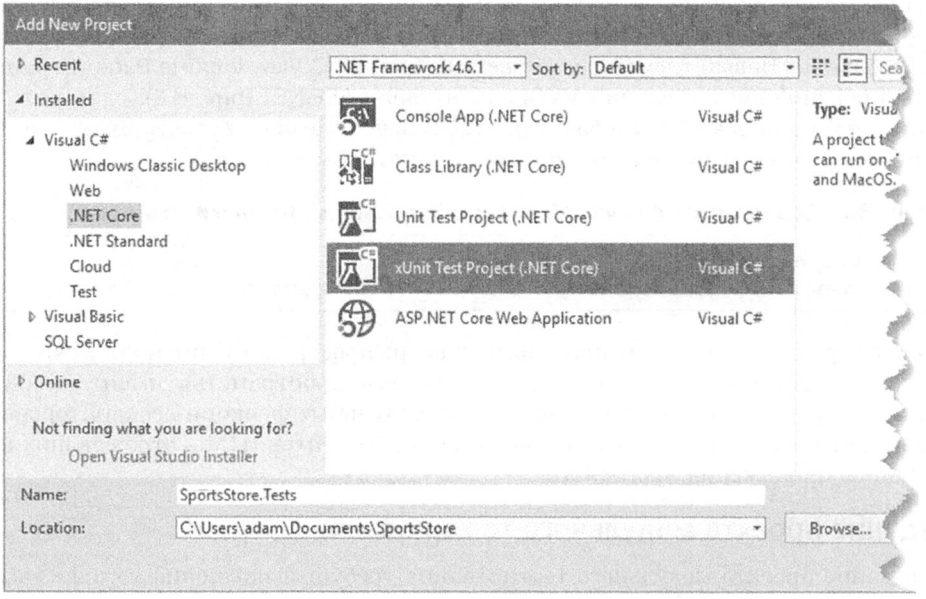


Рис. 8.4. Создание проекта модульного тестирования

После того, как проект модульного тестирования создан, щелкнем правой кнопкой мыши на проекте SportsStore.Tests в окне Solution Explorer и выберем в контекстном меню пункт Edit SportsStore.Tests.csproj (Редактировать SportsStore.Tests.csproj). Внесем в файл SportsStore.Tests.csproj изменения, показанные в листинге 8.3, чтобы добавить пакет Moq в проект SportsStore.Tests и создать ссылку на главный проект SportsStore. Удостоверимся, что для пакета Moq указана та же версия, что и в листинге 8.3.

Листинг 8.3. Добавление пакета в файле SportsStore.Tests.csproj из папки SportsStore.Tests

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>

    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\SportsStore\SportsStore.csproj" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk"
      Version="15.3.0-preview-20170628-02" />
    <PackageReference Include="xunit" Version="2.2.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
    <PackageReference Include="Moq" Version="4.7.99" />
  </ItemGroup>
</Project>
```

Как только изменения будут сохранены, среда Visual Studio загрузит и установит пакет Moq в проект модульного тестирования и создаст ссылку на главный проект SportsStore, так что содержащиеся в нем классы смогут использоваться в тестах.

Проверка и запуск приложения

Проекты приложения и модульного тестирования созданы, сконфигурированы и готовы к дальнейшей разработке. Окно Solution Explorer должно содержать элементы, представленные на рис. 8.5. Если вы видите другие элементы или элементы расположены не в тех позициях, тогда возникнут проблемы, поэтому уделите время проверке, что все элементы присутствуют и находятся на своих местах.

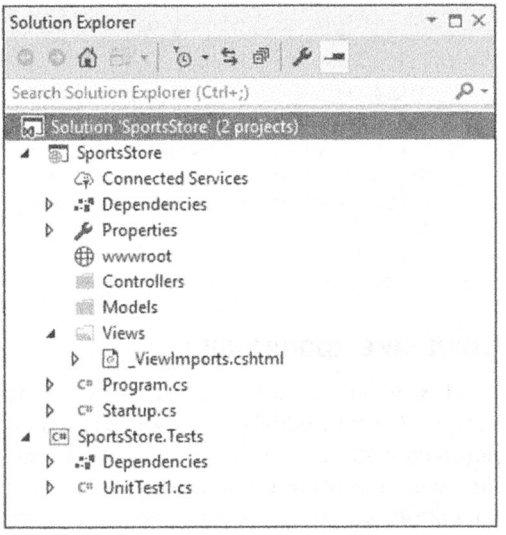


Рис. 8.5. Окно Solution Explorer для проектов приложения SportsStore и модульного тестирования

В результате выбора в меню Debug (Отладка) пункта Start Debugging (Запустить отладку) или Start Without Debugging (Запустить без отладки), если предпочтение отдается итеративному стилю разработки, описанному в главе 6, появится страница ошибки (рис. 8.6). Сообщение об ошибке отображается из-за того, что в настоящий момент в приложении нет контроллеров для обработки запросов; проблему мы вскоре устраним.

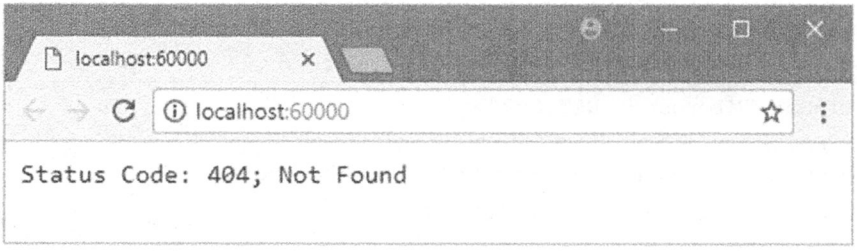


Рис. 8.6. Выполнение приложения SportsStore

Начало работы с моделью предметной области

Все проекты начинаются с создания модели предметной области, которая является центральной частью приложения MVC. Поскольку мы строим приложение электронной коммерции, то наиболее очевидная модель, которая необходима, относится к товару. Добавим в папку `Models` файл класса по имени `Product.cs` с определением, приведенным в листинге 8.4.

Листинг 8.4. Содержимое файла `Product.cs` из папки `Models`

```
namespace SportsStore.Models {
  public class Product {
    public int ProductID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
  }
}
```

Создание хранилища

Нам нужен какой-нибудь способ получения объектов `Product` из базы данных. В главе 3 упоминалось, что модель включает в себя логику для сохранения и извлечения данных из постоянного хранилища. Пока можно не беспокоиться о том, как будет реализовано постоянство данных, но необходимо начать процесс определения интерфейса для него. Добавим в папку `Models` новый файл по имени `IProductRepository.cs` и поместим в него определение интерфейса, показанное в листинге 8.5.

Листинг 8.5. Содержимое файла IProductRepository.cs из папки Models

```
using System.Linq;

namespace SportsStore.Models {

  public interface IProductRepository {
    IQueryable<Product> Products { get; }
  }
}
```

Интерфейс `IProductRepository` использует `IQueryable<T>`, чтобы позволить вызывающему коду получать последовательность объектов `Product`. Интерфейс `IQueryable<T>` является производным от более знакомого интерфейса `IEnumerable<T>` и представляет коллекцию объектов, которые можно запрашивать, вроде тех, что управляются базой данных.

Класс, зависящий от интерфейса `IProductRepository`, способен получать объекты `Product` без необходимости знать детали того, как они хранятся, или каким образом класс реализации будет их доставлять.

Интерфейсы IEnumerable<T> и IQueryable<T>

Интерфейс `IQueryable<T>` удобен, потому что он позволяет эффективно организовать запрашивание коллекции объектов. Позже в главе будет добавлена поддержка для извлечения из базы данных подмножества объектов `Product`. Применение интерфейса `IQueryable<T>` дает возможность запрашивать у базы данных именно те объекты, которые требуются, с помощью стандартных операторов LINQ, не нуждаясь в информации о том, какой сервер баз данных хранит данные или как он обрабатывает запрос. Без интерфейса `IQueryable<T>` пришлось бы извлекать из базы данных все объекты `Product` и затем отбрасывать ненужные, что с ростом объема данных, используемых приложением, превращается в затратную операцию. Именно по этой причине в интерфейсах и классах хранилища в форме базы данных обычно вместо `IEnumerable<T>` применяется `IQueryable<T>`.

Однако во время использования интерфейса `IQueryable<T>` следует соблюдать осторожность, поскольку каждый раз, когда происходит перечисление коллекции объектов, запрос будет оцениваться заново, т.е. базе данных отправится новый запрос. В результате выгоды в плане эффективности от применения `IQueryable<T>` могут быть сведены на нет. В таких ситуациях `IQueryable<T>` можно преобразовать в более предсказуемую форму, используя расширяющий метод `ToList()` или `ToArray()`.

Создание фиктивного хранилища

Теперь, когда определен интерфейс, можно было бы реализовать механизм постоянства и привязать его к базе данных, но сначала необходимо добавить ряд других частей приложения. Для этого мы создадим фиктивную реализацию интерфейса `IProductRepository`, которая будет замещать хранилище данных до тех пор, пока мы им не займемся. Чтобы создать фиктивное хранилище, добавим в папку `Models` файл класса по имени `FakeProductRepository.cs` и поместим в него определение, представленное в листинге 8.6.

Листинг 8.6. Содержимое файла FakeProductRepository.cs из папки Models

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class FakeProductRepository : IProductRepository {

    public IQueryable<Product> Products => new List<Product> {
      new Product { Name = "Football", Price = 25 },
      new Product { Name = "Surf board", Price = 179 },
      new Product { Name = "Running shoes", Price = 95 }
    }.AsQueryable<Product>();
  }
}
```

Класс `FakeProductRepository` реализует интерфейс `IProductRepository`, возвращая в качестве значения свойства `Products` фиксированную коллекцию объектов `Product`. Метод `AsQueryable()` применяется для преобразования фиксированной коллекции объектов в `IQueryable<Product>`, чтобы реализовать интерфейс `IProductRepository` и позволить создавать совместимое фиктивное хранилище, не имея дело с реальными запросами.

Регистрация службы хранилища

В MVC придается особое значение использованию *слабо связанных компонентов*, что означает возможность вносить изменения в одну часть приложения, не внося соответствующие изменения где-то в другом месте. Такой подход трактует части приложения как *службы*, предлагающие функциональные средства, которые задействуются в других частях приложения. Класс, предоставляющий службу, впоследствии может быть модифицирован или заменен, не требуя внесения изменений в классы, в которых он применяется. Подход более подробно объясняется в главе 18, а в случае приложения SportsStore понадобится создать службу хранилища, которая позволит контроллерам получать реализующие интерфейс `IProductRepository` объекты, не зная, какой класс используется. В итоге появится возможность начать разработку приложения с применением простого класса `FakeProductRepository`, созданного в предыдущем разделе, и позже заменить его реальным хранилищем, не внося изменения во все классы, которым необходим доступ в хранилище. Службы регистрируются в методе `ConfigureServices()` класса `Startup`; в листинге 8.7 определена новая служба для хранилища.

Листинг 8.7. Создание службы хранилища в файле Startup.cs из папки SportsStore

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;

namespace SportsStore {
```

```
public class Startup {
  public void ConfigureServices(IServiceCollection services) {
    services.AddTransient<IProductRepository, FakeProductRepository>();
    services.AddMvc();
  }
  public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseMvc(routes => {
    });
  }
}
```

Добавленный в метод `ConfigureServices()` оператор сообщает инфраструктуре ASP.NET Core о том, что когда компоненту вроде контроллера необходима реализация интерфейса `IProductRepository`, она должна получить экземпляр класса `FakeProductRepository`. Метод `AddTransient()` указывает, что каждый раз, когда требуется реализация интерфейса `IProductRepository`, должен создаваться новый объект `FakeProductRepository`. Не беспокойтесь, если смысл кода пока не особенно понятен; вскоре вы увидите, как он вписывается в приложение, а детали того, что происходит, будут представлены в главе 18.

Отображение списка товаров

Оставшийся материал главы можно было бы посвятить построению модели предметной области и хранилища, вообще не касаясь других частей приложения. Однако такой подход выглядит довольно скучным, поэтому мы пойдем другим путем и приступим к интенсивному использованию MVC, вернувшись к добавлению функциональности модели и хранилища, когда она понадобится.

В текущем разделе нам предстоит создать контроллер и метод действия, который может отображать сведения о товарах в хранилище. Пока ими будут только данные из фиктивного хранилища, но позже мы решим эту задачу. Мы также подготовим начальную *конфигурацию маршрутизации*, чтобы инфраструктура MVC знала, каким образом сопоставлять запросы в приложении с контроллером, который будет создан.

Использование инструмента формирования шаблонов MVC в Visual Studio

Везде в книге контроллеры и представления MVC создаются за счет щелчка правой кнопкой мыши на папке в окне Solution Explorer, выбора в контекстном меню пункта Add⇨New Item (Добавить⇨Новый элемент) и указания шаблона элемента в открывшемся диалоговом окне Add New Item (Добавление нового элемента). Существует альтернативный прием, называемый *формированием шаблонов* (scaffolding), при котором среда Visual Studio предлагает в контекстном меню Add (Добавить) пункты, специально предназначенные для создания контроллеров и представлений. После выбора таких пунктов меню предлагается выбрать сценарий для создаваемого компонента, такой как контроллер с действиями, допускающими чтение/запись, или представление, которое содержит форму, применяемую для создания специфического объекта модели.

Формирование шаблонов в книге не используется. Код и разметка, генерируемые инструментом формирования шаблонов, являются настолько общими, что оказываются едва ли не бесполезными, в то время как набор поддерживаемых сценариев ограничен и не решает распространенные задачи разработки. Цель настоящей книги — не только донести до вас знания, каким образом создавать приложения MVC, но также объяснить, как все работает "за кулисами", и сделать это гораздо труднее, когда ответственность за создание компонентов возлагается на инструмент формирования шаблонов.

Тем не менее, мы имеем еще одну ситуацию, когда ваш стиль разработки может отличаться от моего, и в своих проектах вы вполне можете предпочесть работу с инструментом формирования шаблонов. Ваше решение полностью приемлемо, хотя я рекомендую уделить время на выяснение того, что именно инструмент формирования шаблонов делает, чтобы знать, куда затем смотреть, если будут получены не те результаты, которые ожидались.

Добавление контроллера

Чтобы создать первый контроллер в приложении, добавим в папку `Controllers` файл класса по имени `ProductController.cs` с определением, показанным в листинге 8.8.

Листинг 8.8. Содержимое файла `ProductController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

  public class ProductController : Controller {
    private IProductRepository repository;

    public ProductController(IProductRepository repo) {
      repository = repo;
    }
  }
}
```

Когда инфраструктуре MVC необходимо создать новый экземпляр класса `ProductController` для обработки HTTP-запроса, MVC проинспектирует конструктор и выяснит, что он требует объекта, который реализует интерфейс `IProductRepository`. Чтобы определить, какой класс реализации должен использоваться, инфраструктура MVC обращается к конфигурации в классе `Startup`, которая сообщает о том, что нужно применять класс `FakeRepository` и каждый раз создавать его новый экземпляр. Инфраструктура MVC создает новый объект `FakeRepository` и использует его для вызова конструктора `ProductController` с целью создания объекта контроллера, который будет обрабатывать HTTP-запрос.

Такой подход известен под названием *внедрение зависимостей* и позволяет объекту `ProductController` получать доступ к хранилищу приложения через интерфейс `IProductRepository` без необходимости в знании того, какой класс реализации был сконфигурирован. Позже мы заменим фиктивное хранилище реальным, а благодаря внедрению зависимостей контроллер продолжит работать безо всяких изменений.

На заметку! Некоторым разработчикам не нравится внедрение зависимостей, поскольку они считают, что оно усложняет приложения. Я не придерживаюсь такой точки зрения, но если вы не знакомы с внедрением зависимостей, тогда рекомендую вам прочитать главу 18, прежде чем принимать решение.

Далее добавим метод действия по имени `List()`, который будет визуализировать представление, отображающее полный список товаров из хранилища (листинг 8.9).

Листинг 8.9. Добавление метода действия в файле `ProductController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

  public class ProductController : Controller {
    private IProductRepository repository;

    public ProductController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult List() => View(repository.Products);
  }
}
```

Вызов метода `View()` подобного рода (без указания имени представления) сообщает инфраструктуре MVC о том, что нужно визуализировать стандартное представление для метода действия. Передача методу `View()` коллекции объектов `Product` из хранилища снабжает инфраструктуру данными, которыми необходимо заполнить объект `Model` в строго типизированном представлении.

Добавление и конфигурирование представления

Нам требуется создать представление для отображения содержимого пользователю, но предстоит проделать ряд подготовительных шагов, чтобы упростить написание представления. Сначала понадобится создать разделяемую компоновку, в которой будет определено общее содержимое, включаемое во все отправляемые клиентам HTML-ответы. Разделяемые компоновки — удобный способ обеспечить согласованность представлений и наличие в них важных файлов JavaScript и таблиц стилей CSS; их работа объяснялась в главе 5.

Создадим папку `Views/Shared` и добавим в нее новый файл с шаблоном MVC View Layout Page (Страница компоновки представлений MVC) и именем `_Layout.cshtml`, которое является стандартным именем, назначаемым средой Visual Studio элементу такого типа. В листинге 8.10 приведено содержимое файла `_Layout.cshtml`. В стандартное содержимое внесено одно изменение, связанное с установкой внутренностей элемента `title` в `SportsStore`.

Листинг 8.10. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
```

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>SportsStore</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

Затем необходимо сконфигурировать приложение, чтобы файл `_Layout.cshtml` применялся по умолчанию, для чего в папку `Views` добавляется файл с шаблоном MVC View Start Page (Файл запуска представления MVC) и именем `_ViewStart.cshtml`. Стандартное содержимое, добавляемое Visual Studio (листинг 8.11), выбирает компоновку по имени `_Layout.cshtml`, которая соответствует файлу из листинга 8.10.

Листинг 8.11. Содержимое файла `_ViewStart.cshtml` из папки `Views`

```
@{
  Layout = "_Layout";
}
```

Теперь нужно добавить представление, которое будет отображаться, когда для обработки запроса используется метод действия `List()`. Создадим папку `Views/Product` и добавим в нее файл представления Razor по имени `List.cshtml`. Поместим в него разметку, показанную в листинге 8.12.

Листинг 8.12. Содержимое файла `List.cshtml` из папки `Views/Product`

```
@model IEnumerable<Product>

@foreach (var p in Model) {
  <div>
    <h3>@p.Name</h3>
    @p.Description
    <h4>@p.Price.ToString("c")</h4>
  </div>
}
```

Выражение `@model` в начале файла указывает, что представление будет получать от метода действия последовательность объектов `Product` в качестве данных модели. С помощью выражения `@foreach` осуществляется проход по этой последовательности с генерацией простого набора HTML-элементов для каждого полученного объекта `Product`.

Представление не знает, откуда поступили объекты `Product`, как они были получены или охватывают ли они все товары, известные приложению. Взамен представление имеет дело только со способом отображения деталей каждого объекта `Product` с применением HTML-элементов, что согласуется с принципом разделения обязанностей, который был описан в главе 3.

Совет. Значение свойства `Price` преобразуется в строку с использованием метода `ToString("c")`, который визуализирует числовые значения в виде денежных значений в соответствии с настройками культуры, действующими на сервере. Например, если в качестве настройки культуры сервера установлено `en-US`, то вызов `(1002.3).ToString("c")` возвратит значение `$1,002.30`, но если для сервера установлена культура `en-GB`, тогда тот же самый вызов возвратит `£1,002.30`.

Установка стандартного маршрута

Инфраструктуре MVC понадобится сообщить, что она должна отправлять запросы, поступающие для корневого URL приложения (`http://мой-сайт/`), методу действия `List()` класса `ProductController`. Задача решается редактированием оператора в классе `Startup`, который настраивает классы MVC, обрабатывающие HTTP-запросы (листинг 8.13).

Листинг 8.13. Изменение стандартного маршрута в файле `Startup.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;

namespace SportsStore {

  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IProductRepository, FakeProductRepository>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Product}/{action=List}/{id?}");
      });
    }
  }
}
```

Метод `Configure()` класса `Startup` применяется для настройки конвейера запросов, состоящего из классов (известных как *промежуточное программное обеспечение*), которые будут инспектировать HTTP-запросы и генерировать ответы.

Метод `UseMvc()` настраивает промежуточное программное обеспечение MVC, причем одним из параметров конфигурации является схема, которая будет использоваться для сопоставления URL с контроллерами и методами действий. Система маршрутизации подробно рассматривается в главах 15 и 16, а пока просто следует знать, что изменения, выделенные в листинге 8.13, сообщают инфраструктуре MVC о необходимости отправки запросов методу действия `List()` контроллера `Product`, если только в URL запроса не указано иное.

Совет. Обратите внимание, что в листинге 8.13 имя контроллера установлено в `Product`, а не `ProductController`, являющееся именем класса. Это часть соглашения об именовании MVC, в рамках которого имена классов контроллеров обычно заканчиваются словом `Controller`, но при ссылке на класс данная часть имени опускается. Соглашение об именовании и его влияние объясняются в главе 31.

Запуск приложения

Все основные компоненты в наличии. Мы имеем контроллер с методом действия, который MVC будет применять, когда запрашивается стандартный URL для приложения. Инфраструктура MVC создаст экземпляр класса `FakeRepository`, после чего будет использовать его для создания нового объекта контроллера, обрабатывающего запрос. Фиктивное хранилище снабдит контроллер простыми тестовыми данными, которые его метод действия передаст представлению Razor, так что HTML-ответ для браузера будет включать детали каждого объекта `Product`. При генерации HTML-ответа инфраструктура MVC объединит данные из представления, выбранные методом действия, с содержимым разделяемой компоновки, порождая завершенный HTML-документ, который браузер в состоянии разобрать и отобразить. Запустив приложение, можно увидеть результат, показанный на рис. 8.7.

Итак, мы исследовали типовой шаблон разработки для инфраструктуры ASP.NET Core MVC. Начальные затраты времени на необходимую настройку являются обязательными, но затем основные средства приложения будут собираться очень быстро.

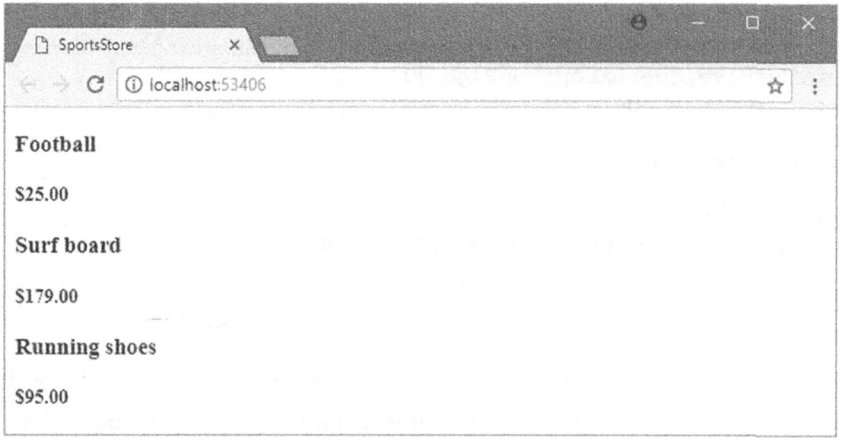


Рис. 8.7. Просмотр основной функциональности приложения

Подготовка базы данных

Мы способны отображать простое представление, содержащее сведения о товарах, но применяем тестовые данные, которые находятся в фиктивном хранилище. Прежде чем можно будет реализовать хранилище с реальными данными, необходимо настроить базу данных и заполнить ее данными.

Для базы данных будет использоваться SQL Server, а доступ к ней будет осуществляться с применением Entity Framework Core (EF Core) — инфраструктуры объектно-реляционного отображения (object-relational mapping — ORM) для Microsoft .NET. Инфраструктура ORM представляет таблицы, столбцы и строки реляционной базы данных посредством обычных объектов C#.

На заметку! Это область, где можно выбирать из широкого диапазона инструментальных средств и технологий. Доступны не только различные реляционные базы данных, но можно также работать с хранилищами объектов, хранилищами документов и рядом экзотических альтернатив. Кроме того, существуют другие инфраструктуры ORM для .NET, каждая из которых использует слегка отличающийся подход; такие вариации позволяют выбрать то, что лучше всего подходит для конкретных проектов.

Инфраструктура Entity Framework Core применяется по нескольким причинам: ее просто запустить в работу, она прекрасно интегрируется с LINQ (а мне нравится использовать LINQ) и она хорошо работает с ASP.NET Core MVC. Ранним выпускам EF Core были присущи мелкие недостатки, но текущие версии элегантны и богаты возможностями.

Продукт SQL Server располагает удобным средством *LocalDB*, которое представляет собой не требующую администрирования реализацию основной функциональности SQL Server, специально спроектированную для разработчиков. Благодаря LocalDB можно пропускать процесс настройки базы данных на время построения проекта, а развертывание проводить в полном экземпляре SQL Server позже. Большинство приложений MVC развертываются в размещаемых средах, которые обслуживаются профессиональными администраторами, и наличие средства LocalDB означает, что конфигурирование баз данных остается в руках администраторов баз данных, позволяя разработчикам приложений заниматься написанием кода.

Совет. Если при установке Visual Studio вы не выбрали LocalDB, то придется сделать это сейчас. Средство LocalDB может быть выбрано через раздел Individual Components (Индивидуальные компоненты) в программе установки Visual Studio. Если вы следовали инструкциям, приведенным в главе 2, тогда средство LocalDB должно быть установлено и готово к применению.

Установка пакета инструментов командной строки Entity Framework Core

Основная функциональность Entity Framework Core добавляется в проект по умолчанию, когда Visual Studio создает проект. Нам необходим один дополнительный пакет NuGet, предоставляющий инструменты командной строки, которые используются для создания классов, готовящих базу данных к хранению данных приложения. Такие классы известны как *миграции*.

Чтобы добавить пакет в проект, щелкнем правой кнопкой мыши на элементе проекта SportsStore в окне Solution Explorer, выберем в контекстном меню пункт Edit SportsStore.csproj (Редактировать SportsStore.csproj) и внесем в файл SportsStore.csproj изменения, показанные в листинге 8.14. Важно применять версию, указанную в листинге. Кроме того, пакет добавляется с использованием элемента DotNetCliToolReference, а не PackageReference, который предназначен для добавления существующего пакета.

На заметку! Пакет инструментов командной строки Entity Framework Core должен устанавливаться за счет редактирования файла SportsStore.csproj. Пакеты такого типа не могут быть добавлены с применением диспетчера пакетов NuGet либо инструмента командной строки dotnet.

Листинг 8.14. Добавление пакета в файле SportsStore.csproj из папки SportsStore

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
      Version="2.0.0" />
  </ItemGroup>

</Project>
```

После сохранения файла среда Visual Studio загрузит и установит пакет инструментов командной строки Entity Framework Core, а также добавит их в проект.

Создание классов базы данных

Класс контекста базы данных — это шлюз между приложением и EF Core, который обеспечивает доступ к данным приложения с использованием объектов моделей. Чтобы создать класс контекста базы данных для приложения SportsStore, добавим в папку Models файл класса по имени ApplicationDbContext.cs с определением, приведенным в листинге 8.15.

Листинг 8.15. Содержимое файла ApplicationDbContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {

  public class ApplicationDbContext : DbContext {
```

```
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
      : base(options) { }

    public DbSet<Product> Products { get; set; }
  }
}
```

Базовый класс `DbContext` предоставляет доступ к лежащей в основе функциональности Entity Framework Core, а свойство `Products` обеспечивает доступ к объектам `Product` в базе данных. Класс `ApplicationDbContext` является производным от `DbContext` и добавляет свойства, которые будут применяться для чтения и записи данных приложения. В текущий момент имеется только одно свойство, которое предоставит доступ к объектам.

Создание класса хранилища

Хотя в настоящее время может показаться иначе, но большая часть работы, требуемой для настройки базы данных, завершена. Следующий шаг заключается в создании класса, который реализует интерфейс `IProductRepository` и получает данные с использованием инфраструктуры Entity Framework Core. Добавим в папку `Models` файл класса по имени `EFProductRepository.cs` с определением класса хранилища, представленным в листинге 8.16.

Листинг 8.16. Содержимое файла `EFProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

    public EFProductRepository(ApplicationDbContext ctx) {
      context = ctx;
    }

    public IQueryable<Product> Products => context.Products;
  }
}
```

По мере добавления средств к приложению в класс будет добавляться соответствующая функциональность, а пока реализация хранилища просто отображает свойство `Products`, определенное в интерфейсе `IProductRepository`, на свойство `Products`, которое определено в классе `ApplicationDbContext`. Свойство `Products` в классе контекста возвращает объект `DbSet<Product>`, который реализует интерфейс `IQueryable<T>` и облегчает реализацию интерфейса `IProductRepository`, когда применяется Entity Framework Core. В результате гарантируется, что запросы к базе данных будут извлекать только те объекты, которые требуются, как объяснялось ранее в главе.

Определение строки подключения

Строка подключения указывает местоположение и имя базы данных, а также предоставляет конфигурационные настройки, с которыми приложение должно под-

ключаться к серверу базы данных. Строки подключения хранятся в файле JSON под названием `appsettings.json`, который создан в проекте `SportsStore` с использованием шаблона элемента ASP.NET Configuration File (Конфигурационный файл ASP.NET) из раздела General (Общие) диалогового окна Add New Item.

При создании файла `appsettings.json` среда Visual Studio добавляет в него заполнитель для строки подключения, который необходимо заменить содержимым листинга 8.17.

Совет. Строка подключения должна быть выражена как единственная неразрывная строка, что выглядит нормально в редакторе Visual Studio, но не умещается на печатной странице, чем и объясняется неуклюжее форматирование в листинге 8.17. При определении строки подключения в собственном проекте удостоверьтесь, что значение элемента `ConnectionString` находится в одной строке.

Листинг 8.17. Редактирование строки подключения в файле `appsettings.json` из папки `SportsStore`

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;
      Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}
```

Внутри раздела `Data` конфигурационного файла имя строки подключения устанавливается в `SportStoreProducts`. Значение элемента `ConnectionString` указывает, что для базы данных по имени `SportsStore` должно применяться средство LocalDB.

Конфигурирование приложения

Далее необходимо прочитать строку подключения и сконфигурировать приложение для ее использования при подключении к базе данных. В листинге 8.18 показаны изменения, которые потребуется внести в класс `Startup`, чтобы получать детали данных конфигурации, содержащихся в файле `appsettings.json`, и применять их для конфигурирования Entity Framework Core. (Работа, связанная с чтением файла JSON, обеспечивается классом `Program`, который будет описан в главе 14.)

Листинг 8.18. Конфигурирование приложения в файле `Startup.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
```

```
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreProducts:ConnectionString"]));
      services.AddTransient<IProductRepository, EFProductRepository>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Product}/{action=List}/{id?}");
      });
    }
  }
}
```

Добавленный в класс `Startup` конструктор получает данные конфигурации из файла `appsettings.json`, которые представлены посредством объекта, реализующего интерфейс `IConfiguration`. Конструктор присваивает объект реализации `IConfiguration` свойству по имени `Configuration`, так что он может использоваться в остальном коде класса `Startup`.

В главе 14 будет показано, как читать и обращаться к данным конфигурации. Для приложения SportsStore мы добавили в метод `ConfigureServices()` последовательность вызовов методов, которая настраивает Entity Framework Core:

```
...
services.AddDbContext<ApplicationDbContext>(options =>
  options.UseSqlServer(
    Configuration["Data:SportStoreProducts:ConnectionString"]));
...
```

Расширяющий метод `AddDbContext()` настраивает службы, предоставляемые инфраструктурой Entity Framework Core, для контекста базы данных, который был создан в листинге 8.15. Как объясняется в главе 14, многие методы, применяемые в классе `Startup`, позволяют конфигурировать службы и средства промежуточного программного обеспечения с использованием аргументов `options`. Аргументом метода `AddDbContext()` является лямбда-выражение, которое получает объект `options`, конфигурирующий базу данных для класса контекста. В этом случае база данных конфигурируется с помощью метода `UseSqlServer()` и указания строки подключения, которая получена из свойства `Configuration`.

Еще одним изменением, внесенным в класс Startup, была замена фиктивного хранилища реальным:

```
...
services.AddTransient<IProductRepository, EFProductRepository>();
...
```

Компоненты в приложении, работающие с интерфейсом IProductRepository, к которым в настоящий момент относится только контроллер Product, при создании будут получать объект EFProductRepository, предоставляющий им доступ к информации в базе данных. Подробные объяснения приводятся в главе 18, а пока просто знайте, что результатом будет гладкая замена фиктивных данных реальными из базы данных без необходимости в изменении класса ProductController.

Отключение проверки области видимости

Применение Entity Framework Core требует внесения изменения в конфигурацию средства внедрения зависимостей, которое рассматривается в главе 18. Класс Program несет ответственность за запуск и конфигурирование ASP.NET Core перед передачей управления классу Startup, и в листинге 8.19 показано необходимое изменение. Без такого изменения попытка создать схему базы данных в следующем разделе приведет к генерации исключения.

Листинг 8.19. Подготовка для работы с Entity Framework Core в файле Program.cs из папки SportsStore

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace SportsStore {
  public class Program {
    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseDefaultServiceProvider(options =>
         options.ValidateScopes = false)
        .Build();
  }
}
```

Конфигурирование ASP.NET Core подробно объясняется в главе 14, а просто отметим, что это единственное изменение, которое требуется для приложения SportsStore.

Создание миграции базы данных

Инфраструктура Entity Framework Core способна генерировать схему для базы данных, используя классы моделей, с помощью средства, которое называется *миграциями*. При подготовке миграции инфраструктура EF Core создает класс C#, содержащий команды SQL, которые нужны для подготовки базы данных. Если необходимо модифицировать классы моделей, тогда можно создать новую миграцию, которая содержит команды SQL, требуемые для отражения изменений. В результате не приходится беспокоиться о написании вручную и тестировании команд SQL и можно сосредоточиться на классах модели C# в приложении.

Команды Entity Framework Core выполняются из командной строки. Откроем окно командной строки или окно PowerShell, перейдем в папку проекта `SportsStore` (ту, что содержит файлы `Startup.cs` и `appsettings.json`) и запустим следующую команду, чтобы создать класс миграции, который подготовит базу данных к первому использованию:

```
dotnet ef migrations add Initial
```

Когда команда завершит свое выполнение, в окне Solution Explorer среды Visual Studio появится папка `Migrations`. Именно в ней инфраструктура Entity Framework Core хранит классы миграции. Одно из имен файлов будет выглядеть как отметка времени, за которой следует `_Initial.cs`, и в этом файле определен класс, применяемый для создания начальной схемы базы данных. Просмотрев содержимое такого файла, можно выяснить, каким образом класс модели `Product` использовался для создания схемы.

Что насчет команд `Add-Migration` и `Update-Database`?

Если вы имеете опыт разработки с помощью Entity Framework, то могли привыкнуть к применению команды `Add-Migration` для создания миграции базы данных и команды `Update-Database` для ее применения к базе данных.

С появлением платформы .NET Core в инфраструктуру Entity Framework Core были добавлены команды, интегрированные в инструмент командной строки `dotnet`, который использует пакет `Microsoft.EntityFrameworkCore.Tools.DotNet`, добавленный к проекту в листинге 8.14. Такие команды применяются в настоящей главе, поскольку они согласуются с другими командами .NET и могут использоваться в любом окне командной строки или PowerShell в отличие от команд `Add-Migration` и `Update-Database`, которые работают только в специальном окне Visual Studio.

Создание начальных данных

Чтобы заполнить базу данных и предоставить какие-то тестовые данные, добавим в папку `Models` файл класса по имени `SeedData.cs` с определением, приведенным в листинге 8.20.

Листинг 8.20. Содержимое файла `SeedData.cs` из папки `Models`

```
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {
```

```
public static class SeedData {
  public static void EnsurePopulated(IApplicationBuilder app) {
    ApplicationDbContext context = app.ApplicationServices
      .GetRequiredService<ApplicationDbContext>();
    context.Database.Migrate();
    if (!context.Products.Any()) {
      context.Products.AddRange(
        new Product {
          Name = "Kayak",
          Description = "A boat for one person",
          Category = "Watersports", Price = 275 },
        new Product {
          Name = "Lifejacket",
          Description = "Protective and fashionable",
          Category = "Watersports", Price = 48.95m },
        new Product {
          Name = "Soccer Ball",
          Description = "FIFA-approved size and weight",
          Category = "Soccer", Price = 19.50m },
        new Product {
          Name = "Corner Flags",
          Description = "Give your playing field a professional touch",
          Category = "Soccer", Price = 34.95m },
        new Product {
          Name = "Stadium",
          Description = "Flat-packed 35,000-seat stadium",
          Category = "Soccer", Price = 79500 },
        new Product {
          Name = "Thinking Cap",
          Description = "Improve brain efficiency by 75%",
          Category = "Chess", Price = 16 },
        new Product {
          Name = "Unsteady Chair",
          Description = "Secretly give your opponent a disadvantage",
          Category = "Chess", Price = 29.95m },
        new Product {
          Name = "Human Chess Board",
          Description = "A fun game for the family",
          Category = "Chess", Price = 75 },
        new Product {
          Name = "Bling-Bling King",
          Description = "Gold-plated, diamond-studded King",
          Category = "Chess", Price = 1200
        }
      );
      context.SaveChanges();
    }
  }
}
```

Статический метод `EnsurePopulated()` получает аргумент типа `IApplication Builder`, представляющий собой интерфейс, который применяется в методе `Configure()` класса `Startup` для регистрации компонентов промежуточного про-

граммного обеспечения, обрабатывающих HTTP-запросы, и именно здесь будет обеспечено наличие в базе данных содержимого.

Метод `EnsurePopulated()` получает объект `ApplicationDbContext` через интерфейс `IApplicationBuilder` и вызывает метод `Database.Migrate()`, чтобы гарантировать применение миграции, что означает создание и подготовку базы данных к хранению объектов `Product`. Далее проверяется количество объектов `Product` в базе данных. Если объекты в базе данных отсутствуют, то база данных заполняется с использованием коллекции объектов `Product` посредством метода `AddRange()` и затем записывается с помощью метода `SaveChanges()`.

Финальное изменение заключается в заполнении базы данных начальной информацией при запуске приложения, для чего в класс `Startup` добавляется вызов метода `EnsurePopulated()`, как показано в листинге 8.21.

Листинг 8.21. Заполнение базы данных начальной информацией в файле `Startup.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

  public class Startup {
    public Startup(IConfiguration configuration) =>
      Configuration = configuration;
    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreProducts:ConnectionString"]));
      services.AddTransient<IProductRepository, EFProductRepository>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Product}/{action=List}/{id?}");
      });
      SeedData.EnsurePopulated(app);
    }
  }
}
```

В результате запуска приложения база данных создается, заполняется и применяется для обеспечения приложения своими данными. (Запаситесь терпением; создание базы данных может занять некоторое время).

Когда браузер запрашивает стандартный URL для приложения, конфигурация приложения сообщает MVC о необходимости создания контроллера `Product` для обработки запроса. Создание контроллера `Product` означает вызов конструктора класса `ProductController`, которому требуется объект, реализующий интерфейс `IProductRepository`, и новая конфигурация указывает MVC о том, что для этого должен быть создан и применен объект `EFProductRepository`. Объект `EFProductRepository` обращается к функциональности Entity Framework Core, которая загружает данные из SQL Server и преобразует их в объекты `Product`. Вся упомянутая работа скрыта от класса `ProductController`, который просто получает объект, реализующий интерфейс `IProductRepository`, и пользуется данными, которые он предоставляет. В итоге окно браузера отображает тестовую информацию из базы данных (рис. 8.8).

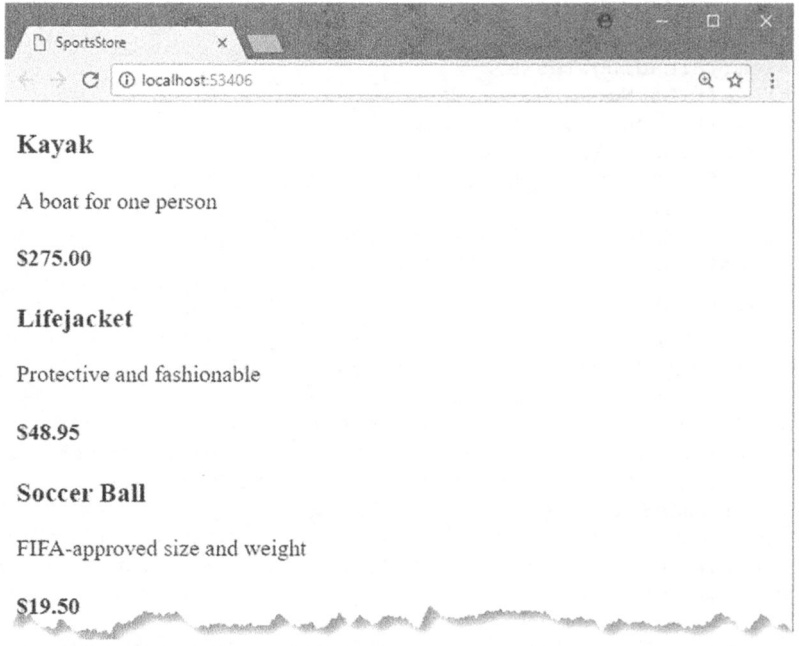


Рис. 8.8. Использование хранилища в виде базы данных

Такой подход с применением Entity Framework Core для представления базы данных SQL Server в виде последовательности объектов моделей отличается простотой и легкостью, позволяя сосредоточить внимание на инфраструктуре ASP.NET Core MVC. Я опустил множество деталей, касающихся оперирования EF Core, и большое количество доступных конфигурационных параметров. Мне нравится инфраструктура Entity Framework Core, и я рекомендую уделить время на ее изучение. Хорошей отправной точкой послужит веб-сайт Microsoft для Entity Framework Core (`https://docs.microsoft.com/ru-ru/ef/#pivot=efcore`) или моя будущая книга по Entity Framework Core, которая выйдет в издательстве Apress.

Добавление поддержки разбиения на страницы

На рис. 8.8 видно, что представление `List.cshtml` отображает сведения о товарах в базе данных на одной странице. В этом разделе мы добавим поддержку разбиения на страницы, чтобы представление отображало на странице меньшее число товаров, а пользователь мог переходить со страницы на страницу для просмотра всего каталога. Для решения задачи добавим в метод действия `List()` контроллера `Product` параметр (листинг 8.22).

Листинг 8.22. Добавление поддержки разбиения на страницы в файле `ProductController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

  public class ProductController : Controller {
    private IProductRepository repository;
    public int PageSize = 4;

    public ProductController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult List(int productPage = 1)
      => View(repository.Products
        .OrderBy(p => p.ProductID)
        .Skip((productPage - 1) * PageSize)
        .Take(PageSize));
  }
}
```

Поле `PageSize` указывает, что на одной странице должны отображаться сведения о четырех товарах. В метод `List()` добавлен необязательный параметр, который означает, что в случае вызова метода без параметра (`List()`) вызов трактуется так, словно ему было передано значение, указанное в определении параметра (`List(1)`). В результате метод действия отображает первую страницу сведений о товарах, когда инфраструктура MVC вызывает его без аргумента. Внутри тела метода действия мы получаем объекты `Product`, упорядочиваем их по первичному ключу, пропускаем товары, которые располагаются до начала текущей страницы, и выбираем количество товаров, указанное в поле `PageSize`.

Модульное тестирование: разбиение на страницы

Модульное тестирование средства разбиения на страницы можно провести, создав имитированное хранилище, внедрив его в конструктор класса `ProductController` и вызвав метод `List()`, чтобы запросить конкретную страницу. Затем полученные объекты `Product` можно сравнить с теми, которые ожидались от тестовых данных в имитированной реализации. Написание модульных тестов обсуждалось в главе 7. Ниже показан созданный для этой цели модульный тест, который находится в файле класса по имени `ProductControllerTests.cs`, добавленном в проект `SportsStore.Tests`:

```
using System.Collections.Generic;
using System.Linq;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

  public class ProductControllerTests {

    [Fact]
    public void Can_Paginate() {
      // Организация
      Mock<IProductRepository> mock = new Mock<IProductRepository>();
      mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
      }).AsQueryable<Product>());

      ProductController controller = new ProductController(mock.Object);
      controller.PageSize = 3;

      // Действие
      IEnumerable<Product> result =
        controller.List(2).ViewData.Model as IEnumerable<Product>;

      // Утверждение
      Product[] prodArray = result.ToArray();
      Assert.True(prodArray.Length == 2);
      Assert.Equal("P4", prodArray[0].Name);
      Assert.Equal("P5", prodArray[1].Name);
    }
  }
}
```

Получение данных, возвращаемых из метода действия, выглядит несколько неуклюже. Результатом является объект `ViewResult`, и значение его свойства `ViewData.Model` должно быть приведено к ожидаемому типу данных. В главе 17 рассматриваются разнообразные результирующие типы, которые могут возвращать методы действий, а также способы работы с ними.

Отображение ссылок на страницы

Запустив приложение, вы увидите, что теперь на странице отображаются четыре позиции каталога. Если нужно просмотреть другую страницу, тогда в конец URL можно добавить параметр строки запроса, например:

```
http://localhost:5000/?productPage=2
```

Вы должны указать в URL номер порта, который был назначен вашему проекту. Используя такие строки запросов, можно перемещаться по каталогу товаров.

У пользователей нет какого-либо способа выяснить о существовании таких параметров строки запроса, но даже если бы он был, то вряд ли бы они захотели иметь

дело с навигацией подобного вида. Взамен мы должны визуализировать в нижней части каждого списка товаров ссылки на страницы, чтобы пользователи могли переходить между страницами. Мы реализуем вспомогательную функцию дескриптора, которая будет генерировать HTML-разметку для требуемых ссылок.

Добавление модели представления

Чтобы обеспечить поддержку вспомогательной функции дескриптора, мы собираемся передавать представлению информацию о количестве доступных страниц, текущей странице и общем числе товаров в хранилище. Проще всего это сделать, создав класс модели представления, который специально применяется для передачи данных между контроллером и представлением. Создадим в проекте `SportsStore` папку `Models/ViewModels` и добавим в нее файл класса с содержимым, приведенным в листинге 8.23.

Листинг 8.23. Содержимое файла `PagingInfo.cs` из папки `Models/ViewModels`

```
using System;

namespace SportsStore.Models.ViewModels {

  public class PagingInfo {
    public int TotalItems { get; set; }
    public int ItemsPerPage { get; set; }
    public int CurrentPage { get; set; }

    public int TotalPages =>
      (int)Math.Ceiling((decimal)TotalItems / ItemsPerPage);
  }
}
```

Добавление класса вспомогательной функции дескриптора

Теперь, имея модель представления, можно создать вспомогательную функцию дескриптора. Создадим в проекте `SportsStore` папку `Infrastructure` и добавим в нее файл класса по имени `PageLinkTagHelper.cs` с определением, показанным в листинге 8.24. Вспомогательные функции дескрипторов являются крупной частью инфраструктуры ASP.NET Core MVC, и в главах 23–25 объясняется, как они работают и каким образом их создавать.

Совет. В папку `Infrastructure` будут помещаться классы, которые предоставляют приложению связующий код, но не имеют отношения к предметной области приложения.

Листинг 8.24. Содержимое файла `PageLinkTagHelper.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;
```

```
namespace SportsStore.Infrastructure {
  [HtmlTargetElement("div", Attributes = "page-model")]
  public class PageLinkTagHelper : TagHelper {
    private IUrlHelperFactory urlHelperFactory;

    public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
      urlHelperFactory = helperFactory;
    }

    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewContext { get; set; }

    public PagingInfo PageModel { get; set; }

    public string PageAction { get; set; }

    public override void Process(TagHelperContext context,
          TagHelperOutput output) {
      IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
      TagBuilder result = new TagBuilder("div");
      for (int i = 1; i <= PageModel.TotalPages; i++) {
        TagBuilder tag = new TagBuilder("a");
        tag.Attributes["href"] = urlHelper.Action(PageAction,
          new { productPage = i });
        tag.InnerHtml.Append(i.ToString());
        result.InnerHtml.AppendHtml(tag);
      }
      output.Content.AppendHtml(result.InnerHtml);
    }
  }
}
```

Класс вспомогательной функции дескриптора `PageLinkTagHelper` заполняет элемент `div` элементами `a`, которые соответствуют страницам товаров. Сейчас мы не будем вдаваться в какие-либо детали относительно вспомогательных функций дескрипторов; достаточно знать, что они предлагают один из наиболее удобных способов помещения логики C# в представления.

Код для вспомогательной функции дескриптора может выглядеть запутанным, потому что смешивать C# и HTML непросто. Но использование вспомогательных функций дескрипторов является предпочтительным способом включения блоков кода C# в представление, поскольку вспомогательную функцию дескриптора можно легко подвергать модульному тестированию.

Большинство компонентов MVC, таких как контроллеры и представления, обнаруживаются автоматически, но вспомогательные функции дескрипторов должны быть зарегистрированы.

В листинге 8.25 приведено содержимое файла `_ViewImports.cshtml` из папки `Views` с добавленным оператором, который сообщает MVC о том, что вспомогательные функции дескрипторов следует искать в пространстве имен `SportsStore.Infrastructure`. Кроме того, добавлено также выражение `@using`, чтобы на классы моделей представлений можно было ссылаться в представлениях, не указывая пространство имен.

Листинг 8.25. Регистрация вспомогательной функции дескриптора в файле _ViewImports.cshtml из папки Views/Home

```
@using SportsStore.Models
@using SportsStore.Models.ViewModels
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper SportsStore.Infrastructure.*, SportsStore
```

Модульное тестирование: создание ссылок на страницы

Чтобы протестировать класс вспомогательной функции дескриптора `PageLinkTagHelper`, вызывается метод `Process()` с тестовыми данными и предоставляется объект `TagHelperOutput`, который инспектируется на предмет сгенерированной HTML-разметки. Тест определен в новом файле `PageLinkTagHelperTests.cs` внутри проекта `SportsStore.Tests`:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Moq;
using SportsStore.Infrastructure;
using SportsStore.Models.ViewModels;
using Xunit;

namespace SportsStore.Tests {

  public class PageLinkTagHelperTests {

    [Fact]
    public void Can_Generate_Page_Links() {
      // Организация
      var urlHelper = new Mock<IUrlHelper>();
      urlHelper.SetupSequence(x => x.Action(It.IsAny<UrlActionContext>()))
        .Returns("Test/Page1")
        .Returns("Test/Page2")
        .Returns("Test/Page3");

      var urlHelperFactory = new Mock<IUrlHelperFactory>();
      urlHelperFactory.Setup(f =>
        f.GetUrlHelper(It.IsAny<ActionContext>()))
          .Returns(urlHelper.Object);

      PageLinkTagHelper helper =
        new PageLinkTagHelper(urlHelperFactory.Object) {
          PageModel = new PagingInfo {
            CurrentPage = 2,
            TotalItems = 28,
            ItemsPerPage = 10
          },
          PageAction = "Test"
        };
        TagHelperContext ctx = new TagHelperContext(
          new TagHelperAttributeList(),
          new Dictionary<object, object>(), "");
```

```
        var content = new Mock<TagHelperContent>();
        TagHelperOutput output = new TagHelperOutput("div",
          new TagHelperAttributeList(),
          (cache, encoder) => Task.FromResult(content.Object));
        // Действие
        helper.Process(ctx, output);
        // Утверждение
        Assert.Equal(@"<a href=""Test/Page1"">1</a>"
          + @"<a href=""Test/Page2"">2</a>"
          + @"<a href=""Test/Page3"">3</a>",
          output.Content.GetContent());
    }
  }
}
```

Сложность теста связана с построением объектов, которые требуются для создания и применения вспомогательной функции дескриптора. Вспомогательные функции дескрипторов используют объекты реализаций `IUrlHelperFactory` для генерации URL, которые указывают на разные части приложения. Для создания реализаций интерфейса `IUrlHelperFactory` и связанного с ним `IUrlHelper`, предоставляющего тестовые данные, используется инфраструктура Moq.

Основная часть теста проверяет вывод вспомогательной функции дескриптора с применением литерального строкового значения, которое содержит двойные кавычки. Язык C# позволяет работать с такими строками при условии, что строка предварена символом `@`, а вместо одной двойной кавычки внутри строки используется набор из двух двойных кавычек (""). Вы должны помнить о том, что разносить литеральную строку по нескольким строкам файла нельзя, если только строка, с которой производится сравнение, не разнесена аналогичным образом. Например, литерал, применяемый в тестовом методе, был размещен в нескольких строках из-за недостаточной ширины печатной страницы. Символ новой строки не добавлялся, иначе тест не прошел бы.

Добавление данных модели представления

Мы пока не готовы использовать вспомогательную функцию дескриптора, т.к. представление еще не снабжено экземпляром класса модели представления `PagingInfo`. Это можно было бы сделать с применением объекта `ViewBag`, но лучше поместить все данные, подлежащие передаче из контроллера в представление, внутрь единственного класса модели представления. Добавим в папку `Models/ViewModels` проекта `SportsStore` файл класса по имени `ProductsListViewModel.cs` с содержимым, приведенным в листинге 8.26.

Листинг 8.26. Содержимое файла `ProductsListViewModel.cs` из папки `Models/ViewModels`

```
using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {
  public class ProductsListViewModel {
    public IEnumerable<Product> Products { get; set; }
    public PagingInfo PagingInfo { get; set; }
  }
}
```

Далее можно обновить метод действия `List()` класса `ProductController`, чтобы он использовал класс `ProductsListViewModel` для снабжения представления сведениями о товарах, отображаемых на страницах, и информацией о разбиении на страницы (листинг 8.27).

Листинг 8.27. Обновление метода действия `List()` в файле `ProductController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

  public class ProductController : Controller {
    private IProductRepository repository;
    public int PageSize = 4;

    public ProductController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult List(int productPage = 1)
     => View(new ProductsListViewModel {
       Products = repository.Products
         .OrderBy(p => p.ProductID)
         .Skip((productPage - 1) * PageSize)
         .Take(PageSize),
       PagingInfo = new PagingInfo {
         CurrentPage = productPage,
         ItemsPerPage = PageSize,
         TotalItems = repository.Products.Count()
       }
     });
  }
}
```

Внесенные изменения обеспечивают передачу представлению объекта `ProductsListViewModel` как данных модели.

Модульное тестирование: данные разбиения на страницы для модели представления

Нам необходимо удостовериться в том, что контроллер отправляет представлению корректную информацию о разбиении на страницы. Ниже показан модульный тест, добавленный в класс `ProductControllerTests` внутри тестового проекта, который обеспечивает такую проверку:

```
...
[Fact]
public void Can_Send_Pagination_View_Model() {
  // Организация
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
```

```
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  }).AsQueryable<Product>());
  // Организация
  ProductController controller =
    new ProductController(mock.Object) { PageSize = 3 };
  // Действие
  ProductsListViewModel result =
    controller.List(2).ViewData.Model as ProductsListViewModel;
  // Утверждение
  PagingInfo pageInfo = result.PagingInfo;
  Assert.Equal(2, pageInfo.CurrentPage);
  Assert.Equal(3, pageInfo.ItemsPerPage);
  Assert.Equal(5, pageInfo.TotalItems);
  Assert.Equal(2, pageInfo.TotalPages);
}
...
```

Потребуется также модифицировать ранее созданный модульный тест для проверки разбиения на страницы, содержащийся в методе `Can_Paginate()`. Он полагается на метод действия `List()`, возвращающий объект `ViewResult`, свойством `Model` которого является последовательность объектов `Product`, но мы поместили эти данные внутрь еще одного типа модели представления. Вот переделанный тест:

```
...
[Fact]
public void Can_Paginate() {
  // Организация
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  }).AsQueryable<Product>());

  ProductController controller = new ProductController(mock.Object);
  controller.PageSize = 3;

  // Действие
  ProductsListViewModel result =
    controller.List(2).ViewData.Model as ProductsListViewModel;

  // Утверждение
  Product[] prodArray = result.Products.ToArray();
  Assert.True(prodArray.Length == 2);
  Assert.Equal("P4", prodArray[0].Name);
  Assert.Equal("P5", prodArray[1].Name);
}
...
```

Учитывая объем дублированного кода в двух тестовых методах, обычно следовало бы создать общий метод начальной установки. Однако, поскольку модульные тесты описаны в индивидуальных врезках вроде текущей, их код представлен по отдельности, чтобы с каждым тестом можно было ознакомиться независимо от других.

В настоящий момент представление ожидает последовательность объектов Product, поэтому файл List.cshtml необходимо обновить, как показано в листинге 8.28, чтобы иметь дело с новым типом модели представления.

Листинг 8.28. Обновление файла List.cshtml из папки Views/Product

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  <div>
    <h3>@p.Name</h3>
    @p.Description
    <h4>@p.Price.ToString("c")</h4>
  </div>
}
```

Выражение @model было изменено для указания механизму Razor на то, что теперь мы работаем с другим типом данных. Цикл foreach был обновлен, чтобы источником данных стало свойство Products объекта модели.

Отображение ссылок на страницы

Наконец-то все готово для добавления в представление List ссылок на страницы. Мы создали модель представления, которая содержит информацию о разбиении на страницы, обновили контроллер, чтобы эта информация передавалась представлению, и модифицировали выражение @model, приведя его в соответствие с новым типом модели представления. Осталось добавить HTML-элемент, который вспомогательная функция дескриптора будет обрабатывать для создания ссылок на страницы (листинг 8.29).

Листинг 8.29. Добавление ссылок на страницы в файле List.cshtml из папки Views/Product

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  <div>
    <h3>@p.Name</h3>
    @p.Description
    <h4>@p.Price.ToString("c")</h4>
  </div>
}

<div page-model="@Model.PagingInfo" page-action="List"></div>
```

Запустив приложение, вы увидите новые ссылки на страницы (рис. 8.9). Стиль отображения довольно примитивен, но далее в главе он будет улучшен. Пока важно то, что ссылки позволяют перемещаться по страницам каталога и знакомиться с товарами, выставленными на продажу. Когда механизм Razor обнаруживает атрибут page-model в элементе div, он обращается к классу PageLinkTagHelper для преобразования элемента, что и дает набор ссылок, показанных на рис. 8.9.

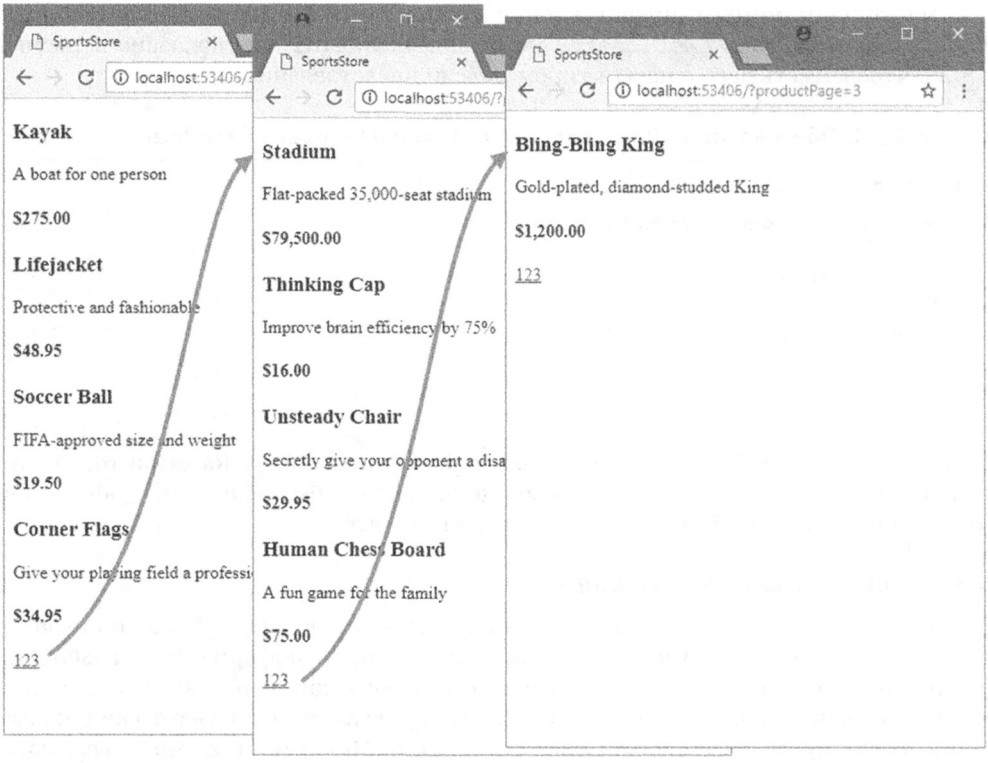


Рис. 8.9. Отображение ссылок для навигации по страницам

Почему бы просто не воспользоваться элементом управления `GridView`?

Если вы ранее работали с ASP.NET, то вам может показаться, что такой большой объем работы привел к довольно скромным результатам. Пришлось написать немало кода лишь для того, чтобы получить список товаров с разбиением на страницы. Если бы мы прибегли к услугам инфраструктуры Web Forms, то такого же результата можно было бы достичь с применением готового элемента управления `GridView` или `ListView` из ASP.NET Web Forms, привязав его напрямую к таблице `Products` базы данных.

То, что было сделано в главе, выглядит не слишком впечатляюще, но серьезно отличается от процесса перетаскивания какого-то элемента управления на поверхность визуального проектирования. Во-первых, мы строим приложение с надежной и удобной в сопровождении архитектурой, которая обеспечивает надлежащее разделение обязанностей. В отличие от простейшего использования `ListView` мы не связываем напрямую пользовательский интерфейс и базу данных, что представляет собой подход, который дает быстрые результаты, но вызовет проблемы и сложности в будущем. Во-вторых, в ходе дела мы создаем модульные тесты, что позволяет проверять поведение приложения естественным образом, который практически невозможен в случае применения сложного элемента управления Web Forms. В-третьих, не забывайте, что значительная часть главы посвящена созданию инфраструктуры, на основе которой строится приложение. Например, определить и реализовать хранилище нужно только один раз, и теперь, когда оно имеется в нашем распоряжении, новые функциональные средства можно строить и тестировать легко и быстро, как будет продемонстрировано в последующих главах.

Разумеется, ничто из сказанного отнюдь не умаляет значимость безотлагательных результатов, которые способна предложить инфраструктура Web Forms, но, как объяснялось в главе 3, такая безотлагательность имеет свою цену, которая может оказаться высокой и болезненной в крупных и сложных проектах.

Улучшение URL

Ссылки на страницы работают, но для передачи информации серверу они по-прежнему используют строку запроса следующего вида:

```
http://localhost/?productPage=2
```

Чтобы получить более привлекательные URL, необходимо создать схему, которая следует шаблону *компонуемых URL*. Компонуемым называется URL, имеющий смысл для пользователя, такой как показанный ниже:

```
http://localhost/Page2
```

Инфраструктура MVC позволяет легко изменять схему URL в приложении, потому что применяет средство *маршрутизации* ASP.NET, которое отвечает за обработку URL для выяснения, на какую часть приложения они указывают. Понадобится лишь добавить новый маршрут при регистрации промежуточного программного обеспечения MVC в методе `Configure()` класса `Startup` (листинг 8.30).

Листинг 8.30. Добавление нового маршрута в файле `Startup.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreProducts:ConnectionString"]));
      services.AddTransient<IProductRepository, EFProductRepository>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
```

```
        app.UseStatusCodePages();
        app.UseStaticFiles();
        app.UseMvc(routes => {
         routes.MapRoute(
           name: "pagination",
           template: "Products/Page{productPage}",
           defaults: new { Controller = "Product", action = "List" });
          routes.MapRoute(
            name: "default",
            template: "{controller=Product}/{action=List}/{id?}");
        });
        SeedData.EnsurePopulated(app);
    }
  }
}
```

Новый маршрут важно поместить перед стандартным маршрутом (по имени `default`), который уже присутствует в методе. Как будет показано в главе 15, система маршрутизации обрабатывает маршруты в порядке их перечисления, а нам нужно, чтобы новый маршрут имел преимущество перед существующим маршрутом.

Это единственное изменение, которое требуется внести, чтобы изменить схему URL для разбиения на страницы списка товаров. Инфраструктура MVC тесно интегрирована с функцией маршрутизации, а потому приложение автоматически отражает изменение подобного рода в URL, используемых приложением, включая URL, которые генерируются вспомогательными функциями дескрипторов вроде применяемых для генерации ссылок на страницы. Не беспокойтесь, если маршрутизация пока не особенно понятна — она будет подробно рассматриваться в главах 15 и 16.

Запустив приложение и щелкнув на ссылке для какой-нибудь страницы, можно наблюдать новую схему URL в действии (рис. 8.10).

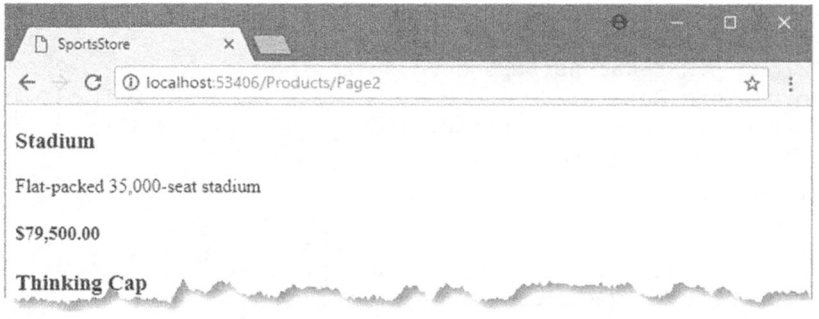


Рис. 8.10. Новая схема URL, отображаемая в браузере

Стилизация содержимого

Мы построили значительную часть инфраструктуры и базовые средства приложения и готовы к сборке всего воедино, но пока совершенно не уделяли внимание его внешнему виду. Хотя данная книга не посвящена веб-дизайну или CSS, внешний вид приложения SportsStore настолько примитивен, что умаляет его технические достоинства. В настоящем разделе мы постараемся исправить ситуацию. Мы собираемся реализовать классическую компоновку, содержащую два столбца и заголовок (рис. 8.11).

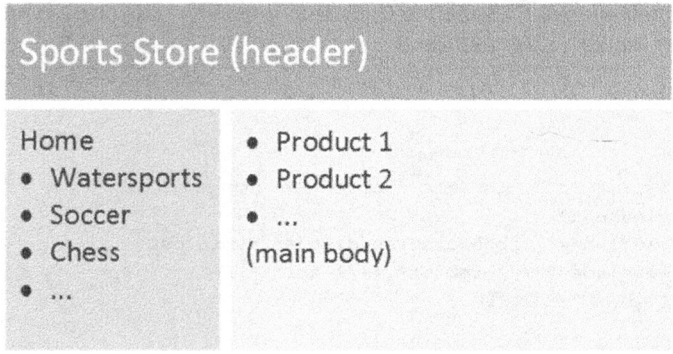


Рис. 8.11. Целевой дизайн для приложения SportsStore

Установка пакета Bootstrap

Для предоставления стилей CSS, которые будут применены к приложению, мы планируем использовать пакет Bootstrap. При установке пакета Bootstrap мы будем полагаться на встроенную в Visual Studio поддержку инструмента Bower. Выберем шаблон элемента Bower Configuration File (Файл конфигурации Bower) из категории General в диалоговом окне Add New Item, чтобы создать в проекте `SportsStore` файл по имени `bower.json`, как демонстрировалось в главе 6. Добавим в раздел `dependencies` созданного файла пакет Bootstrap, как показано в листинге 8.31. Ранее уже упоминалось, что в примерах книги применяется предварительный выпуск Bootstrap.

Листинг 8.31. Добавление пакета Bootstrap в файле `bower.json` из папки `SportsStore`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
    }
}
```

После сохранения изменений, внесенных в файл `bower.json`, среда Visual Studio с помощью инструмента Bower загрузит пакет Bootstrap в папку `wwwroot/lib/bootstrap`. Пакет Bootstrap зависит от пакета jQuery, который также автоматически добавится в проект.

Применение стилей Bootstrap к компоновке

В главе 5 объяснялось, как работают представления Razor, как они используются и как встраивать в них компоновки. Файл запуска представления, добавленный в начале главы, указывает, что в качестве стандартной компоновки должен выступать файл по имени `_Layout.cshtml`, так что начальную стилизацию Bootstrap мы применим именно к нему (листинг 8.32).

Листинг 8.32. Применение CSS-файла Bootstrap к файлу _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet"
       asp-href-include="/lib/bootstrap/dist/**/*.min.css"
       asp-href-exclude="**/*-reboot*,**/*-grid*" />
  <title>SportsStore</title>
</head>
<body>
  <div class="navbar navbar-inverse bg-inverse" role="navigation">
    <a class="navbar-brand" href="#">SPORTS STORE</a>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      Put something useful here later
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

Элемент `link` имеет атрибуты `asp-href-include` и `asp-href-exclude`, которые представляют собой пример использования встроенного класса вспомогательной функции дескриптора. В этом случае вспомогательная функция дескриптора просматривает значения атрибутов и генерирует элементы `link` для всех файлов по пути, указанному атрибутом `include`, и по путям, указанным атрибутами `exclude`. Пути в атрибутах могут содержать групповые символы, что делает такой прием удобным способом обеспечения возможности добавления и удаления файлов из структуры папок `wwwroot`, не нарушая работу приложения. Но, как объясняется в главе 25, следует проявлять внимание, чтобы из указанных путей выбирались только те файлы, которые ожидается получить.

Добавление в компоновку таблицы стилей CSS из Bootstrap означает, что определенные в ней стили можно применять в любом представлении, которое полагается на данную компоновку. В листинге 8.33 стилизация используется в файле `List.cshtml`.

Листинг 8.33. Стилизация содержимого в файле List.cshtml из папки Views/Product

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  <div class="card card-outline-primary m-1 p-1">
    <div class="bg-faded p-1">
      <h4>
        @p.Name
        <span class="badge badge-pill badge-primary" style="float:right">
```

```
        <small>@p.Price.ToString("c")</small>
      </span>
    </h4>
  </div>
  <div class="card-text p-1">@p.Description</div>
 </div>
}

<div page-model="@Model.PagingInfo" page-action="List" page-classes-enabled="true"
    page-class="btn" page-class-normal="btn-secondary"
    page-class-selected="btn-primary" class="btn-group pull-right m-1">
</div>
```

Нам необходимо стилизовать кнопки, которые генерирует класс `PageLinkTagHelper`, но жестко встраивать классы Bootstrap в код нежелательно, т.к. в итоге затруднится повторное использование вспомогательной функции дескриптора где-то в другом месте приложения либо изменение внешнего вида кнопок. Взамен мы определяем в элементе `div` специальные атрибуты, которые указывают требуемые классы и соответствуют свойствам, добавленным во вспомогательную функцию дескриптора. Затем они применяются для стилизации создаваемых элементов `a` (листинг 8.34).

Листинг 8.34. Добавление классов к генерируемым элементам в файле PageLinkTagHelper.cs из папки Infrastructure

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure {

  [HtmlTargetElement("div", Attributes = "page-model")]
  public class PageLinkTagHelper : TagHelper {
    private IUrlHelperFactory urlHelperFactory;

    public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
      urlHelperFactory = helperFactory;
    }

    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewContext { get; set; }

    public PagingInfo PageModel { get; set; }

    public string PageAction { get; set; }

    public bool PageClassesEnabled { get; set; } = false;
    public string PageClass { get; set; }
    public string PageClassNormal { get; set; }
    public string PageClassSelected { get; set; }

    public override void Process(TagHelperContext context,
          TagHelperOutput output) {
      IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
      TagBuilder result = new TagBuilder("div");
```

```
        for (int i = 1; i <= PageModel.TotalPages; i++) {
          TagBuilder tag = new TagBuilder("a");
          tag.Attributes["href"] = urlHelper.Action(PageAction,
            new { productPage = i });
          if (PageClassesEnabled) {
            tag.AddCssClass(PageClass);
            tag.AddCssClass(i == PageModel.CurrentPage
              ? PageClassSelected : PageClassNormal);
          }
          tag.InnerHtml.Append(i.ToString());
          result.InnerHtml.AppendHtml(tag);
        }
        output.Content.AppendHtml(result.InnerHtml);
      }
    }
}
```

Значения атрибутов автоматически используются для установки значений свойств во вспомогательной функции дескриптора, учитывая отображение между форматом имени атрибута HTML (`page-class-normal`) и форматом имени свойства C# (`PageClassNormal`). Это позволяет вспомогательным функциям дескрипторов по-разному реагировать в зависимости от атрибутов HTML-элемента, обеспечивая более гибкий способ генерации содержимого в приложении MVC.

Запустив приложение, легко заметить, что его внешний вид улучшился — во всяком случае, в какой-то степени (рис. 8.12).

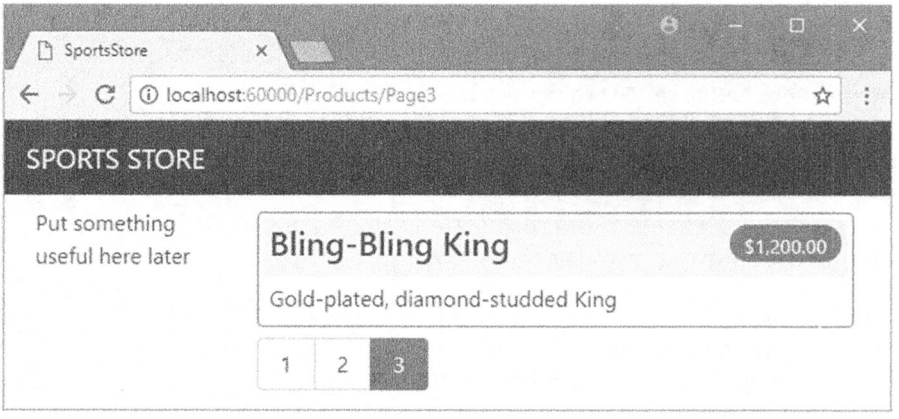


Рис. 8.12. Приложение SportsStore с улучшенным дизайном

Создание частичного представления

В качестве завершающего штриха в главе мы проведем рефакторинг приложения, чтобы упростить представление `List.cshtml`. Мы создадим *частичное представление*, являющееся фрагментом содержимого, которое можно внедрять в другое представление подобно шаблону. Частичные представления подробно рассматриваются в главе 21. Они помогают сократить дублирование, когда одно и то же содержимое

должно появляться в разных местах приложения. Вместо того чтобы копировать и вставлять одинаковую разметку Razor во множество представлений, можно определить ее единожды в частичном представлении. Чтобы создать частичное представление, добавим в папку Views/Shared файл представления Razor по имени ProductSummary.cshtml и поместим в него разметку, показанную в листинге 8.35.

Листинг 8.35. Содержимое файла ProductSummary.cshtml из папки Views/Shared

```
@model Product

<div class="card card-outline-primary m-1 p-1">
  <div class="bg-faded p-1">
    <h4>
      @Model.Name
      <span class="badge badge-pill badge-primary" style="float:right">
        <small>@Model.Price.ToString("c")</small>
      </span>
    </h4>
  </div>
<div class="card-text p-1">@Model.Description</div>
</div>
```

Теперь необходимо модифицировать файл List.cshtml из папки Views/Product, чтобы в нем применялось частичное представление (листинг 8.36).

Листинг 8.36. Использование частичного представления в файле List.cshtml из папки Views/Product

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  @Html.Partial("ProductSummary", p)
}

<div page-model="@Model.PagingInfo" page-action="List" page-classes-
enabled="true"
    page-class="btn" page-class-normal="btn-secondary"
    page-class-selected="btn-primary" class="btn-group pull-right m-1">
</div>
```

Мы взяли код разметки, который ранее размещался в цикле foreach внутри представления List.cshtml, и перенесли его в новое частичное представление. Обращение к частичному представлению производится с помощью вспомогательного метода Html.Partial(), которому в аргументах передаются имя представления и объект модели представления. Подобного рода переход на частичное представление является рекомендуемым приемом, поскольку он позволяет вставлять одну и ту же разметку в любое представление, которое нуждается в отображении сводки о товаре. Как видно на рис. 8.13, добавление частичного представления не изменяет внешний вид приложения; оно просто меняет место, где механизм Razor находит содержимое, которое применяет для генерации ответа, отправляемого браузеру.

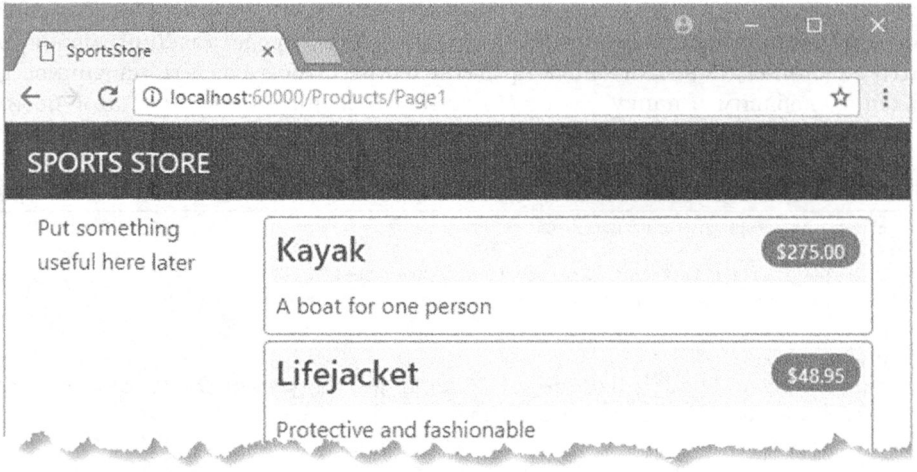


Рис. 8.13. Использование частичного представления

Резюме

В главе была построена основная инфраструктура для приложения SportsStore. Пока что она не содержит достаточного набора функциональных средств, чтобы их можно было продемонстрировать пользователю прямо сейчас, но "за кулисами" уже имеются зачатки модели предметной области с хранилищем товаров, которое поддерживается SQL Server и Entity Framework Core. В приложении присутствует единственный контроллер `Product`, который может создавать разбитые на страницы списки товаров, а также настроена ясная и дружественная к пользователям схема URL.

В настоящей главе могло показаться, что для получения весьма незначительных выгод пришлось провести излишне трудоемкую начальную подготовку, но в следующей главе равновесие будет восстановлено. Теперь, когда фундаментальная структура на месте, можно двигаться дальше и добавлять все средства, ориентированные на пользователя: навигацию по категориям, корзину для покупок и начальную форму для оплаты.

ГЛАВА 9

SportsStore: навигация

В настоящей главе мы продолжим построение примера приложения SportsStore, добавив к нему поддержку навигации по приложению и начав создавать корзину для покупок.

Добавление навигационных элементов управления

Приложение SportsStore станет более удобным, если пользователи получат возможность навигации среди товаров по категории. Мы будем решать задачу в три этапа.

- Расширим метод действия `List()` в классе `ProductController`, чтобы он был способен фильтровать объекты `Product` в хранилище.
- Пересмотрим и расширим схему URL.
- Создадим список категорий, который будет находиться в боковой панели сайта, подсвечивая текущую категорию и предоставляя ссылки на остальные категории.

Фильтрация списка товаров

Мы начнем с расширения класса модели представления `ProductsListViewModel`, который был добавлен в проект `SportsStore` в предыдущей главе. Нам нужно обеспечить взаимодействие текущей категории с представлением, чтобы визуализировать боковую панель, и это хорошая отправная точка. В листинге 9.1 показаны изменения, внесенные в файл `ProductsListViewModel.cs` из папки `Models/ViewModels`.

Листинг 9.1. Добавление свойства в файле `ProductsListViewModel.cs` из папки `Models/ViewModels`

```
using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

  public class ProductsListViewModel {
    public IEnumerable<Product> Products { get; set; }
    public PagingInfo PagingInfo { get; set; }
    public string CurrentCategory { get; set; }
  }
}
```

В класс ProductsListViewModel добавлено свойство по имени CurrentCategory. Следующий шаг заключается в обновлении класса ProductController, чтобы метод действия List() фильтровал объекты Product по категории и использовал только что добавленное в модель представления свойство для указания категории, выбранной в текущий момент. Изменения приведены в листинге 9.2.

Листинг 9.2. Добавление поддержки категорий к методу действия List() в файле ProductController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

  public class ProductController : Controller {
    private IProductRepository repository;
    public int PageSize = 4;

    public ProductController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult List(string category, int productPage = 1)
      => View(new ProductsListViewModel {
        Products = repository.Products
          .Where(p => category == null || p.Category == category)
          .OrderBy(p => p.ProductID)
          .Skip((productPage - 1) * PageSize)
          .Take(PageSize),
        PagingInfo = new PagingInfo {
          CurrentPage = productPage,
          ItemsPerPage = PageSize,
          TotalItems = repository.Products.Count()
        },
        CurrentCategory = category
      });
  }
}
```

В метод действия List() внесены три изменения. Первое — добавлен новый параметр по имени category. Он применяется вторым изменением, которое связано с расширением запроса LINQ. Если значение category не равно null, тогда выбираются только объекты Product с соответствующим значением в свойстве Category. Последнее, третье, изменение касается установки значения свойства CurrentCategory, которое было добавлено в класс ProductsListViewModel. Однако в результате таких изменений значение PagingInfo.TotalItems вычисляется некорректно, потому что оно не принимает во внимание фильтр по категории. Со временем мы все исправим.

Модульное тестирование: обновление существующих модульных тестов

Мы изменили сигнатуру метода действия `List()`, поэтому некоторые существующие методы модульного тестирования перестали компилироваться. Чтобы решить возникшую проблему, в модульных тестах, которые работают с контроллером, методу действия `List()` понадобится передавать в первом параметре значение `null`. Например, раздел действия тестового метода `Can_Paginate()` в файле `ProductControllerTests.cs` должен выглядеть следующим образом:

```
...
[Fact]
public void Can_Paginate() {
  // Организация
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  }).AsQueryable<Product>());

  ProductController controller = new ProductController(mock.Object);
  controller.PageSize = 3;

  // Действие
  ProductsListViewModel result =
    controller.List(null, 2).ViewData.Model as ProductsListViewModel;

  // Утверждение
  Product[] prodArray = result.Products.ToArray();
  Assert.True(prodArray.Length == 2);
  Assert.Equal("P4", prodArray[0].Name);
  Assert.Equal("P5", prodArray[1].Name);
}
...
```

Указывая `null` для `category`, мы получаем все объекты `Product`, которые контроллер извлекает из хранилища, что воспроизводит ситуацию перед добавлением нового параметра. Такого же рода изменение необходимо внести в тестовый метод `Can_Send_Pagination_View_Model()`:

```
...
[Fact]
public void Can_Send_Pagination_View_Model() {
  // Организация
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  }).AsQueryable<Product>());
  // Организация
  ProductController controller =
    new ProductController(mock.Object) { PageSize = 3 };
```

```
    // Действие
    ProductsListViewModel result =
      controller.List(null, 2).ViewData.Model as ProductsListViewModel;

    // Утверждение
    PagingInfo pageInfo = result.PagingInfo;
    Assert.Equal(2, pageInfo.CurrentPage);
    Assert.Equal(3, pageInfo.ItemsPerPage);
    Assert.Equal(5, pageInfo.TotalItems);
    Assert.Equal(2, pageInfo.TotalPages);
}
...
```

Когда вы примете образ мышления, ориентированный на тестирование, поддержание модульных тестов в синхронизированном состоянии с изменениями кода быстро станет вашей второй натурой.

Чтобы увидеть результат фильтрации по категории, запустим приложение и выберем категорию с помощью показанной ниже строки запроса (замените номер порта тем, который был назначен вашему проекту средой Visual Studio), позаботившись о том, что `Soccer` начинается с прописной буквы `S`:

```
http://localhost:60000/?category=Soccer
```

Отобразятся только товары из категории `Soccer` (рис. 9.1).

Вряд ли пользователи захотят переходить по категориям с применением URL, но здесь было показано, что незначительные изменения в приложении MVC могут оказывать крупное влияние, если базовая структура на месте.

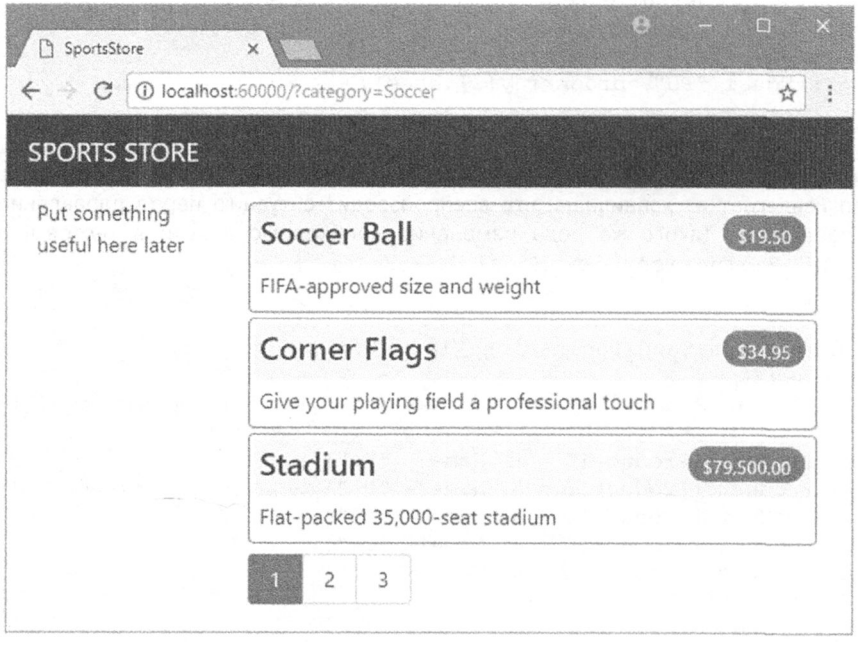


Рис. 9.1. Использование строки запроса для фильтрации по категории

Модульное тестирование: фильтрация по категории

Нам необходим модульный тест для проверки функциональности фильтрации по категории, чтобы удостовериться в том, что фильтр может корректно генерировать сведения о товарах указанной категории. Ниже приведен тестовый метод, добавленный в класс `ProductControllerTests`:

```
...
[Fact]
public void Can_Filter_Products() {
  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
    new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
    new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
    new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
    new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
  }).AsQueryable<Product>());

  // Организация - создание контроллера и установка размера страницы
  // в три элемента
  ProductController controller = new ProductController(mock.Object);
  controller.PageSize = 3;

  // Действие
  Product[] result =
    (controller.List("Cat2", 1).ViewData.Model as ProductsListViewModel)
      .Products.ToArray();

  // Утверждение
  Assert.Equal(2, result.Length);
  Assert.True(result[0].Name == "P2" && result[0].Category == "Cat2");
  Assert.True(result[1].Name == "P4" && result[1].Category == "Cat2");
}
...
```

Тест создает имитированное хранилище, содержащее объекты `Product`, которые относятся к диапазону категорий. С использованием метода действия `List()` запрашивается одна специфическая категория, и результаты проверяются на предмет наличия корректных объектов в правильном порядке.

Улучшение схемы URL

Мало кому понравится видеть либо иметь дело с неуклюжими URL вроде `/?category=Soccer`. Для решения этой проблемы мы намерены изменить схему маршрутизации в методе `Configure()` класса `Startup`, чтобы создать более удобный набор URL (листинг 9.3).

Внимание! Новые маршруты в листинге 9.3 важно добавлять в показанном порядке. Маршруты применяются в порядке, в котором они определены, поэтому изменение порядка может привести к нежелательным эффектам.

Листинг 9.3. Изменение схемы маршрутизации в файле `Startup.cs` из папки `SportsStore`

```
...
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  app.UseDeveloperExceptionPage();
  app.UseStatusCodePages();
  app.UseStaticFiles();
  app.UseMvc(routes => {

    routes.MapRoute(
      name: null,
      template: "{category}/Page{productPage:int}",
      defaults: new { controller = "Product", action = "List" }
    );

    routes.MapRoute(
      name: null,
      template: "Page{productPage:int}",
      defaults: new { controller = "Product",
      action = "List", productPage = 1 }
    );

    routes.MapRoute(
      name: null,
      template: "{category}",
      defaults: new { controller = "Product",
      action = "List", productPage = 1 }
    );

    routes.MapRoute(
      name: null,
      template: "",
      defaults: new { controller = "Product", action = "List",
        productPage = 1 });

    routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");
  });
  SeedData.EnsurePopulated(app);
}
...
```

В табл. 9.1 описана схема URL, которую представляют определенные маршруты. Система маршрутизации подробно объясняется в главах 15 и 16.

Таблица 9.1. Сводка по маршрутам

URL	Что делает
`/`	Выводит первую страницу списка товаров всех категорий
`/Page2`	Выводит указанную страницу (в данном случае страницу 2), отображая товары всех категорий
`/Soccer`	Выводит первую страницу товаров указанной категории (`Soccer`)
`/Soccer/Page2`	Выводит указанную страницу (страницу 2) товаров заданной категории (`Soccer`)

Система маршрутизации ASP.NET используется инфраструктурой MVC для обработки входящих запросов от пользователей, но также генерирует исходящие URL, которые соответствуют схеме URL и потому могут встраиваться в веб-страницы. Применение системы маршрутизации для обработки входящих запросов и генерации исходящих URL позволяет гарантировать согласованность всех URL в приложении.

Интерфейс `IUrlHelper` предоставляет доступ к функциональности генерации URL. Мы использовали этот интерфейс и определяемый им метод `Action()` в классе вспомогательной функции дескриптора, созданном в предыдущей главе. Теперь, когда нужно генерировать более сложные URL, необходим способ получения дополнительной информации от представления, не добавляя дополнительные свойства к классу вспомогательной функции дескриптора. К счастью, классы вспомогательных функций дескрипторов обладают удобным средством, которое позволяет получать в одной коллекции все свойства с общим префиксом (листинг 9.4).

Листинг 9.4. Получение значений атрибутов, снабженных префиксом, в файле `PageLinkTagHelper.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;
using System.Collections.Generic;

namespace SportsStore.Infrastructure {

  [HtmlTargetElement("div", Attributes = "page-model")]
  public class PageLinkTagHelper : TagHelper {
    private IUrlHelperFactory urlHelperFactory;

    public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
      urlHelperFactory = helperFactory;
    }
    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewContext { get; set; }
    public PagingInfo PageModel { get; set; }
    public string PageAction { get; set; }
    [HtmlAttributeName(DictionaryAttributePrefix = "page-url-")]
    public Dictionary<string, object> PageUrlValues { get; set; }
      = new Dictionary<string, object>();
    public bool PageClassesEnabled { get; set; } = false;
    public string PageClass { get; set; }
    public string PageClassNormal { get; set; }
    public string PageClassSelected { get; set; }

    public override void Process(TagHelperContext context,
          TagHelperOutput output) {
      IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
      TagBuilder result = new TagBuilder("div");
      for (int i = 1; i <= PageModel.TotalPages; i++) {
        TagBuilder tag = new TagBuilder("a");
        PageUrlValues["productPage"] = i;
        tag.Attributes["href"] = urlHelper.Action(PageAction, PageUrlValues);
```

```
        if (PageClassesEnabled) {
          tag.AddCssClass(PageClass);
          tag.AddCssClass(i == PageModel.CurrentPage
            ? PageClassSelected : PageClassNormal);
        }
        tag.InnerHtml.Append(i.ToString());
        result.InnerHtml.AppendHtml(tag);
      }
      output.Content.AppendHtml(result.InnerHtml);
    }
  }
}
```

Декорирование свойства в классе вспомогательной функции дескриптора посредством атрибута `HtmlAttributeName` позволяет указывать префикс для имен атрибутов элемента, которым в данном случае будет `page-url-`. Значение любого атрибута, чье имя начинается с такого префикса, будет добавлено в словарь, присваиваемый свойству `PageUrlValues`, которое затем передается методу `IUrlHelper.Action()`, чтобы сгенерировать URL для атрибута `href` элементов `a`, выпускаемых вспомогательной функцией дескриптора.

В листинге 9.5 к элементу `div` добавлен новый атрибут, который обрабатывается вспомогательной функцией дескриптора и указывает категорию, применяемую для генерации URL. В представление был добавлен только один атрибут, но в словарь добавился бы любой атрибут с тем же самым префиксом.

Листинг 9.5. Добавление нового атрибута в файле `List.cshtml` из папки `Views/Home`

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  @Html.Partial("ProductSummary", p)
}
<div page-model="@Model.PagingInfo" page-action="List"
                page-classes-enabled="true"
  page-class="btn" page-class-normal="btn-secondary"
  page-class-selected="btn-primary" page-url-category="@Model.CurrentCategory"
    class="btn-group pull-right m-1">
</div>
```

До внесения этого изменения ссылки для перехода на страницы генерировались так:

```
http://<сервер>:<порт>/Page1
```

Если пользователь щелкнет на страничной ссылке подобного типа, то фильтр по категории утрачивается и приложение отобразит страницу, содержащую товары всех категорий. За счет добавления текущей категории, получаемой из модели представления, взамен генерируются URL следующего вида:

```
http://<сервер>:<порт>/Chess/Page1
```

Когда пользователь щелкает на ссылке такого рода, текущая категория передается методу действия `List()` и фильтрация сохраняется. После внесения данного изменения можно посещать URL вроде `/Chess` или `/Soccer` и наблюдать, что страничные ссылки, расположенные внизу, корректно включают категорию.

Построение меню навигации по категориям

Нам нужно предложить пользователям способ выбора категории, который не предусматривает ввод URL, что означает необходимость в отображении списка доступных категорий с отмеченной текущей категорией, если она есть. По мере построения приложения список категорий будет задействован в нескольких контроллерах, а потому он должен быть самодостаточным и многократно используемым.

В инфраструктуре ASP.NET Core MVC поддерживается концепция *компонентов представлений*, которые идеально подходят для создания единиц вроде многократно используемого навигационного элемента управления. Компонент представления — это класс C#, который предлагает небольшой объем многократно используемой прикладной логики с возможностью выбора и отображения частичных представлений Razor. Компоненты представлений подробно рассматриваются в главе 22.

В данном случае мы создадим компонент представления, который визуализирует навигационное меню и интегрирует его в приложение за счет обращения к этому компоненту из разделяемой компоновки. Подход подобного рода дает нам обычный класс C#, который может содержать любую необходимую прикладную логику и который можно подвергать модульному тестированию подобно любому другому классу. Такой способ удобен при создании небольших сегментов приложения с одновременным сохранением общего подхода MVC.

Создание навигационного компонента представления

Создадим папку по имени `Components`, которая по соглашению является местом хранения компонентов представлений, и добавим в нее файл класса под названием `NavigationMenuViewComponent.cs` с определением, показанным в листинге 9.6.

Листинг 9.6. Содержимое файла `NavigationMenuViewComponent.cs` из папки `Components`

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Components {

  public class NavigationMenuViewComponent : ViewComponent {

    public string Invoke() {
      return "Hello from the Nav View Component";
    }
  }
}
```

Метод `Invoke()` компонента представления вызывается, когда компонент применяется в представлении Razor, а результат, возвращаемый методом `Invoke()`, вставляется в HTML-разметку, отправляемую браузеру. Мы начали с простого компонента представления, который возвращает строку, но вскоре заменим его динамическим HTML-содержимым.

Список категорий должен присутствовать на всех страницах, поэтому мы собираемся использовать компонент представления в разделяемой компоновке, а не в отдельном представлении. Внутри компоновки компоненты представлений применяются через выражение `@await Component.InvokeAsync()`, как показано в листинге 9.7.

Листинг 9.7. Использование компонента представления в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet"
       asp-href-include="/lib/bootstrap/dist/**/*.min.css"
       asp-href-exclude="**/*-reboot*,**/*-grid*" />
  <title>SportsStore</title>
</head>
<body>
  <div class="navbar navbar-inverse bg-inverse" role="navigation">
    <a class="navbar-brand" href="#">SPORTS STORE</a>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      @await Component.InvokeAsync("NavigationMenu")
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

Текст заполнителя заменен вызовом метода `Component.InvokeAsync()`. Аргументом метода является имя класса компонента без части `ViewComponent`, т.е. с помощью `NavigationMenu` указывается класс `NavigationMenuViewComponent`. Запустив приложение, можно заметить, что вывод из метода `InvokeAsync()` включен в HTML-разметку, отправляемую браузеру (рис. 9.2).

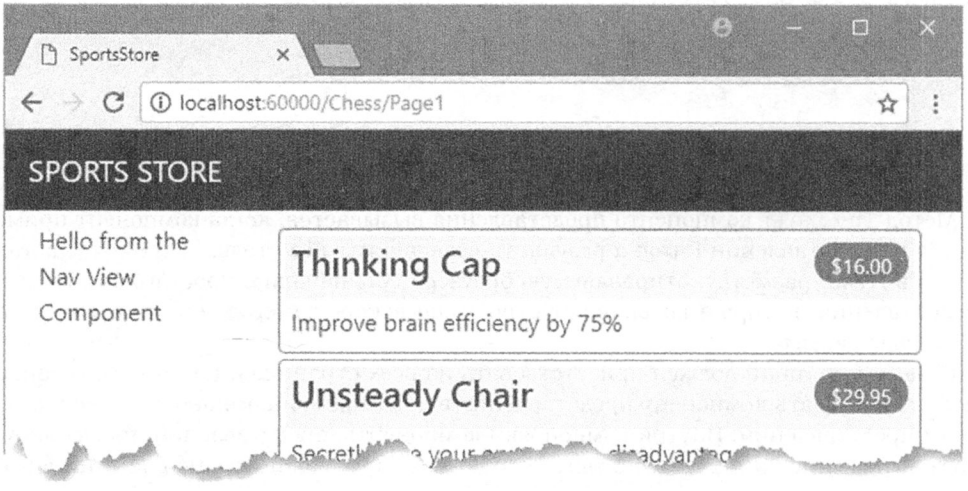


Рис. 9.2. Применение компонента представления

Генерация списков категорий

Теперь можно возвратиться к навигационному компоненту представления и сгенерировать реальный набор категорий. Построить HTML-разметку для категорий можно было бы программно, как делалось во вспомогательной функции дескриптора для страничных ссылок, но одно из преимуществ работы с компонентами представлений заключается в том, что они могут визуализировать частичные представления Razor. Это означает, что компонент представления можно использовать для генерации списка категорий и затем применить более выразительный синтаксис Razor для визуализации HTML-разметки, которая отобразит данный список. Первым делом нужно обновить компонент представления, как показано в листинге 9.8.

Листинг 9.8. Добавление категорий в файле `NavigationMenuViewComponent.cs` из папки `Components`

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components {

  public class NavigationMenuViewComponent : ViewComponent {
    private IProductRepository repository;

    public NavigationMenuViewComponent(IProductRepository repo) {
      repository = repo;
    }

    public IViewComponentResult Invoke() {
      return View(repository.Products
        .Select(x => x.Category)
        .Distinct()
        .OrderBy(x => x));
    }
  }
}
```

Конструктор, определенный в листинге 9.8, принимает аргумент типа `IProductRepository`. Когда инфраструктуре MVC необходимо создать экземпляр класса компонента представления, она отметит потребность в предоставлении этого аргумента и просмотрит конфигурацию в классе `Startup`, чтобы выяснить, какой объект реализации должен использоваться. Мы имеем дело с тем же самым средством внедрения зависимостей, которое применялось в контроллере из главы 8, и результат будет аналогичным — обеспечение компоненту представления доступа к данным без необходимости знать используемую реализацию хранилища, как описано в главе 18.

В методе `Invoke()` с помощью LINQ выбирается и упорядочивается набор категорий в хранилище, после чего он передается в качестве аргумента методу `View()`, который визуализирует стандартное частичное представление Razor. Детали частичного представления возвращаются из метода с применением объекта реализации `IViewComponentResult` (данный процесс будет подробно рассматриваться в главе 22).

Модульное тестирование: генерация списка категорий

Модульный тест, предназначенный для проверки возможности генерации списка категорий, относительно прост. Цель заключается в том, чтобы создать список, который отсортирован в алфавитном порядке и не содержит дубликатов. Проще всего это сделать, построив тестовые данные, которые *имеют* дублированные категории и *не* отсортированы должным образом, передав их вспомогательной функции дескриптора и установив утверждение, что данные были соответствующим образом очищены. Вот модульный тест, который определяется в новом файле класса по имени `NavigationMenuViewComponentTests.cs` внутри проекта `SportsStore.Tests`:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Moq;
using SportsStore.Components;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

  public class NavigationMenuViewComponentTests {

    [Fact]
    public void Can_Select_Categories() {
      // Организация
      Mock<IProductRepository> mock = new Mock<IProductRepository>();
      mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Apples"},
        new Product {ProductID = 2, Name = "P2", Category = "Apples"},
        new Product {ProductID = 3, Name = "P3", Category = "Plums"},
        new Product {ProductID = 4, Name = "P4", Category = "Oranges"},
      }).AsQueryable<Product>());

      NavigationMenuViewComponent target =
        new NavigationMenuViewComponent(mock.Object);

      // Действие - получение набора категорий
      string[] results = ((IEnumerable<string>)(target.Invoke()
        as ViewViewComponentResult).ViewData.Model).ToArray();

      // Утверждение
      Assert.True(Enumerable.SequenceEqual(new string[] { "Apples",
        "Oranges", "Plums" }, results));
    }
  }
}
```

Мы создаем имитированную реализацию хранилища, которая содержит повторяющиеся и несортированные категории. Затем мы устанавливаем утверждение о том, что дубликаты удалены и восстановлен алфавитный порядок.

Создание представления

При работе с представлениями, которые выбираются компонентами представлений, механизм Razor использует разнообразные соглашения. И стандартное имя представления, и местоположения, в которых ищется представление, отличаются от тех,

которые приняты для контроллеров. Создадим папку `Views/Shared/Components/NavigationMenu` и добавим в нее файл представления по имени `Default.cshtml` с содержимым, приведенным в листинге 9.9.

Листинг 9.9. Содержимое файла `Default.cshtml` из папки `Views/Shared/Components/NavigationMenu`

```
@model IEnumerable<string>

<a class="btn btn-block btn-secondary"
   asp-action="List"
   asp-controller="Product"
   asp-route-category="">
     Home
</a>

@foreach (string category in Model) {
  <a class="btn btn-block btn-secondary"
     asp-action="List"
     asp-controller="Product"
     asp-route-category="@category"
     asp-route-productPage="1">
       @category
  </a>
}
```

В представлении применяется один из встроенных классов вспомогательных функций дескрипторов (описанных в главах 24 и 25) для создания элементов `a`, атрибут `href` которых содержит URL, выбирающий определенную категорию товаров.

После запуска приложения появятся ссылки на категории (рис. 9.3). Щелчок на какой-то категории приводит к тому, что список товаров обновляется, отображая только товары выбранной категории.



Рис. 9.3. Генерация ссылок на категории с помощью компонента представления

Подсветка текущей категории

В настоящий момент пользователь не располагает какой-нибудь визуальной подсказкой о выбранной категории. Хотя, основываясь на товарах в списке, можно выдвинуть предположение относительно категории, гораздо лучше предоставить более наглядную визуальную обратную связь. Компоненты ASP.NET Core MVC, такие как контроллеры и компоненты представлений, могут получать информацию о текущем запросе, обращаясь к объекту контекста. Большую часть времени заботу о получении объекта контекста можно поручить базовым классам, которые используются для создания компонентов, подобно тому, как базовый класс `Controller` применяется для создания контроллеров.

Базовый класс `ViewComponent` не является исключением и обеспечивает доступ к объектам контекста через набор свойств. Одно из свойств называется `RouteData` и предоставляет информацию о том, как URL запроса был обработан системой маршрутизации.

В листинге 9.10 свойство `RouteData` используется для доступа к данным запроса, чтобы получить значение выбранной в текущий момент категории. Значение категории можно было бы передать представлению путем создания еще одного класса модели представления (и так бы делалось в реальном проекте), но ради разнообразия применим объект `ViewBag`, который был введен в главе 2.

Листинг 9.10. Передача выбранной категории в файле NavigationMenuViewComponent.cs из папки Components

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components {

  public class NavigationMenuViewComponent : ViewComponent {
    private IProductRepository repository;

    public NavigationMenuViewComponent(IProductRepository repo) {
      repository = repo;
    }

    public IViewComponentResult Invoke() {
      ViewBag.SelectedCategory = RouteData?.Values["category"];
      return View(repository.Products
        .Select(x => x.Category)
        .Distinct()
        .OrderBy(x => x));
    }
  }
}
```

Внутри метода `Invoke()` мы динамически создаем свойство `SelectedCategory` в объекте `ViewBag` и устанавливаем его значение равным значению текущей категории, которое получаем через объект контекста, возвращенный свойством `RouteData`. Как объяснялось в главе 2, `ViewBag` представляет собой динамический объект, который позволяет определять новые свойства, просто присваивая им значения.

Модульное тестирование: сообщение о выбранной категории

Для выполнения проверки того, что компонент представления корректно добавил детали о выбранной категории, в модульном тесте можно прочитать значение свойства `ViewBag`, которое доступно через класс `ViewViewComponentResult`, описанный в главе 22. Ниже показан модульный тест, добавленный в класс `NavigatioMenuViewComponentTests`:

```
...
[Fact]
public void Indicates_Selected_Category() {

  // Организация
  string categoryToSelect = "Apples";
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Apples"},
    new Product {ProductID = 4, Name = "P2", Category = "Oranges"},
  }).AsQueryable<Product>());

  NavigationMenuViewComponent target =
    new NavigationMenuViewComponent(mock.Object);
  target.ViewComponentContext = new ViewComponentContext {
    ViewContext = new ViewContext {
      RouteData = new RouteData()
    }
  };
  target.RouteData.Values["category"] = categoryToSelect;

  // Действие
  string result = (string)(target.Invoke() as
    ViewViewComponentResult).ViewData["SelectedCategory"];

  // Утверждение
  Assert.Equal(categoryToSelect, result);
}
...
```

Модульный тест снабжает компонент представления данными маршрутизации через свойство `ViewComponentContext`, посредством которого компоненты представлений получают все свои данные контекста. Свойство `ViewComponentContext` обеспечивает доступ к данным контекста, специфичным для представления, с помощью своего свойства `ViewContext`, которое в свою очередь обеспечивает доступ к информации о маршрутизации через собственное свойство `RouteData`. Большая часть кода в модульном тесте связана с созданием объектов контекста, которые будут предоставлять выбранную категорию таким же способом, как она бы предлагалась во время выполнения приложения, когда данные контекста предоставляются инфраструктурой ASP.NET Core MVC.

Теперь, когда доступна информация о том, какая категория выбрана, можно обновить представление, выбираемое компонентом представления, чтобы задействовать эту информацию, и изменить классы CSS, которые используются для стилизации ссылок, сделав представление текущей категории отличающимся от остальных категорий. В листинге 9.11 приведено изменение, внесенное в файл `Default.cshtml`.

Листинг 9.11. Подсветка текущей категории в файле `Default.cshtml` из папки `Views/Shared/Components/NavigationMenu`

```
@model IEnumerable<string>

<a class="btn btn-block btn-secondary"
   asp-action="List"
   asp-controller="Product"
   asp-route-category="">
     Home
</a>

@foreach (string category in Model) {
  <a class="btn btn-block
    @(category == ViewBag.SelectedCategory ? "btn-primary": "btn-secondary")"
    asp-action="List"
    asp-controller="Product"
    asp-route-category="@category"
    asp-route-productPage="1">
      @category
  </a>
}
```

С помощью выражения Razor внутри атрибута `class` мы применяем класс `btn-primary` к элементу, который представляет выбранную категорию, и класс `btn-secondary` к остальным элементам. Указанные классы применяют разные стили Bootstrap и делают активную кнопку визуально отличающейся (рис. 9.4).

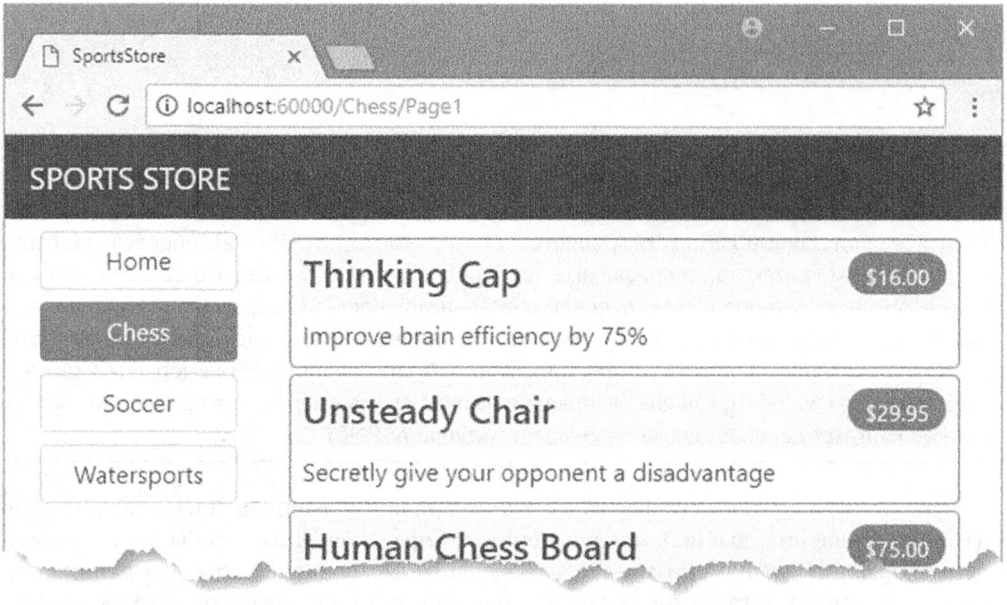


Рис. 9.4. Подсвечивание выбранной категории

Корректировка счетчика страниц

Мы должны скорректировать ссылки на страницы, чтобы они правильно работали, когда выбрана какая-то категория. В настоящий момент количество ссылок на страницы определяется общим числом товаров в хранилище, а не количеством товаров выбранной категории. Это значит, что пользователь может щелкнуть на ссылке для страницы 2 категории `Chess` и получить пустую страницу, поскольку товаров данной категории не хватает для заполнения второй страницы. Проблема демонстрируется на рис. 9.5.

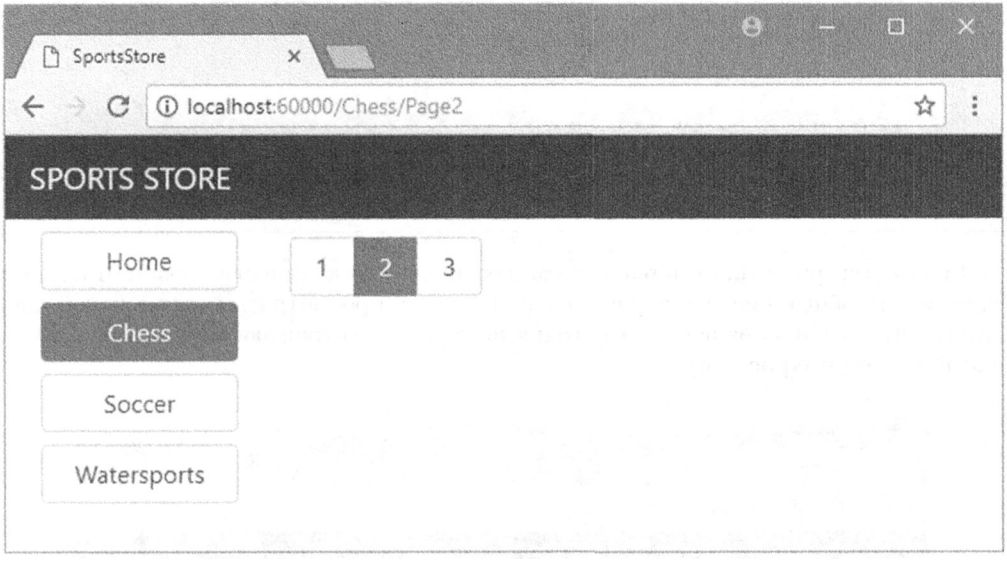


Рис. 9.5. Отображение некорректных ссылок на страницы, когда выбрана какая-то категория

Проблему можно устранить, модифицировав метод действия `List()` в контроллере `Product` так, чтобы при разбиении на страницы категории принимались во внимание (листинг 9.12).

Листинг 9.12. Создание данных о разбиении на страницы, учитывающих категории, в файле `ProductController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

  public class ProductController : Controller {
    private IProductRepository repository;
    public int PageSize = 4;

    public ProductController(IProductRepository repo) {
      repository = repo;
    }
```

```
    public ViewResult List(string category, int productPage = 1)
      => View(new ProductsListViewModel {
        Products = repository.Products
          .Where(p => category == null || p.Category == category)
          .OrderBy(p => p.ProductID)
          .Skip((productPage - 1) * PageSize)
          .Take(PageSize),
          PagingInfo = new PagingInfo {
            CurrentPage = productPage,
            ItemsPerPage = PageSize,
            TotalItems = category == null ?
              repository.Products.Count() :
              repository.Products.Where(e =>
                e.Category == category).Count()
          },
          CurrentCategory = category
      });
  }
}
```

Если категория была выбрана, тогда возвращается количество позиций в ней, а если нет, то общее число товаров. Теперь во время просмотра товаров в какой-либо категории ссылки в нижней части страницы корректно отражают количество товаров в этой категории (рис. 9.6).

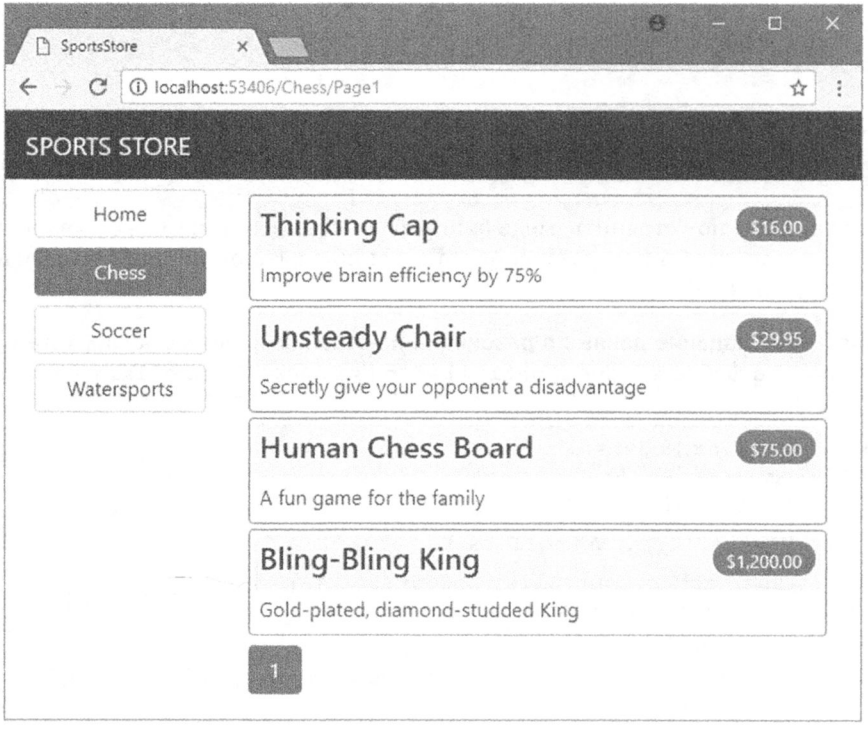


Рис. 9.6. Отображение ссылок на страницы с учетом выбранной категории

Модульное тестирование: счетчик товаров определенной категории

Протестировать возможность генерации корректных счетчиков товаров для различных категорий очень просто. Мы создадим имитированное хранилище, которое содержит известные данные в определенном диапазоне категорий, и затем вызовем метод действия `List()`, запрашивая каждую категорию по очереди. Вот модульный тест, добавленный в класс `ProductControllerTests`:

```
...
[Fact]
public void Generate_Category_Specific_Product_Count() {
  // Организация
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
    new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
    new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
    new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
    new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
  }).AsQueryable<Product>());

  ProductController target = new ProductController(mock.Object);
  target.PageSize = 3;

  Func<ViewResult, ProductsListViewModel> GetModel = result =>
    result?.ViewData?.Model as ProductsListViewModel;

  // Действие
  int? res1 = GetModel(target.List("Cat1"))?.PagingInfo.TotalItems;
  int? res2 = GetModel(target.List("Cat2"))?.PagingInfo.TotalItems;
  int? res3 = GetModel(target.List("Cat3"))?.PagingInfo.TotalItems;
  int? resAll = GetModel(target.List(null))?.PagingInfo.TotalItems;

  // Утверждение
  Assert.Equal(2, res1);
  Assert.Equal(2, res2);
  Assert.Equal(1, res3);
  Assert.Equal(5, resAll);
}
...
```

Обратите внимание, что в модульном тесте также вызывается метод `List()` без указания категории, чтобы удостовериться в правильности подсчета общего количества товаров.

Построение корзины для покупок

Приложение успешно расширяется, но продавать какие-либо товары до тех пор, пока не будет реализована корзина для покупок, не удастся. В настоящем разделе мы создадим корзину для покупок согласно иллюстрации на рис. 9.7. Если вы приобретали что-нибудь в электронных магазинах, то она должна выглядеть знакомой.

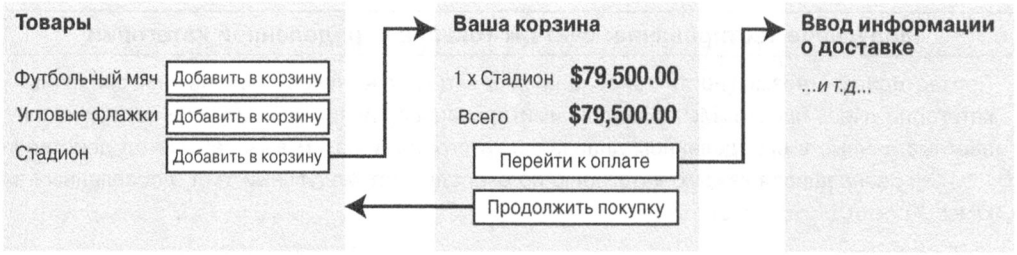


Рис. 9.7. Базовый поток корзины для покупок

Кнопка добавления в корзину (Add to cart) будет отображаться рядом с каждым товаром в каталоге. Щелчок на ней будет приводить к выводу сводки по товарам, которые уже были выбраны пользователем, включая их общую стоимость. В этой точке пользователь может с помощью кнопки продолжения покупки (Continue shopping) возвратиться в каталог товаров, а посредством кнопки перехода к оплате (Checkout now) сформировать заказ и завершить сеанс покупки.

Определение модели корзины

Начнем с добавления в папку `Models` файла класса по имени `Cart.cs` с определениями, показанными в листинге 9.13.

Листинг 9.13. Содержимое файла `Cart.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class Cart {
    private List<CartLine> lineCollection = new List<CartLine>();

    public virtual void AddItem(Product product, int quantity) {
      CartLine line = lineCollection
        .Where(p => p.Product.ProductID == product.ProductID)
        .FirstOrDefault();

      if (line == null) {
        lineCollection.Add(new CartLine {
          Product = product,
          Quantity = quantity
        });
      } else {
        line.Quantity += quantity;
      }
    }

    public virtual void RemoveLine(Product product) =>
      lineCollection.RemoveAll(l => l.Product.ProductID == product.ProductID);

    public virtual decimal ComputeTotalValue() =>
      lineCollection.Sum(e => e.Product.Price * e.Quantity);

    public virtual void Clear() => lineCollection.Clear();

    public virtual IEnumerable<CartLine> Lines => lineCollection;
  }
```

```
  public class CartLine {
    public int CartLineID { get; set; }
    public Product Product { get; set; }
    public int Quantity { get; set; }
  }
}
```

Класс `Cart` использует класс `CartLine`, который определен в том же самом файле и представляет товар, выбранный пользователем, а также приобретаемое количество товара. Мы определили методы для добавления элемента в корзину, удаления элемента из корзины, вычисления общей стоимости элементов в корзине и очистки корзины путем удаления всех элементов. Мы также предоставили свойство, которое позволяет обратиться к содержимому корзины с применением `IEnumerable<CartLine>`. Все они легко реализуются с помощью кода C# и небольшой доли кода LINQ.

Модульное тестирование: проверка корзины

Класс `Cart` относительно прост, но в нем присутствует ряд важных линий поведения, которые должны корректно работать. Неправильно функционирующая корзина нарушит работу всего приложения SportsStore. Мы разобьем средства на части и протестируем их по отдельности. Для размещения тестов в проекте `SportsStore.Tests` создан новый файл по имени `CartTests.cs`.

Первая линия поведения относится к добавлению элемента в корзину. При первоначальном добавлении в корзину объекта `Product` должен быть добавлен новый экземпляр `CartLine`. Ниже представлен тестовый метод вместе с определением класса модульного тестирования.

```
using System.Linq;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

  public class CartTests {

    [Fact]
    public void Can_Add_New_Lines() {
      // Организация - создание нескольких тестовых товаров
      Product p1 = new Product { ProductID = 1, Name = "P1" };
      Product p2 = new Product { ProductID = 2, Name = "P2" };

      // Организация - создание новой корзины
      Cart target = new Cart();

      // Действие
      target.AddItem(p1, 1);
      target.AddItem(p2, 1);
      CartLine[] results = target.Lines.ToArray();

      // Утверждение
      Assert.Equal(2, results.Length);
      Assert.Equal(p1, results[0].Product);
      Assert.Equal(p2, results[1].Product);
    }
  }
}
```

Но если пользователь уже добавлял объект `Product` в корзину, тогда необходимо увеличить количество в соответствующем экземпляре `CartLine`, а не создавать новый. Вот модульный тест:

```
...
[Fact]
public void Can_Add_Quantity_For_Existing_Lines() {
  // Организация - создание нескольких тестовых товаров
  Product p1 = new Product { ProductID = 1, Name = "P1" };
  Product p2 = new Product { ProductID = 2, Name = "P2" };

  // Организация - создание новой корзины
  Cart target = new Cart();

  // Действие
  target.AddItem(p1, 1);
  target.AddItem(p2, 1);
  target.AddItem(p1, 10);
  CartLine[] results = target.Lines
    .OrderBy(c => c.Product.ProductID).ToArray();

  // Утверждение
  Assert.Equal(2, results.Length);
  Assert.Equal(11, results[0].Quantity);
  Assert.Equal(1, results[1].Quantity);
}
...
```

Также понадобится проверить, что пользователи имеют возможность менять свое решение и удалять товары из корзины. Эта линия поведения реализуется методом `RemoveLine()`. Модульный тест выглядит следующим образом:

```
...
[Fact]
public void Can_Remove_Line() {
  // Организация - создание нескольких тестовых товаров
  Product p1 = new Product { ProductID = 1, Name = "P1" };
  Product p2 = new Product { ProductID = 2, Name = "P2" };
  Product p3 = new Product { ProductID = 3, Name = "P3" };

  // Организация - создание новой корзины
  Cart target = new Cart();

  // Организация - добавление некоторых товаров в корзину
  target.AddItem(p1, 1);
  target.AddItem(p2, 3);
  target.AddItem(p3, 5);
  target.AddItem(p2, 1);

  // Действие
  target.RemoveLine(p2);

  // Утверждение
  Assert.Equal(0, target.Lines.Where(c => c.Product == p2).Count());
  Assert.Equal(2, target.Lines.Count());
}
...
```

Далее проверяется линия поведения, связанная с возможностью вычисления общей стоимости элементов в корзине. Вот модульный тест:

```
...
[Fact]
public void Calculate_Cart_Total() {
  // Организация - создание нескольких тестовых товаров
  Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
  Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

  // Организация - создание новой корзины
  Cart target = new Cart();

  // Действие
  target.AddItem(p1, 1);
  target.AddItem(p2, 1);
  target.AddItem(p1, 3);
  decimal result = target.ComputeTotalValue();

  // Утверждение
  Assert.Equal(450M, result);
}
...
```

Последняя проверка очень проста. Мы должны удостовериться, что в результате очистки корзины ее содержимое корректно удаляется. Ниже показан модульный тест.

```
...
[Fact]
public void Can_Clear_Contents() {
  // Организация - создание нескольких тестовых товаров
  Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
  Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

  // Организация - создание новой корзины
  Cart target = new Cart();

  // Организация - добавление нескольких элементов в корзину
  target.AddItem(p1, 1);
  target.AddItem(p2, 1);

  // Действие - очистка корзины
  target.Clear();

  // Утверждение
  Assert.Equal(0, target.Lines.Count());
}
...
```

Временами, как в данном случае, код для тестирования функциональности класса получается намного длиннее и сложнее, чем код самого класса. Не допускайте, чтобы это приводило к отказу от написания модульных тестов. Дефекты в простых классах, особенно в тех, которые играют настолько важную роль, как `Cart` в приложении SportsStore, могут оказывать разрушительное воздействие.

Создание кнопок добавления в корзину

Нам необходимо модифицировать частичное представление `Views/Shared/ProductSummary.cshtml`, добавив кнопки к спискам товаров. В качестве подготовки добавим в папку `Infrastructure` файл класса по имени `UrlExtensions.cs` с определением расширяющего метода, приведенного в листинге 9.14.

Листинг 9.14. Содержимое файла `UrlExtensions.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;

namespace SportsStore.Infrastructure {

  public static class UrlExtensions {

    public static string PathAndQuery(this HttpRequest request) =>
      request.QueryString.HasValue
        ? $"{request.Path}{request.QueryString}"
        : request.Path.ToString();
  }
}
```

Расширяющий метод `PathAndQuery()` работает с классом `HttpRequest`, используемый в ASP.NET Core для описания HTTP-запроса. Расширяющий метод генерирует URL, по которому браузер будет возвращаться после обновления корзины, принимая во внимание строку запроса, если она есть. В листинге 9.15 в файл импортирования представлений добавляется пространство имен, которое содержит расширяющий метод, так что его можно применять в частичном представлении.

Листинг 9.15. Добавление пространства имен в файле `_ViewImports.cshtml` из папки `Views`

```
@using SportsStore.Models
@using SportsStore.Models.ViewModels
@using SportsStore.Infrastructure
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper SportsStore.Infrastructure.*, SportsStore
```

В листинге 9.16 показано частичное представление с описанием каждого товара, которое обновлено для включения кнопки Add To Cart (Добавить в корзину).

Листинг 9.16. Добавление кнопки в файле `ProductSummary.cshtml` из папки `Views/Shared`

```
@model Product

<div class="card card-outline-primary m-1 p-1">
  <div class="bg-faded p-1">
    <h4>
      @Model.Name
      <span class="badge badge-pill badge-primary" style="float:right">
        <small>@Model.Price.ToString("c")</small>
      </span>
    </h4>
  </div>
  <form id="@Model.ProductID" asp-action="AddToCart"
      asp-controller="Cart" method="post">
    <input type="hidden" asp-for="ProductID" />
    <input type="hidden" name="returnUrl"
        value="@ViewContext.HttpContext.Request.PathAndQuery()" />
    <span class="card-text p-1">
      @Model.Description
      <button type="submit"
          class="btn btn-success btn-sm pull-right" style="float:right">
```

```
        Add To Cart
      </button>
    </span>
  </form>
</div>
```

Мы добавили элемент `form`, содержащий скрытые элементы `input`, которые устанавливают значение `ProductID` из модели представления и URL, куда браузер должен возвращаться после обновления корзины. Элемент `form` и один из элементов `input` конфигурируются с использованием встроенных вспомогательных функций дескрипторов, что является удобным способом генерирования форм, которые содержат значения модели и нацелены на контроллеры и действия в приложении, как описано в главе 24. Во втором элементе `input` применяется расширяющий метод, созданный для установки URL возврата. Кроме того, добавлен элемент `button`, который будет отправлять форму приложению.

На заметку! Обратите внимание, что атрибут `method` элемента `form` установлен в `post`, что инструктирует браузер относительно отправки данных формы с использованием HTTP-метода `POST`. Вы можете это изменить и заставить форму применять HTTP-метод `GET`, но должны соблюдать осторожность. Спецификация HTTP требует, чтобы запросы `GET` были *идемпотентными*, т.е. они не должны приводить к изменениям, а добавление товара в корзину определенно считается изменением. Более подробно такая тема будет обсуждаться в главе 16, включая объяснение того, что может произойти, если проигнорировать требование идемпотентности запросов `GET`.

Включение поддержки сеансов

Мы собираемся сохранять детали корзины пользователя с использованием состояния сеанса, что представляет собой данные, которые хранятся на сервере и ассоциируются с последовательностью запросов, сделанных пользователем. Инфраструктура ASP.NET предлагает целый ряд разных способов хранения состояния сеанса, в том числе хранение его в памяти, что мы и будем применять. Преимуществом такого подхода является простота, но данные сеанса будут утеряны, когда приложение останавливается или перезапускается. Включение поддержки сеансов требует добавления в класс `Startup` служб и промежуточного программного обеспечения (листинг 9.17).

Листинг 9.17. Включение поддержки сеансов в файле `Startup.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
```

```
namespace SportsStore {
  public class Startup {
    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreProducts:ConnectionString"]));
      services.AddTransient<IProductRepository, EFProductRepository>();
      services.AddMvc();
      services.AddMemoryCache();
      services.AddSession();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseSession();
      app.UseMvc(routes => {

        // ...для краткости конфигурация маршрутизации не показана...

      });
      SeedData.EnsurePopulated(app);
    }
  }
}
```

Вызов метода `AddMemoryCache()` настраивает хранилище данных в памяти. Метод `AddSession()` регистрирует службы, используемые для доступа к данным сеанса, а метод `UseSession()` позволяет системе сеансов автоматически ассоциировать запросы с сеансами, когда они поступают от клиента.

Реализация контроллера для корзины

Для обработки щелчков на кнопках Add To Cart понадобится создать контроллер. Добавим в папку `Controllers` файл класса по имени `CartController.cs` с определением, представленным в листинге 9.18.

Листинг 9.18. Содержимое файла `CartController.cs` из папки `Controllers`

```
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;

namespace SportsStore.Controllers {

  public class CartController : Controller {
    private IProductRepository repository;

    public CartController(IProductRepository repo) {
      repository = repo;
    }
```

```
    public RedirectToActionResult AddToCart(int productId, string returnUrl) {
      Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);
      if (product != null) {
        Cart cart = GetCart();
        cart.AddItem(product, 1);
        SaveCart(cart);
      }
      return RedirectToAction("Index", new { returnUrl });
    }
    public RedirectToActionResult RemoveFromCart(int productId,
          string returnUrl) {
      Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);
      if (product != null) {
        Cart cart = GetCart();
        cart.RemoveLine(product);
        SaveCart(cart);
      }
      return RedirectToAction("Index", new { returnUrl });
    }
    private Cart GetCart() {
      Cart cart = HttpContext.Session.GetJson<Cart>("Cart") ?? new Cart();
      return cart;
    }
    private void SaveCart(Cart cart) {
      HttpContext.Session.SetJson("Cart", cart);
    }
  }
}
```

Относительно класса контроллера `CartController` необходимо сделать несколько замечаний. Для сохранения и извлечения объектов `Cart` применяется средство состояния сеанса ASP.NET, с которым и взаимодействует метод `GetCart()`. Промежуточное программное обеспечение, зарегистрированное в предыдущем разделе, использует cookie-наборы или переписывание URL, чтобы ассоциировать вместе множество запросов от определенного пользователя с целью формирования отдельного сеанса просмотра. Связанным средством является состояние сеанса, которое ассоциирует данные с сеансом. Ситуация идеально подходит для класса `Cart`: мы хотим, чтобы каждый пользователь имел собственную корзину, и она сохранялась между запросами. Данные, связанные с сеансом, удаляются по истечении времени существования сеанса (обычно из-за того, что пользователь не отправляет запрос какое-то время), т.е. управлять хранилищем или жизненным циклом объектов `Cart` не придется.

В методах действий `AddToCart()` и `RemoveFromCart()` применялись имена параметров, которые соответствуют именам элементов `input` в HTML-формах, созданных в представлении `ProductSummary.cshtml`. Это позволяет инфраструктуре MVC ассоциировать входящие переменные HTTP-запроса `POST` формы с параметрами и означает, что делать что-то самостоятельно для обработки формы не нужно. Такой процесс называется *привязкой модели* и с его помощью можно значительно упрощать классы контроллеров, как будет объясняться в главе 26.

Определение расширяющих методов состояния сеанса

Средство состояния сеанса в ASP.NET Core хранит только значения `int`, `string` и `byte[]`. Поскольку мы хотим сохранять объект `Cart`, необходимо определить расширяющие методы для интерфейса `ISession`, которые предоставят доступ к данным состояния сеанса с целью сериализации объектов `Cart` в формат JSON и их обратного преобразования. Добавим в папку `Infrastructure` файл класса по имени `SessionExtensions.cs` с определениями расширяющих методов, показанными в листинге 9.19.

Листинг 9.19. Содержимое файла `SessionExtensions.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

namespace SportsStore.Infrastructure {

  public static class SessionExtensions {

    public static void SetJson(this ISession session, string key,
                               object value) {
      session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T GetJson<T>(this ISession session, string key) {
      var sessionData = session.GetString(key);
      return sessionData == null
        ? default(T) : JsonConvert.DeserializeObject<T>(sessionData);
    }
  }
}
```

При сериализации объектов в формат JSON (JavaScript Object Notation — система обозначений для объектов JavaScript) расширяющие методы полагаются на пакет Json.NET, который вы снова встретите в главе 20. Пакет Json.NET не требуется добавлять к проекту, потому что он уже используется "за кулисами" инфраструктурой MVC для поддержки средства заголовков JSON, которое описано в главе 21. (По адресу `https://www.newtonsoft.com/json` доступна информация о том, как работать с пакетом Json.NET напрямую.)

Расширяющие методы облегчают сохранение и извлечение объектов `Cart`. Для добавления объекта `Cart` к состоянию сеанса в контроллере применяется следующий вызов:

```
...
HttpContext.Session.SetJson("Cart", cart);
...
```

Свойство `HttpContext` определено в базовом классе `Controller`, от которого обычно унаследованы контроллеры, и возвращает объект `HttpContext`. Этот объект предоставляет данные контекста о запросе, который был получен, и ответе, находящемся в процессе подготовки. Свойство `HttpContext.Session` возвращает объект, реализующий интерфейс `ISession`. Данный интерфейс является именно тем типом, где мы определили метод `SetJson()`, принимающий аргументы, в которых указываются ключ и объект, подлежащий добавлению в состояние сеанса.

Расширяющий метод сериализирует объект и добавляет его в состояние сеанса, используя функциональность, которая лежит в основе интерфейса ISession.

Для извлечения объекта Cart применяется другой расширяющий метод, которому передается тот же самый ключ:ы

```
...
Cart cart = HttpContext.Session.GetJson<Cart>("Cart");
...
```

Параметр типа позволяет указать тип объекта, который ожидается извлечь; этот тип используется в процессе десериализации.

Отображение содержимого корзины

Финальное замечание о контроллере Cart касается того, что методы AddToCart() и RemoveFromCart() вызывают метод RedirectToAction(). Результатом будет отправка клиентскому браузеру HTTP-инструкции перенаправления, которая заставит браузер запросить новый URL. В данном случае браузер запросит URL, который вызывает метод действия Index() контроллера Cart.

Мы планируем реализовать метод Index() и применять его для отображения содержимого объекта Cart. Взглянув на рис. 9.7 еще раз, можно заметить, что это рабочий поток, инициируемый по щелчку пользователя на кнопке добавления в корзину.

Представлению, которое будет отображать содержимое корзины, необходимо передать две порции информации: объект Cart и URL для отображения в случае, если пользователь щелкнет на кнопке Continue shopping (Продолжить покупку). Для такой цели мы создадим простой класс модели представления. Добавим в папку Models/ViewModels проекта SportsStore файл класса по имени CartIndexViewModel.cs с содержимым, приведенным в листинге 9.20.

Листинг 9.20. Содержимое файла CartIndexViewModel.cs из папки Models/ViewModels

```
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

  public class CartIndexViewModel {
    public Cart Cart { get; set; }
    public string ReturnUrl { get; set; }
  }
}
```

Имея модель представления, можно реализовать метод действия Index() в классе CartController (листинг 9.21).

Листинг 9.21. Реализация метода действия Index() в файле CartController.cs из папки Controllers

```
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;
using SportsStore.Models.ViewModels;
```

```
namespace SportsStore.Controllers {
  public class CartController : Controller {
    private IProductRepository repository;
    public CartController(IProductRepository repo) {
      repository = repo;
    }
    public ViewResult Index(string returnUrl) {
      return View(new CartIndexViewModel {
        Cart = GetCart(),
        ReturnUrl = returnUrl
      });
    }
    // ...для краткости другие методы не показаны...
  }
}
```

Метод действия `Index()` извлекает объект `Cart` из состояния сеанса и использует его при создании объекта `CartIndexViewModel`, который затем передается методу `View()` для применения в качестве модели представления.

Последний шаг, относящийся к отображению содержимого корзины, предусматривает создание представления, которое будет визуализировать действие `Index`. Создадим папку `Views/Cart` и поместим в нее файл представления Razor по имени `Index.cshtml` с разметкой, приведенной в листинге 9.22.

Листинг 9.22. Содержимое файла `Index.cshtml` из папки `Views/Cart`

```
@model CartIndexViewModel

<h2>Your cart</h2>
<table class="table table-bordered table-striped">
  <thead>
    <tr>
      <th>Quantity</th>
      <th>Item</th>
      <th class="text-right">Price</th>
      <th class="text-right">Subtotal</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var line in Model.Cart.Lines) {
      <tr>
        <td class="text-center">@line.Quantity</td>
        <td class="text-left">@line.Product.Name</td>
        <td class="text-right">@line.Product.Price.ToString("c")</td>
        <td class="text-right">
          @((line.Quantity * line.Product.Price).ToString("c"))
        </td>
      </tr>
    }
  </tbody>
```

```
  <tfoot>
    <tr>
      <td colspan="3" class="text-right">Total:</td>
      <td class="text-right">
        @Model.Cart.ComputeTotalValue().ToString("c")
      </td>
    </tr>
  </tfoot>
</table>

<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>
```

Представление проходит по элементам в корзине и добавляет в HTML-таблицу строку для каждого элемента вместе со стоимостью и итоговой суммой по корзине. Классы, назначенные элементам, соответствуют стилям Bootstrap для таблиц и выравнивания текста.

В результате становится доступной базовая функциональность корзины для покупок. Во-первых, товары выводятся вместе с кнопками Add To Cart (Добавить в корзину), как показано на рис. 9.8.

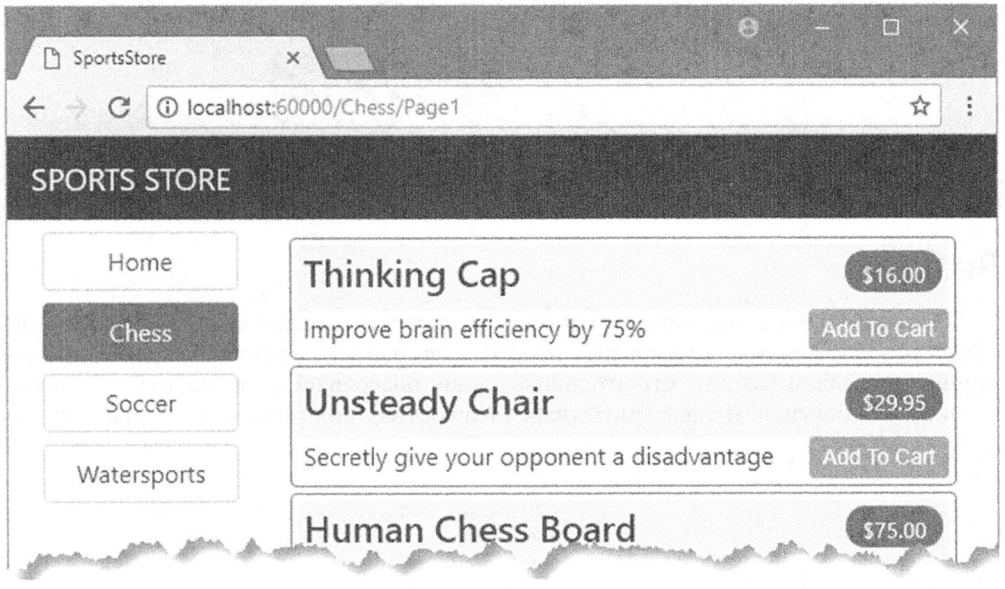


Рис. 9.8. Кнопки Add To Cart

Во-вторых, щелчок пользователя на кнопке Add To Cart приводит к добавлению соответствующего товара в его корзину и отображению сводной информации по корзине (рис. 9.9). Щелчок на кнопке Continue shopping (Продолжить покупку) возвращает пользователя на страницу товара, из которой произошел переход.

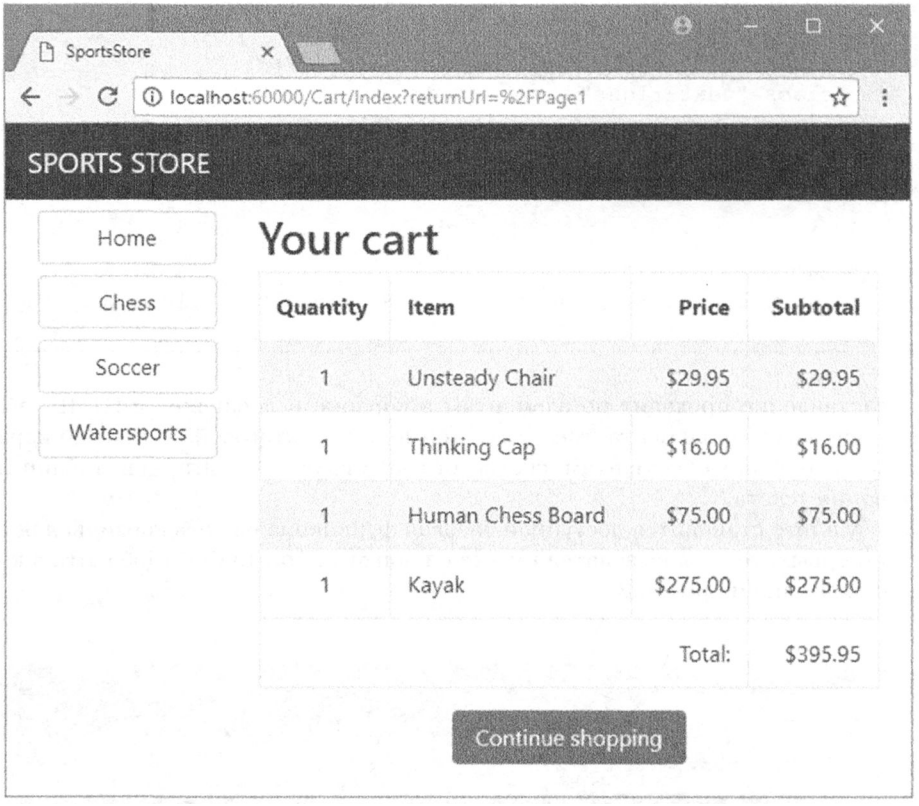


Рис. 9.9. Отображение содержимого корзины для покупок

Резюме

В главе мы начали расширять пользовательские части приложения SportsStore. Мы предоставили средства, с помощью которых пользователь может переходить по категориям, и создали базовые строительные блоки, позволяющие добавлять элементы в корзину для покупок. В следующей главе разработка приложения будет продолжена.

ГЛАВА 10

SportsStore: завершение построения корзины для покупок

В настоящей главе мы продолжим строить пример приложения SportsStore. В предыдущей главе мы добавили базовую поддержку корзины для покупок, а теперь собираемся усовершенствовать и завершить создание этой функциональности.

Усовершенствование модели корзины с помощью службы

В предыдущей главе был определен класс модели `Cart` и продемонстрирована возможность его сохранения с использованием средства состояния сеанса, что давало возможность пользователю формировать набор товаров для покупки. Обязанность по управлению постоянством класса `Cart` возлагалась на контроллер `Cart`, в котором явно определялись методы для получения и сохранения объектов `Cart`.

Проблема такого подхода в том, что нам пришлось дублировать код для получения и сохранения объектов `Cart` во всех компонентах, которые его применяли. В текущем разделе мы воспользуемся средством служб, которое находится в самой основе инфраструктуры ASP.NET Core, чтобы упростить способ управления объектами `Cart`, освобождая индивидуальные компоненты, такие как контроллер `Cart`, от необходимости напрямую иметь дело с деталями.

Службы чаще всего применяются для сокрытия деталей реализации интерфейсов от компонентов, которые от них зависят. Вы видели пример, когда создавалась служба для интерфейса `IProductRepository`, которая позволила гладко заменить фиктивный класс хранилища реальным хранилищем Entity Framework Core. Но службы могут использоваться для решения множества других задач, а также применяться для придания и изменения формы приложения, даже если работа производится с конкретными классами наподобие `Cart`.

Создание класса корзины, осведомленного о хранилище

Первым шагом по приведению в порядок способа использования класса `Cart` будет создание подкласса, который осведомлен о том, как сохранять самого себя с применением состояния сеанса. Добавим в папку `Models` файл класса по имени `SessionCart.cs` и поместим в него определение, показанное в листинге 10.1.

Листинг 10.1. Содержимое файла SessionCart.cs из папки Models

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Newtonsoft.Json;
using SportsStore.Infrastructure;

namespace SportsStore.Models {
  public class SessionCart: Cart {
    public static Cart GetCart(IServiceProvider services) {
      ISession session = services.GetRequiredService<IHttpContextAccessor>()?
        .HttpContext.Session;
      SessionCart cart = session?.GetJson<SessionCart>("Cart")
        ?? new SessionCart();
      cart.Session = session;
      return cart;
    }
    [JsonIgnore]
    public ISession Session { get; set; }
    public override void AddItem(Product product, int quantity) {
      base.AddItem(product, quantity);
      Session.SetJson("Cart", this);
    }
    public override void RemoveLine(Product product) {
      base.RemoveLine(product);
      Session.SetJson("Cart", this);
    }
    public override void Clear() {
      base.Clear();
      Session.Remove("Cart");
    }
  }
}
```

Класс `SessionCart` является производным от класса `Cart` и переопределяет методы `AddItem()`, `RemoveLine()` и `Clear()`, так что они вызывают базовые реализации и затем сохраняют обновленное состояние в сеансе, используя расширяющие методы интерфейса `ISession`, которые были определены в главе 9. Статический метод `GetCart()` — это фабрика для создания объектов `SessionCart` и снабжения их объектом реализации `ISession`, чтобы они могли себя сохранять.

Получение объекта реализации `ISession` несколько затруднено. Мы должны получить экземпляр службы `IHttpContextAccessor`, который предоставит доступ к объекту `HttpContext`, а тот в свою очередь — к объекту реализации `ISession`. Такой непрямой подход требуется из-за того, что сеанс не предоставляется как обычная служба.

Регистрация службы

Следующий шаг заключается в создании службы для класса `Cart`. Цель в том, чтобы удовлетворять запросы для объектов `Cart` выдачей объектов `SessionCart`, которые будут сохранять себя самостоятельно. Создание службы иллюстрируется в листинге 10.2.

Листинг 10.2. Создание службы корзины в файле `Startup.cs` из папки `SportsStore`

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreProducts:ConnectionString"]));
  services.AddTransient<IProductRepository, EFProductRepository>();
  services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
  services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
  services.AddMvc();
  services.AddMemoryCache();
  services.AddSession();
}
...
```

Метод `AddScoped()` указывает, что для удовлетворения связанных запросов к экземплярам `Cart` должен применяться один и тот же объект. Способ связывания запросов может быть сконфигурирован, но по умолчанию это значит, что в ответ на любой запрос экземпляра `Cart` со стороны компонентов, которые обрабатывают тот же самый HTTP-запрос, будет выдаваться один и тот же объект.

Вместо предоставления методу `AddScoped()` отображения между типами, как делалось для хранилища, указывается лямбда-выражение, которое будет выполняться для удовлетворения запросов к `Cart`. Лямбда-выражение получает коллекцию служб, которые были зарегистрированы, и передает ее методу `GetCart()` класса `SessionCart`. В результате запросы для службы `Cart` будут обрабатываться путем создания объектов `SessionCart`, которые сериализируют сами себя как данные сеанса, когда они модифицируются.

Мы также добавили службу с использованием метода `AddSingleton()`, который указывает, что всегда должен применяться один и тот же объект. Созданная служба сообщает инфраструктуре MVC о том, что когда требуются реализации интерфейса `IHttpContextAccessor`, необходимо использовать класс `HttpContextAccessor`. Данная служба обязательна, поэтому в классе `SessionCart` можно получать доступ к текущему сеансу, как делалось в листинге 10.1.

Упрощение контроллера `Cart`

Преимущество создания службы такого вида связано с тем, что она позволит упростить контроллеры, в которых применяются объекты `Cart`. В листинге 10.3 приведен переделанный класс `CartController`, где задействована новая служба.

Листинг 10.3. Использование службы `Cart` в файле `CartController.cs` из папки `Controllers`

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

  public class CartController : Controller {
    private IProductRepository repository;
    private Cart cart;
```

```
    public CartController(IProductRepository repo, Cart cartService) {
      repository = repo;
      cart = cartService;
    }
    public ViewResult Index(string returnUrl) {
      return View(new CartIndexViewModel {
        Cart = cart,
        ReturnUrl = returnUrl
      });
    }
    public RedirectToActionResult AddToCart(int productId, string returnUrl)
    {
      Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);
      if (product != null) {
        cart.AddItem(product, 1);
      }
      return RedirectToAction("Index", new { returnUrl });
    }
    public RedirectToActionResult RemoveFromCart(int productId,
          string returnUrl) {
      Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);
      if (product != null) {
        cart.RemoveLine(product);
      }
      return RedirectToAction("Index", new { returnUrl });
    }
  }
}
```

Класс `CartController` указывает на то, что он нуждается в объекте `Cart`, за счет объявления аргумента конструктора. Это позволяет удалить методы, которые читают и записывают данные в сеанс, а также код, требующийся для записи обновлений. Результатом оказывается контроллер, который не только проще, но и более сосредоточен на своей роли в приложении, не беспокоясь о том, как объекты `Cart` создаются либо хранятся. И поскольку службы доступны по всему приложению, любой компонент может получать корзину пользователя с применением одного и того же приема.

Завершение функциональности корзины

После ввода службы `Cart` наступило время завершить построение функциональности корзины, добавив два новых средства. Первое средство позволит пользователю удалять элемент из корзины. Второе средство даст возможность отображать итоговую информацию по корзине в верхней части страницы.

Удаление элементов из корзины

Мы уже определили и протестировали метод действия `RemoveFromCart()` в контроллере, а потому для предоставления пользователям возможности удаления элементов достаточно лишь открыть доступ к данному методу в представлении, доба-

вив кнопки Remove (Удалить) ко всем строкам в итоговой информации по корзине. Изменения, которые понадобится внести в файл `Views/Cart/Index.cshtml`, показаны в листинге 10.4.

Листинг 10.4. Добавление кнопок Remove в файле `Index.cshtml` из папки `Views/Cart`

```
@model CartIndexViewModel

<h2>Your cart</h2>
<table class="table table-bordered table-striped">
  <thead>
    <tr>
      <th>Quantity</th>
      <th>Item</th>
      <th class="text-right">Price</th>
      <th class="text-right">Subtotal</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var line in Model.Cart.Lines) {
      <tr>
        <td class="text-center">@line.Quantity</td>
        <td class="text-left">@line.Product.Name</td>
        <td class="text-right">@line.Product.Price.ToString("c")</td>
        <td class="text-right">
          @((line.Quantity * line.Product.Price).ToString("c"))
        </td>
        <td>
          <form asp-action="RemoveFromCart" method="post">
            <input type="hidden" name="ProductID"
                  value="@line.Product.ProductID" />
            <input type="hidden" name="returnUrl"
                  value="@Model.ReturnUrl" />
            <button type="submit" class="btn btn-sm btn-danger">
              Remove
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
  <tfoot>
    <tr>
      <td colspan="3" class="text-right">Total:</td>
      <td class="text-right">
        @Model.Cart.ComputeTotalValue().ToString("c")
      </td>
    </tr>
  </tfoot>
</table>

<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>
```

Мы добавили к каждой строке таблицы новый столбец, содержащий элемент `form` со скрытыми элементами `input`, которые указывают товар, подлежащий удалению, и URL возврата, а также кнопку для отправки формы.

Чтобы увидеть кнопки Remove в работе, понадобится запустить приложение и добавить несколько элементов в корзину для покупок. Поскольку корзина уже обладает функциональностью удаления элементов, ее можно протестировать, щелкнув на одной из новых кнопок (рис. 10.1).



Рис. 10.1. Удаление элемента из корзины для покупок

Добавление виджета с итоговой информацией по корзине

Мы имеем функционирующую корзину, но еще должны определить способ ее встраивания в пользовательский интерфейс. Пользователи могут выяснять, что находится в их корзинах, только за счет просмотра экрана со сводкой по корзине. Но попасть на этот экран можно только путем добавления нового элемента в корзину.

Для решения упомянутой задачи мы создадим виджет (графический элемент) с итоговой информацией по содержимому корзины, щелчок на котором будет приводить к отображению товаров, находящихся в корзине, и сделаем его доступным в любом месте приложения. Он будет реализован почти так же, как виджет для навигации — в виде компонента представления, вывод которого может включаться в разделяемую компоновку Razor.

Добавление пакета Font Awesome

В качестве части итоговой информации по корзине мы будем отображать кнопку, которая позволит пользователю перейти к оплате. Вместо отображения каких-либо слов на кнопке мы хотим использовать символ корзины. При отсутствии художественных навыков можно применить пакет с открытым кодом Font Awesome, предлагающий великолепный набор значков, которые допускается интегрировать в приложения как шрифты, где каждый символ шрифта представляет собой отдельное изображение. Получить дополнительные сведения о пакете Font Awesome, а также просмотреть содержащиеся в нем значки, можно по адресу `https://fontawesome.com/`.

Выберем проект SportsStore и щелкнем на кнопке Show All Items (Показать все элементы) в верхней части окна Solution Explorer, чтобы отобразить файл bower.json. Добавим пакет Font Awesome в раздел dependencies (листинг 10.5).

Листинг 10.5. Добавление пакета Font Awesome в файле bower.json из папки SportsStore

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "fontawesome": "4.7.0"
  }
}
```

После сохранения файла bower.json среда Visual Studio с помощью инструмента Bower загрузит и установит пакет Font Awesome в папку wwwroot/lib/fontawesome.

Создание класса компонента представления и представления

Добавим в папку Components файл класса по имени CartSummaryViewComponent.cs и определим в нем компонент представления, показанный в листинге 10.6.

Листинг 10.6. Содержимое файла CartSummaryViewComponent.cs из папки Components

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Components {

  public class CartSummaryViewComponent : ViewComponent {
    private Cart cart;

    public CartSummaryViewComponent(Cart cartService) {
      cart = cartService;
    }

    public IViewComponentResult Invoke() {
      return View(cart);
    }
  }
}
```

Компонент представления CartSummaryViewComponent способен задействовать в своих интересах службу, созданную ранее в главе для получения объекта Cart, принимая ее как аргумент конструктора. Результатом оказывается простой компонент представления, который передает объект Cart методу View(), чтобы сгенерировать фрагмент HTML-разметки для включения в компоновку. Чтобы создать компоновку, создадим папку Views/Shared/Components/CartSummary, добавим в нее файл представления Razor по имени Default.cshtml и поместим в него разметку, приведенную в листинге 10.7.

Листинг 10.7. Содержимое файла `Default.cshtml` из папки `Views/Shared/Components/CartSummary`

```
@model Cart

<div class="">
  @if (Model.Lines.Count() > 0) {
    <small class="navbar-text">
      <b>Your cart:</b>
      @Model.Lines.Sum(x => x.Quantity) item(s)
      @Model.ComputeTotalValue().ToString("c")
    </small>
  }
  <a class="btn btn-sm btn-secondary navbar-btn"
     asp-controller="Cart" asp-action="Index"
     sp-route-returnurl="@ViewContext.HttpContext.Request.PathAndQuery()">
       <i class="fa fa-shopping-cart"></i>
  </a>
</div>
```

Представление отображает кнопку со значком корзины Font Awesome и когда в корзине присутствуют товары, предоставляет снимок, который сообщает количество элементов и общую сумму. Теперь, имея компонент представления и представление, можно модифицировать разделяемую компоновку, чтобы итоговая информация по корзине включалась в ответы, генерируемые контроллерами приложения (листинг 10.8).

Листинг 10.8. Добавление итоговой информации по корзине в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet"
       asp-href-include="/lib/bootstrap/dist/**/*.min.css"
       asp-href-exclude="**/*-reboot*,**/*-grid*" />
  <link rel="stylesheet" asp-href-include="/lib/fontawesome/css/*.css" />
  <title>SportsStore</title>
</head>
<body>
  <div class="navbar navbar-inverse bg-inverse" role="navigation">
    <div class="row">
      <a class="col navbar-brand" href="#">SPORTS STORE</a>
      <div class="col-4 text-right">
        @await Component.InvokeAsync("CartSummary")
      </div>
    </div>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      @await Component.InvokeAsync("NavigationMenu")
    </div>
```

```
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

Запустив приложение, можно увидеть итоговую информацию по корзине. Когда корзина пуста, отображается только кнопка Continue shopping (Продолжить покупки). По мере добавления элементов в корзину выводится их количество и общая стоимость (рис. 10.2). Благодаря такому дополнению пользователь знает, какие товары находятся в корзине, и получает очевидный путь для перехода к оплате покупок.



Рис. 10.2. Отображение итоговой информации по корзине

Отправка заказов

Итак, мы добрались до финального пользовательского средства приложения SportsStore: возможности перехода к оплате и оформлению заказа. В последующих разделах мы расширим модель предметной области, чтобы обеспечить поддержку получения от пользователя подробной информации о доставке, и добавим в приложение обработку этой информации.

Создание класса модели

Добавим в папку `Models` файл класса по имени `Order.cs` и приведем его содержимое в соответствие с листингом 10.9. Класс `Order` будет использоваться для представления информации о доставке пользователю.

Листинг 10.9. Содержимое файла `Order.cs` из папки `Models`

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

  public class Order {
```

```
    [BindNever]
    public int OrderID { get; set; }
    [BindNever]
    public ICollection<CartLine> Lines { get; set; }
    [Required(ErrorMessage = "Please enter a name")]
                            // Введите имя
    public string Name { get; set; }
    [Required(ErrorMessage = "Please enter the first address line")]
                            // Введите первую строку адреса
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string Line3 { get; set; }
    [Required(ErrorMessage = "Please enter a city name")]
                            // Введите название города
    public string City { get; set; }
    [Required(ErrorMessage = "Please enter a state name")]
                            // Введите название штата
    public string State { get; set; }
    public string Zip { get; set; }
    [Required(ErrorMessage = "Please enter a country name")]
                            // Введите название страны
    public string Country { get; set; }
    public bool GiftWrap { get; set; }
  }
}
```

Здесь применяются атрибуты проверки достоверности из пространства имен `System.ComponentModel.DataAnnotations`, как мы делали в главе 2. Проверка достоверности подробно рассматривается в главе 27.

Кроме того, в классе `Order` используется атрибут `BindNever`, который предотвращает предоставление пользователем значений для снабженных этим атрибутом свойств в HTTP-запросе. Такая возможность системы привязки моделей, описанная в главе 26, останавливает применение инфраструктурой MVC значений из HTTP-запроса для заполнения конфиденциальных или важных свойств модели.

Добавление реализации процесса оплаты

Наша цель заключается в том, чтобы обеспечить пользователям возможность ввода информации о доставке и отправки заказа. Для начала потребуется добавить к представлению итоговой информации по корзине кнопку Checkout (Перейти к оплате). Изменения, которые необходимо внести в файл `Views/Cart/Index.cshtml`, показаны в листинге 10.10.

Листинг 10.10. Добавление кнопки Checkout в файле `Index.cshtml` из папки `Views/Cart`

```
...
<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
  <a class="btn btn-primary" asp-action="Checkout"
     asp-controller="Order">
    Checkout
  </a>
</div>
...
```

Изменения обеспечивают генерацию ссылки, стилизованной в виде кнопки, щелчок на которой приводит к вызову метода действия `Checkout()` контроллера `Order`, создаваемого в следующем разделе. На рис. 10.3 показано, как выглядит кнопка Checkout.

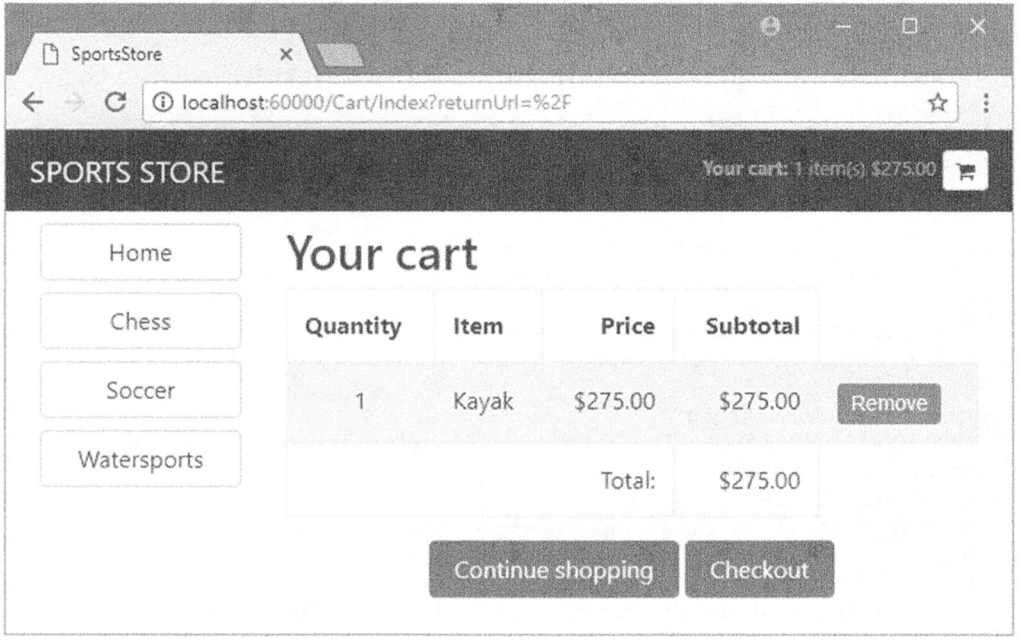


Рис. 10.3. Кнопка Checkout

Теперь понадобится определить контроллер `Order`. Добавим в папку `Controllers` файл класса по имени `OrderController.cs` с определением, приведенным в листинге 10.11.

Листинг 10.11. Содержимое файла `OrderController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

  public class OrderController : Controller {

    public ViewResult Checkout() => View(new Order());
  }
}
```

Метод `Checkout()` возвращает стандартное представление и передает новый объект `ShippingDetails` в качестве модели представления. Чтобы создать представление, создадим папку `Views/Order` и поместим в нее файл представления Razor по имени `Checkout.cshtml` с разметкой, показанной в листинге 10.12.

Листинг 10.12. Содержимое файла `Checkout.cshtml` из папки `Views/Order`

```
@model Order

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

<form asp-action="Checkout" method="post">
  <h3>Ship to</h3>
  <div class="form-group">
    <label>Name:</label><input asp-for="Name" class="form-control" />
  </div>
  <h3>Address</h3>
  <div class="form-group">
    <label>Line 1:</label><input asp-for="Line1" class="form-control" />
  </div>
  <div class="form-group">
    <label>Line 2:</label><input asp-for="Line2" class="form-control" />
  </div>
  <div class="form-group">
    <label>Line 3:</label><input asp-for="Line3" class="form-control" />
  </div>
  <div class="form-group">
    <label>City:</label><input asp-for="City" class="form-control" />
  </div>
  <div class="form-group">
    <label>State:</label><input asp-for="State" class="form-control" />
  </div>
  <div class="form-group">
    <label>Zip:</label><input asp-for="Zip" class="form-control" />
  </div>
  <div class="form-group">
    <label>Country:</label><input asp-for="Country" class="form-control" />
  </div>
  <h3>Options</h3>
  <div class="checkbox">
    <label>
      <input asp-for="GiftWrap" /> Gift wrap these items
    </label>
  </div>
  <div class="text-center">
    <input class="btn btn-primary" type="submit" value="Complete Order" />
  </div>
</form>
```

Для каждого свойства в модели мы создали элементы `label` и `input` для пользовательского ввода, сформатированные с помощью Bootstrap. Атрибут `asp-for` в элементах `input` обрабатывается встроенной вспомогательной функцией дескриптора, которая генерирует атрибуты `type`, `id`, `name` и `value` на основе указанного свойства модели, как объясняется в главе 24.

Чтобы увидеть результат добавления нового метода действия и представления (рис. 10.4), запустим приложение, щелкнем на кнопке со значком корзины в верхней части страницы и затем на кнопке Checkout (Оплата). Попасть на данное представление можно также, запросив URL вида `/Order/Checkout`.

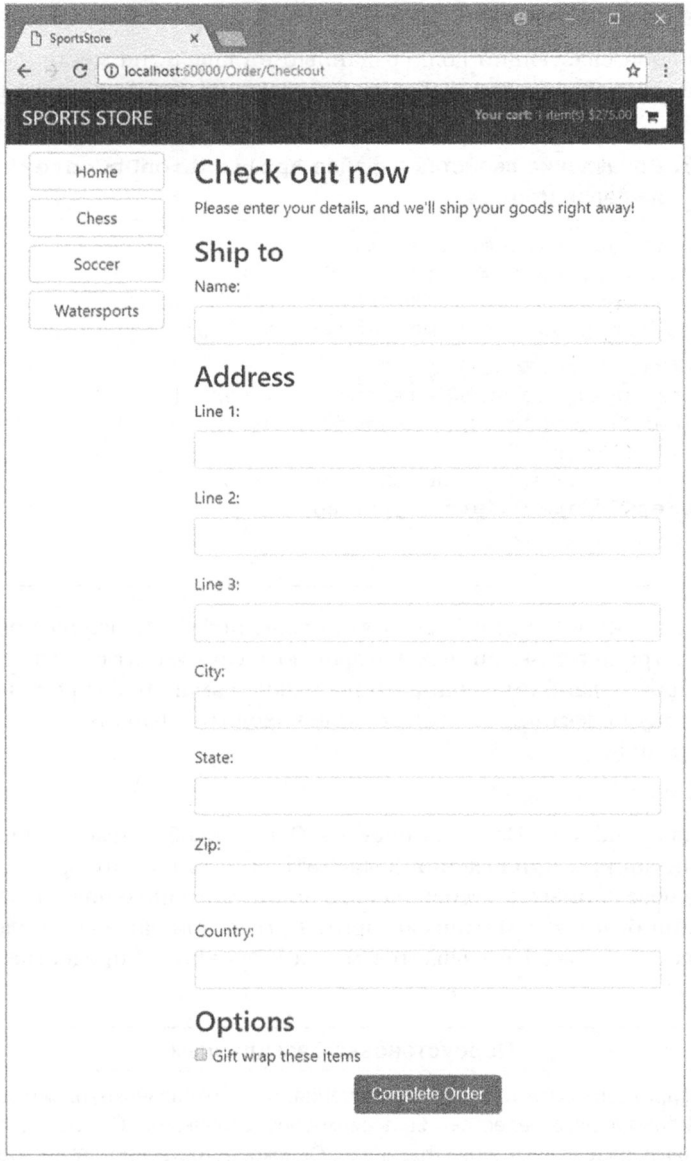


Рис. 10.4. Форма для сбора деталей о доставке

Реализация обработки заказов

Мы будем обрабатывать заказы, записывая их в базу данных. Разумеется, большинство сайтов электронной коммерции на этом не останавливаются, но мы не станем заниматься обработкой кредитных карт или других форм оплаты. Чтобы сосредоточиться на инфраструктуре MVC, вполне достаточно простого сохранения в базе данных.

Расширение базы данных

Когда основной связующий код, созданный в главе 8, на месте, добавить в базу данных новый вид модели легко. Добавим в класс контекста базы данных новое свойство, как показано в листинге 10.13.

Листинг 10.13. Добавление свойства в файле `ApplicationDbContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {
  public class ApplicationDbContext : DbContext {
   public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
     : base(options) { }
    public DbSet<Product> Products { get; set; }
    public DbSet<Order> Orders { get; set; }
  }
}
```

Такого изменения достаточно для того, чтобы инфраструктура Entity Framework Core создала миграцию базы данных, которая позволит объектам `Order` сохраняться в базе данных. Для создания миграции откроем окно командной строки или PowerShell, перейдем в папку проекта `SportsStore` (где находится файл `Startup.cs`) и введем следующую команду:

```
dotnet ef migrations add Orders
```

Эта команда сообщает Entity Framework Core о необходимости получить новый снимок модели данных приложения, выяснить его отличия от предыдущей версии, хранящейся в базе данных, и сгенерировать новую миграцию под названием `Orders`. Новая миграция будет автоматически применяться при запуске приложения, потому что в классе `SeedData` вызывается метод `Migrate()`, предоставляемый Entity Framework Core.

Переустановка базы данных

Если в модель приходится часто вносить изменения, то может возникнуть ситуация, когда миграции и схема базы данных перестают быть синхронизированными. Самое простое, что можно сделать — удалить базу данных и начать заново. Однако, естественно, такой подход приемлем только на стадии разработки, потому что все сохраненные данные будут утрачены.

Чтобы удалить базу данных, необходимо ввести приведенную ниже команду, находясь в папке проекта `SportsStore`:

```
dotnet ef database drop --force
```

После того, как база данных удалена, ввод следующей команды в папке проекта `SportsStore` приводит к построению базы данных заново и применению созданных ранее миграций:

```
dotnet ef database update
```

В результате база данных переустанавливается, точно отражая модель, что дает возможность возвратиться к разработке приложения.

Создание хранилища заказов

Чтобы предоставить доступ к объектам `Order`, мы последуем тому же самому шаблону, который использовался для хранилища товаров. Добавим в папку `Models` новый файл по имени `IOrderRepository.cs` и определим в нем интерфейс, показанный в листинге 10.14.

Листинг 10.14. Содержимое файла `IOrderRepository.cs` из папки `Models`

```
using System.Linq;

namespace SportsStore.Models {

  public interface IOrderRepository {

    IQueryable<Order> Orders { get; }
    void SaveOrder(Order order);
  }
}
```

Для реализации интерфейса хранилища заказов добавим в папку `Models` файл класса по имени `EFOrderRepository.cs` с определением, представленным в листинге 10.15.

Листинг 10.15. Содержимое файла `EFOrderRepository.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace SportsStore.Models {

  public class EFOrderRepository : IOrderRepository {
    private ApplicationDbContext context;

    public EFOrderRepository(ApplicationDbContext ctx) {
      context = ctx;
    }

    public IQueryable<Order> Orders => context.Orders
      .Include(o => o.Lines)
      .ThenInclude(l => l.Product);

    public void SaveOrder(Order order) {
      context.AttachRange(order.Lines.Select(l => l.Product));
      if (order.OrderID == 0) {
        context.Orders.Add(order);
      }
      context.SaveChanges();
    }
  }
}
```

Класс `EFOrderRepository` реализует интерфейс `IOrderRepository` с применением Entity Framework Core, позволяя извлекать набор сохраненных объектов `Order` и создавать либо изменять заказы.

Особенности хранилища заказов

Реализация хранилища для заказов в листинге 10.15 требует небольшой дополнительной работы. Инфраструктуру Entity Framework Core необходимо проинструктировать о загрузке связанных данных, если они охватывают несколько таблиц. В листинге 10.15 с помощью методов `Include()` и `ThenInclude()` указано, что когда объект `Order` читается из базы данных, то также должна загружаться коллекция, ассоциированная со свойством `Lines`, наряду с объектами `Product`, которые связаны с элементами коллекции:

```
...
public IQueryable<Order> Orders => context.Orders
  .Include(o => o.Lines)
  .ThenInclude(l => l.Product);
...
```

Такой подход гарантирует получение всех нужных объектов данных, не выполняя запросы и не собирая данные напрямую.

Дополнительный шаг требуется и при сохранении объекта `Order` в базе данных. Когда данные корзины пользователя десериализируются из состояния сеанса, пакет JSON создает новые объекты, не известные инфраструктуре Entity Framework Core, которая затем пытается записать все объекты в базу данных. В случае объектов `Product` это означает, что инфраструктура Entity Framework Core попытается записать объекты, которые уже были сохранены, что приведет к ошибке. Во избежание проблемы мы уведомляем Entity Framework Core о том, что объекты существуют и не должны сохраняться в базе данных до тех пор, пока они не будут модифицированы:

```
...
context.AttachRange(order.Lines.Select(l => l.Product));
...
```

В результате инфраструктура Entity Framework Core не будет пытаться записывать десериализированные объекты `Product`, которые ассоциированы с объектом `Order`.

В листинге 10.16 хранилище заказов регистрируется как служба внутри метода `ConfigureServices()` класса `Startup`.

Листинг 10.16. Регистрация службы хранилища заказов в файле `Startup.cs` из папки `SportsStore`

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreProducts:ConnectionString"]));
  services.AddTransient<IProductRepository, EFProductRepository>();
  services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
  services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
  services.AddTransient<IOrderRepository, EFOrderRepository>();
  services.AddMvc();
  services.AddMemoryCache();
  services.AddSession();
}
...
```

Завершение построения контроллера Order

Для завершения класса OrderController понадобится модифицировать конструктор так, чтобы он получал службы, требующиеся ему для обработки заказа, и добавить новый метод действия, который будет обрабатывать HTTP-запрос POST формы, когда пользователь щелкает на кнопке Complete order (Завершить заказ). Оба изменения показаны в листинге 10.17.

Листинг 10.17. Завершение контроллера в файле OrderController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {
  public class OrderController : Controller {
    private IOrderRepository repository;
    private Cart cart;
    public OrderController(IOrderRepository repoService, Cart cartService) {
      repository = repoService;
      cart = cartService;
    }
    public ViewResult Checkout() => View(new Order());
    [HttpPost]
    public IActionResult Checkout(Order order) {
      if (cart.Lines.Count() == 0) {
        ModelState.AddModelError("", "Sorry, your cart is empty!");
                                    // Корзина пуста!
      }
      if (ModelState.IsValid) {
        order.Lines = cart.Lines.ToArray();
        repository.SaveOrder(order);
        return RedirectToAction(nameof(Completed));
      } else {
        return View(order);
      }
    }
    public ViewResult Completed() {
      cart.Clear();
      return View();
    }
  }
}
```

Метод действия Checkout() декорирован атрибутом HttpPost, т.е. он будет вызываться для запроса POST — в этом случае, когда пользователь отправляет форму. Мы снова полагаемся на систему привязки моделей, так что можно получить объект Order, дополнить его данными из объекта Cart и сохранить в хранилище.

Инфраструктура MVC контролирует ограничения проверки достоверности, которые были применены к классу Order посредством атрибутов аннотаций данных, и через свойство ModelState сообщает методу действия о любых проблемах. Чтобы выяснить, есть ли проблемы, мы проверяем свойство ModelState.IsValid. Мы вызы-

ваем метод `ModelState.AddModelError()` для регистрации сообщения об ошибке, если в корзине нет элементов. Вскоре мы объясним, как отображать такие сообщения об ошибках, а более подробное описание привязки моделей и проверки достоверности будет представлено в главах 27 и 28.

Модульное тестирование: обработка заказа

Чтобы выполнить модульное тестирование класса `OrderController`, необходимо проверить поведение версии `POST` метода `Checkout()`. Хотя метод выглядит коротким и простым, использование привязки моделей MVC означает наличие многих вещей, происходящих "за кулисами", которые должны быть протестированы.

Мы хотим обрабатывать заказ, только если в корзине присутствуют элементы *и* пользователь предоставил достоверные детали о доставке. При любых других обстоятельствах пользователю должно быть сообщено об ошибке. Вот первый тестовый метод, который определен в файле класса по имени `OrderControllerTests.cs` внутри проекта `SportsStore.Tests`:

```
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {
  public class OrderControllerTests {
    [Fact]
    public void Cannot_Checkout_Empty_Cart() {
      // Организация - создание имитированного хранилища заказов
      Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
      // Организация - создание пустой корзины
      Cart cart = new Cart();
      // Организация - создание заказа
      Order order = new Order();
      // Организация - создание экземпляра контроллера
      OrderController target = new OrderController(mock.Object, cart);
      // Действие
      ViewResult result = target.Checkout(order) as ViewResult;
      // Утверждение - проверка, что заказ не был сохранен
      mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Never);
      // Утверждение - проверка, что метод возвращает стандартное представление
      Assert.True(string.IsNullOrEmpty(result.ViewName));
      // Утверждение - проверка, что представлению передана
      // недопустимая модель
      Assert.False(result.ViewData.ModelState.IsValid);
    }
  }
}
```

Тест проверяет отсутствие возможности перехода к оплате при пустой корзине. Мы удостоверяемся, что метод `SaveOrder()` имитированной реализации `IOrderRepository` никогда не вызывается, что метод возвращает стандартное представление (которое повторно отобразит введенные пользователем данные, давая ему шанс откорректировать их) и что состояние модели, передаваемое представлению, помечено как недопустимое. Это может выглядеть как излишне ограничивающий набор утверждений, но для проверки правильности

поведения нужны все три утверждения. Следующий тестовый метод работает в основном так же, но внедряет в модель представления ошибку, эмулирующую проблему, о которой сообщает средство привязки модели (что должно происходить в производственной среде, когда пользователь вводит некорректные данные о доставке):

```
...
[Fact]
public void Cannot_Checkout_Invalid_ShippingDetails() {
  // Организация - создание имитированного хранилища заказов
  Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
  // Организация - создание корзины с одним элементом
  Cart cart = new Cart();
  cart.AddItem(new Product(), 1);
  // Организация - создание экземпляра контроллера
  OrderController target = new OrderController(mock.Object, cart);
  // Организация - добавление ошибки в модель
  target.ModelState.AddModelError("error", "error");
  // Действие - попытка перехода к оплате
  ViewResult result = target.Checkout(new Order()) as ViewResult;
  // Утверждение - проверка, что заказ не был сохранен
  mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Never);
  // Утверждение - проверка, что метод возвращает стандартное представление
  Assert.True(string.IsNullOrEmpty(result.ViewName));
  // Утверждение - проверка, что представлению передается недопустимая модель
  Assert.False(result.ViewData.ModelState.IsValid);
}
...
```

Удостоверившись в том, что пустая корзина или некорректные данные о доставке предотвращают сохранение заказа, необходимо проверить, что нормальные заказы сохраняются должным образом. Ниже приведен тест.

```
...
[Fact]
public void Can_Checkout_And_Submit_Order() {
  // Организация - создание имитированного хранилища заказов
  Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
  // Организация - создание корзины с одним элементом
  Cart cart = new Cart();
  cart.AddItem(new Product(), 1);
  // Организация - создание экземпляра контроллера
  OrderController target = new OrderController(mock.Object, cart);
  // Действие - попытка перехода к оплате
  RedirectToActionResult result =
    target.Checkout(new Order()) as RedirectToActionResult;
  // Утверждение - проверка, что заказ был сохранен
  mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Once);
  // Утверждение - проверка, что метод перенаправляется на действие Completed
  Assert.Equal("Completed", result.ActionName);
}
...
```

Тестировать возможность идентификации допустимых сведений о доставке не нужно. Это автоматически обрабатывается средством привязки моделей с использованием атрибутов, примененных к свойствам класса `Order`.

Отображение сообщений об ошибках проверки достоверности

Для проверки пользовательских данных инфраструктура MVC будет использовать атрибуты проверки достоверности, примененные к классу `Order`. Тем не менее, чтобы отобразить сообщения о проблемах, понадобится внести небольшое изменение. Здесь задействована еще одна встроенная вспомогательная функция дескриптора, которая инспектирует состояние проверки достоверности данных, предоставленных пользователем, и добавляет предупреждающие сообщения для каждой обнаруженной проблемы. В листинге 10.18 демонстрируется добавление в файл `Checkout.cshtml` элемента HTML, который будет обрабатываться этой вспомогательной функцией дескриптора.

Листинг 10.18. Добавление области с итогами проверки достоверности в файле `Checkout.cshtml` из папки `Views/Order`

```
@model Order

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>
<div asp-validation-summary="All" class="text-danger"></div>
<form asp-action="Checkout" method="post">
  <h3>Ship to</h3>
...
```

Благодаря такому простому изменению пользователю отображаются сообщения об ошибках проверки достоверности. Чтобы увидеть результат, понадобится посетить URL вида `/Order/Checkout` и попробовать перейти к оплате, не выбрав ни одного товара или не указав сведения о доставке (рис. 10.5). Вспомогательная функция дескриптора, генерирующая такие сообщения, является частью системы проверки достоверности моделей, которая подробно рассматривается в главе 27.

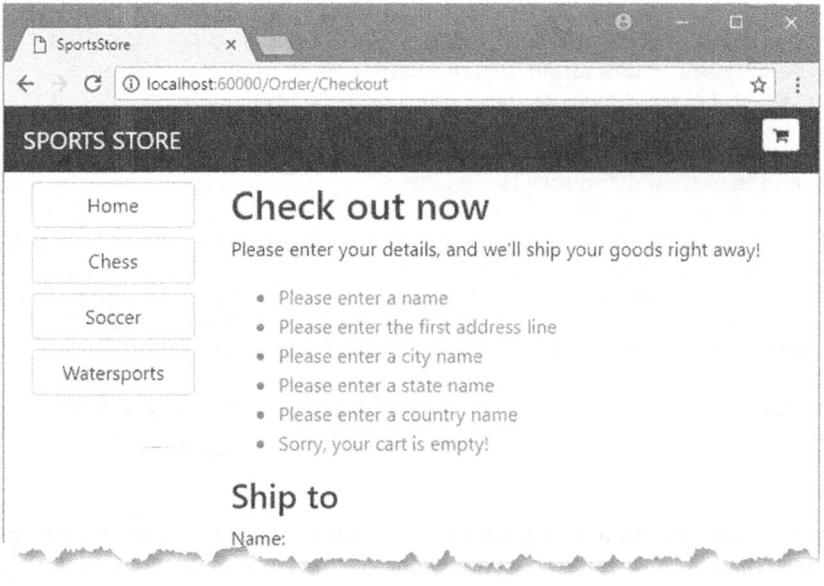


Рис. 10.5. Отображение сообщений об ошибках проверки достоверности

Совет. Данные, отправленные пользователем, перед проверкой посылаются серверу, что известно как *проверка достоверности на стороне сервера*, для которой инфраструктура MVC предлагает великолепную поддержку. Проблема с проверкой достоверности на стороне сервера заключается в том, что пользователю сообщается об ошибках лишь после того, как данные отправлены серверу и обработаны, а также сгенерирована результирующая страница — на занятом сервере все это может занять несколько секунд. По указанной причине проверка достоверности на стороне сервера обычно дополняется проверкой достоверности на стороне клиента, при которой введенные пользователем значения проверяются с помощью кода JavaScript до отправки серверу данных формы. Проверка достоверности на стороне клиента будет описана в главе 27.

Отображение итоговой страницы

Чтобы завершить реализацию процесса оплаты, необходимо создать представление, которое будет отображаться, когда браузер перенаправляется на действие `Completed` контроллера `Order`. Добавим в папку `Views/Order` файл представления Razor по имени `Completed.cshtml` и поместим в него разметку, приведенную в листинге 10.19.

Листинг 10.19. Содержимое файла `Completed.cshtml` из папки `Views/Order`

```
<h2>Thanks!</h2>
<p>Thanks for placing your order.</p>
<p>We'll ship your goods as soon as possible.</p>
```

Для интеграции представления `Completed` в приложение никаких изменений в код вносить не придется, поскольку требуемые операторы уже были добавлены при определении метода действия `Completed()` в листинге 10.17. Теперь пользователь может проходить через весь процесс, начиная с выбора товаров и заканчивая переходом к оплате. При условии, что пользователь предоставил корректные сведения о доставке (и в корзине есть какие-то товары), после щелчка на кнопке Complete order он увидит итоговую страницу (рис. 10.6).

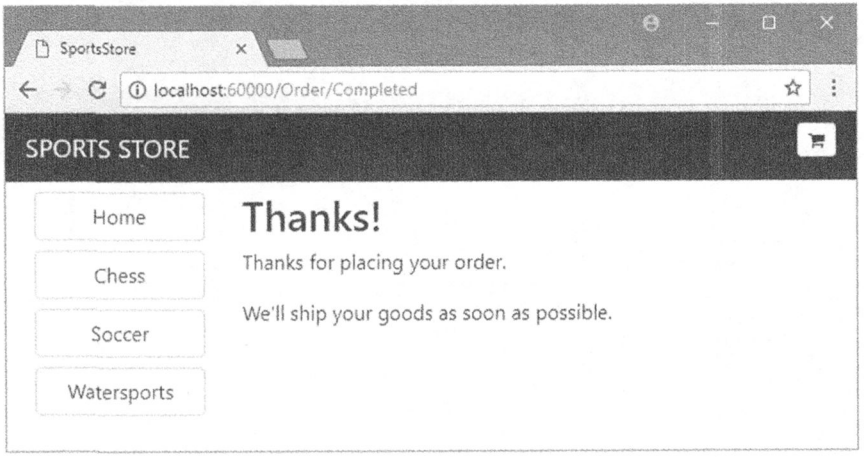


Рис. 10.6. Итоговая страница завершенного заказа

Резюме

Мы завершили все основные части приложения SportsStore, отвечающие за взаимодействие с пользователями. Конечно, до сайта Amazon приложению далеко, но в нем имеется каталог товаров с возможностью просмотра по категориям и страницам, аккуратная корзина для покупок и простой процесс оплаты.

Архитектура с хорошим разделением означает, что мы можем легко изменять поведение любой порции приложения, не беспокоясь о возникновении проблем или несовместимости где-либо в приложении. Например, мы могли бы изменить способ сохранения заказов, что никак бы не повлияло на корзину для покупок, каталог товаров или любую другую область приложения. В следующей главе мы добавим средства, необходимые для администрирования приложения SportsStore.

ГЛАВА 11

SportsStore: администрирование

В настоящей главе мы продолжим строить приложение SportsStore, чтобы предоставить администратору сайта способ управления заказами и товарами.

Управление заказами

В предыдущей главе была добавлена поддержка для получения заказов от пользователей и сохранения их в базе данных. В этой главе мы собираемся создать простой инструмент администрирования, который позволит просматривать полученные заказы и помечать их как отгруженные.

Расширение модели

Первым изменением, которое необходимо внести, является расширение модели, чтобы можно было фиксировать, какие заказы были отгружены. В листинге 11.1 показано добавление нового свойства в класс `Order`, который определен в файле `Order.cs` внутри папки `Models`.

Листинг 11.1. Добавление свойства в файле `Order.cs` из папки `Models`

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

  public class Order {

    [BindNever]
    public int OrderID { get; set; }
    [BindNever]
    public ICollection<CartLine> Lines { get; set; }

    [BindNever]
    public bool Shipped { get; set; }

    [Required(ErrorMessage = "Please enter a name")]
                            // Введите имя
    public string Name { get; set; }

    [Required(ErrorMessage = "Please enter the first address line")]
                             // Введите первую строку адреса
```

```
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string Line3 { get; set; }

    [Required(ErrorMessage = "Please enter a city name")]
                            // Введите название города
    public string City { get; set; }

    [Required(ErrorMessage = "Please enter a state name")]
                            // Введите название штата
    public string State { get; set; }
    public string Zip { get; set; }

    [Required(ErrorMessage = "Please enter a country name")]
                            // Введите название страны
    public string Country { get; set; }
    public bool GiftWrap { get; set; }
  }
}
```

Такой итеративный подход к расширению и адаптации модели с целью поддержки различных средств типичен при разработке приложений MVC. В идеальном мире у нас была бы возможность полностью определить классы моделей в начале проекта и просто строить приложение на их основе, но так случается только в простейших проектах, а на практике следует ожидать итеративную разработку по мере понимания того, что требуется разрабатывать и развивать.

Миграции Entity Framework Core облегчают этот процесс, поскольку нам не приходится вручную удерживать схему базы данных в синхронизированном состоянии с классами моделей, создавая и запуская команды SQL. Обновим базу данных, чтобы отразить добавление свойства `Shipped` в класс `Order`, открыв окно командной строки или PowerShell, перейдя в папку проекта `SportsStore` (ту, что содержит файл `Startup.cs`) и выполнив следующую команду:

```
dotnet ef migrations add ShippedOrders
```

Миграция будет применена автоматически, когда приложение запустится и в классе `SeedData` произойдет обращение к методу `Migrate()`, предоставленному инфраструктурой Entity Framework Core.

Добавление действий и представления

Функциональность, требуемая для отображения и обновления набора заказов в базе данных, относительно проста, потому что она строится на основе средств и инфраструктуры, которые были созданы в предшествующих главах. В листинге 11.2 к контроллеру `Order` добавляются два метода действий.

Листинг 11.2. Добавление двух методов действий в файле `OrderController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

  public class OrderController : Controller {
    private IOrderRepository repository;
```

```
    private Cart cart;

    public OrderController(IOrderRepository repoService, Cart cartService) {
      repository = repoService;
      cart = cartService;
    }

    public ViewResult List() =>
      View(repository.Orders.Where(o => !o.Shipped));

    [HttpPost]
    public IActionResult MarkShipped(int orderID) {
      Order order = repository.Orders
        .FirstOrDefault(o => o.OrderID == orderID);
      if (order != null) {
        order.Shipped = true;
        repository.SaveOrder(order);
      }
      return RedirectToAction(nameof(List));
    }

    public ViewResult Checkout() => View(new Order());

    [HttpPost]
    public IActionResult Checkout(Order order) {
      if (cart.Lines.Count() == 0) {
        ModelState.AddModelError("", "Sorry, your cart is empty!");
      }
      if (ModelState.IsValid) {
        order.Lines = cart.Lines.ToArray();
        repository.SaveOrder(order);
        return RedirectToAction(nameof(Completed));
      } else {
        return View(order);
      }
    }

    public ViewResult Completed() {
      cart.Clear();
      return View();
    }
  }
}
```

Метод `List()` выбирает все объекты `Order` в хранилище, свойство `Shipped` которых имеет значение `false`, и передает их стандартному представлению. Метод действия `List()` будет использоваться для отображения администратору списка неотгруженных заказов.

Метод `MarkShipped()` будет получать запрос `POST`, указывающий идентификатор заказа, который применяется для извлечения соответствующего объекта `Order` из хранилища, чтобы установить его свойство `Shipped` в `true` и сохранить.

Для отображения списка неотгруженных заказов добавим в папку `Views/Order` файл представления Razor по имени `List.cshtml` и поместим в него разметку из листинга 11.3. Элемент `table` используется для отображения ряда деталей, включая сведения о приобретенных товарах.

Листинг 11.3. Содержимое файла `List.cshtml` из папки `Views/Order`

```
@model IEnumerable<Order>

@{
  ViewBag.Title = "Orders";
  Layout = "_AdminLayout";
}
@if (Model.Count() > 0) {
  <table class="table table-bordered table-striped">
    <tr><th>Name</th><th>Zip</th><th colspan="2">Details</th><th></th></tr>
    @foreach (Order o in Model) {
      <tr>
        <td>@o.Name</td><td>@o.Zip</td><th>Product</th><th>Quantity</th>
        <td>
          <form asp-action="MarkShipped" method="post">
            <input type="hidden" name="orderId" value="@o.OrderID" />
            <button type="submit" class="btn btn-sm btn-danger">
              Ship
            </button>
          </form>
        </td>
      </tr>
      @foreach (CartLine line in o.Lines) {
        <tr>
          <td colspan="2"></td>
          <td>@line.Product.Name</td><td>@line.Quantity</td>
          <td></td>
        </tr>
      }
    }
  </table>
} else {
  <div class="text-center">No Unshipped Orders</div>
}
```

Каждый заказ отображается с кнопкой Ship (Отгрузить), которая отправляет форму методу действия `MarkShipped()`. С помощью свойства `Layout` для представления `List` указана другая компоновка, которая переопределяет компоновку, заданную в файле `_ViewStart.cshtml`.

Для добавления компоновки создадим в папке `Views/Shared` файл по имени `_AdminLayout.cshtml` с применением шаблона элемента MVC View Layout Page (Страница компоновки представления MVC) и поместим в него разметку, показанную в листинге 11.4.

Листинг 11.4. Содержимое файла `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @RenderBody()
</body>
</html>
```

Чтобы просматривать и управлять заказами в приложении, запустим приложение, выберем некоторые товары и перейдем к оплате. Затем посетим URL вида `/Order/List`. Появится сводка по созданным заказам (рис. 11.1). Щелкнем на кнопке Ship; база данных обновится, а список ожидающих заказов будет пуст.

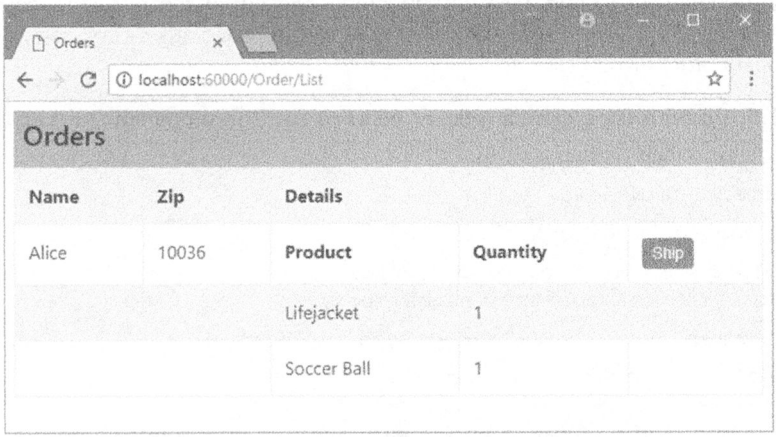


Рис. 11.1. Управление заказами

На заметку! В настоящий момент ничто не может воспрепятствовать запросу пользователем URL вида `/Order/List` и администрированию своего заказа. В главе 12 объясняется, как ограничить доступ к методам действий.

Добавление средств управления каталогом

Соглашение для управления более сложными коллекциями элементов предусматривает предоставление пользователю страниц двух типов: страницы *списка* и страницы *редактирования* (рис. 11.2).

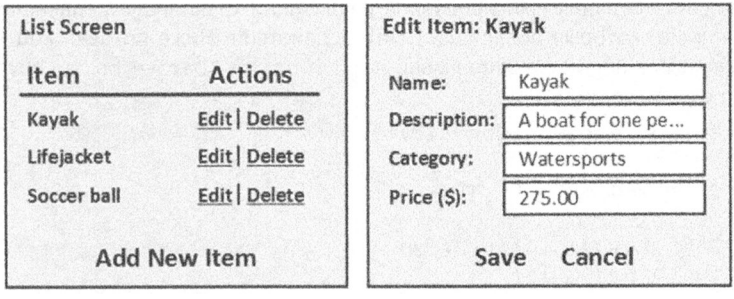


Рис. 11.2. Эскиз пользовательского интерфейса CRUD для каталога товаров

Вместе эти страницы позволяют пользователю создавать, читать, обновлять и удалять (create, read, update, delete — CRUD) элементы в коллекции. Такие действия называются *операциями CRUD*. Разработчики нуждаются в реализации операций CRUD настолько часто, что средство формирования шаблонов в Visual Studio предлагает сценарии для создания контроллеров CRUD с заранее определенными методами действий (включение средства формирования шаблонов рассматривалось в главе 8). Но, как и со всеми шаблонами Visual Studio, я считаю, что изучать возможности ASP.NET Core MVC лучше напрямую.

Создание контроллера CRUD

Начнем с создания отдельного контроллера для управления каталогом товаров. Добавим в папку `Controllers` файл класса по имени `AdminController.cs` с кодом, приведенным в листинге 11.5.

Листинг 11.5. Содержимое файла `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

    public AdminController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult Index() => View(repository.Products);
  }
}
```

В конструкторе контроллера объявлена зависимость от интерфейса `IProductRepository`, которая будет распознаваться при создании экземпляров. В классе контроллера определен единственный метод действия `Index()`, который вызывает метод `View()`, чтобы выбрать стандартное представление для действия, и передает ему в качестве модели представления набор товаров из базы данных.

Модульное тестирование: метод действия `Index()`

Нас интересует поведение метода действия `Index()` в контроллере `Admin`, которое заключается в корректном возвращении объектов `Product` из хранилища. Протестировать его можно за счет создания имитированной реализации хранилища и сравнения тестовых данных с данными, которые возвращает метод действия. Ниже показан код модульного теста, помещенный в новый файл по имени `AdminControllerTests.cs` внутри проекта `SportsStore.Tests`.

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;
```

```
namespace SportsStore.Tests {
  public class AdminControllerTests {
    [Fact]
    public void Index_Contains_All_Products() {
      // Организация - создание имитированного хранилища
      Mock<IProductRepository> mock = new Mock<IProductRepository>();
      mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
      }).AsQueryable<Product>());
      // Организация - создание контроллера
      AdminController target = new AdminController(mock.Object);
      // Действие
      Product[] result
        = GetViewModel<IEnumerable<Product>>(target.Index())?.ToArray();
      // Утверждение
      Assert.Equal(3, result.Length);
      Assert.Equal("P1", result[0].Name);
      Assert.Equal("P2", result[1].Name);
      Assert.Equal("P3", result[2].Name);
    }
    private T GetViewModel<T>(IActionResult result) where T : class {
      return (result as ViewResult)?.ViewData.Model as T;
    }
  }
}
```

В тест был добавлен метод `GetViewModel()` для распаковки результата, возвращаемого методом действия, и получения данных модели представления. Далее в главе будут реализованы дополнительные тесты, которые используют этот метод.

Реализация представления списка

Следующим шагом будет добавление представления для метода действия `Index()` контроллера `Admin`. Создадим папку `Views/Admin` и добавим в нее файл представления Razor по имени `Index.cshtml` с содержимым, приведенным в листинге 11.6.

Листинг 11.6. Содержимое файла `Index.cshtml` из папки `Views/Admin`

```
@model IEnumerable<Product>

@{
  ViewBag.Title = "All Products";
  Layout = "_AdminLayout";
}

<table class="table table-striped table-bordered table-sm">
  <tr>
    <th class="text-right">ID</th>
    <th>Name</th>
    <th class="text-right">Price</th>
    <th class="text-center">Actions</th>
  </tr>
```

```
  @foreach (var item in Model) {
    <tr>
      <td class="text-right">@item.ProductID</td>
      <td>@item.Name</td>
      <td class="text-right">@item.Price.ToString("c")</td>
      <td class="text-center">
        <form asp-action="Delete" method="post">
          <a asp-action="Edit" class="btn btn-sm btn-warning"
             asp-route-productId="@item.ProductID">
            Edit
          </a>
          <input type="hidden" name="ProductID" value="@item.ProductID" />
          <button type="submit" class="btn btn-danger btn-sm">
            Delete
          </button>
        </form>
      </td>
    </tr>
  }
</table>
<div class="text-center">
  <a asp-action="Create" class="btn btn-primary">Add Product</a>
</div>
```

Представление содержит таблицу, в которой для каждого товара предусмотрена строка с ячейками, содержащими наименование и цену товара. Кроме того, в каждой строке присутствуют кнопки, которые позволят редактировать сведения о товаре и удалять его, отправляя запросы к действиям `Edit` и `Delete`. В дополнение к таблице имеется кнопка Add Product (Добавить товар), нацеленная на действие `Create`. Мы добавим действия `Edit`, `Delete` и `Create` в последующих разделах, а пока можно посмотреть, как отображаются товары, запустив приложение и запросив URL вида `/Admin/Index` (рис. 11.3).

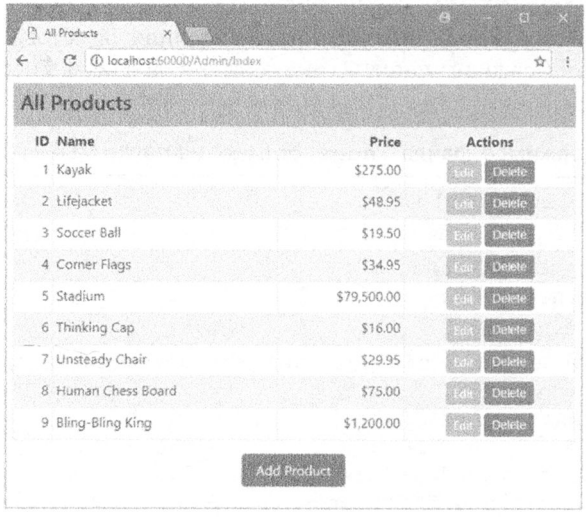


Рис. 11.3. Отображение списка товаров

Совет. В листинге 11.6 кнопка Edit (Редактировать) находится внутри элемента `form`, так что две кнопки располагаются рядом благодаря интервалу, примененному Bootstrap. Кнопка Edit будет посылать серверу HTTP-запрос `GET` для получения текущих сведений о товаре; это не требует элемента `form`. Однако поскольку кнопка Delete (Удалить) будет вносить изменение в состояние приложения, необходимо использовать HTTP-запрос `POST`, который требует элемента `form`.

Редактирование сведений о товарах

Чтобы предоставить средства создания и обновления, мы добавим страницу редактирования сведений о товаре, подобную показанной на рис. 11.2. Задача состоит из двух частей:

- отобразить страницу, которая позволит администратору изменять значения для свойств товара;
- добавить метод действия, который обработает внесенные изменения, когда они будут отправлены.

Создание метода действия `Edit()`

В листинге 11.7 приведен код метода действия `Edit()`, добавленного в контроллер `Admin`, который будет получать HTTP-запрос, отправляемый браузером, когда пользователь щелкает на кнопке Edit.

Листинг 11.7. Добавление метода действия `Edit()` в файле `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

    public AdminController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult Index() => View(repository.Products);

    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));
  }
}
```

Этот простой метод ищет товар с идентификатором, соответствующим значению параметра `productId`, и передает его как объект модели представления методу `View()`.

Модульное тестирование: метод действия Edit()

В методе действия Edit() нам необходимо протестировать две линии поведения. Первая заключается в том, что мы получаем запрашиваемый товар, когда предоставляем допустимое значение идентификатора, чтобы удостовериться в редактировании ожидаемого товара. Вторая проверяемая линия поведения связана с тем, что мы не должны получать товар при запросе значения идентификатора, отсутствующего в хранилище. Ниже показаны тестовые методы, добавленные в файл класса AdminControllerTests.cs.

```
...
[Fact]
public void Can_Edit_Product() {
  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
  }).AsQueryable<Product>());
  // Организация - создание контроллера
  AdminController target = new AdminController(mock.Object);

  // Действие
  Product p1 = GetViewModel<Product>(target.Edit(1));
  Product p2 = GetViewModel<Product>(target.Edit(2));
  Product p3 = GetViewModel<Product>(target.Edit(3));

  // Утверждение
  Assert.Equal(1, p1.ProductID);
  Assert.Equal(2, p2.ProductID);
  Assert.Equal(3, p3.ProductID);
}

[Fact]
public void Cannot_Edit_Nonexistent_Product() {
  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
  }).AsQueryable<Product>());

  // Организация - создание контроллера
  AdminController target = new AdminController(mock.Object);

  // Действие
  Product result = GetViewModel<Product>(target.Edit(4));

  // Утверждение
  Assert.Null(result);
}
...
```

Создание представления редактирования

Теперь, располагая методом действия, можно создать представление для отображения. Добавим в папку `Views/Admin` файл представления Razor по имени `Edit.cshtml` и поместим в него разметку, приведенную в листинге 11.8.

Листинг 11.8. Содержимое файла `Edit.cshtml` из папки `Views/Admin`

```
@model Product

@{
  ViewBag.Title = "Edit Product";
  Layout = "_AdminLayout";
}

<form asp-action="Edit" method="post">
  <input type="hidden" asp-for="ProductID" />
  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Description"></label>
    <textarea asp-for="Description" class="form-control"></textarea>
  </div>
  <div class="form-group">
    <label asp-for="Category"></label>
    <input asp-for="Category" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Price"></label>
    <input asp-for="Price" class="form-control" />
  </div>
  <div class="text-center">
    <button class="btn btn-primary" type="submit">Save</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
  </div>
</form>
```

В представлении имеется форма HTML, большая часть содержимого которой генерируется посредством вспомогательных функций дескрипторов, включая установку целей для элементов `form` и `a`, установку содержимого элементов `label` и выдачу атрибутов `name`, `id` и `value` для элементов `input` и `textarea`.

Чтобы увидеть HTML-разметку, генерируемую представлением, запустим приложение, перейдем на URL типа `/Admin/Index` и щелкнем на кнопке Edit для одного из товаров (рис. 11.4).

Совет. Скрытый элемент `input` для свойства `ProductID` применяется ради простоты. Значение `ProductID` генерируется базой данных как первичный ключ, когда новый объект сохраняется инфраструктурой Entity Framework Core, и его безопасное изменение может оказаться сложным процессом. Для большинства приложений проще всего предотвратить изменение значения со стороны пользователя.

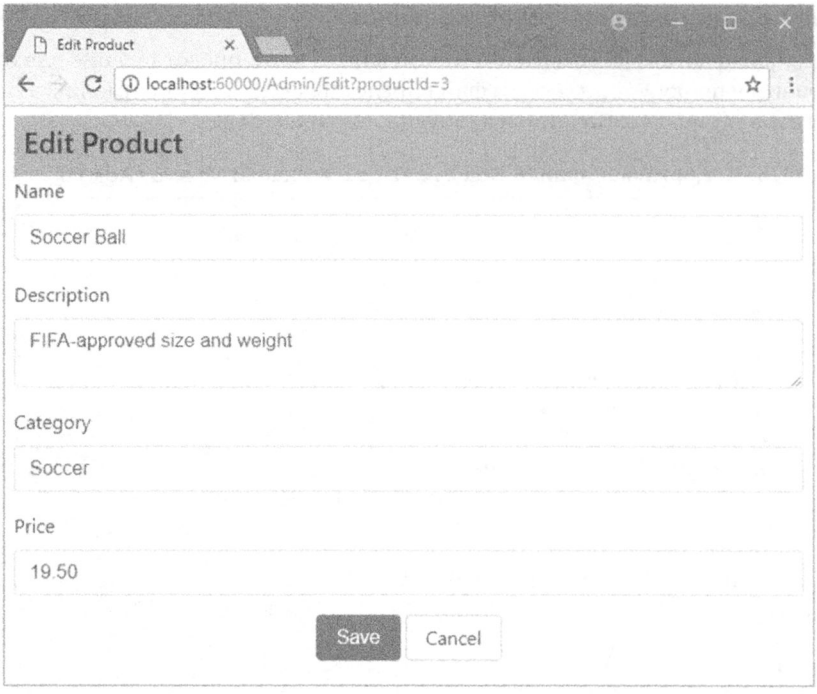


Рис. 11.4. Отображение сведений о товаре для редактирования

Обновление хранилища товаров

Прежде чем можно будет обрабатывать результаты редактирования, хранилище товаров понадобится расширить, добавив возможность сохранения изменений. Первым делом необходимо добавить к интерфейсу `IProductRepository` новый метод (листинг 11.9).

Листинг 11.9. Добавление метода в файл `IProductRespository.cs` из папки `Models`

```
using System.Linq;
namespace SportsStore.Models {
  public interface IProductRepository {
    IQueryable<Product> Products { get; }
    void SaveProduct(Product product);
  }
}
```

Затем можно добавить новый метод к реализации хранилища с помощью Entity Framework Core, которая определена в файле `EFProductRepository.cs` (листинг 11.10).

Листинг 11.10. Реализация нового метода в файле `EFProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

    public EFProductRepository(ApplicationDbContext ctx) {
      context = ctx;
    }

    public IQueryable<Product> Products => context.Products;
    public void SaveProduct(Product product) {
      if (product.ProductID == 0) {
        context.Products.Add(product);
      } else {
        Product dbEntry = context.Products
          .FirstOrDefault(p => p.ProductID == product.ProductID);
        if (dbEntry != null) {
          dbEntry.Name = product.Name;
          dbEntry.Description = product.Description;
          dbEntry.Price = product.Price;
          dbEntry.Category = product.Category;
        }
      }
      context.SaveChanges();
    }
  }
}
```

Реализация метода `SaveChanges()` добавляет товар в хранилище, если значение `ProductID` равно 0; в противном случае применяются изменения к существующей записи в базе данных.

Мы не хотим здесь вдаваться в детали инфраструктуры Entity Framework Core, поскольку, как упоминалось ранее, это отдельная крупная тема, к тому же она не является частью ASP.NET Core MVC. Тем не менее, кое-какой код в методе `SaveProduct()` оказывает влияние на проектное решение, положенное в основу приложения MVC.

Нам известно, что обновление должно выполняться, когда получен параметр `Product`, который имеет ненулевое значение `ProductID`. Задача решается путем извлечения из хранилища объекта `Product` с тем же самым значением `ProductID` и обновления всех его свойств, чтобы они соответствовали значениям свойств объекта, переданного в качестве параметра.

Причина таких действий в том, что инфраструктура Entity Framework Core отслеживает объекты, которые она создает из базы данных. Объект, переданный методу `SaveChanges()`, создается системой привязки моделей MVC, т.е. инфраструктура Entity Framework Core ничего не знает о новом объекте `Product`, и она не будет применять обновление к базе данных, когда объект `Product` модифицирован. Существует множество способов решения указанной проблемы, но мы принимаем самый простой из них, предполагающий поиск соответствующего объекта, о котором известно инфраструктуре Entity Framework Core, и его явное обновление.

Добавление нового метода в интерфейс IProductRepository нарушает работу класса имитированного хранилища FakeProductRepository, который был создан в главе 8. Имитированное хранилище использовалось в целях быстрого старта процесса разработки и демонстрации возможности применения служб для гладкой замены реализаций интерфейса, не изменяя компоненты, которые на них опираются. Имитированное хранилище больше не понадобится. В листинге 11.11 видно, что интерфейс IProductRepository удален из объявления класса, поэтому продолжать модификацию класса по мере добавления функций хранилища не придется.

Листинг 11.11. Удаление интерфейса в файле FakeProductRepository.cs из папки Models

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class FakeProductRepository /* : IProductRepository */ {

    public IQueryable<Product> Products => new List<Product> {
      new Product { Name = "Football", Price = 25 },
      new Product { Name = "Surf board", Price = 179 },
      new Product { Name = "Running shoes", Price = 95 }
    }.AsQueryable<Product>();
  }
}
```

Обработка запросов POST в методе действия Edit()

К настоящему моменту все готово для реализации в контроллере Admin перегруженной версии метода действия Edit(), которая будет обрабатывать запросы POST, инициируемые по щелчку администратором на кнопке Save (Сохранить). Код нового метода приведен в листинге 11.12.

Листинг 11.12. Определение метода действия в файле AdminController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

    public AdminController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult Index() => View(repository.Products);

    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));

    [HttpPost]
```

```
    public IActionResult Edit(Product product) {
      if (ModelState.IsValid) {
        repository.SaveProduct(product);
        TempData["message"] = $"{product.Name} has been saved";
        return RedirectToAction("Index");
      } else {
        // Что-то не так со значениями данных
        return View(product);
      }
    }
  }
}
```

Мы выясняем, смог ли процесс привязки модели проверить достоверность отправленных пользователем данных, для чего читаем значение свойства `ModelState.IsValid`. Если здесь все в порядке, тогда мы сохраняем изменения в хранилище и направляем пользователя на действие `Index`, так что он увидит модифицированный список товаров. При наличии какой-нибудь проблемы с данными мы снова визуализируем стандартное представление, чтобы пользователь получил возможность внести корректировки.

После сохранения изменений в хранилище сообщение сохраняется с использованием *средства* `TempData`, которое является частью средства состояния сеанса ASP.NET Core. Это словарь пар "ключ/значение", похожий на применяемые ранее средства данных сеанса и `ViewBag`. Основное отличие объекта `TempData` от данных сеанса в том, что он хранится до тех пор, пока не будет прочитан.

В такой ситуации использовать `ViewBag` невозможно, потому что объект `ViewBag` передает данные между контроллером и представлением, и он не может удерживать данные дольше, чем длится текущий HTTP-запрос. Когда редактирование успешно, браузер перенаправляется на новый URL, поэтому данные `ViewBag` утрачиваются. Мы могли бы прибегнуть к средству данных сеанса, но тогда сообщение хранилось бы вплоть до его явного удаления, чего делать бы не хотелось.

Таким образом, объект `TempData` подходит как нельзя лучше. Данные ограничиваются сеансом одного пользователя (пользователи не видят объекты `TempData` друг друга) и хранятся достаточно долго, чтобы быть прочитанными. Мы будем читать данные в представлении, которое визуализируется методом действия, куда перенаправлялся пользователь, и определяется в следующем разделе.

Модульное тестирование: метод действия `Edit()`, обрабатывающий запросы `POST`

В методе действия `Edit()`, обрабатывающем запросы `POST`, мы должны удостовериться, что хранилищу товаров для сохранения передаются допустимые обновления объекта `Product`, полученного в качестве аргумента метода. Кроме того, необходимо проверить, что недопустимые обновления (т.е. содержащие ошибки проверки достоверности модели) в хранилище не передаются. Ниже приведены тестовые методы, которые добавлены в файл `AdminControllerTests.cs`.

```
...
[Fact]
public void Can_Save_Valid_Changes() {
  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
```

```
    // Организация - создание имитированных временных данных
    Mock<ITempDataDictionary> tempData = new Mock<ITempDataDictionary>();
    // Организация - создание контроллера
    AdminController target = new AdminController(mock.Object) {
      TempData = tempData.Object
    };
    // Организация - создание товара
    Product product = new Product { Name = "Test" };

    // Действие - попытка сохранить товар
    IActionResult result = target.Edit(product);

    // Утверждение - проверка того, что к хранилищу было произведено обращение
    mock.Verify(m => m.SaveProduct(product));
    // Утверждение - проверка, что типом результата является перенаправление
    Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Index", (result as RedirectToActionResult).ActionName);
  }

  [Fact]
  public void Cannot_Save_Invalid_Changes() {
    // Организация - создание имитированного хранилища
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    // Организация - создание контроллера
    AdminController target = new AdminController(mock.Object);
    // Организация - создание товара
    Product product = new Product { Name = "Test" };
    // Организация - добавление ошибки в состояние модели
    target.ModelState.AddModelError("error", "error");

    // Действие - попытка сохранить товар
    IActionResult result = target.Edit(product);

    // Утверждение - проверка того, что к хранилищу было произведено обращение
    mock.Verify(m => m.SaveProduct(It.IsAny<Product>()), Times.Never());
    // Утверждение - проверка типа результата метода
    Assert.IsType<ViewResult>(result);
  }
  ...
```

Отображение подтверждающего сообщения

Мы будем иметь дело с сообщением, сохраненным с помощью `TempData`, в файле компоновки `_AdminLayout.cshtml` (листинг 11.13). За счет обработки сообщения в компоновке мы можем создавать сообщения в любом представлении, которое применяет данную компоновку, без необходимости в создании дополнительных выражений Razor.

Листинг 11.13. Обработка сообщения `TempData` в файле `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>
```

Совет. Преимущество работы с сообщением внутри компоновки заключается в том, что пользователи будут видеть его на любой странице, визуализированной после сохранения изменений. В данный момент мы возвращаем его списку товаров, но рабочий поток можно изменить с целью визуализации какого-то другого представления, и пользователи будут по-прежнему видеть сообщение (при условии, что следующее представление использует ту же самую компоновку).

Теперь мы располагаем всеми фрагментами для редактирования сведений о товарах. Чтобы увидеть, как все они работают, запустим приложение, перейдем на URL вида `/Admin/Index`, щелкнем на кнопке Edit и внесем изменение. Затем щелкнем на кнопке Save. Произойдет перенаправление на `/Admin/Index` и отобразится сообщение из `TempData` (рис. 11.5). Если перезагрузить страницу со списком товаров, то сообщение исчезнет, поскольку после чтения объект `TempData` удаляется. Подход очень удобен, т.к. не приходится иметь дело со старыми сообщениями.

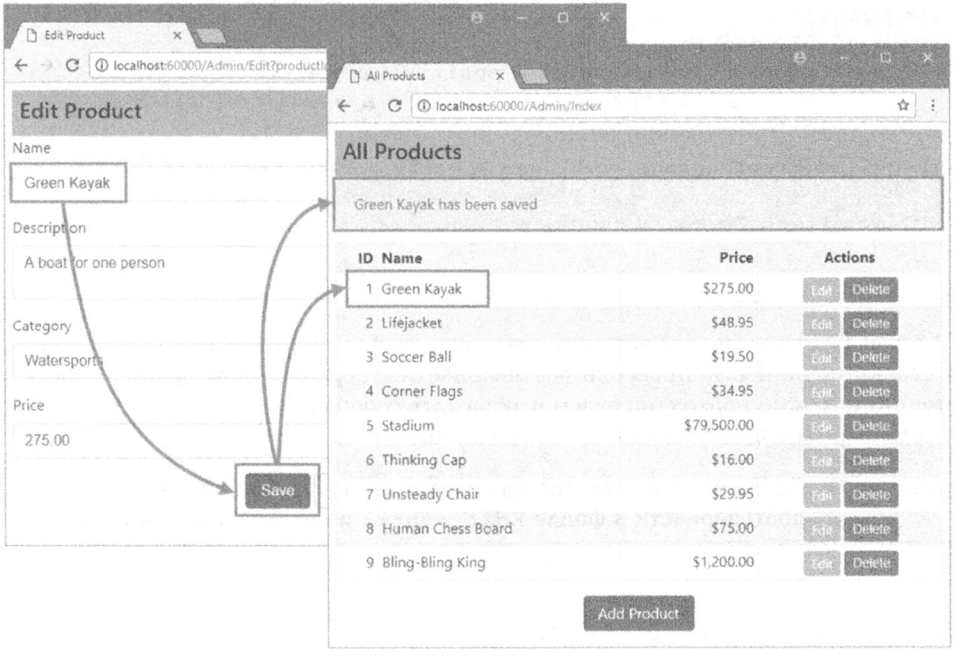


Рис. 11.5. Редактирование сведений о товаре и отображение сообщения из `TempData`

Добавление проверки достоверности модели

Мы добрались до точки, когда к классам модели необходимо добавить правила проверки достоверности. Пока что администратор может вводить отрицательные значения для цен или оставлять описания пустыми — и приложение SportsStore благополучно сохранит эти данные в базе. Смогут ли недопустимые данные успешно сохраниться, зависит от того, удовлетворяют ли они ограничениям в таблицах SQL, созданных инфраструктурой Entity Framework Core, и для большинства приложений таких мер безопасности будет недостаточно. Чтобы защититься от недопустимых значений данных, свойства класса `Product` декорируются с помощью атрибутов, как делалось в классе `Order` из главы 10 (листинг 11.14).

Листинг 11.14. Применение атрибутов проверки достоверности в файле `Product.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

  public class Product {
    public int ProductID { get; set; }

    [Required(ErrorMessage = "Please enter a product name")]
                              // Введите наименование товара
    public string Name { get; set; }

    [Required(ErrorMessage = "Please enter a description")]
                             // Введите описание
    public string Description { get; set; }

    [Required]
    [Range(0.01, double.MaxValue,
      ErrorMessage = "Please enter a positive price")]
                    // Введите положительное значение для цены
    public decimal Price { get; set; }

    [Required(ErrorMessage = "Please specify a category")]
                              // Укажите категорию
    public string Category { get; set; }
  }
}
```

В главе 10 использовалась вспомогательная функция дескриптора для отображения сводки по ошибкам проверки достоверности в верхней части формы. Здесь мы применим похожий подход, но будем отображать сообщения об ошибках рядом с элементами формы в представлении `Edit` (листинг 11.15).

Листинг 11.15. Добавление элементов для отображения ошибок проверки достоверности в файле `Edit.cshtml` из папки `Views/Admin`

```
@model Product

@{
  ViewBag.Title = "Edit Product";
  Layout = "_AdminLayout";
}
```

```
<form asp-action="Edit" method="post">
  <input type="hidden" asp-for="ProductID" />
  <div class="form-group">
    <label asp-for="Name"></label>
    <div><span asp-validation-for="Name" class="text-danger"></span></div>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Description"></label>
    <div><span asp-validation-for="Description" class="text-danger"></span>
    </div>
    <textarea asp-for="Description" class="form-control"></textarea>
  </div>
  <div class="form-group">
    <label asp-for="Category"></label>
    <div><span asp-validation-for="Category" class="text-danger"></span>
    </div>
    <input asp-for="Category" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Price"></label>
    <div><span asp-validation-for="Price" class="text-danger"></span>
    </div>
    <input asp-for="Price" class="form-control" />
  </div>
  <div class="text-center">
    <button class="btn btn-primary" type="submit">Save</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
  </div>
</form>
```

Когда атрибут `asp-validation-for` применяется к элементу `span`, он использует вспомогательную функцию дескриптора, которая добавляет сообщение об ошибке проверки достоверности для указанного свойства, если при проверке возникли какие-то проблемы.

Вспомогательные функции дескрипторов будут вставлять сообщение об ошибке в элемент `span` и добавлять элемент в класс `input-validation-error`, который позволит легко применять стили CSS к элементам с сообщениями об ошибках (листинг 11.16).

Листинг 11.16. Добавление стиля CSS в файле `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
  <style>
    .input-validation-error { border-color: red; background-color: #fee ; }
  </style>
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success mt-1">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>
```

Определенный здесь стиль CSS выбирает элементы, которые являются членами класса `input-validation-error`, и устанавливает для них границу красного цвета и фон.

Совет. Явная установка стилей, когда используется библиотека CSS наподобие Bootstrap, может привести к несоответствиям в случае применения тем содержимого. В главе 27 будет продемонстрирован альтернативный подход, при котором для применения классов Bootstrap к элементам с сообщениями об ошибках проверки достоверности используется код JavaScript, сохраняя все в согласованном состоянии.

Вспомогательные функции дескрипторов для сообщений проверки достоверности можно применять где угодно в представлении, но по соглашению (что вполне разумно) принято размещать их поближе к проблемному элементу, чтобы ввести пользователя в курс дела. На рис. 11.6 показано, как выглядят сообщения об ошибках проверки достоверности и отображаемые подсказки, для чего нужно запустить приложение, отредактировать сведения о товаре и отправить недопустимые данные.

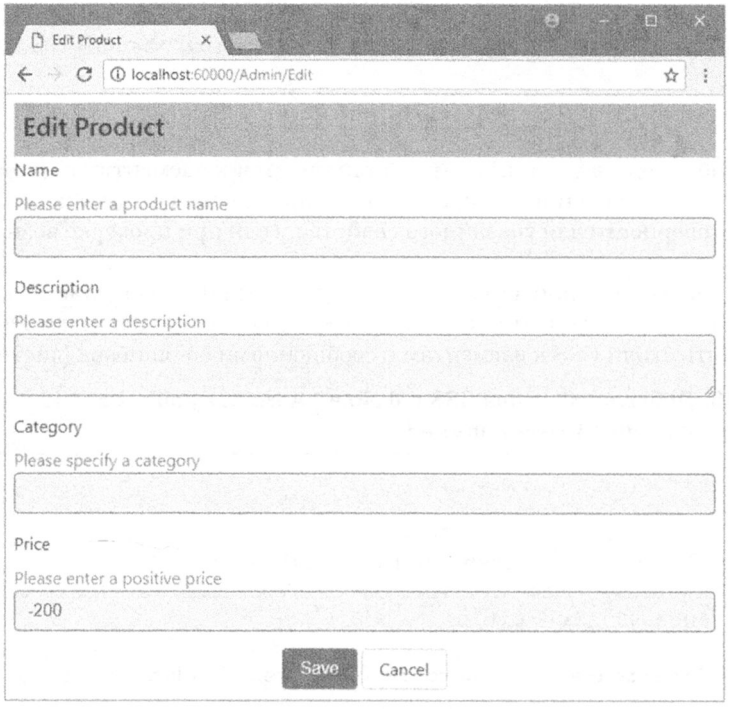


Рис. 11.6. Проверка достоверности данных при редактировании сведений о товаре

Включение проверки достоверности на стороне клиента

В текущий момент проверка достоверности данных применяется, только когда пользователь-администратор отправляет результаты редактирования серверу, но большинство пользователей ожидают немедленного отклика при наличии проблем с введенными данными. Именно потому разработчики часто предпочитают выполнять *проверку достоверности на стороне клиента*, при которой данные проверяются в браузере с использованием JavaScript. Приложения MVC могут выполнять проверку достоверности на стороне клиента на основе аннотаций данных, применяемых к классу модели предметной области.

Прежде всего, понадобится добавить библиотеки JavaScript, которые предоставят приложению средство проверки достоверности на стороне клиента, что делается в файле `bower.json` (листинг 11.17).

Листинг 11.17. Добавление пакетов JavaScript в файле `bower.json` из папки `SportsStore`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "fontawesome": "4.7.0",
    "jquery": "3.2.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Проверка достоверности на стороне клиента построена на основе популярной библиотеки jQuery, которая упрощает работу с API-интерфейсом DOM браузера. Следующий шаг связан с добавлением файлов JavaScript в компоновку, чтобы они загружались, когда используются средства администрирования приложения SportsStore (листинг 11.18).

Листинг 11.18. Добавление библиотек проверки достоверности в файле `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
  <style>
    .input-validation-error { border-color: red; background-color: #fee ; }
  </style>
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script src="/lib/jquery-validation/dist/jquery.validate.min.js">
  </script>
  <script
src="/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
  </script>
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success mt-1">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>
```

Включение проверки достоверности на стороне клиента не приводит к каким-то визуальным изменениям. Но ограничения, которые указаны с помощью атрибутов, примененных к классу модели C#, вступают в силу на уровне браузера, предотвращая отправку пользователем формы с недопустимыми данными и обеспечивая немедленный отклик при наличии проблемы. За дополнительными сведениями обращайтесь в главу 27.

Создание новых товаров

Далее мы реализуем метод действия `Create()`, который указан для кнопки Add Product на главной странице со списком товаров. Он позволит администратору добавлять новые элементы в каталог товаров. Добавление возможности создания новых товаров требует одного небольшого дополнения в приложении, что является великолепной демонстрацией мощи и гибкости хорошо структурированного приложения MVC. Для начала добавим в контроллер `Admin` метод `Create()`, как показано в листинге 11.19.

Листинг 11.19. Добавление метода действия `Create()` в файле `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {
  public class AdminController : Controller {
    private IProductRepository repository;
    public AdminController(IProductRepository repo) {
      repository = repo;
    }
    public ViewResult Index() => View(repository.Products);
    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));

    [HttpPost]
    public IActionResult Edit(Product product) {
      if (ModelState.IsValid) {
        repository.SaveProduct(product);
        TempData["message"] = $"{product.Name} has been saved";
        return RedirectToAction("Index");
      } else {
        // Что-то не так со значениями данных
        return View(product);
      }
    }
```

```
    public ViewResult Create() => View("Edit", new Product());
  }
}
```

Метод `Create()` не визуализирует свое стандартное представление. Взамен он указывает, что должно использоваться представление `Edit`. Применение в методе действия представления, которое обычно связано с другим методом действия, вполне допустимо. В рассматриваемом случае мы указываем в качестве модели представления новый объект `Product`, так что представление `Edit` заполняется пустыми полями.

На заметку! Модульный тест для метода действия `Create()` не добавляется. Он позволил бы проверить только способность обработки инфраструктурой ASP.NET Core MVC результата, возвращаемого методом действия — то, что мы считаем само собой разумеющимся. (Обычно тесты для функциональных средств инфраструктуры не пишутся, если только нет подозрения о наличии дефекта.)

Это единственное изменение, которое потребовалось внести, поскольку метод действия `Edit()` уже настроен на получение объектов `Product` от системы привязки моделей и на их сохранение в базе данных. Чтобы протестировать имеющуюся функциональность, запустим приложение, перейдем на URL вида `/Admin/Index`, щелкнем на кнопке Add Product, заполним форму и отправим ее. Информация, введенная на форме, будет использоваться для создания нового товара в базе данных, который затем появится в списке (рис. 11.7).

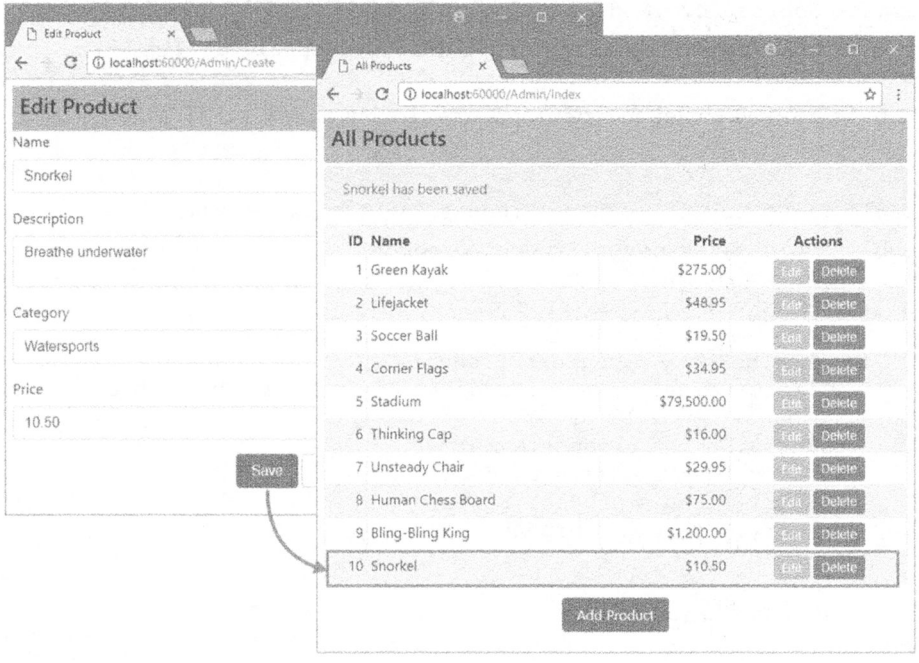


Рис. 11.7. Добавление нового товара в каталог

Удаление товаров

Обеспечить поддержку удаления элементов из каталога довольно просто. Для начала добавим в интерфейс `IProductRepository` новый метод, как показано в листинге 11.20.

Листинг 11.20. Добавление метода для удаления товаров в файле `IProductRepository.cs` из папки `Models`

```
using System.Linq;

namespace SportsStore.Models {
  public interface IProductRepository {
    IQueryable<Product> Products { get; }
    void SaveProduct(Product product);
    Product DeleteProduct(int productID);
  }
}
```

Затем реализуем метод `DeleteProduct()` в классе хранилища Entity Framework Core, т.е. `EFProductRepository` (листинг 11.21).

Листинг 11.21. Реализация поддержки удаления в файле `EFProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {
  public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

    public EFProductRepository(ApplicationDbContext ctx) {
      context = ctx;
    }

    public IQueryable<Product> Products => context.Products;

    public void SaveProduct(Product product) {
      if (product.ProductID == 0) {
        context.Products.Add(product);
      } else {
        Product dbEntry = context.Products
          .FirstOrDefault(p => p.ProductID == product.ProductID);
        if (dbEntry != null) {
          dbEntry.Name = product.Name;
          dbEntry.Description = product.Description;
          dbEntry.Price = product.Price;
          dbEntry.Category = product.Category;
        }
      }
      context.SaveChanges();
    }
```

```
    public Product DeleteProduct(int productID) {
      Product dbEntry = context.Products
        .FirstOrDefault(p => p.ProductID == productID);
      if (dbEntry != null) {
        context.Products.Remove(dbEntry);
        context.SaveChanges();
      }
      return dbEntry;
    }
  }
}
```

Финальный шаг связан с реализацией метода действия `Delete()` в контроллере `Admin`. Метод действия `Delete()` должен поддерживать только запросы `POST`, потому что удаление объектов не является идемпотентной операцией. Как будет показано в главе 16, браузеры и кеши вольны выдавать запросы `GET` без явного согласия пользователя, поэтому мы должны проявить осторожность, чтобы избежать внесения изменений как следствия запросов `GET`. Код нового метода действия приведен в листинге 11.22.

Листинг 11.22. Добавление метода действия `Delete()` в файле `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

    public AdminController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult Index() => View(repository.Products);

    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));

    [HttpPost]
    public IActionResult Edit(Product product) {

      if (ModelState.IsValid) {
        repository.SaveProduct(product);
        TempData["message"] = $"{product.Name} has been saved";
        return RedirectToAction("Index");
      } else {
        // Что-то не так со значениями данных
        return View(product);
      }
    }
```

```
    public IActionResult Create() => View("Edit", new Product());

    [HttpPost]
    public IActionResult Delete(int productId) {
      Product deletedProduct = repository.DeleteProduct(productId);
      if (deletedProduct != null) {
        TempData["message"] = $"{deletedProduct.Name} was deleted";
      }
      return RedirectToAction("Index");
    }
  }
}
```

Модульное тестирование: удаление товаров

Нам нужно протестировать основное поведение метода действия `Delete()`, которое заключается в том, что при передаче в качестве параметра допустимого идентификатора `ProductID` метод действия должен вызвать метод `DeleteProduct()` хранилища и передать ему корректное значение `ProductID` удаляемого товара. Вот тест, добавленный в файл `AdminControllerTests.cs`:

```
...
[Fact]
public void Can_Delete_Valid_Products() {
  // Организация - создание объекта Product
  Product prod = new Product { ProductID = 2, Name = "Test" };

  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    prod,
    new Product {ProductID = 3, Name = "P3"},
  }).AsQueryable<Product>());

  // Организация - создание контроллера
  AdminController target = new AdminController(mock.Object);

  // Действие - удаление товара
  target.Delete(prod.ProductID);

  // Утверждение - проверка того, что был вызван метод удаления
  // в хранилище с корректным объектом Product
  mock.Verify(m => m.DeleteProduct(prod.ProductID));
}
...
```

Чтобы увидеть средство удаления в работе, запустим приложение, перейдем на URL вида `/Admin/Index` и щелкнем на одной из кнопок Delete (Удалить) внутри страницы со списком товаров (рис. 11.8). На рисунке можно заметить, что с помощью переменной `TempData` отображается сообщение об удалении товара из каталога.

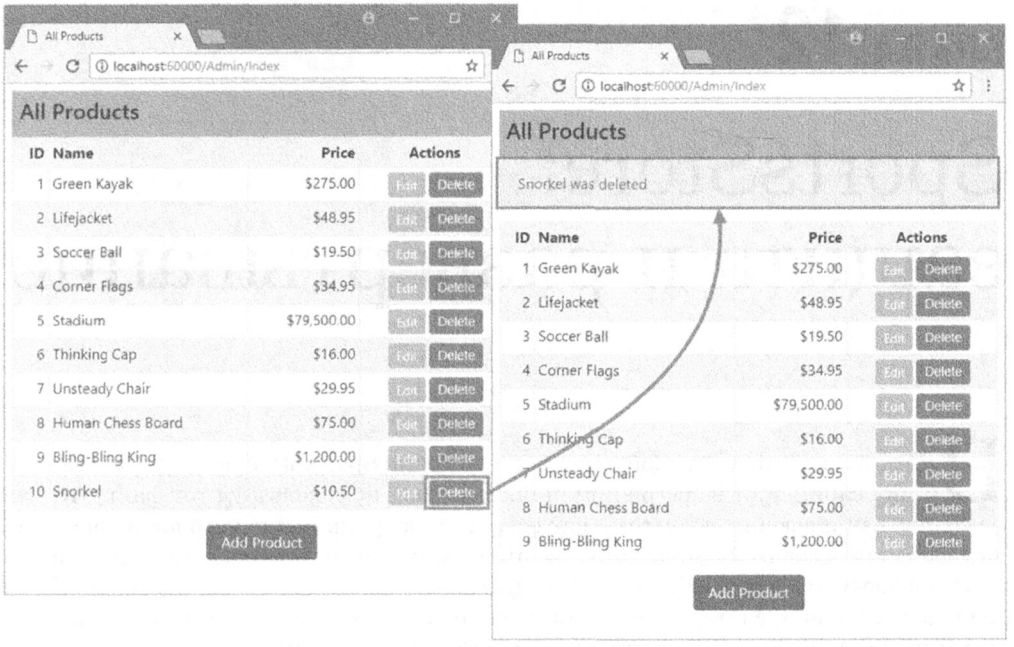


Рис. 11.8. Удаление товара из каталога

На заметку! При попытке удалить товар, для которого ранее был создан заказ, возникнет ошибка. Когда объект `Order` сохраняется в базе данных, он превращается в запись внутри таблицы базы данных, которая содержит ссылку на связанный с ним объект `Product`, что известно как отношение внешнего ключа. Это означает, что по умолчанию база данных не позволит удалить объект `Product`, если для него был создан объект `Order`, поскольку такое действие внесло бы несогласованность в базу данных. Существует несколько способов решения указанной проблемы, включая удаление объектов `Order` при удалении объекта `Product`, к которому они относятся, либо изменение отношения между объектами `Product` и `Order`. Подробные сведения можно найти в документации Entity Framework Core.

Резюме

В главе были введены средства администрирования и показано, как реализовать операции CRUD, которые позволяют администратору создавать, читать, обновлять и удалять товары из хранилища и помечать заказы как отгруженные. В следующей главе мы продемонстрируем способ защиты административных функций, чтобы они не были доступными абсолютно всем пользователям, и развернем приложение SportsStore в производственной среде.

ГЛАВА 12

SportsStore: защита и развертывание

В предыдущей главе мы добавили в приложение SportsStore поддержку администрирования, и от вашего внимания, вероятно, не ускользнул тот факт, что если развернуть приложение в том виде как есть, то модифицировать каталог товаров сможет любой пользователь. Для этого ему лишь нужно знать, что средства администрирования доступны через URL вида `/Admin/Index` и `/Order/List`. В настоящей главе мы покажем, как предотвратить использование административных функций случайными посетителями, защитив их паролем. После добавления возможности защиты мы объясним, каким образом подготовить и развернуть приложение SportsStore в производственной среде.

Защита средств администрирования

Аутентификация и авторизация предоставляются системой ASP.NET Core Identity, которая аккуратно интегрируется как в платформу ASP.NET Core, так и в приложения MVC. В последующих разделах мы создадим базовую настройку защиты, которая позволит одному пользователю по имени `Admin` проходить аутентификацию и получать доступ к административным функциям в приложении. Система ASP.NET Core Identity предлагает множество других средств для аутентификации пользователей, а также авторизации доступа к функциям и данным приложения. Более подробные сведения вы найдете в главах 28–30, где будет показано, как создавать и управлять пользовательскими учетными записями, каким образом применять роли и политики и как поддерживать аутентификацию от третьих сторон вроде Microsoft, Google, Facebook и Twitter. Однако цель текущей главы — создать лишь столько функциональности, сколько достаточно для предотвращения доступа пользователей к чувствительным частям приложения SportsStore, что будет содействовать пониманию того, каким образом аутентификация и авторизация вписываются в приложение MVC.

Создание базы данных `Identity`

Система ASP.NET Core Identity чрезвычайно конфигурируема и расширяема, поддерживая многочисленные варианты хранения данных о пользователях. Мы собираемся использовать наиболее распространенный вариант, который предусматривает хранение данных с применением Microsoft SQL Server и доступ к ним с помощью Entity Framework Core.

Создание класса контекста

Нам необходимо создать файл контекста базы данных, который будет действовать в качестве шлюза между базой данных и объектами моделей базы данных Identity, предоставляющими к ней доступ. Добавим в папку Models файл класса по имени AppIdentityDbContext.cs с определением, приведенным в листинге 12.1.

На заметку! Вы могли привыкнуть к тому, что для получения дополнительных функциональных средств вроде защиты к проекту необходимо добавлять пакеты. Но с выпуском ASP.NET Core 2 пакеты NuGet, требующиеся для системы Identity, уже включены в проект через метапакет, который был добавлен в файл SportsStore.csproj как часть шаблона проекта.

Листинг 12.1. Содержимое файла AppIdentityDbContext.cs из папки Models

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
namespace SportsStore.Models {
  public class AppIdentityDbContext : IdentityDbContext<IdentityUser> {
   public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext> options)
     : base(options) { }
  }
}
```

Класс AppIdentityDbContext является производным от класса IdentityDbContext, который предлагает связанные с Identity средства для Entity Framework Core. В параметре типа используется IdentityUser — встроенный класс, применяемый для представления пользователей. В главе 28 будет продемонстрировано использование специального класса, который можно расширять с целью добавления дополнительной информации о пользователях приложения.

Определение строки подключения

Далее определяется строка подключения, предназначенная для базы данных. В листинге 12.2 показано добавление в файл appsettings.json проекта SportsStore, которое следует тому же самому формату, что и строка подключения, определенная для базы данных товаров в главе 8.

Листинг 12.2. Определение строки подключения в файле appsettings.json из папки SportsStore

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString":
      "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;
              Trusted_Connection=True;MultipleActiveResultSets=true"
    },
    "SportStoreIdentity": {
     "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=Identity;
                 Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}
```

Вспомните, что строка подключения должна определяться в файле appsettings.json как единственная неразрывная строка кода, а в листинге она выглядит так из-за ограниченной ширины печатной страницы. Добавленная разметка определяет строку подключения по имени SportsStoreIdentity, в которой указывается база данных LocalDB под названием Identity.

Конфигурирование приложения

Подобно другим средствам ASP.NET Core система Identity конфигурируется в классе Start. В листинге 12.3 приведены добавления для настройки Identity в проекте SportsStore с применением ранее определенного класса контекста и строки подключения.

Листинг 12.3. Конфигурирование средства Identity в файле Startup.cs из папки SportsStore

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

namespace SportsStore {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreProducts:ConnectionString"]));

      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));
      services.AddIdentity<IdentityUser, IdentityRole>()
        .AddEntityFrameworkStores<AppIdentityDbContext>()
        .AddDefaultTokenProviders();

      services.AddTransient<IProductRepository, EFProductRepository>();
      services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
      services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
      services.AddTransient<IOrderRepository, EFOrderRepository>();
      services.AddMvc();
      services.AddMemoryCache();
      services.AddSession();
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseSession();
      app.UseAuthentication();
      app.UseMvc(routes => {
        // ...для краткости маршруты не показаны...
      });
      SeedData.EnsurePopulated(app);
    }
  }
}
```

В методе `ConfigureServices()` конфигурация Entity Framework Core расширена для регистрации класса контекста и с помощью метода `AddIdentity()` устанавливает службы Identity, используя встроенные классы для представления пользователей и ролей. Внутри метода `Configure()` вызывается метод `UseAuthentication()` для установки компонентов, которые будут перехватывать запросы и ответы для внедрения политики безопасности.

Создание и применение миграции базы данных

Основная конфигурация на месте, и самое время воспользоваться средством миграций Entity Framework Core для определения схемы и ее применения к базе данных. Откроем окно командной строки или PowerShell и введем показанную ниже команду в папке проекта `SportsStore`, чтобы создать новую миграцию для базы данных `Identity`:

```
dotnet ef migrations add Initial --context AppIdentityDbContext
```

Существенное отличие от предшествующих команд базы данных связано с тем, что здесь используется аргумент `--context` для указания имени класса контекста, ассоциированного с базой данных, с которой необходимо работать — `AppIdentityDbContext`. При наличии множества баз данных в приложении важно гарантировать работу с правильным классом контекста.

После того как инфраструктура Entity Framework Core сгенерировала начальную миграцию, выполним следующую команду, чтобы создать базу данных и запустить команды миграции:

```
dotnet ef database update --context AppIdentityDbContext
```

Результатом будет новая база данных LocalDB по имени `Identity`, которую можно просмотреть с применением проводника объектов SQL Server (SQL Server Object Explorer) среды Visual Studio.

Определение начальных данных

Мы планируем явно создать пользователя `Admin`, наполняя базу данных начальными данными при запуске приложения. Добавим в папку `Models` файл класса по имени `IdentitySeedData.cs` и определим в нем статический класс (листинг 12.4).

Листинг 12.4. Содержимое файла `IdentitySeedData.cs` из папки `Models`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {

  public static class IdentitySeedData {
    private const string adminUser = "Admin";
    private const string adminPassword = "Secret123$";

    public static async void EnsurePopulated(IApplicationBuilder app) {

      UserManager<IdentityUser> userManager = app.ApplicationServices
        .GetRequiredService<UserManager<IdentityUser>>();

      IdentityUser user = await userManager.FindByIdAsync(adminUser);
      if (user == null) {
        user = new IdentityUser("Admin");
        await userManager.CreateAsync(user, adminPassword);
      }
    }
  }
}
```

В коде используется класс `UserManager<T>`, который система ASP.NET Core Identity предоставляет в виде службы для управления пользователями, как обсуждается в главе 28. В базе данных производится поиск учетной записи пользователя `Admin`, которая в случае ее отсутствия создается (с паролем `Secret123$`). Не изменяйте жестко закодированный пароль в этом примере, поскольку система Identity имеет политику проверки достоверности, которая требует, чтобы пароли содержали цифры и диапазон символов. Способ изменения настроек, относящихся к проверке достоверности, описан в главе 28.

Внимание! Жесткое кодирование деталей учетной записи администратора часто требуется для того, чтобы можно было войти в приложение после его развертывания и начать администрирование. Поступая так, вы должны помнить о необходимости изменения пароля для учетной записи, которую создали. В главе 28 приведены детали того, как изменять пароли, используя Identity.

Чтобы обеспечить начальное заполнение базы данных `Identity` во время запуска приложения, добавим в метод `Configure()` класса `Startup` оператор, как показано в листинге 12.5.

Листинг 12.5. Начальное заполнение базы данных `Identity` в файле `Startup.cs` из папки `SportsStore`

```
...
public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
  app.UseDeveloperExceptionPage();
  app.UseStatusCodePages();
  app.UseStaticFiles();
  app.UseSession();
```

```
    app.UseAuthentication();
    app.UseMvc(routes => {
      // ...для краткости маршруты не показаны...
    });
    SeedData.EnsurePopulated(app);
    IdentitySeedData.EnsurePopulated(app);
}
...
```

Применение базовой политики авторизации

Теперь, когда система ASP.NET Core Identity установлена и сконфигурирована, можно применить политику авторизации к тем частям приложения, которые необходимо защитить. Мы собираемся использовать самую базовую политику авторизации, которая предусматривает разрешение доступа любому пользователю, прошедшему аутентификацию. Хотя она может оказаться полезной политикой также и в реальном приложении, существуют возможности для создания более детализированных элементов управления авторизацией (как описано в главах 28–30), но из-за того, что в приложении SportsStore имеется только один пользователь, вполне достаточно провести различие между анонимными и аутентифицированными запросами.

Атрибут `Authorize` применяется для ограничения доступа к методам действий, и в листинге 12.6 видно, что этот атрибут используется для защиты доступа к административным действиям в контроллере `Order`.

Листинг 12.6. Ограничение доступа в файле `OrderController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;

namespace SportsStore.Controllers {

  public class OrderController : Controller {
    private IOrderRepository repository;
    private Cart cart;

    public OrderController(IOrderRepository repoService,
                           Cart cartService) {
      repository = repoService;
      cart = cartService;
    }

    [Authorize]
    public ViewResult List() =>
      View(repository.Orders.Where(o => !o.Shipped));

    [HttpPost]

    [Authorize]
    public IActionResult MarkShipped(int orderID) {
      Order order = repository.Orders
        .FirstOrDefault(o => o.OrderID == orderID);
```

```
      if (order != null) {
        order.Shipped = true;
        repository.SaveOrder(order);
      }
      return RedirectToAction(nameof(List));
    }

    public ViewResult Checkout() => View(new Order());

    [HttpPost]
    public IActionResult Checkout(Order order) {
      if (cart.Lines.Count() == 0) {
        ModelState.AddModelError("", "Sorry, your cart is empty!");
      }
      if (ModelState.IsValid) {
        order.Lines = cart.Lines.ToArray();
        repository.SaveOrder(order);
        return RedirectToAction(nameof(Completed));
      } else {
        return View(order);
      }
    }

    public ViewResult Completed() {
      cart.Clear();
      return View();
    }
  }
}
```

Мы не хотим препятствовать доступу пользователей, не прошедших аутентификацию, к остальным методам действий в контроллере `Order`, так что атрибут `Authorize` применен только к методам `List()` и `MarkShipped()`. Нам нужно защитить все методы действий, определяемые контроллером `Admin`, чего можно достичь за счет применения атрибута `Authorize` к самому классу контроллера, что приведет к применению политики авторизации ко всем содержащимся в нем методам действий (листинг 12.7).

Листинг 12.7. Ограничение доступа в файле `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;

namespace SportsStore.Controllers {

  [Authorize]
  public class AdminController : Controller {
    private IProductRepository repository;

    public AdminController(IProductRepository repo) {
      repository = repo;
    }
```

```
    public ViewResult Index() => View(repository.Products);

    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));

    [HttpPost]
    public IActionResult Edit(Product product) {
      if (ModelState.IsValid) {
        repository.SaveProduct(product);
        TempData["message"] = $"{product.Name} has been saved";
        return RedirectToAction("Index");
      } else {
        // Что-то не так со значениями данных
        return View(product);
      }
    }

    public ViewResult Create() => View("Edit", new Product());

    [HttpPost]
    public IActionResult Delete(int productId) {
      Product deletedProduct = repository.DeleteProduct(productId);
      if (deletedProduct != null) {
        TempData["message"] = $"{deletedProduct.Name} was deleted";
      }
      return RedirectToAction("Index");
    }
  }
}
```

Создание контроллера `Account` и представлений

Когда пользователь, не прошедший аутентификацию, посылает запрос, который требует авторизации, он перенаправляется на URL вида `/Account/Login`, где приложение может предложить пользователю ввести свои учетные данные. В качестве подготовки создадим модель представления для учетных данных пользователя, добавив в папку `Models/ViewModels` файл класса по имени `LoginModel.cs` с определением, показанным в листинге 12.8.

Листинг 12.8. Содержимое файла `LoginModel.cs` из папки `Models/ViewModels`

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Models.ViewModels {

  public class LoginModel {

    [Required]
    public string Name { get; set; }

    [Required]
    [UIHint("password")]
    public string Password { get; set; }

    public string ReturnUrl { get; set; } = "/";
  }
}
```

Свойства Name и Password декорированы атрибутом Required, который использует проверку достоверности модели для обеспечения того, что значения были предоставлены. Свойство Password декорировано атрибутом UIHint, поэтому в случае применения атрибута asp-for внутри элемента input представления Razor, предназначенного для входа, вспомогательная функция дескриптора установит атрибут type в password; таким образом, вводимый пользователем текст не будет виден на экране. Использование атрибута UIHint описано в главе 24.

Далее добавим в папку Controllers файл класса по имени AccountController.cs с определением контроллера, приведенным в листинге 12.9. Данный контроллер будет отвечать на запросы к URL вида /Account/Login.

Листинг 12.9. Содержимое файла AccountController.cs из папки Controllers

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<IdentityUser> userManager;
    private SignInManager<IdentityUser> signInManager;

    public AccountController(UserManager<IdentityUser> userMgr,
        SignInManager<IdentityUser> signInMgr) {
      userManager = userMgr;
      signInManager = signInMgr;
    }

    [AllowAnonymous]
    public ViewResult Login(string returnUrl) {
      return View(new LoginModel {
        ReturnUrl = returnUrl
      });
    }

    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginModel loginModel) {
      if (ModelState.IsValid) {
        IdentityUser user =
          await userManager.FindByNameAsync(loginModel.Name);
        if (user != null) {
          await signInManager.SignOutAsync();
          if ((await signInManager.PasswordSignInAsync(user,
              loginModel.Password, false, false)).Succeeded) {
            return Redirect(loginModel?.ReturnUrl ?? "/Admin/Index");
          }
        }
      }
```

```
      ModelState.AddModelError("", "Invalid name or password");
                                  // Неправильное имя или пароль
      return View(loginModel);
    }
    public async Task<RedirectResult> Logout(string returnUrl = "/") {
      await signInManager.SignOutAsync();
      return Redirect(returnUrl);
    }
  }
}
```

Когда пользователь перенаправляется на URL вида `/Account/Login`, версия `GET` метода действия `Login()` визуализирует стандартное представление для страницы и создает объект модели представления, включающий URL, на который браузер должен быть перенаправлен, если запрос на аутентификацию завершился успешно.

Учетные данные аутентификации отправляются версии `POST` метода действия `Login()`, которая применяет службы `UserManager<IdentityUser>` и `SignInManager<IdentityUser>`, полученные через конструктор класса контроллера, для аутентификации пользователя и его входа в систему. Работа упомянутых классов объясняется в главах 28–30, а пока достаточно знать, что в случае отказа в аутентификации создается ошибка проверки достоверности модели и визуализируется стандартное представление. Если же аутентификация прошла успешно, тогда пользователь перенаправляется на URL, к которому он хотел получить доступ перед тем, как ему было предложено ввести свои учетные данные.

Внимание! В целом использование проверки достоверности данных на стороне клиента является хорошей практикой. Она освобождает от определенной работы сервер и обеспечивает пользователям немедленный отклик о предоставленных ими данных. Тем не менее, не поддавайтесь искушению выполнять на стороне клиента аутентификацию, поскольку обычно это предусматривает передачу клиенту допустимых учетных данных, которые будут применяться при проверке вводимых имени пользователя и пароля, или, по меньшей мере, наличие доверия сообщению клиента о том, что аутентификация завершилась успешно. Аутентификация должна всегда выполняться на сервере.

Чтобы снабдить метод `Login()` представлением для визуализации, создадим папку `Views/Account` и поместим в нее файл представления Razor по имени `Login.cshtml` с содержимым, показанным в листинге 12.10.

Листинг 12.10. Содержимое файла `Login.cshtml` из папки `Views/Account`

```
@model LoginModel
@{
  ViewBag.Title = "Log In";
  Layout = "_AdminLayout";
}
<div class="text-danger" asp-validation-summary="All"></div>
<form asp-action="Login" asp-controller="Account" method="post">
  <input type="hidden" asp-for="ReturnUrl" />
  <div class="form-group">
    <label asp-for="Name"></label>
    <div><span asp-validation-for="Name" class="text-danger"></span></div>
    <input asp-for="Name" class="form-control" />
  </div>
```

```
  <div class="form-group">
    <label asp-for="Password"></label>
    <div><span asp-validation-for="Password" class="text-danger"></span>
    </div>
    <input asp-for="Password" class="form-control" />
  </div>
  <button class="btn btn-primary" type="submit">Log In</button>
</form>
```

Финальный шаг связан с изменением разделяемой компоновки для администрирования, чтобы добавить кнопку Log Out (Выход), которая позволит текущему пользователю выходить из приложения за счет отправки запроса действию `Logout` (листинг 12.11). Это удобное средство, облегчающее тестирование приложения, без которого пришлось бы очищать cookie-наборы браузера, чтобы возвращаться в состояние, когда аутентификация еще не прошла.

Листинг 12.11. Добавление кнопки Log Out в файле `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
  <style>
    .input-validation-error {
      border-color: red;
      background-color: #fee;
    }
  </style>
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script src="/lib/jquery-validation/dist/jquery.validate.min.js">
  </script>
  <script
    src=
"/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
  </script>
</head>
<body class="m-1 p-1">
  <div class="bg-info p-2 row">
    <div class="col">
      <h4>@ViewBag.Title</h4>
    </div>
    <div class="col-2">
      <a class="btn btn-sm btn-primary"
         asp-action="Logout" asp-controller="Account">Log Out</a>
    </div>
  </div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success mt-1">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>
```

Тестирование политики безопасности

Теперь можно протестировать политику безопасности, запустив приложение и запросив URL вида `/Admin/Index`. Поскольку в настоящий момент мы еще не прошли аутентификацию и пытаемся обратиться к действию, которое требует авторизации, браузер будет перенаправлен на URL вида `/Account/Login`. Введем **`Admin`** и **`Secret123$`** в качестве имени и пароля и отправим форму. Контроллер `Account` сравнит предоставленные учетные данные с начальными данными, добавленными в базу данных `Identity`, и (при условии ввода правильных сведений) аутентифицирует нас, после чего перенаправит обратно на `/Account/Login`, куда теперь имеется доступ. Процесс проиллюстрирован на рис. 12.1.

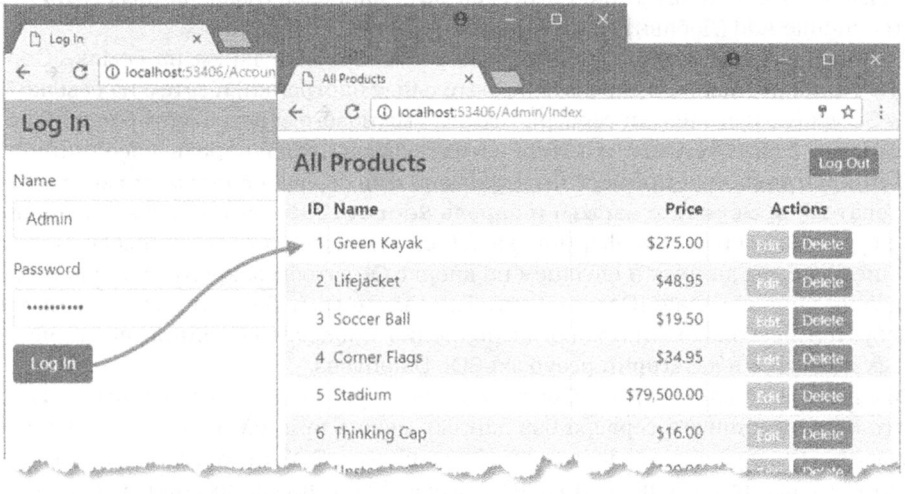


Рис. 12.1. Процесс аутентификации/авторизации для административных функций

Развертывание приложения

Все средства и функциональность приложения SportsStore на месте, так что наступил момент для его подготовки и развертывания в производственной среде. В отношении приложений ASP.NET Core MVC доступно множество вариантов размещения, и в настоящей главе используется один из них — платформа Microsoft Azure. Она выбрана из-за того, что поступает от Microsoft и предлагает бесплатные учетные записи, т.е. вы можете полностью проработать пример SportsStore, даже если не хотите применять Azure в собственных проектах.

На заметку! В данном разделе понадобится учетная запись Azure. Если у вас ее пока нет, тогда можете создать бесплатную учетную запись на `https://azure.microsoft.com/`.

Создание баз данных

Начальной задачей является создание баз данных, которые приложение SportsStore будет использовать в производственной среде. Такую задачу можно выполнять как часть процесса разработки в Visual Studio, но трудность ситуации в том, что нужно

знать строки подключений для баз данных до развертывания, а это относится к процессу, который создает базы данных.

Внимание! Портал Azure часто меняется по мере того, как в Microsoft добавляют новые средства и пересматривают существующие. Инструкции, приводимые в настоящем разделе, были точными на время его написания, но к моменту выхода книги необходимые шаги могут слегка измениться. Базовый подход должен остаться таким же, но имена полей данных и точный порядок шагов может потребовать определенного экспериментирования, чтобы добиться правильных результатов.

Самый простой подход предусматривает вход на портал `https://portal.azure.com` с применением своей учетной записи Azure и создание баз данных вручную. После входа выберем категорию ресурсов SQL Databases (Базы данных SQL) и щелкнем на кнопке Add (Добавить), чтобы создать новую базу данных.

Для первой базы данных укажем имя **`products`**. Щелкнем на ссылке Configure Required Settings (Конфигурировать обязательные настройки) и затем на ссылке Create a New Server (Создать новый сервер). Введем имя нового сервера, которое должно быть уникальным в рамках Azure, и выберем имя пользователя и пароль для администратора. В рассматриваемом примере было указано имя сервера **`sportsstorecore2db`**, имя пользователя **`sportsstoreadmin`** и пароль **`Secret123$`**. Вам понадобится использовать другое имя сервера и сформировать более надежный пароль. Выберем местоположение для базы данных и щелкнем на кнопке OK, чтобы закрыть экран параметров, и далее на кнопке Create (Создать), чтобы создать саму базу данных. Порталу Azure потребуется несколько минут для выполнения процесса создания, после чего база данных появится в категории ресурсов SQL Databases.

Создадим еще один сервер баз данных SQL, указав на этот раз имя **`identity`**. Вместо создания нового сервера баз данных можно применять тот, который был создан ранее. Результатом окажутся две базы данных SQL Server, размещаемые Azure, детали которых приведены в табл. 12.1. У вас будут другие имена серверов баз данных и наверняка более надежные пароли.

Таблица 12.1. Базы данных Azure для приложения SportsStore

Имя базы данных	Имя сервера	Имя пользователя-администратора	Пароль
`products`	`sportsstorecore2db`	`sportsstoreadmin`	`Secret123$`
`identity`	`sportsstorecore2db`	`sportsstoreadmin`	`Secret123$`

Открытие доступа в брандмауэре для конфигурирования

Далее необходимо создать схемы баз данных, и сделать это проще всего, открыв доступ в брандмауэре Azure, чтобы можно было запускать команды Entity Framework Core из машины разработки.

Выберем одну их двух баз данных в категории ресурсов SQL Databases, щелкнем на кнопке Tools (Сервис) и затем щелкнем на ссылке Open in Visual Studio (Открыть в Visual Studio). Теперь щелкнем на ссылке Configure Your Firewall (Конфигурировать брандмауэр), щелкнем на кнопке Add Client IP (Добавить IP-адрес клиента) и щелкнем на кнопке Save (Сохранить). В результате текущий IP-адрес получит возможность достигать сервера баз данных и выполнять команды конфигурирования. (Проинспектировать схему базы данных можно, щелкнув на кнопке Open In Visual Studio, чтобы открыть

Visual Studio и воспользоваться окном SQL Server Object Explorer для исследования базы данных.)

Получение строк подключений

Вскоре понадобятся строки подключений для новых баз данных. Портал Azure предоставляет такую информацию по щелчку на базе данных в категории ресурсов SQL Databases через ссылку Show Database Connection Strings (Показать строки подключений для баз данных). Строки подключений предлагаются для разных платформ разработки; приложениям .NET требуются строки ADO.NET. Вот строка подключения, которую портал Azure предоставляет для базы данных `products`:

```
Server=tcp:sportsstorecore2db.database.windows.net,1433;
Initial Catalog=products;
Persist Security Info=False;
User ID={ваше_имя_пользователя};Password={ваш_пароль};
MultipleActiveResultSets=True;Encrypt=True;
TrustServerCertificate=False;Connection Timeout=30;
```

В зависимости от того, как портал Azure подготовил базу данных, вы будете видеть разные параметры конфигурации. Обратите внимание на выделенные полужирным заполнители для имени пользователя и пароля, которые должны быть изменены, когда вы применяете строку подключения при конфигурировании приложения.

Подготовка приложения

Перед тем, как приложение можно будет развернуть, предстоит выполнить определенные подготовительные шаги, чтобы привести его в готовность к производственной среде. В последующих разделах будет изменен способ отображения сообщений об ошибках и настроены строки подключения для производственных баз данных.

Создание контроллера и представления для отображения сообщений об ошибках

В текущий момент приложение сконфигурировано на использование страниц ошибок, дружественных к разработчику, которые предоставляют полезную информацию при возникновении проблемы. Конечные пользователи не должны видеть такую информацию, поэтому добавим в папку `Controllers` файл класса по имени `ErrorController.cs` с определением простого контроллера, показанного в листинге 12.12.

Листинг 12.12. Содержимое файла `ErrorController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers {

  public class ErrorController : Controller {
    public ViewResult Error() => View();
  }
}
```

В контроллере определено действие `Error`, которое визуализирует стандартное представление. Чтобы снабдить контроллер представлением, создадим папку `Views/Error` и добавим в нее файл представления Razor по имени `Error.cshtml` с разметкой, приведенной в листинге 12.13.

Листинг 12.13. Содержимое файла `Error.cshtml` из папки `Views/Error`

```
@{
  Layout = null;
}
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
  <title>Error</title>
</head>
<body>
  <h2 class="text-danger">Error.</h2>
  <h3 class="text-danger">An error occurred while processing your request.</h3>
</body>
</html>
```

Страница ошибки подобного рода является последним средством, а потому ее лучше сохранить как можно более простой и не полагаться на разделяемые компоновки, компоненты представлений или другие расширенные возможности. В данном случае мы отключаем разделяемые компоновки и определяем простой HTML-документ с сообщением о возникновении ошибки, не предоставляя никакой информации о том, что произошло.

Определение настроек производственных баз данных

Следующим шагом будет создание файла, который снабдит приложение строками подключения к его базам данных в производственной среде. Добавим в проект `SportsStore` новый файл по имени `appsettings.production.json` с применением шаблона элемента ASP.NET Configuration File (Файл конфигурации ASP.NET) и поместим в него содержимое, показанное в листинге 12.14.

Совет. В списке файлов окна Solution Explorer файл `appsettings.production.json` находится внутри узла `appsettings.json`, который понадобится раскрыть, если позже вы захотите отредактировать данный файл.

Листинг 12.14. Содержимое файла `appsettings.production.json` из папки `SportsStore`

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString":
      "Server=tcp:sportsstorecore2db.database.windows.net,1433;
      Initial Catalog=products;Persist Security Info=False;
      User ID={ваше_имя_пользователя};
      Password={ваш_пароль};MultipleActiveResultSets=True;Encrypt=True;
      TrustServerCertificate=False;Connection Timeout=30;"
},
    "SportStoreIdentity": {
      "ConnectionString":
      "Server=tcp:sportsstorecore2db.database.windows.net,1433;
```

```
      Initial Catalog=identity;Persist Security Info=False;
      User ID={ваше_имя_пользователя};
      Password={ваш_пароль};MultipleActiveResultSets=True;Encrypt=True;
      TrustServerCertificate=False;Connection Timeout=30;"
    }
  }
}
```

Файл неудобен для чтения, т.к. строки подключений разрывать нельзя. Содержимое данного файла дублирует раздел строк подключений файла `appsettings.json`, но здесь используются строки подключений Azure. (Не забудьте заменить заполнители для имени пользователя и пароля.) Кроме того, параметр `MultipleActiveResultSets` был установлен в `True`, что делает возможными множество параллельных запросов и устраняет условия для возникновения распространенной ошибки, которые появляются при выполнении сложных запросов LINQ в отношении данных приложения.

На заметку! При вставке в строки подключения своего имени пользователя и пароля удалите символы фигурных скобок, чтобы в итоге получилось `Password=MyPassword`, но не `Password={MyPassword}`.

Конфигурирование приложения

Теперь можно изменить код класса `Startup`, чтобы в случае нахождения в производственной среде приложение вело себя по-другому. Изменения представлены в листинге 12.15.

Листинг 12.15. Конфигурирование приложения в файле `Startup.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

namespace SportsStore {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreProducts:ConnectionString"]));
```

```
    services.AddDbContext<AppIdentityDbContext>(options =>
      options.UseSqlServer(
        Configuration["Data:SportStoreIdentity:ConnectionString"]));

    services.AddIdentity<IdentityUser, IdentityRole>()
      .AddEntityFrameworkStores<AppIdentityDbContext>()
      .AddDefaultTokenProviders();

    services.AddTransient<IProductRepository, EFProductRepository>();
    services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    services.AddTransient<IOrderRepository, EFOrderRepository>();
    services.AddMvc();
    services.AddMemoryCache();
    services.AddSession();
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    if (env.IsDevelopment()) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
    } else {
      app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseSession();
    app.UseAuthentication();
    app.UseMvc(routes => {
      routes.MapRoute(name: "Error", template: "Error",
        defaults: new { controller = "Error", action = "Error" });
      routes.MapRoute(name: null,
        template: "{category}/Page{productPage:int}",
        defaults: new { controller = "Product", action = "List" }
      );
      routes.MapRoute(name: null,template: "Page{productPage:int}",
        defaults: new { controller = "Product",
        action = "List", productPage = 1 }
      );
      routes.MapRoute(name: null, template: "{category}",
        defaults: new { controller = "Product",
        action = "List", productPage = 1 }
      );
      routes.MapRoute(name: null,template: "",
        defaults: new { controller = "Product",
        action = "List", productPage = 1 }
      );
      routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");
    });
    // SeedData.EnsurePopulated(app);
    // IdentitySeedData.EnsurePopulated(app);
  }
}
}
```

Интерфейс IHostingEnvironment используется для предоставления информации о среде, в которой функционирует приложение, такой как среда разработки или производственная среда. В случае производственной среды ASP.NET Core будет загружать содержимое файла appsettings.production.json, чтобы переопределить настройки в файле appsettings.json, а это значит, что Entity Framework Core будет подключаться к базам данных Azure вместо LocalDB. Доступно множество параметров для настройки конфигурации приложения в различных средах, которые будут рассматриваться в главе 14.

Также были закомментированы операторы, которые производят начальное заполнение баз данных, что объясняется в разделе "Управление начальным заполнением баз данных" далее в главе.

Применение миграций к базам данных

Чтобы настроить базы данных со схемами, требующимися приложению, откроем окно командной строки или PowerShell и перейдем в папку проекта SportsStore. Установка среды так, что инструмент командной строки dotnet будет использовать строки подключения для Azure, требует установки переменной среды. В случае применения окна PowerShell установка переменной среды выполняется с помощью следующей команды:

```
$env:ASPNETCORE_ENVIRONMENT="Production"
```

Если же используется окно командной строки, то переменная среды устанавливается посредством такой команды:

```
set ASPNETCORE_ENVIRONMENT=Production
```

Для применения миграций в проекте к базам данных Azure нужно запустить показанные далее команды, находясь в папке проекта SportsStore:

```
dotnet ef database update --context ApplicationDbContext
dotnet ef database update --context AppIdentityDbContext
```

Переменная среды указывает размещающую среду, которая применяется для получения строк подключения к базам данных. Если приведенные команды не работают, тогда понадобится проверить, сконфигурирован ли брандмауэр Azure для разрешения доступа к машине разработки, как было описано ранее в главе, а также корректно ли скопированы и модифицированы строки подключения.

Управление начальным заполнением баз данных

В листинге 12.15 операторы, выполняющие начальное заполнение баз данных в классе Startup, закомментированы. Дело в том, что команды Entity Framework Core, используемые в предыдущем разделе для применения миграций к базе данных, полагаются на службы, устанавливаемые классом Startup. При наличии упомянутых операторов код начального заполнения баз данных вызывался бы перед применением миграций, приводя к ошибке и препятствуя нормальной работе миграций. Когда базы данных установлены, проблема не возникает. В случае базы данных products причина в том, что метод SeedData.EnsurePopulated() применяет миграции перед заполнением начальными данными, и в том, что начальные данные Identity не добавляются в приложение до тех пор, пока к базе данных не будет применена миграция.

В отношении производственной среды мы хотим принять другой подход к заполнению начальными данными. Для пользовательских учетных записей мы собираемся поместить в базу данных учетную запись администратора, когда производится попытка входа. Мы добавим к инструменту администрирования средство для начального заполнения базы данных товаров, чтобы производственная система могла заполняться тестовыми данными или оставаться пустой для поступления реальных данных.

На заметку! Начальное заполнение данными аутентификации в производственной системе должно делаться осторожно, а приложение обязано использовать функциональные средства, описанные в главах 28–30, для изменения пароля сразу же после того, как приложение развернуто.

Начальное заполнение данными о пользователях

Первый шаг в изменении способа заполнения данными о пользователях предусматривает упрощение кода в классе `IdentitySeedData` (листинг 12.16).

Листинг 12.16. Упрощение кода в файле `IdentitySeedData.cs` из папки `Models`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.DependencyInjection;
using System.Threading.Tasks;

namespace SportsStore.Models {

  public static class IdentitySeedData {
    private const string adminUser = "Admin";
    private const string adminPassword = "Secret123$";

    public static async Task EnsurePopulated(UserManager<IdentityUser>
                                             userManager) {

      IdentityUser user = await userManager.FindByIdAsync(adminUser);
      if (user == null) {
        user = new IdentityUser("Admin");
        await userManager.CreateAsync(user, adminPassword);
      }
    }
  }
}
```

Вместо получения самой службы `UserManager<IdentityUser>` метод `EnsurePopulated()` получает в качестве аргумента объект, что позволяет интегрировать начальное заполнение базы данных в класс `AccountController` (листинг 12.17).

Листинг 12.17. Начальное заполнение базы данных в файле `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;
```

```
using SportsStore.Models;

namespace SportsStore.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<IdentityUser> userManager;
    private SignInManager<IdentityUser> signInManager;

    public AccountController(UserManager<IdentityUser> userMgr,
        SignInManager<IdentityUser> signInMgr) {
      userManager = userMgr;
      signInManager = signInMgr;
      IdentitySeedData.EnsurePopulated(userMgr).Wait();
    }

    // ...для краткости другие методы не показаны...

  }
}
```

Внесенные изменения гарантируют, что база данных `Identity` будет заполняться каждый раз, когда создается объект `AccountController` для обработки HTTP-запроса. Конечно, это далеко от идеала, но не существует хорошего способа начального заполнения базы данных, и такой подход обеспечивает возможность администрирования приложения в производственной среде и среде разработки, хотя за счет ряда дополнительных запросов к базе данных.

Начальное заполнение данными о товарах

Что касается данных о товарах, то мы собираемся снабдить администратора кнопкой, щелчок на которой приведет к начальному заполнению базы данных, когда она пуста. Первым делом изменим код начального заполнения, чтобы задействовать в нем интерфейс, который позволит обращаться к службам, предоставляемым контроллером, а не классом `Startup` (листинг 12.18). Кроме того, мы закомментировали операторы для автоматического применения любых ожидающих миграций, которые могут привести к утере данных и должны использоваться с особой осторожностью в производственных системах.

Листинг 12.18. Подготовка к ручному начальному заполнению в файле `SeedData.cs` из папки `Models`

```
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using System;

namespace SportsStore.Models {

  public static class SeedData {

    public static void EnsurePopulated(IServiceProvider services) {
      ApplicationDbContext context =
        services.GetRequiredService<ApplicationDbContext>();

      // context.Database.Migrate();
      if (!context.Products.Any()) {
        context.Products.AddRange(
```

```
        // ...для краткости другие операторы не показаны...
      );
      context.SaveChanges();
    }
  }
}
```

Следующий шаг заключается в обновлении контроллера Admin с целью добавления метода действия, который инициирует операцию начального заполнения (листинг 12.19).

Листинг 12.19. Начальное заполнение базы данных в файле AdminController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;

namespace SportsStore.Controllers {

  [Authorize]
  public class AdminController : Controller {
    private IProductRepository repository;

    public AdminController(IProductRepository repo) {
      repository = repo;
    }

    public ViewResult Index() => View(repository.Products);

    // ...для краткости другие методы не показаны...

    [HttpPost]
    public IActionResult SeedDatabase() {
      SeedData.EnsurePopulated(HttpContext.RequestServices);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Новое действие декорируется атрибутом HttpPost, так что оно может быть целью запросов POST, и после начального заполнения базы данных перенаправляет браузер на метод действия Index(). Осталось лишь создать кнопку для начального заполнения базы данных, которая будет отображаться, когда база данных пуста (листинг 12.20).

Листинг 12.20. Добавление кнопки для начального заполнения базы данных в файле Index.cshtml из папки Views/Admin

```
@model IEnumerable<Product>

@{
  ViewBag.Title = "All Products";
  Layout = "_AdminLayout";
}
```

```
@if (Model.Count() == 0) {
  <div class="text-center m-2">
    <form asp-action="SeedDatabase" method="post">
      <button type="submit" class="btn btn-danger">Seed Database</button>
    </form>
  </div>
} else {
  <table class="table table-striped table-bordered table-sm">
    <tr>
      <th class="text-right">ID</th>
      <th>Name</th>
      <th class="text-right">Price</th>
      <th class="text-center">Actions</th>
    </tr>
    @foreach (var item in Model) {
      <tr>
        <td class="text-right">@item.ProductID</td>
        <td>@item.Name</td>
        <td class="text-right">@item.Price.ToString("c")</td>
        <td class="text-center">
          <form asp-action="Delete" method="post">
            <a asp-action="Edit" class="btn btn-sm btn-warning"
               asp-route-productId="@item.ProductID">
              Edit
            </a>
            <input type="hidden" name="ProductID"
                  value="@item.ProductID" />
            <button type="submit" class="btn btn-danger btn-sm">
              Delete
            </button>
          </form>
        </td>
      </tr>
    }
  </table>
}
<div class="text-center">
  <a asp-action="Create" class="btn btn-primary">Add Product</a>
</div>
```

Развертывание приложения

Чтобы развернуть приложение, щелкнем правой кнопкой мыши на элементе проекта `SportsStore` в окне Solution Explorer (проекта, но не решения) и выберем в контекстном меню пункт Publish (Опубликовать). Среда Visual Studio предложит выбрать метод опубликования (рис. 12.2).

Что делать, если развертывание терпит неудачу?

Самой главной причиной неудачи развертывания являются строки подключения, либо потому, что они были некорректно скопированы из среды Azure, либо из-за того, что при редактировании для вставки имени пользователя и пароля была допущена ошибка.

В случае отказа развертывания начинать поиск проблемы нужно со строк подключения. Если команды `dotnet ef database update` из раздела "Применение миграций к базам данных" выше в главе не приводят к ожидаемым результатам, то развертывание потерпит неудачу. Если команды отработали успешно, но развертывание закончилось неудачей, тогда необходимо проверить, установлена ли переменная среды, поскольку есть вероятность того, что вместо базы данных в облаке была подготовлена локальная база данных.

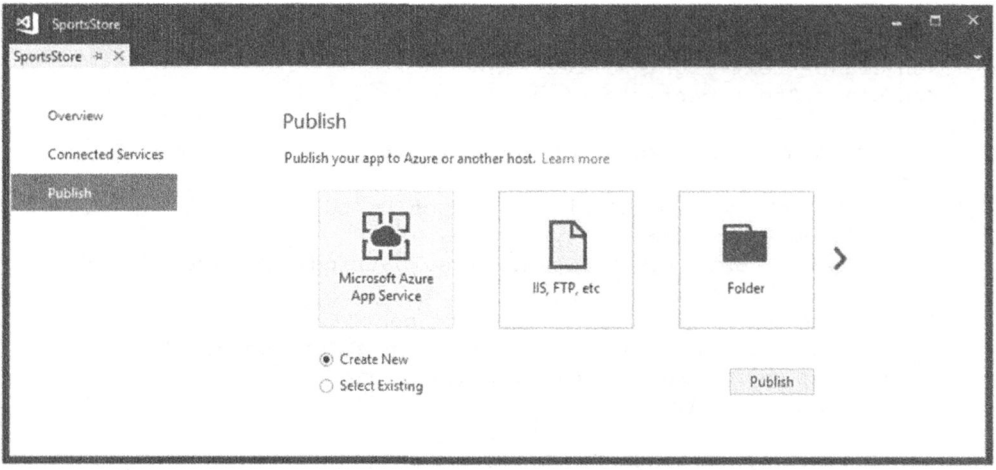


Рис. 12.2. Выбор метода опубликования

Выберем вариант Microsoft Azure App Service (Служба приложений Microsoft Azure) и удостоверимся в выборе переключателя Create New (Создать новую); переключатель Select Existing (Выбрать существующую) используется для обновления существующего развернутого приложения. Будет предложено предоставить детали для развертывания. Начнем со щелчка на кнопке Add an Account (Добавить учетную запись) и ввода учетных данных Azure.

После ввода учетных данных можно выбрать имя для развернутого приложения и ввести информацию относительно службы, которая будет зависеть от типа имеющейся учетной записи Azure, желаемой области для развертывания и требующейся службы развертывания (рис. 12.3).

После того, как служба сконфигурирована, щелкнем на кнопке Create (Создать). Как только служба установится, будет выдана сводка по операции опубликования, которая отправит приложение в службу размещения (рис. 12.4).

Щелкнем на кнопке Publish (Опубликовать), чтобы запустить процесс развертывания. Выбрав пункт Web Publish Activity (Активность веб-развертывания) в меню View⇨Other Windows (Вид⇨Другие окна) среды Visual Studio, можно наблюдать за ходом развертывания. Наберитесь терпения, т.к. отправка всех файлов из проекта службе Azure может занять некоторое время. Последующие обновления будут проходить быстрее, потому что передаче подлежат только измененные файлы.

По завершении развертывания среда Visual Studio откроет новое окно браузера для развернутого приложения. Поскольку база данных товаров пуста, отобразится компоновка, показанная на рис. 12.5.

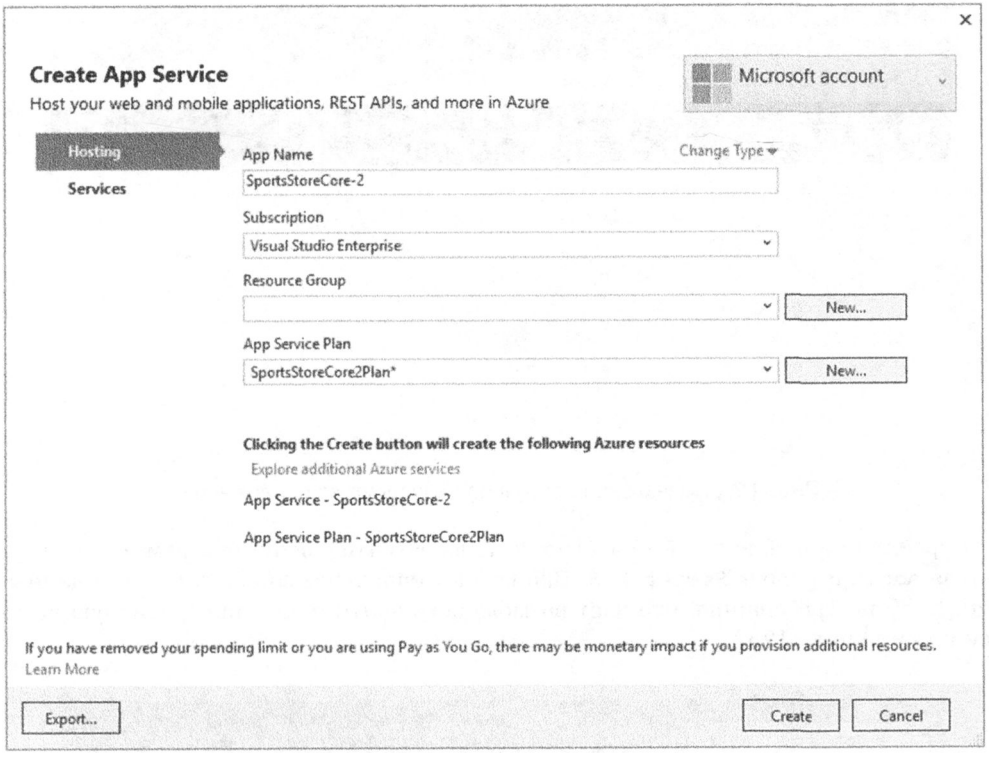


Рис. 12.3. Создание новой службы приложений Azure

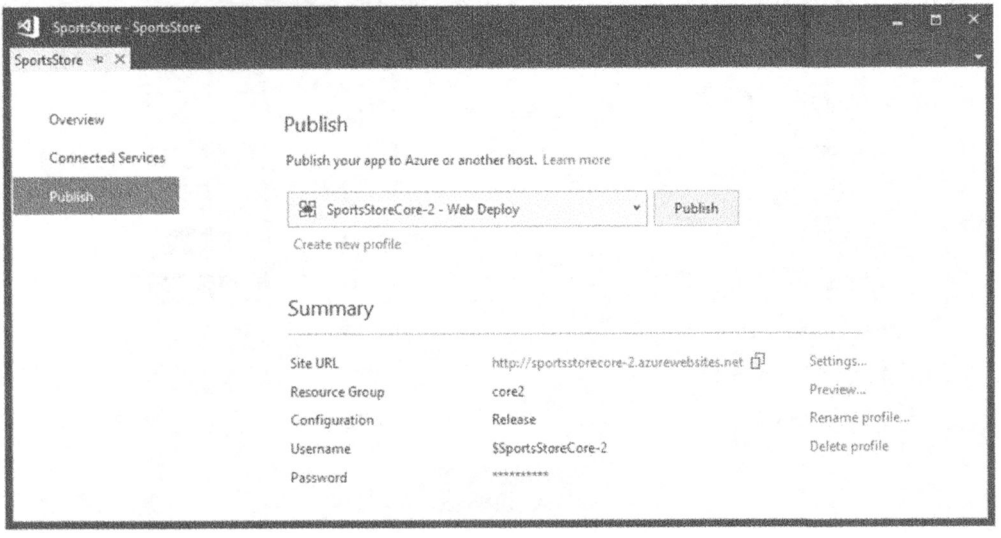


Рис. 12.4. Сводка по операции опубликования

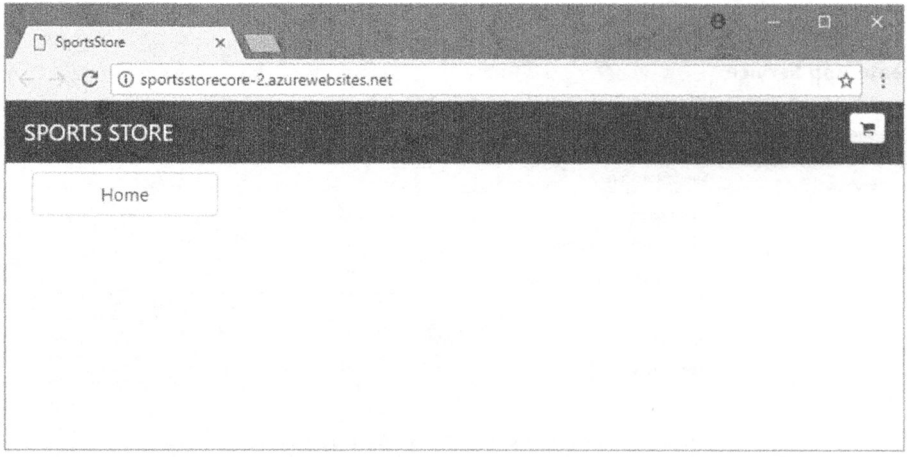


Рис. 12.5. Начальное состояние развернутого приложения

Перейдем на URL вида `/Admin/Index` и для аутентификации введем имя пользователя **`Admin`** и пароль **`Secret123$`**. Произойдет начальное заполнение базы данных `Identity` по требованию, что даст возможность войти в административную часть приложения (рис. 12.6).

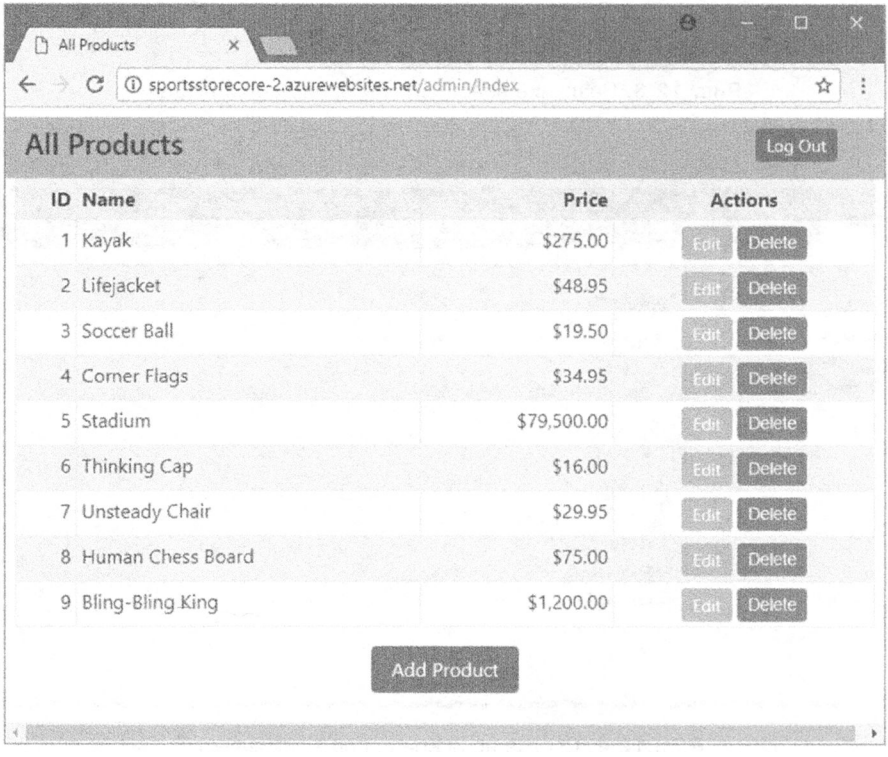


Рис. 12.6. Экран администрирования

Щелкнем на кнопке Seed Database (Заполнить базу данных), чтобы заполнить базу данных товаров, что приведет к результату, представленному на рис. 12.7. Затем можно перейти на корневой URL приложения и работать с ним обычным образом.



Рис. 12.7. Заполненная база данных

Резюме

В этой и предшествующих главах демонстрировалось применение инфраструктуры ASP.NET Core MVC для создания реалистичного приложения электронной коммерции. В приведенном расширенном примере были проиллюстрированы многие основные средства MVC: контроллеры, методы действий, маршрутизация, представления, метаданные, проверка достоверности, компоновки, аутентификация и т.д. Вы также видели, как пользоваться рядом ключевых технологий, имеющих отношение к MVC, в том числе Entity Framework Core, внедрение зависимостей и модульное тестирование. Результатом стало приложение с ясной и ориентированной на компоненты архитектурой, обеспечивающей разделение обязанностей, и кодовой базой, которую будет легко расширять и сопровождать. Итак, разработка приложения SportsStore завершена. В следующей главе будет показано, как применять Visual Studio Code для создания приложений ASP.NET Core MVC.

ГЛАВА 13

Работа с Visual Studio Code

В настоящей главе будет показано, как создать приложение ASP.NET Core MVC с применением Visual Studio Code — межплатформенного редактора с открытым кодом производства Microsoft. Несмотря на название, продукт Visual Studio Code не имеет отношения к Visual Studio и основан на инфраструктуре Electron, используемой редактором Atom, который популярен среди разработчиков, применяющих другие инфраструктуры для построения веб-приложений, такие как Angular.

Продукт Visual Studio Code поддерживает операционные системы Windows, macOS и наиболее популярные дистрибутивы Linux. Он сформировался в удобную и полнофункциональную среду разработки, хотя и лишен всех украшений, предлагаемых Visual Studio. Я обнаружил, что использую среду Visual Studio Code все чаще и чаще, поскольку с ней просто работать, она быстрая и располагает хорошей поддержкой других языков вроде JavaScript и TypeScript.

Настройка среды разработки

Процесс настройки Visual Studio Code требует выполнения определенной работы, т.к. некоторая функциональность, включенная в состав Visual Studio, в Visual Studio Code обрабатывается внешними инструментами. Одни инструменты идентичны тем, которые среда Visual Studio применяет "за кулисами", но другие являются новыми в мире разработки для .NET и могут быть незнакомыми. Хорошая новость в том, что эти инструменты широко используются разработчиками, которые ориентируются на другие инфраструктуры для построения веб-приложений, причем качество и функциональные средства находятся на высоком уровне. В последующих разделах мы исследуем процесс установки Visual Studio Code наряду с важными инструментами и дополнениями, требующимися для разработки приложений MVC.

Установка Node.js

В мире разработки клиентской стороны Node.js (или просто Node) представляет собой исполняющую среду, на которую полагаются многие популярные инструменты разработки. Продукт Node был создан в 2009 году как простая и эффективная исполняющая среда для серверных приложений, написанных на языке JavaScript. Она основана на механизме JavaScript, применяемом в браузере Chrome, и предлагает API-интерфейс для выполнения кода JavaScript за пределами среды браузера.

Исполняющая среда Node.js достигла определенных успехов в качестве сервера приложений, но в данной главе она интересна тем, что предоставляет основу для нового поколения межплатформенных инструментов построения и диспетчеров пакетов. Ряд интеллектуальных проектных решений, внедренных командой разработ-

чиков Node, и межплатформенная поддержка, обеспечиваемая исполняющей средой JavaScript браузера Chrome, создали благоприятную возможность, которой воспользовались полные энтузиазма проектировщики инструментов, особенно те из них, кто желал поддерживать разработку веб-приложений.

На заметку! Доступны две версии Node.js. Версия LTS (Long Term Support — долгосрочная поддержка) предоставляет стабильный фундамент для развертывания в производственных средах, где изменения должны быть сведены к минимуму. Обновления версии LTS выпускаются каждые 6 месяцев и сопровождаются в течение 18 месяцев. Версия Current (текущая) является более часто изменяющимся выпуском, в котором преимущество отдается новым средствам взамен стабильности. В настоящей главе мы будем применять выпуск Current.

Установка Node.js в Windows

Загрузим и запустим установщик Node.js для Windows из веб-сайта `http://nodejs.org`. При установке Node.js понадобится обеспечить выбор варианта Add to PATH (Добавить в переменную `PATH`). На рис. 13.1 показан установщик для Windows, который предлагает модифицировать переменную среды `PATH` в качестве варианта установки.

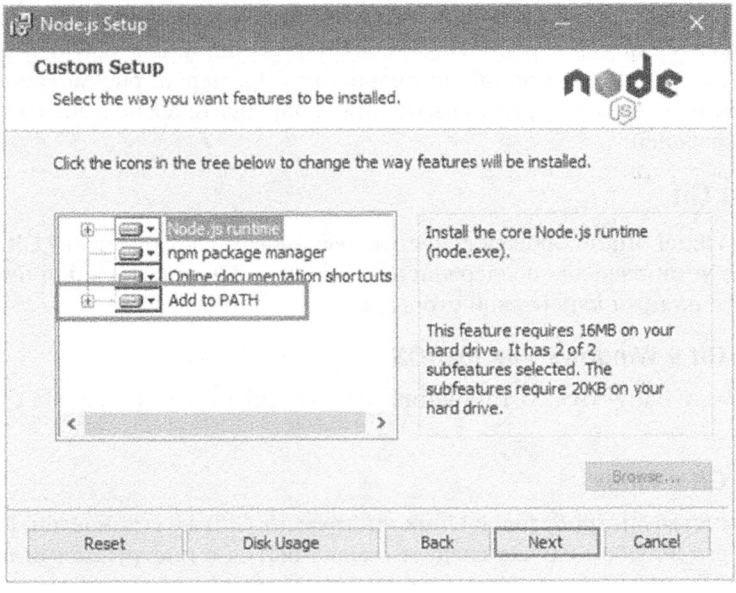


Рис. 13.1. Добавление пути к Node.js в переменную среды `PATH`

Установка Node.js в macOS

Установщик для macOS можно загрузить из веб-сайта `https://nodejs.org`. Запустим установщик и примем стандартные настройки. Когда установка завершится, удостоверимся в наличии пути `/usr/local/bin` внутри переменной `$PATH`.

Установка Node.js в Linux

Самый простой способ установки Node.js в Linux предполагает использование диспетчера пакетов; команда разработчиков Node предоставила инструкции для основных дистрибутивов по адресу `https://nodejs.org/en/download/package-manager/`. В среде Ubuntu для загрузки и установки Node.js применяются команды, приведенные в листинге 13.1.

Листинг 13.1. Установка Node.js

```
sudo curl -sL https://deb.nodesource.com/setup_6.x |
sudo -E bash -sudo apt-get install -y nodejs
```

Проверка установки Node

После завершения установки откроем новое окно командной строки и запустим команду, показанную в листинге 13.2, чтобы проверить, работает ли Node, и отобразить номер установленной версии.

Листинг 13.2. Проверка успешности установки Node

```
node -v
```

Если установка прошла успешно, и путь к Node добавился в переменную среды `PATH`, то появится номер версии. На момент написания главы текущей версией Node была 6.11.2. В случае если во время проработки примеров, рассмотренных в главе, вы получаете неожиданные результаты, тогда попробуйте воспользоваться указанной конкретной версией.

Установка Git

Продукт Visual Studio Code включает интегрированную поддержку Git, но требуется отдельная установка для поддержки инструмента Bower, который применяется при управлении пакетами клиентской стороны.

Установка Git в Windows или macOS

Загрузим и запустим установщик из веб-сайта `https://git-scm.com/downloads`.

Установка Git в Linux

В большинстве дистрибутивов Linux инструмент Git уже установлен. При желании установить его в любом случае следует ознакомиться с инструкциями по установке для своего дистрибутива, которые доступны по адресу:

`https://git-scm.com/download/linux`

В среде Ubuntu используется команда, представленная в листинге 13.3.

Листинг 13.3. Установка Git в Ubuntu

```
sudo apt-get install git
```

Проверка установки Git

После завершения установки запустим в новом окне командной строки/терминала команду, приведенную в листинге 13.4, чтобы проверить, установлен и доступен ли инструмент Git:

Листинг 13.4. Проверка успешности установки Git

```
git --version
```

Команда выведет версию установленного пакета Git. На момент написания главы последней версией Git для Windows и macOS была 2.14.1, а для Linux — 2.7.4.

Установка Bower

Продукт Node.js поступает с диспетчером пакетов Node (Node Package Manager — NPM), который применяется для загрузки и установки пакетов разработки, написанных на JavaScript. Для целей текущей главы нужен единственный пакет Bower, который используется для управления пакетами клиентской стороны и был описан в главе 6. Запустим команду, показанную в листинге 13.5, чтобы загрузить и установить Bower в Windows.

Листинг 13.5. Установка пакета Bower в Windows

```
npm install -g bower@1.8.0
```

В средах Linux и macOS применяется та же самая команда, но с обязательной командой `sudo` (листинг 13.6).

Листинг 13.6. Установка пакета Bower в Linux и macOS

```
sudo npm install -g bower@1.8.0
```

Установка .NET Core

При разработке приложений ASP.NET Core MVC требуется исполняющая среда .NET Core. Для каждой поддерживаемой платформы предусмотрен собственный процесс установки, описание которого доступно по адресу `www.microsoft.com/net/core`. В Microsoft предлагают установщики для Windows и macOS, а также предоставляют инструкции для установки в Linux с использованием архивов `tar`.

Установка .NET Core в Windows и macOS

Чтобы установить .NET Core в Windows или в macOS, нужно просто загрузить и запустить установщик .NET Core SDK.

Установка .NET Core в Linux

По адресу `www.microsoft.com/net/core` предлагаются инструкции по установке .NET Core в большинстве популярных дистрибутивов Linux. В главе применяется Ubuntu, и процесс требует первоначальной настройки новой подачи для `apt-get` с использованием команд из листинга 13.7.

Листинг 13.7. Подготовка к установке .NET Core в Ubuntu Linux

```
sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/
dotnet-release/xenial main" > /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
  --recv-keys 417A0893
sudo apt-get update
```

Далее производится установка .NET Core (листинг 13.8).

Листинг 13.8. Установка .NET Core в Ubuntu Linux

```
sudo apt-get install dotnet-sdk-2.0.0
```

Проверка установки .NET Core

Независимо от имеющейся платформы проверка того, что инфраструктура .NET Core установлена и готова к использованию, производится одинаково. Откроем новое окно командной строки/терминала и выполним команду, приведенную в листинге 13.9.

Листинг 13.9. Проверка версии .NET Core

```
dotnet --version
```

Команда `dotnet` запускает исполняющую среду .NET, после чего отобразится номер версии установленного пакета .NET Core. На момент написания главы текущим выпуском был 2.0.0, но ко времени выхода книги он с высокой вероятностью будет заменен более новым выпуском.

Установка Visual Studio Code

Самым важным шагом является загрузка и установка редактора Visual Studio Code, который доступен на веб-сайте `https://code.visualstudio.com`. Установочные пакеты предусмотрены для Windows, macOS и популярных дистрибутивов Linux. Понадобится загрузить и установить пакеты для выбранной платформы.

На заметку! Компания Microsoft предлагает новые выпуски Visual Studio Code ежемесячно, а это значит, что устанавливаемая вами версия будет отличаться от версии, которая применяется в главе. Хотя основы должны остаться теми же, выполнение некоторых примеров может потребовать определенной доли экспериментирования.

Установка Visual Studio Code в Windows

Чтобы установить Visual Studio Code для Windows, необходимо просто запустить установщик. После завершения процесса Visual Studio Code запустится и появится окно редактора (рис. 13.2).

Установка Visual Studio Code в macOS

Продукт Visual Studio Code предоставляется в виде архива `zip` для Mac, который доступен для загрузки по адресу `https://go.microsoft.com/fwlink/?LinkID=620882`. Раскроем архив и дважды щелкнем на содержащемся в нем файле `Visual Studio Code.app`, чтобы запустить Visual Studio Code и получить окно редактора (см. рис. 13.2).

Установка Visual Studio Code в Linux

Компания Microsoft предлагает файл `.deb` для Debian и Ubuntu и файл `.rpm` для Red Hat, Fedora и CentOS. Необходимо загрузить и установить файл для предпочитаемого дистрибутива Linux. Поскольку в главе используется Ubuntu, нужно загрузить файл `.deb` и установить его с помощью центра приложений Ubuntu.

После завершения установки выполним команду, приведенную в листинге 13.10, чтобы запустить Visual Studio Code и получить окно редактора, показанное на рис. 13.2.

Листинг 13.10. Запуск Visual Studio Code в Linux

```
/usr/share/code/code
```

Проверка установки Visual Studio Code

Тестирование успешности установки Visual Studio Code сводится к проверке возможности запуска приложения и открытию окна редактора (см. рис. 13.2). (Цветовая схема была изменена, т.к. стандартные темные цвета не очень хорошо подходят для получения экранных снимков.)

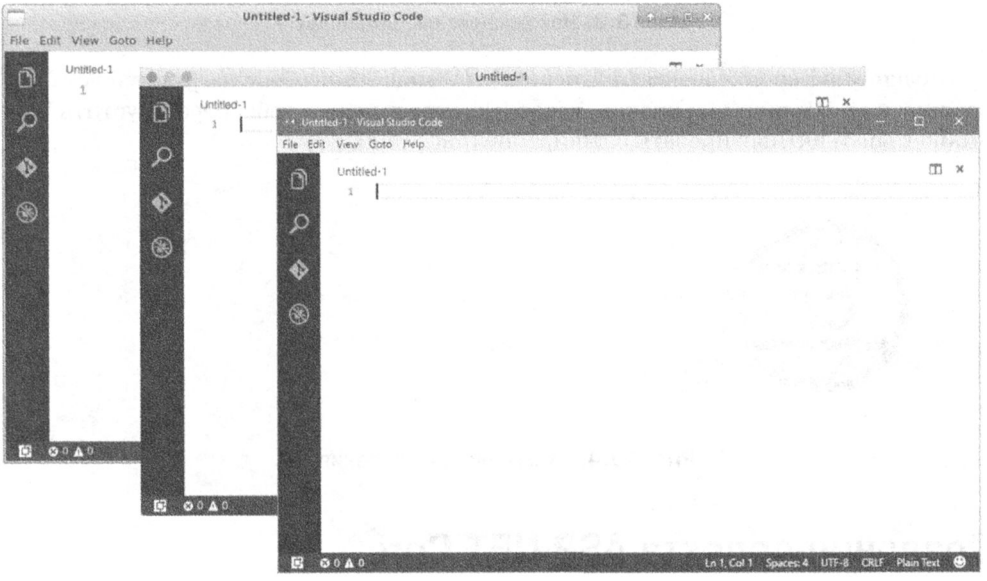


Рис. 13.2. Запуск Visual Studio Code в средах Windows, macOS и Ubuntu Linux

Установка расширения C# для Visual Studio Code

Редактор Visual Studio Code поддерживает функциональность, специфичную для языка, через расширения, хотя это не те же самые расширения, которые поддерживаются средой Visual Studio. Самое важное расширение, касающееся разработки приложений ASP.NET Core MVC, добавляет поддержку для языка C#, отсутствие которой может выглядеть как странное упущение в базовой установке. Тем не менее, такое положение дел отражает тот факт, что в Microsoft позиционировали Visual Studio Code

как универсальный межплатформенный редактор, который охватывает широчайший спектр языков и инфраструктур.

Чтобы установить расширение C#, щелкнем на значке Extensions (Расширения), расположенном в левой части окна Visual Studio Code. В поле поиска введем **csharp** и отыщем в списке расширение C# for Visual Studio Code (C# для Visual Studio Code), как показано на рис. 13.3.



Рис. 13.3. Нахождение расширения C#

Щелкнем на кнопке Install (Установить) и Visual Studio Code загрузит и установит расширение. Щелкнем на кнопке Reload (Перезагрузить), чтобы перезапустить Visual Studio Code и активизировать расширение (рис. 13.4).

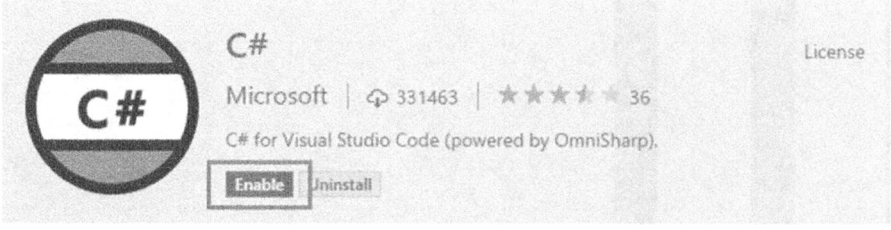


Рис. 13.4. Включение расширения C#

Создание проекта ASP.NET Core

Редактор Visual Studio Code не имеет интегрированной поддержки для создания проектов ASP.NET Core, но создавать новые проекты можно с применением командной строки `dotnet`.

Для корректной настройки проекта важно создать правильную структуру папок, особенно в случае, когда проект сопровождается модульными тестами. Создадим проект под названием `InvitesProjects` в подходящем местоположении — папке, которая будет содержать проект ASP.NET Core MVC и проект модульного тестирования.

Далее создадим внутри папки `InvitesProjects` папку по имени `PartyInvites`, с использованием окна командной строки перейдем в эту папку и запустим команду, приведенную в листинге 13.11.

Листинг 13.11. Создание нового проекта в папке `PartyInvites`

```
dotnet new web --language C# --framework netcoreapp2.0
```

Команда `dotnet new` предоставляет доступ посредством командной строки к шаблонам проектов, и указанный в листинге 13.11 шаблон `web` соответствует шаблону Empty (Пустой) из Visual Studio, который применялся в предшествующих главах. В табл. 13.1 описан набор шаблонов проектов, предназначенных для разработки приложений ASP.NET Core. (Существуют и другие шаблоны, но при построении приложений ASP.NET Core они не используются. Просмотреть полный список можно посредством команды `dotnet new --help`.)

Таблица 13.1. Шаблоны `dotnet new` для разработки приложений ASP.NET Core

Имя	Описание
`web`	Это шаблон Empty, применяемый в предшествующих главах, который создает проект ASP.NET Core, но не включает поддержку инфраструктуры MVC
`mvc`	Это шаблон Web Application (Model-View-Controller) (Веб-приложение (модель-представление-контроллер)), используемый в главе 2, который создает проект, включающий поддержку инфраструктуры MVC, а также заполнители для контроллеров и представлений
`xunit`	Это шаблон xUnit Test Project (.NET Core) (Проект тестирования xUnit (.NET Core)), который настраивает модульное тестирование с помощью пакета xUnit, как объяснялось в главе 7

Подготовка проекта с помощью Visual Studio Code

Чтобы открыть проект в Visual Studio Code, выберем пункт Open Folder (Открыть папку) в меню File (Файл), перейдем в папку `InvitesProjects` и щелкнем на кнопке Select Folder (Выбрать папку). Среда Visual Studio Code откроет проект и автоматически установит набор пакетов, требующихся для редактирования и отладки приложений C#. Спустя несколько секунд после начала редактирования файла появится сообщение, предлагающее добавить элементы в проект (рис. 13.5).



Рис. 13.5. Приглашение добавить активы в проект

Щелкнем на кнопке Yes (Да). Среда Visual Studio Code создаст папку `.vscode` и добавит в нее файлы, которые конфигурируют процесс построения. По умолчанию Visual Studio Code применяет компоновку из трех разделов. Боковая панель, выделенная на рис. 13.6, обеспечивает доступ к главным областям функциональности.

Самая верхняя кнопка позволяет открыть панель проводника, которая отобразит содержимое ранее открытой папки. Остальные кнопки предоставляют доступ к средству поиска, к интегрированному управлению исходным кодом, к отладчику и к набору установленных расширений.

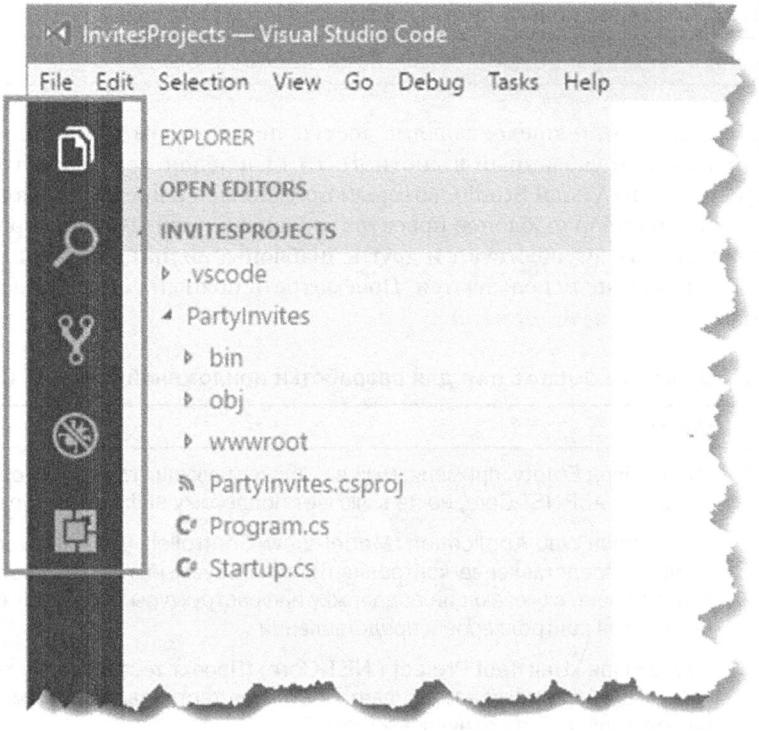


Рис. 13.6. Боковая панель Visual Studio Code

Щелчок на имени файла в панели проводника приводит к открытию файла для редактирования. Допускается редактировать несколько файлов одновременно. Можно создавать новые панели редактора, щелкая на кнопке Split Editor (Разделить окно редактора) в правой верхней части окна. Редактор Visual Studio Code довольно хорош, обладая неплохой поддержкой IntelliSense и содействием в завершении имен и версий пакетов NuGet и Bower.

Кроме содержимого папки проекта панель проводника показывает, какие файлы в текущий момент редактируются. Это позволяет легко сосредоточиться на подмножестве файлов, с которыми производится работа, что является удобным дополнением, когда приходится иметь дело с поднабором связанных файлов в крупном проекте.

Управление пакетами клиентской стороны

Как и в Visual Studio, при управлении пакетами клиентской стороны в проектах Visual Studio Code используется инструмент Bower, хотя требуется дополнительная работа.

Первым делом нужно добавить файл по имени `.bowerrc`, который применяется для того, чтобы указать Bower, где устанавливать пакеты. Щелкнем правой кнопкой мыши на папке `PartyInvites` и выберем в контекстном меню пункт New File (Новый файл), как демонстрируется на рис. 13.7.

Установим имя файла в `.bowerrc` (обратите внимание на наличие в имени двух букв `r`) и поместим в него содержимое, приведенное в листинге 13.12.

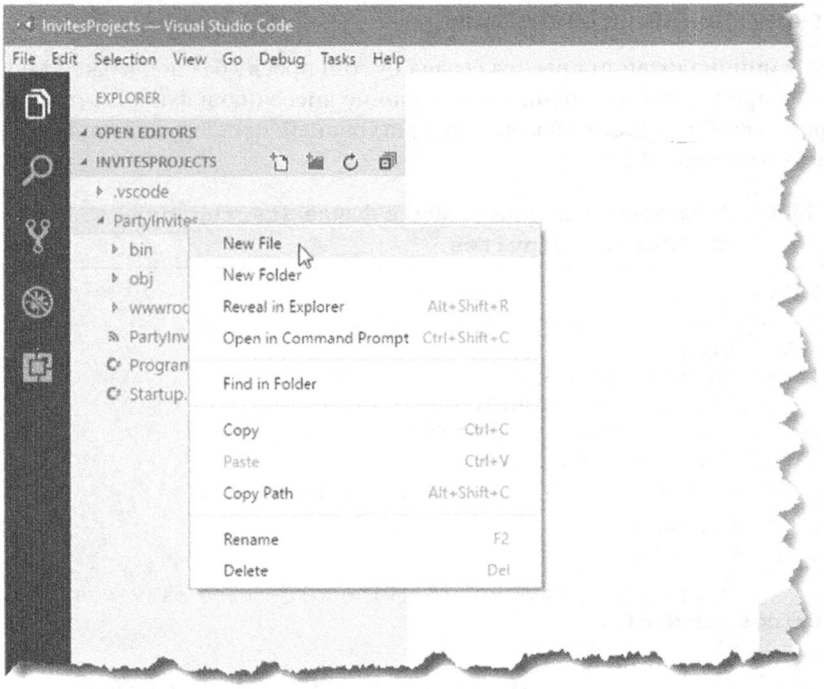


Рис. 13.7. Создание нового файла

Листинг 13.12. Содержимое файла `.bowerrc`

```
{
  "directory": "wwwroot/lib"
}
```

Затем создадим файл по имени `bower.json` с содержимым, представленным в листинге 13.13.

Листинг 13.13. Содержимое файла `bower.json`

```
{
  "name": "PartyInvites",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Воспользуемся окном командной строки/терминала, чтобы выполнить в папке `PartyInvites` команду из листинга 13.14, которая применяет инструмент Bower для загрузки и установки пакетов клиентской стороны, указанных в файле `bower.json`.

Листинг 13.14. Установка пакетов клиентской стороны

```
bower install
```

Конфигурирование приложения

Процесс инициализации проекта создал пустой проект без поддержки MVC. В листинге 13.15 приведены изменения, подлежащие внесению в файл `Startup.cs`, чтобы настроить MVC с наиболее базовой конфигурацией, используя операторы, которые были описаны в главе 14.

Листинг 13.15. Добавление поддержки MVC в файле `Startup.cs` из папки `PartyInvites`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace PartyInvites {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Построение и запуск проекта

Чтобы построить и запустить проект, понадобится выполнить в окне командной строки/терминала команду из листинга 13.16, находясь в папке `PartyInvites`.

Листинг 13.16. Запуск приложения

```
dotnet run
```

Среда Visual Studio Code скомпилирует код в проекте и с помощью сервера приложений Kestrel, рассматриваемого в главе 14, запустит приложение, ожидая HTTP-запросы на порте 5000.

Поскольку в Visual Studio Code не предоставляется поддержка для обнаружения изменений в файлах классов C#, после их модификации потребуется остановить приложение и запустить его снова.

Чтобы протестировать приложение, необходимо открыть окно браузера и перейти на URL вида `http://localhost:5000`. Будет получен ответ, представленный на рис. 13.8. Ошибка 404 связана с тем, что в текущий момент в проекте отсутствуют какие-либо контроллеры для обработки запросов.

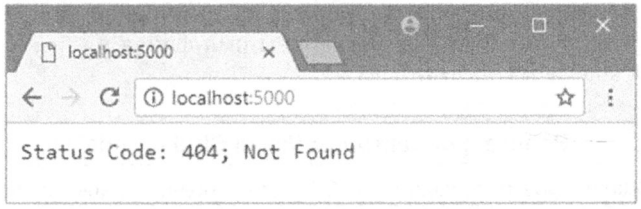


Рис. 13.8. Тестирование примера приложения

Воссоздание приложения PartyInvites

Все подготовительные шаги завершены, т.е. мы можем сосредоточиться на создании приложения MVC. Мы собираемся воссоздать простое приложение `PartyInvites` из главы 2, но с несколькими изменениями и дополнениями, которые подчеркнут особенности работы с Visual Studio Code.

Создание модели и хранилища

Для начала щелкнем правой кнопкой мыши на папке `PartyInvites` и выберем в контекстном меню пункт New Folder (Новая папка), как показано на рис. 13.9. Установим имя папки в `Models`.

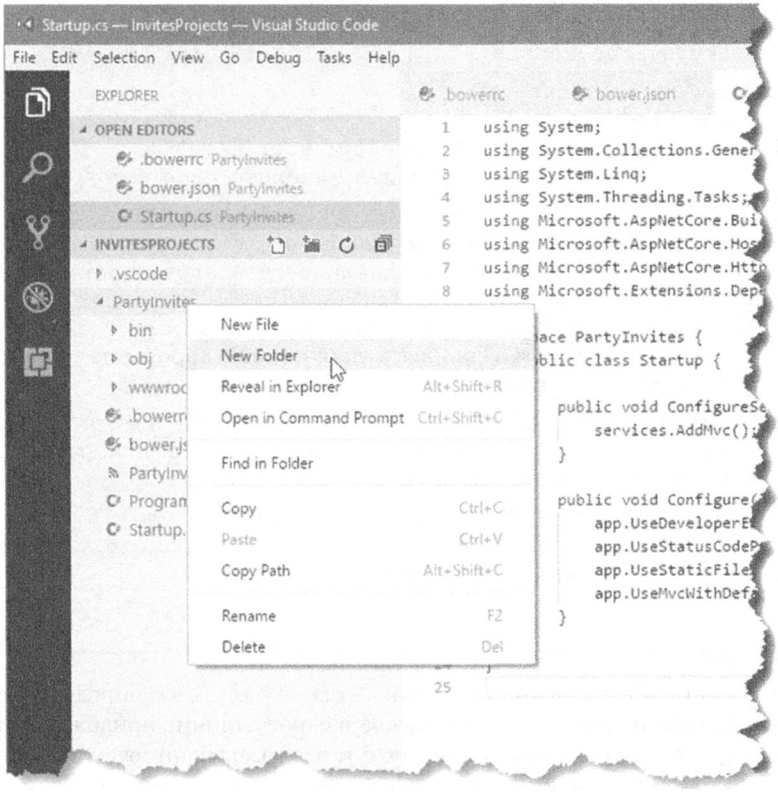


Рис. 13.9. Создание новой папки

Щелкнем правой кнопкой мыши на папке `Models` в панели проводника, выберем в контекстном меню пункт New File, установим имя файла в `GuestResponse.cs` и поместим в файл код C# из листинга 13.17.

Работа с редактором Visual Studio Code

Продукт Visual Studio Code (и расширение C#, установленное ранее в главе) предлагает полнофункциональные средства для редактирования файлов C#, а также файлов распространенных веб-форматов, подобных JavaScript, CSS и HTML. Во многих отношениях написание приложения MVC в Visual Studio Code имеет массу общего с тем же процессом в редакторе Visual Studio: доступна поддержка IntelliSense, кодирование цветом и подсветка ошибок (с советами по их исправлению).

Основной недочетом Visual Studio Code является отсутствие возможности настройки, особенно когда речь идет о форматировании кода. На момент написания главы были доступны варианты конфигурации для других языков, но расширение C# не допускает настройки, что несколько затрудняет работу, если предпочитаемый вами стиль написания кода не совпадает со стилем, предлагаемым по умолчанию. Однако в целом редактор отличается быстрой реакцией и легкостью в применении, так что написание приложений MVC в среде macOS или Linux не выглядит второсортным процессом.

Листинг 13.17. Содержимое файла `GuestResponse.cs` из папки `PartyInvites/Models`

```
using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {

  public class GuestResponse {
    public int id {get; set; }
    [Required(ErrorMessage = "Please enter your name")]
                            // Пожалуйста, введите свое имя
    public string Name { get; set; }

    [Required(ErrorMessage = "Please enter your email address")]
                        // Пожалуйста, введите свой адрес электронной почты
    [RegularExpression(".+\\@.+\\..+",
      ErrorMessage = "Please enter a valid email address")]
                  // Пожалуйста, введите допустимый адрес электронной почты
    public string Email { get; set; }

    [Required(ErrorMessage = "Please enter your phone number")]
                            // Пожалуйста, введите свой номер телефона
    public string Phone { get; set; }

    [Required(ErrorMessage = "Please specify whether you'll attend")]
                            // Пожалуйста, укажите, примете ли участие
    public bool? WillAttend { get; set; }
  }
}
```

Добавим в папку `Models` файл по имени `IRepository.cs` с определением интерфейса, приведенным в листинге 13.18. Самое важное отличие приложения в настоящей главе от приложения из главы 2 связано с тем, что мы планируем хранить данные модели в постоянной базе данных. Интерфейс `IRepository` описывает, каким образом приложение будет получать доступ к данным модели, не указывая реализацию.

Листинг 13.18. Содержимое файла IRepository.cs из папки PartyInvites/Models

```
using System.Collections.Generic;

namespace PartyInvites.Models {

  public interface IRepository {
    IEnumerable<GuestResponse> Responses {get; }

    void AddResponse(GuestResponse response);
  }
}
```

Добавим в папку Models файл по имени ApplicationDbContext.cs и определим в нем класс контекста базы данных (листинг 13.19).

Листинг 13.19. Содержимое файла ApplicationDbContext.cs из папки PartyInvites/Models

```
using Microsoft.EntityFrameworkCore;

namespace PartyInvites.Models {

  public class ApplicationDbContext : DbContext {

    public ApplicationDbContext() {}

    protected override void OnConfiguring(DbContextOptionsBuilder builder) {
      builder.UseSqlite("Filename=./PartyInvites.db");
    }

    public DbSet<GuestResponse> Invites {get; set;}
  }
}
```

Средство SQLite хранит данные в файле, который указывается классом контекста. В создаваемом примере приложения данные будут храниться в файле по имени PartyInvites.db, что определено в методе OnConfiguring().

Чтобы завершить набор классов, необходимых для хранения и доступа к данным модели, требуется реализация интерфейса IRepository, которая использует класс контекста базы данных. Добавим в папку Models файл по имени EFRepository.cs и поместим в него код, представленный в листинге 13.20.

Листинг 13.20. Содержимое файла EFRepository.cs из папки PartyInvites/Models

```
using System.Collections.Generic;

namespace PartyInvites.Models {

  public class EFRepository : IRepository {
    private ApplicationDbContext context = new ApplicationDbContext();

    public IEnumerable<GuestResponse> Responses => context.Invites;

    public void AddResponse(GuestResponse response) {
      context.Invites.Add(response);
      context.SaveChanges();
    }
  }
}
```

Класс EFRepository следует шаблону, который похож на тот, что применялся в главе 8 для настройки базы данных SportsStore. В листинге 13.21 видно, что к методу ConfigureServices() класса Startup добавлен оператор конфигурирования, который сообщает инфраструктуре ASP.NET Core о необходимости создания экземпляра класса EFRepository, когда требуются реализации интерфейса IRepository, с помощью средства внедрения зависимостей (рассматриваемого в главе 18).

Листинг 13.21. Конфигурирование хранилища в файле Startup.cs из папки PartyInvites

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using PartyInvites.Models;

namespace PartyInvites {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IRepository, EFRepository>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание базы данных

В оставшихся главах книги всякий раз, когда нужно продемонстрировать функциональность, требующую постоянства данных, используется средство LocalDB, которое представляет собой упрощенную версию Microsoft SQL Server. Но средство LocalDB доступно только в среде Windows, поэтому при создании приложений ASP.NET Core MVC на других платформах понадобится какая-то альтернатива. Наилучшей альтернативой LocalDB является SQLite — не нуждающаяся в конфигурировании межплатформенная система управления базами данных. Она может встраиваться в приложения, и для нее компания Microsoft включила поддержку в Entity Framework Core, которая и представляет собой уровень доступа к данным, обычно применяемый в приложениях ASP.NET Core MVC. В последующих разделах мы рассмотрим процесс добавления базы данных SQLite в проект и ее использование в качестве хранилища данных для ответов на приглашения поучаствовать в вечеринке.

Использование SQLite при разработке

Одна из причин того, что LocalDB является настолько полезным инструментом, связана с возможностью разработки с применением механизма баз данных SQL Server, который делает переход в производственную среду SQL Server простым и почти полностью лишенным рисков. SQLite — великолепная база данных, но она не очень хорошо подходит для крупномасштабных веб-приложений, а потому при развертывании приложения MVC требуется переход на другую базу данных. Изменения в конфигурации могут быть упрощены с использованием средств конфигурирования, которые будут описаны в главе 14, но приложение должно быть тщательно протестировано в испытательной среде, чтобы выявить любые отличия, привносимые производственной базой данных.

Почитайте статью по адресу `https://www.sqlite.org/whentouse.html`, если вы не уверены в том, применять ли SQLite в производственной среде. Там приведен обзор ситуаций, когда база данных SQLite может быть хорошим вариантом, а когда нет.

Важно принимать во внимание тот факт, что SQLite не поддерживает полный набор изменений схемы, которые инфраструктура Entity Framework Core способна генерировать для других баз данных. В целом это не проблема, когда SQLite используется при разработке, поскольку вы можете удалить файл базы данных и сгенерировать новый файл с чистой схемой. Тем не менее, ситуация усложняется, если вы обдумываете развертывание приложения, в котором применяется SQLite.

Если вы хотите использовать ту же самую базу данных в среде разработки и в производственной среде, тогда ознакомьтесь со списком поддерживаемых инфраструктурой Entity Framework Core баз данных по адресу `https://docs.microsoft.com/ru-ru/ef/core/providers/`. На момент написания главы список был относительно коротким, но в Microsoft объявили о поддержке баз данных, которые больше подходят для процесса разработки, чем SQLite, и могут функционировать также на платформах, отличающихся от Windows.

Добавление пакета базы данных

Пакет NuGet, содержащий инструменты командной строки, которые используются для создания и применения миграций базы данных, должен быть добавлен в проект вручную. Откроем файл `PartyInvites.csproj` и добавим в него элемент, выделенный в листинге 13.22 полужирным.

Листинг 13.22. Добавление пакета NuGet в файле `PartyInvites.csproj` из папки `PartyInvites`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
        Version="2.0.0" />
  </ItemGroup>
</Project>
```

Сохраним внесенные изменения и, находясь в папке PartyInvites, запустим команду, показанную в листинге 13.23, чтобы обеспечить загрузку и установку нового пакета.

Листинг 13.23. Установка пакета NuGet

```
dotnet restore
```

Создание и применение миграции базы данных

Процедура создания базы данных следует тому же самому процессу, который используется в Visual Studio. Чтобы создать начальную миграцию базы данных, выполним команду из листинга 13.24, находясь в папке PartyInvites.

Листинг 13.24. Создание начальной миграции базы данных

```
dotnet ef migrations add Initial
```

Инфраструктура Entity Framework Core создаст папку по имени Migrations, содержащую файлы классов C#, которые будут использоваться для настройки схемы базы данных. Чтобы применить эту миграцию базы данных, запустим в папке PartyInvites команду из листинга 13.25, которая создаст базу данных внутри папки PartyInvites.

Листинг 13.25. Применение миграции базы данных

```
dotnet ef database update
```

Среда Visual Studio Code не включает поддержку для инспектирования баз данных SQLite, но на веб-сайте http://sqlitebrowser.org можно найти великолепный инструмент с открытым кодом для Windows, macOS и Linux.

Создание контроллеров и представлений

В настоящем разделе мы добавим в приложение контроллер и представления. Начнем с создания папки PartyInvites/Controllers и добавления в нее файла класса по имени HomeController.cs с определением, показанным в листинге 13.26.

Листинг 13.26. Содержимое файла HomeController.cs из папки PartyInvites/Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) =>
      this.repository = repo;
```

```
    public ViewResult Index() {
      int hour = DateTime.Now.Hour;
      ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
      return View("MyView");
    }

    [HttpGet]
    public ViewResult RsvpForm() => View();

    [HttpPost]
    public ViewResult RsvpForm(GuestResponse guestResponse) {
      if (ModelState.IsValid) {
        repository.AddResponse(guestResponse);
        return View("Thanks", guestResponse);
      } else {
        // Имеется ошибка проверки достоверности
        return View();
      }
    }

    public ViewResult ListResponses() =>
      View(repository.Responses.Where(r => r.WillAttend == true));
  }
}
```

Для установки встроенных вспомогательных функций дескрипторов создадим папку `PartyInvites/Views` и добавим в нее файл по имени `_ViewImports.cshtml`, который содержит выражение, приведенное в листинге 13.27.

Листинг 13.27. Содержимое файла `_ViewImports.cshtml` из папки `PartyInvites/Views`

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Далее создадим папку `PartyInvites/Views/Home` и добавим в нее файл по имени `MyView.cshtml`, в котором определяется представление, выбираемое методом действия `Index()` из листинга 13.26. Поместим в него разметку, показанную в листинге 13.28.

Листинг 13.28. Содержимое файла `MyView.cshtml` из папки `PartyInvites/Views/Home`

```
@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body class="p-2">
  <div class="text-center">
```

```
    <h3>We're going to have an exciting party!</h3>
    <h4>And you are invited</h4>
    <a class="btn btn-primary" asp-action="RsvpForm">RSVP Now</a>
  </div>
</body>
</html>
```

Добавим в папку `PartyInvites/Views/Home` файл по имени `RsvpForm.cshtml` с содержимым, приведенным в листинге 13.29. Это представление предлагает HTML-форму, которая будет заполняться пользователем, чтобы принять или отклонить приглашение на вечеринку.

Листинг 13.29. Содержимое файла `RsvpForm.cshtml` из папки `PartyInvites/Views/Home`

```
@model PartyInvites.Models.GuestResponse

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RsvpForm</title>
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
  <div class="m-2">
    <div class="text-center"><h4>RSVP</h4></div>
    <form class="p-1" asp-action="RsvpForm" method="post">
      <div asp-validation-summary="All"></div>
      <div class="form-group">
        <label asp-for="Name">Your name:</label>
        <input class="form-control" asp-for="Name" />
      </div>
      <div class="form-group">
        <label asp-for="Email">Your email:</label>
        <input class="form-control" asp-for="Email" />
      </div>
      <div class="form-group">
        <label asp-for="Phone">Your phone:</label>
        <input class="form-control" asp-for="Phone" />
      </div>
      <div class="form-group">
        <label>Will you attend?</label>
        <select class="form-control" asp-for="WillAttend">
          <option value="">Choose an option</option>
          <option value="true">Yes, I'll be there</option>
          <option value="false">No, I can't come</option>
        </select>
      </div>
```

```
    <div class="text-center">
      <button class="btn btn-primary" type="submit">
        Submit RSVP
      </button>
    </div>
  </form>
 </div>
</body>
</html>
```

Следующий файл представления называется `Thanks.cshtml`, также создается в папке `PartyInvites/Views/Home` и имеет содержимое, показанное в листинге 13.30, которое отображается, когда гость отправляет свой ответ.

Листинг 13.30. Содержимое файла `Thanks.cshtml` из папки `PartyInvites/Views/Home`

```
@model PartyInvites.Models.GuestResponse

@{
  Layout = null;
}
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Thanks</title>
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body class="text-center">
  <p>
    <h1>Thank you, @Model.Name!</h1>
    @if (Model.WillAttend == true) {
      @:It's great that you're coming. The drinks are already in the fridge!
    } else {
      @:Sorry to hear that you can't make it, but thanks for letting us know.
    }
  </p>
  Click <a asp-action="ListResponses">here</a>
  to see who is coming.
</body>
</html>
```

Финальный файл представления имеет имя `ListResponses.cshtml` и подобно другим представлениям в примере добавляется в папку `PartyInvites/Views/Home`. Он отображает список ответов гостей, используя разметку из листинга 13.31.

Листинг 13.31. Содержимое файла `ListResponses.cshtml` из папки `PartyInvites/Views/Home`

```
@model IEnumerable<PartyInvites.Models.GuestResponse>
@{
  Layout = null;
}
```

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
  <title>Responses</title>
</head>
<body>
  <div class="m-1 p-1">
    <h2>Here is the list of people attending the party</h2>
    <table class="table table-sm table-striped table-bordered">
      <thead>
        <tr><th>Name</th><th>Email</th><th>Phone</th></tr>
      </thead>
      <tbody>
        @foreach (PartyInvites.Models.GuestResponse r in Model) {
          <tr><td>@r.Name</td><td>@r.Email</td><td>@r.Phone</td></tr>
        }
      </tbody>
    </table>
  </div>
</body>
</html>
```

Выполним команду `dotnet run` в папке `PartyInvites`, чтобы скомпилировать проект и запустить исполняющую среду ASP.NET Core. Увидеть в работе завершенное приложение можно, перейдя на `http://localhost:5000` (рис. 13.10).

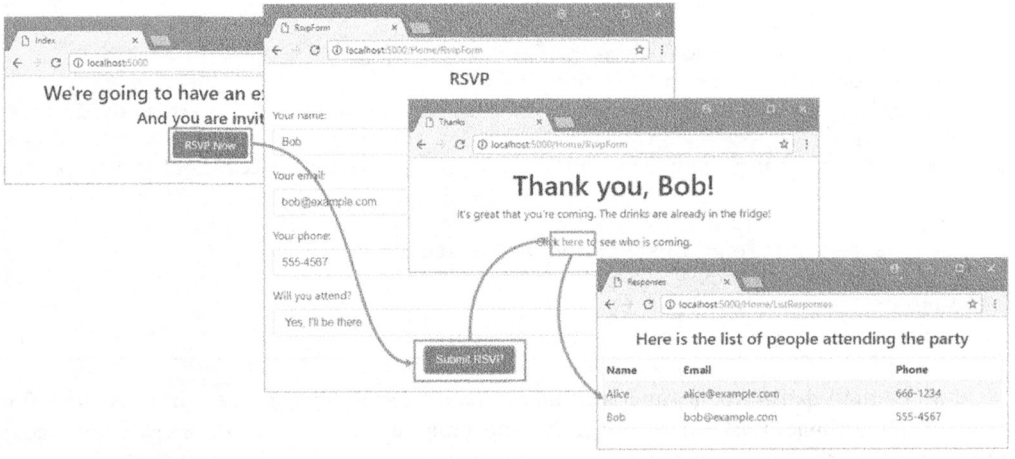


Рис. 13.10. Выполнение завершенного приложения

Модульное тестирование в Visual Studio Code

Процесс модульного тестирования в Visual Studio Code похож на аналогичный процесс в Visual Studio. Первым делом понадобится создать отдельный проект для модульного тестирования.

Создадим внутри папки InvitesProject папку по имени Tests и запустим в ней команду, представленную в листинге 13.32, которая создаст проект модульного тестирования.

Листинг 13.32. Создание проекта модульного тестирования

```
dotnet new xunit --language C# --framework netcoreapp2.0
```

Запустим в папке Tests команду из листинга 13.33, чтобы добавить ссылку на проект приложения, сделав содержащиеся в нем классы доступными для тестирования.

Листинг 13.33. Добавление ссылки на проект приложения

```
dotnet add reference ../PartyInvites/PartyInvites.csproj
```

Создание модульного теста

Модульные тесты создаются так, как было описано в главе 7. Добавим в папку Tests новый файл класса по имени HomeControllerTests.cs с содержимым, показанным в листинге 13.34.

Листинг 13.34. Содержимое файла HomeControllerTests.cs из папки Tests

```
using System;
using System.Collections.Generic;
using PartyInvites.Controllers;
using PartyInvites.Models;
using Xunit;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
namespace Tests {
  public class HomeControllerTests {
    [Fact]
    public void ListActionFiltersNonAttendees() {
      // Организация
      HomeController controller = new HomeController(new FakeRepository());
      // Действие
      ViewResult result = controller.ListResponses();
      // Утверждение
      Assert.Equal(2, (result.Model as IEnumerable<GuestResponse>).Count());
    }
  }
  class FakeRepository : IRepository {
    public IEnumerable<GuestResponse> Responses =>
      new List<GuestResponse> {
        new GuestResponse { Name = "Bob", WillAttend = true },
        new GuestResponse { Name = "Alice", WillAttend = true },
        new GuestResponse { Name = "Joe", WillAttend = false }
      };
    public void AddResponse(GuestResponse response) {
      throw new NotImplementedException();
    }
  }
}
```

Мы создали стандартный тест xUnit, который проверяет действие `ListResponses` в контроллере `Home` и корректно фильтрует в хранилище объекты `GuestResponse` со значением свойства `WillAttend`, равным `false`.

Прогон тестов

Чтобы прогнать модульные тесты в проекте, запустим в папке `Tests` команду, приведенную в листинге 13.35.

Листинг 13.35. Прогон модульных тестов

```
dotnet test
```

Все тесты в проекте выполнятся, в результате чего отобразится вывод вроде показанного ниже:

```
Starting test execution, please wait...
[xUnit.net 00:00:00.6731479]   Discovering: Tests
[xUnit.net 00:00:00.7900132]   Discovered:  Tests
[xUnit.net 00:00:00.8432715]   Starting:    Tests
[xUnit.net 00:00:00.9967614]   Finished:    Tests

Total tests: 2. Passed: 2. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.6974 Seconds
```

В выводе присутствуют результаты двух тестов, поскольку шаблон проекта включает файл по имени `UnitTest1.cs`, который содержит пустой модульный тест. Упомянутый файл можно удалить, как демонстрировалось в главе 7.

Резюме

В главе был предложен краткий обзор работы со средой Visual Studio Code — легковесным инструментом разработки, который поддерживает создание приложений ASP.NET Core MVC в Windows, macOS и Linux. Среда Visual Studio Code пока еще не является заменой полного продукта Visual Studio, но предоставляет основные средства, необходимые при построении приложений MVC, и расширяется компанией Microsoft в ежемесячных выпусках.

Итак, первая часть книги завершена. Во второй части мы начнем погружение в детали и посмотрим, как работают средства, которые использовались для создания примера приложения.

ЧАСТЬ II

Подробные сведения об инфраструктуре ASP.NET Core MVC 2

К настоящему моменту вы уже знаете, для чего существует инфраструктура .NET Core MVC, а также понимаете ее архитектуру и лежащие в основе проектные цели. Вы получили эти знания благодаря построению реалистичного приложения электронной коммерции. Наступило время заняться исследованием внутреннего устройства инфраструктуры.

Во второй части книги мы погрузимся в детали, начав с изучения структуры приложения ASP.NET Core MVC и способа обработки запросов. Затем мы сосредоточим внимание на индивидуальных средствах, таких как маршрутизация, контроллеры и действия, система представлений MVC и вспомогательных функций дескрипторов, а также особенности работы с моделями предметной области.

ГЛАВА 14

Конфигурирование приложений

Тема конфигурации может показаться неинтересной, но она открывает много аспектов, связанных с функционированием приложений MVC и обработкой HTTP-запросов. Вы не должны поддаваться соблазну пропустить текущую главу. Вы обязаны выделить время на изучение способа, которым система конфигурации придает форму веб-приложениям MVC. Данная тема образует прочный фундамент для понимания материала последующих глав.

В главе объясняется, как конфигурировать приложения MVC, и демонстрируется, каким образом инфраструктура MVC опирается на средства, предоставляемые платформой ASP.NET Core. В табл. 14.1 приведена сводка, позволяющая поместить конфигурирование приложений в контекст.

Таблица 14.1. Помещение системы конфигурации в контекст

Вопрос	Ответ
Что это такое?	Классы `Program` и `Startup` и файлы JSON используются для конфигурирования работы приложения, а также указания пакетов, от которых оно зависит
Чем она полезна?	Система конфигурации позволяет подстраивать приложения под их среды и управлять зависимостями от пакетов
Как она используется?	Самым важным компонентом является класс `Startup`, который применяется для создания служб (объектов, предоставляющих общую функциональность повсюду в приложении) и компонентов промежуточного программного обеспечения (ПО), используемых для обработки HTTP-запросов
Существуют ли какие-то скрытые ловушки или ограничения?	В сложных приложениях конфигурация может стать трудной в управлении. В разделе "Работа со сложными конфигурациями" далее в главе описаны средства ASP.NET, предназначенные для решения такой проблемы
Существуют ли альтернативы?	Нет. Система конфигурации — это неотъемлемая часть ASP.NET и средство настройки приложений MVC

В табл. 14.2 представлена сводка по главе.

Таблица 14.2. Сводка по главе

Задача	Решение	Листинг
Добавление функциональности в приложение	Добавьте пакеты NuGet в файл `.csproj`	14.5–14.8
Управление инициализацией приложения ASP.NET	Используйте класс `Program`	14.9–14.11
Конфигурирование приложения	Применяйте методы `ConfigureServices()` и `Configure()` класса `Startup`	14.12, 14.13
Создание общей функциональности	Используйте метод `ConfigureServices()` для создания служб	14.14–14.16
Генерация ответов с содержимым	Создайте промежуточное ПО для генерации содержимого	14.17–14.19
Предотвращение прохождения запросов через конвейер запросов	Создайте промежуточное ПО для обхода	14.20, 14.21
Редактирование запроса перед его обработкой другими компонентами промежуточного ПО	Создайте промежуточное ПО для редактирования запросов	14.22–14.24
Редактирование ответа, который был обработан другими компонентами промежуточного ПО	Создайте промежуточное ПО для редактирования ответов	14.25, 14.26
Настройка функциональности MVC	Применяйте метод `UseMvc()` или `UseMvcWithDefaultRoute()`	14.27
Изменение конфигурации приложения для разных сред	Используйте службу среды размещения	14.28
Обработка ошибок в приложении	Используйте промежуточное ПО для обработки ошибок в среде разработки или производственной среде	14.29, 14.30
Управление несколькими браузерами во время разработки	Применяйте средство Browser Link (Ссылка на браузер)	14.31
Включение файлов изображений, JavaScript и CSS	Включите промежуточное ПО для статического содержимого	14.32
Отделение данных конфигурации от кода C#	Создайте внешние источники конфигурации, такие как файлы JSON	14.33–14.35
Регистрация данных приложения в журнале	Применяйте промежуточное ПО для регистрации в журнале	14.36–14.38
Подготовка системы внедрения зависимостей для использования с Entity Framework Core	Отключите проверку области действия	14.39
Конфигурирование служб MVC	Используйте средства параметров конфигурации	14.40
Конфигурирование сложных приложений	Применяйте несколько внешних файлов или классов	14.41–14.45

Подготовка проекта для примера

В текущей главе мы создадим новый проект по имени ConfiguringApps с использованием шаблона Empty (Пустой). Мы будем конфигурировать приложение позже в главе, но есть несколько базовых средств, которые необходимо поместить на свои места в плане подготовки к предстоящим изменениям.

Мы собираемся применять Bootstrap для стилизации HTML-содержимого, поэтому создадим файл bower.json, используя шаблон элемента Bower Configuration File (Файл конфигурации Bower), и добавим в него пакет, как показано в листинге 14.1.

Листинг 14.1. Добавление Bootstrap в файле bower.json из папки ConfiguringApps

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Создадим папку Controllers и поместим в нее файл класса по имени HomeController.cs с определением контроллера, приведенным в листинге 14.2.

Листинг 14.2. Содержимое файла HomeController.cs из папки Controllers

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

namespace ConfiguringApps.Controllers {

  public class HomeController : Controller {

    public ViewResult Index()
      => View(new Dictionary<string, string> {
        ["Message"] = "This is the Index action"
      });
  }
}
```

Создадим папку Views/Home и добавим в нее файл представления по имени Index.cshtml с содержимым, показанным в листинге 14.3.

Листинг 14.3. Содержимое файла Index.cshtml из папки Views/Home

```
@model Dictionary<string, string>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
  <title>Result</title>
</head>
```

```
<body class="p-1">
  <table class="table table-condensed table-bordered table-striped">
    @foreach (var kvp in Model) {
      <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
    }
  </table>
</body>
</html>
```

При выборе CSS-файлов Bootstrap элемент `link` в представлении полагается на встроенную вспомогательную функцию дескриптора. Чтобы включить встроенные вспомогательные функции дескрипторов, создадим в папке `Views` файл `_ViewImports.cshtml` с применением шаблона элемента MVC View Imports Page (Страница импортирования представлений MVC) и поместим в него выражение из листинга 14.4.

Листинг 14.4. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

В результате запуска приложения отобразится сообщение, представленное на рис. 14.1.

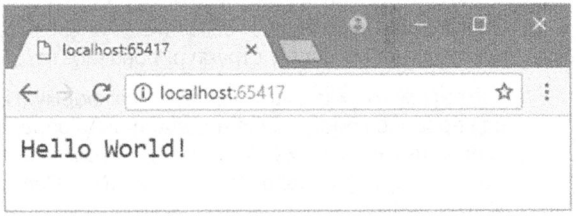


Рис. 14.1. Выполнение примера приложения

Конфигурирование проекта

Самым важным конфигурационным файлом является `<имя_проекта>.csproj`, который заменяет файл `project.json`, используемый в более ранних версиях ASP.NET Core. Такой файл, который в проекте примера называется `ConfiguringApps.csproj`, среда Visual Studio скрывает и разрешает к нему доступ за счет щелчка правой кнопкой мыши на элементе проекта в окне Solution Explorer и выбора в контекстном меню пункта Edit ConfiguringApps.csproj (Редактировать `ConfiguringApps.csproj`). В листинге 14.5 приведено начальное содержимое файла `ConfiguringApps.csproj`, которое было создано Visual Studio как часть шаблона проекта Empty.

Листинг 14.5. Содержимое файла `ConfiguringApps.csproj` из папки `ConfiguringApps`

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
```

```
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>

</Project>
```

Файл `.csproj` применяется для конфигурирования инструмента MSBuild, который используется при построении проектов .NET. Конфигурирование производится с помощью элементов XML, а в табл. 14.3 описаны элементы, находящиеся в стандартном конфигурационном файле. В более поздних примерах будут применяться другие элементы конфигурации, но представленных в табл. 14.3 элементов вполне достаточно, чтобы начать работу над проектом ASP.NET Core MVC.

Таблица 14.3. Элементы конфигурации XML в стандартном файле `.csproj`

Элемент	Описание
`Project`	Это корневой элемент, который обозначает конфигурационный файл MSBuild. Атрибут `Sdk` устанавливается в `Microsoft.NET.Sdk.Web`, чтобы предоставить набор неявно импортируемых пакетов, требующихся для построения проекта
`PropertyGroup`	Этот элемент группирует связанные конфигурационные свойства в целях придания файлу структурированности
`TargetFramework`	Этот элемент указывает целевую платформу .NET Framework для процесса построения и должен быть определен внутри элемента `PropertyGroup`. Стандартным значением является `netcoreapp2.0`, направляющее на .NET Core 2.0
`ItemGroup`	Этот элемент группирует связанные конфигурационные элементы в целях придания файлу структурированности
`Folder`	Этот элемент сообщает MSBuild о том, как иметь дело с той или иной папкой в проекте. Данный элемент в листинге 14.5 указывает инструменту MSBuild о необходимости включения папки `wwwroot`, когда приложение публикуется
`PackageReference`	Этот элемент используется для указания зависимости от пакета NuGet, который идентифицируется через атрибуты `Include` и `Version`. Пакет `Microsoft.AspNetCore.All` применяется для обеспечения доступа ко всем индивидуальным пакетам, которые предоставляют функциональность ASP.NET Core и MVC Framework

Добавление пакетов в проект

Самая важная роль файла `.csproj` — составить список пакетов, от которых зависит проект. Когда среда Visual Studio обнаруживает изменение в файле `.csproj`, она инспектирует список пакетов, загружает любые вновь добавленные пакеты и удаляет любые пакеты, которые больше не нужны.

С выпуском ASP.NET Core 2 вся базовая функциональность, требующаяся для ASP.NET Core MVC, MVC Framework и Entity Framework Core, включена в мета-пакет

`Microsoft.AspNetCore.All` — удобная возможность, которая позволяет избежать необходимости начинать разработку приложения с добавления в проект длинного списка пакетов NuGet.

Но даже при таких условиях по-прежнему придется добавлять пакеты NuGet для сторонних или расширенных функциональных возможностей. Существует три способа добавления пакетов в проект. Первый способ — выбор пункта меню Tools⇨NuGet Package Manager⇨Manage NuGet Packages for Solution (Сервис⇨Диспетчер пакетов NuGet⇨Управлять пакетами NuGet для решения), что позволит управлять пакетами NuGet через легкий в использовании интерфейс. Для новичков в мире разработки приложений .NET такой подход будет наилучшим, потому что он снижает вероятность ошибки при выборе пакета.

Например, чтобы добавить пакет `System.Net.Http`, который обеспечивает поддержку отправки (а не получения) запросов HTTP, можно перейти на вкладку Browse (Обзор) окна диспетчера пакетов NuGet, произвести поиск по имени и увидеть полный список доступных версий, в том числе предварительных версий (рис. 14.2).

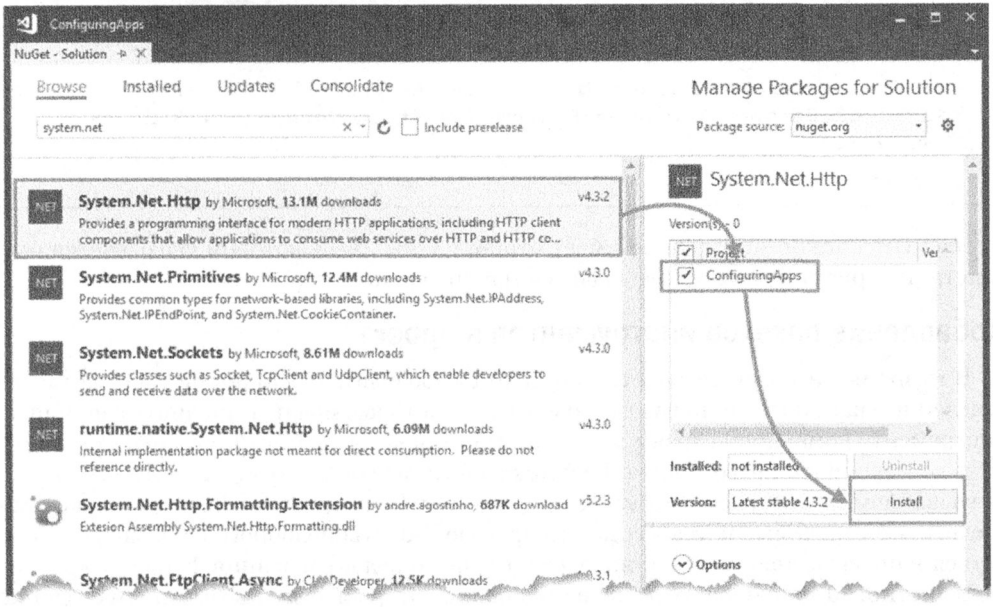


Рис. 14.2. Добавление пакета с применением диспетчера пакетов NuGet

Далее необходимо выбрать пакет интересующей версии, указать проект, для которого требуется этот пакет, и щелкнуть на кнопке Install (Установить). Среда Visual Studio загрузит пакет и обновит файл `.csproj`.

Добавлять пакеты можно также с использованием командной строки, хотя в таком случае нужно знать имя пакета (и в идеале его версию). В листинге 14.6 показана команда, которую пришлось бы выполнить, находясь в папке `ConfiguringApps`, чтобы добавить в проект пакет `System.Net.Http`.

Листинг 14.6. Добавление пакета в проект

```
dotnet add package System.Net.Http --version 4.3.2
```

Как диспетчер пакетов NuGet, так и команда `dotnet add package` добавляют элементы `PackageReference` в файл `.csproj`. При желании пакеты можно добавлять за счет редактирования конфигурационного файла, вручную добавляя в него элементы `PackageReference`. Подход является самым прямым, но требует внимательности во избежание неправильного набора имен пакетов или указания для них несуществующих номеров версий. В листинге 14.7 демонстрируется добавление пакета `System.Net.Http` в файле `.csproj`.

Листинг 14.7. Добавление пакета в файле `ConfiguringApps.csproj` из папки `ConfiguringApps`

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version="4.3.2" />
  </ItemGroup>

</Project>
```

Элемент `PackageReference` имеет атрибут `Include`, в котором указывается имя пакета, и атрибут `Version`, где задается номер версии.

Добавление пакетов инструментов в проект

В то время как обычные пакеты допускается добавлять различными способами, определенные пакеты расширяют диапазон задач, которые можно выполнять с помощью инструмента командной строки `dotnet`, и такие пакеты требуют применения в файле `.csproj` элемента другого вида. Речь идет об элементе `DotNetCliToolReference`, указываемом вместо элемента `PackageReference`, который используется пакетами средств, применяемых приложением напрямую. Пакеты подобного рода могут добавляться в проекты только путем непосредственного редактирования файла `.csproj`.

В листинге 14.8 к проекту добавляется пакет, который делает возможным создание и применение миграций базы данных с использованием команд `dotnet ef`, как было показано в первой части книги.

Листинг 14.8. Добавление пакетов инструментов в файле `ConfiguringApps.csproj` из папки `ConfiguringApps`

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
```

```
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version="4.3.2" />
    <DotNetCliToolReference
      Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
      Version="2.0.0 " />
  </ItemGroup>
</Project>
```

При добавлении пакетов инструментов в проект можно либо поместить элементы `DotNetCliToolReference` в тот же самый элемент `ItemGroup`, где находятся обычные элементы `PackageReference`, что и было сделано в листинге 14.8, либо создать отдельный элемент `ItemGroup`. После сохранения изменений, внесенных в файл `.csproj`, среда Visual Studio загрузит и установит пакеты и задействует их для конфигурирования инструмента командной строки `dotnet`.

Класс Program

Класс `Program` определяется в файле по имени `Program.cs` и предоставляет точку входа для запуска приложения, снабжая инфраструктуру .NET методом `Main()`, который может быть выполнен для конфигурирования среды размещения и выбора класса, завершающего конфигурирование приложения ASP.NET Core. Стандартного содержимого класса `Program` вполне достаточно для разработки большинства проектов; в листинге 14.9 приведен стандартный код, который среда Visual Studio добавляет в проекты.

Листинг 14.9. Стандартное содержимое файла `Program.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace ConfiguringApps {
  public class Program {

    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
  }
}
```

Метод `Main()` обеспечивает точку входа, которую должны предоставлять все приложения .NET, чтобы иметь возможность выполняться исполняющей средой. Внутри метода `Main()` в классе `Program` вызывается метод `BuildWebHost()`, который несет ответственность за конфигурирование ASP.NET Core.

Для конфигурирования ASP.NET Core метод `BuildWebHost()` применяет статические методы, определенные в классе `WebHost`. С выходом ASP.NET Core 2 конфигурирование упростилось за счет появления метода `CreateDefaultBuilder()`, который конфигурирует ASP.NET Core с использованием настроек, подходящих большинству проектов. Метод `UseStartup()` вызывается с целью идентификации класса, который будет предоставлять конфигурацию, специфичную для приложения; по соглашению применяется класс по имени `Startup`, рассматриваемый позже в главе. Метод `Build()` обрабатывает все настройки конфигурации и создает объект, реализующий интерфейс `IWebHost` и возвращаемый методу `Main()`, на котором вызывается метод `Run()`, чтобы начать обработку HTTP-запросов.

Анализ деталей конфигурации

Метод `CreateDefaultBuilder()` является удобной отправной точкой конфигурации ASP.NET Core, но он скрывает много важных деталей, что может стать проблемой, если необходимо изменить способ, которым конфигурируется приложение. В листинге 14.10 вызов метода `CreateDefaultBuilder()` заменен отдельными операторами, создающими стандартную конфигурацию.

Листинг 14.10. Детальные операторы конфигурирования в файле `Program.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
  public class Program {

    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) {
      return new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) => {
          var env = hostingContext.HostingEnvironment;
          config.AddJsonFile("appsettings.json", optional: true,
                             reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                             optional: true, reloadOnChange: true);
```

```
        if (env.IsDevelopment()) {
          var appAssembly =
            Assembly.Load(new AssemblyName(env.ApplicationName));
          if (appAssembly != null) {
            config.AddUserSecrets(appAssembly, optional: true);
          }
        }
        config.AddEnvironmentVariables();
        if (args != null) {
          config.AddCommandLine(args);
        }
      })
      .ConfigureLogging((hostingContext, logging) => {
        logging.AddConfiguration(
          hostingContext.Configuration.GetSection("Logging"));
        logging.AddConsole();
        logging.AddDebug();
      })
      .UseIISIntegration()
      .UseDefaultServiceProvider((context, options) => {
        options.ValidateScopes =
          context.HostingEnvironment.IsDevelopment();
      })
      .UseStartup<Startup>()
      .Build();
  }
}
}
```

Все методы конфигурирования, добавленные в метод `BuildWebHost()`, кратко описаны в табл. 14.4.

Таблица 14.4. Стандартные методы конфигурирования ASP.NET Core

Имя	Описание
`UseKestrel()`	Этот метод конфигурирует веб-сервер Kestrel, как объясняется ниже во врезке "Использование Kestrel напрямую"
`UseContentRoot()`	Этот метод конфигурирует корневой каталог для приложения, который применяется для загрузки файлов конфигурации и доставки статического содержимого, такого как файлы изображений, JavaScript и CSS
`ConfigureAppConfiguration()`	Этот метод используется для подготовки конфигурационных данных приложения, как описано в разделе "Конфигурирование приложения" далее в главе
`AddUserSecrets()`	Этот метод применяется для хранения конфиденциальных данных за пределами файлов кода; подробные сведения доступны по ссылке `https://docs.microsoft.com/ru-ru/aspnet/core/security/app-secrets?view=aspnetcore-2.1&tabs=windows`. Средство несколько неуклюжее и потому оно в книге не используется

Окончание табл. 14.4

Имя	Описание
`ConfigureLogging()`	Этот метод позволяет сконфигурировать регистрацию в журнале для приложения, как объясняется в разделе "Конфигурирование регистрации в журнале" позже в главе
`UseIISIntegration()`	Этот метод включает интеграцию с IIS и IIS Express
`UseDefaultServiceProvider()`	Этот метод применяется для конфигурирования внедрения зависимостей, как описано в разделе "Конфигурирование внедрения зависимостей" далее в главе
`UseStartup()`	Этот метод указывает класс, который будет использоваться для конфигурирования ASP.NET, как описано в разделе "Класс `Startup`" далее в главе

Позже в главе будут объясняться более сложные операторы из листинга 14.10, а пока мы удалим некоторые операторы конфигурирования, чтобы оставить только базовую конфигурацию (листинг 14.11).

Листинг 14.11. Упрощение конфигурации в файле `Program.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
  public class Program {

    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }
    public static IWebHost BuildWebHost(string[] args) {
      return new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();
    }
  }
}
```

Показанные в листинге 14.11 операторы обеспечивают базовую конфигурацию, которая будет работать для большинства приложений ASP.NET Core MVC. Другие операторы будут добавлены снова по мере объяснения функциональных средств, к которым они относятся.

Использование Kestrel напрямую

Kestrel — это межплатформенный веб-сервер, спроектированный для выполнения приложений ASP.NET Core. Он задействуется автоматически, когда запускается приложение ASP.NET Core с применением IIS Express (сервер, предоставляемый средой Visual Studio для использования на стадии разработки) или полной версии IIS, которая была традиционной веб-платформой для приложений .NET.

При желании веб-сервер Kestrel также разрешено запускать напрямую, что означает возможность выполнения приложений ASP.NET Core MVC на любой из поддерживаемых платформ, обходя ограничение только операционной системой Windows, которое накладывает IIS. Существуют два способа запуска приложения с применением Kestrel. Первый из них — щелчок на стрелке рядом с правой гранью кнопки IIS Express в панели инструментов Visual Studio и выбор элемента, который соответствует имени проекта. В результате откроется новое окно командной строки, и приложение запустится с использованием Kestrel.

Той же цели можно достичь, открыв собственное окно командной строки, перейдя в папку с файлами конфигурации приложения (ту, которая содержит файл `.csproj`) и запустив следующую команду:

```
dotnet run
```

По умолчанию веб-сервер Kestrel начинает прослушивать порт 5000 на предмет поступления HTTP-запросов. Если в проекте присутствует файл `Properties/launchSettings.json`, тогда настройки порта HTTP и среды для приложения будут читаться из него.

Класс `Startup`

Класс `Program` отвечает за быстрое начало функционирования приложения, но более важная работа по конфигурированию делегируется через метод `UseStartup()`:

```
...
.UseStartup<Startup>()
...
```

При идентификации класса, который будет конфигурировать ASP.NET Core, метод `UseStartup()` полагается на параметр типа. По соглашению этот класс имеет имя `Startup`, которое применяется шаблонами проектов ASP.NET Core MVC, в том числе шаблоном Empty, используемым для создания проекта примера в текущей главе.

Изучение работы класса `Startup` позволяет уловить суть способа обработки HTTP-запросов и интеграции инфраструктуры MVC с остальными частями платформы ASP.NET Core.

В настоящем разделе мы начнем с простейшего из возможных классов `Startup` и добавим средства для демонстрации влияния различных конфигурационных параметров, получив в итоге конфигурацию, которая будет подходящей в большинстве проектов MVC. В качестве отправной точки в листинге 14.12 показан класс `Startup`, добавляемый средой Visual Studio к проектам Empty, который настраивает всего лишь функциональность, необходимую ASP.NET Core для обработки HTTP-запросов.

Листинг 14.12. Начальное содержимое файла `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {

      if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
      }
      app.Run(async (context) => {
        await context.Response.WriteAsync("Hello World!");
      });
    }
  }
}
```

В классе `Startup` определены два метода, `ConfigureServices()` и `Configure()`, которые настраивают совместно используемые средства, требующиеся приложению, и указывают ASP.NET о том, как они должны применяться.

Когда приложение запускается, инфраструктура ASP.NET Core создает новый экземпляр класса `Startup` и вызывает его метод `ConfigureServices()`, так что приложение может создать свои *службы*. Как объясняется в разделе "Службы ASP.NET" далее в главе, службы — это объекты, которые предоставляют функциональность другим частям приложения. Приведенное описание не отличается особой строгостью по той причине, что службы могут использоваться для снабжения практически любой функциональностью.

После того как службы созданы, ASP.NET вызывает метод `Configure()`. Целью метода `Configure()` является настройка *конвейера запросов*, который представляет собой набор компонентов (называемых *промежуточным программным обеспечением*), используемых для обработки входящих HTTP-запросов и генерации ответов на них. В разделе "Промежуточное программное обеспечение ASP.NET" далее в главе будут даны объяснения, как работает конвейер запросов, и продемонстрировано создание компонентов промежуточного ПО. На рис. 14.3 иллюстрируется взаимодействие ASP.NET с классом `Startup`.

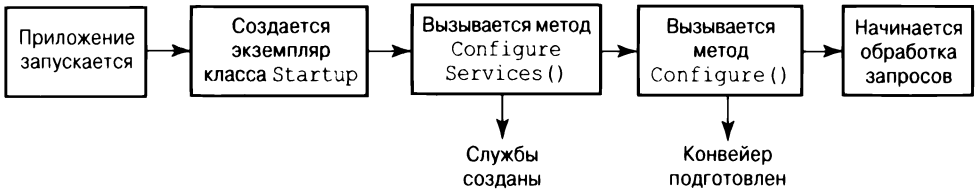


Рис. 14.3. Применение класса `Startup` инфраструктурой ASP.NET для конфигурирования приложения

Класс `Startup`, который для всех запросов просто возвращает одно и то же сообщение `"Hello World!"`, не особенно полезен, поэтому перед подробным обсуждением методов класса имеет смысл забежать чуть вперед и включить MVC (листинг 14.13).

Листинг 14.13. Включение MVC в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Благодаря таким добавлениям (которые объясняются в последующих разделах) в нашем распоряжении оказывается достаточный объем инфраструктуры, чтобы обрабатывать HTTP-запросы и генерировать ответы с использованием контроллеров и представлений. На рис. 14.4 показан результат запуска приложения.

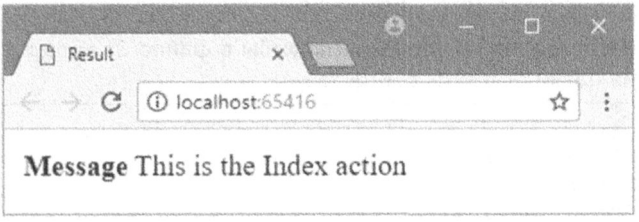


Рис. 14.4. Результат включения MVC

Обратите внимание, что содержимое не стилизовано. Минимальная конфигурация в листинге 14.13 не предлагает какую-либо поддержку для обслуживания статического содержимого, такого как таблицы стилей CSS и файлы JavaScript. В итоге элемент `link` в HTML-разметке, визуализируемой представлением `Index.cshtml`, инициирует запрос к таблице стилей CSS из Bootstrap, который приложение не в состоянии обработать, что предотвращает получение требуемой стилевой информации. Проблема будет устранена в разделе "Добавление оставшихся компонентов промежуточного программного обеспечения" далее в главе.

Службы ASP.NET

Инфраструктура ASP.NET Core вызывает метод `Startup.ConfigureServices()`, так что приложение может настроить требуемые службы. Термин "служба" относится к любому объекту, который предоставляет функциональность другим частям приложения. Как уже отмечалось, такое описание не является строгим, поскольку службы способны делать для приложения все что угодно. В качестве примера создадим в проекте папку `Infrastructure` и добавим в нее файл класса по имени `UptimeService.cs` с определением, приведенным в листинге 14.14.

Листинг 14.14. Содержимое файла `UptimeService.cs` из папки `Infrastructure`

```
using System.Diagnostics;

namespace ConfiguringApps.Infrastructure {

  public class UptimeService {
    private Stopwatch timer;

    public UptimeService() {
      timer = Stopwatch.StartNew();
    }

    public long Uptime => timer.ElapsedMilliseconds;
  }
}
```

Когда создается экземпляр класса `UptimeService`, его конструктор запускает таймер, который отслеживает, сколько времени выполнялось приложение. Это хороший пример службы, потому что класс `UptimeService` обеспечивает функциональность, которая может применяться в остальных частях приложения и создается при запуске приложения.

Службы ASP.NET регистрируются с использованием метода `ConfigureServices()` класса `Startup`; в листинге 14.15 показано, как зарегистрировать класс `UptimeService`.

Листинг 14.15. Регистрация специальной службы в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Метод `ConfigureServices()` получает в качестве аргумента объект, который реализует интерфейс `IServiceCollection`. Службы регистрируются с помощью расширяющего метода, вызываемого на интерфейсе `IServiceCollection`, которому указываются разнообразные конфигурационные параметры. Доступные параметры для создания служб будут описаны в главе 18, а пока мы применяем метод `AddSingleton()`, что означает совместное использование единственного объекта `UptimeService` по всему приложению.

Службы тесно связаны со средством под названием *внедрение зависимостей*, которое позволяет компонентам вроде контроллеров легко получать службы и подробно рассматривается в главе 18. Получить доступ к службам, зарегистрированным в методе `Startup.ConfigureServices()`, можно за счет создания конструктора, который принимает аргумент с требуемым типом службы. В листинге 14.16 показан конструктор, добавленный в контроллер `Home` для доступа к совместно используемому объекту `UptimeService`, который был создан в листинге 14.15. Кроме того, обновлен метод действия `Index()` контроллера, так что он включает в генерируемые данные представления значение свойства `Update` службы.

Листинг 14.16. Доступ к службе в файле `HomeController.cs` из папки `Controllers`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;
namespace ConfiguringApps.Controllers {
  public class HomeController : Controller {
    private UptimeService uptime;
    public HomeController(UptimeService up) => uptime = up;
    public ViewResult Index()
      => View(new Dictionary<string, string> {
          ["Message"] = "This is the Index action",
          ["Uptime"] = $"{uptime.Uptime}ms"
      });
  }
}
```

Когда инфраструктуре MVC необходим экземпляр класса контроллера `Home` для обработки HTTP-запроса, она инспектирует конструктор `HomeController` и обнаруживает, что ему требуется объект `UptimeService`. Затем MVC инспектирует набор служб, которые были сконфигурированы в классе `Startup`, выясняет, что служба `UptimeService` сконфигурирована так, чтобы для всех запросов применялся единственный объект `UptimeService`, и передает этот объект в качестве аргумента конструктору при создании экземпляра `HomeController`.

Службы могут регистрироваться и потребляться более сложными способами, но рассмотренный пример продемонстрировал главную идею, лежащую в основе служб, и показал, как определение службы в классе `Startup` позволяет определять функциональность или данные, которые используются по всему приложению.

В результате запуска приложения и запроса стандартного URL появится ответ, который включает количество миллисекунд, прошедших с момента старта приложения (рис. 14.5). Число миллисекунд получается из объекта `UptimeService`, созданного в классе `Startup`. (Строго говоря, отображаемое значение соответствует времени, прошедшему с момента создания объекта службы `UptimeService`, но оно довольно близко к моменту запуска приложения, чтобы для целей данной главы разница не играла никакой роли.)

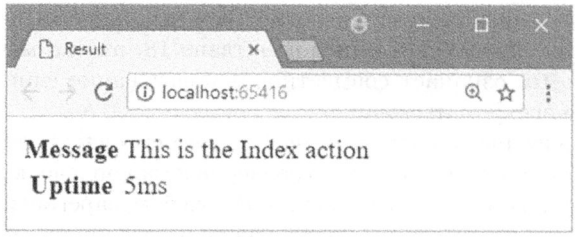


Рис. 14.5. Потребление простой службы

Каждый раз, когда поступает запрос к стандартному URL, инфраструктура MVC создает новый объект `HomeController` и предоставляет ему разделяемый объект `UptimeService` в виде аргумента конструктора. Это позволяет контроллеру `Home` получать доступ к значению времени выполнения приложения, не заботясь о том, каким образом такая информация предоставлена или реализована.

Службы MVC

Пакет с уровнем сложности, как у MVC, имеет дело со многими службами; одни из них предназначены для внутреннего потребления, а другие предлагают функциональность разработчикам. Пакеты определяют расширяющие методы, которые настраивают все требующиеся им службы в единственном вызове метода. Для MVC такой метод имеет имя `AddMvc()` и является одним из двух методов, вызовы которых были добавлены в класс `Startup`, чтобы обеспечить работу MVC:

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddSingleton<UptimeService>();
  services.AddMvc();
}
...
```

Метод `AddMvc()` настраивает каждую службу, в которой нуждается MVC, не загромождая метод `ConfigureServices()` гигантским списком индивидуальных служб.

На заметку! Средство IntelliSense в Visual Studio будет отображать длинный список других расширяющих методов, которые можно вызывать на объекте реализации `IServiceCollection` в методе `ConfigureServices()`. Некоторые из этих методов, такие как `AddSingleton()` и `AddScoped()`, применяются для регистрации служб различными способами. Другие методы, подобные `AddRouting()` или `AddCors()`, добавляют отдельные службы, которые уже использовались методом `AddMvc()`. В результате для большинства приложений метод `ConfigureServices()` содержит небольшое число специальных служб, вызов метода `AddMvc()` и необязательно операторы конфигурирования встроенных служб, которые описаны в разделе "Конфигурирование служб MVC" далее в главе.

Промежуточное программное обеспечение ASP.NET

В ASP.NET Core термин *промежуточное ПО* используется в отношении компонентов, которые объединяются с целью формирования *конвейера запросов*. Конвейер запросов организован аналогично цепочке; когда поступает новый запрос, он передается первому компоненту промежуточного ПО в цепочке. Компонент инспектирует запрос и решает, обработать его и сгенерировать ответ или передать следующему компоненту в цепочке. После того как запрос обработан, ответ, который будет возвращен клиенту, передается по цепочке обратно, что позволяет всем предшествующим компонентам инспектировать или модифицировать его.

Работа компонентов промежуточного ПО может выглядеть несколько странной, но она допускает большую гибкость в том, как приложения собираются вместе. Понимание того, каким образом применение промежуточного ПО придает форму приложению, может оказаться важным, особенно если вы получаете не те ответы, которые ожидали. Чтобы выяснить, как функционирует система промежуточного ПО, мы создадим несколько специальных компонентов, которые продемонстрируют каждый из имеющихся четырех типов промежуточного ПО.

Создание промежуточного ПО для генерации содержимого

Самый важный тип промежуточного ПО генерирует содержимое для клиентов, и именно к данной категории принадлежит MVC. Чтобы создать компонент промежуточного ПО для генерации содержимого, не имея дела со сложностью MVC, добавим в папку `Infrastructure` файл класса по имени `ContentMiddleware.cs` с определением, приведенным в листинге 14.17.

Листинг 14.17. Содержимое файла `ContentMiddleware.cs` из папки `Infrastructure`

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

  public class ContentMiddleware {
    private RequestDelegate nextDelegate;

    public ContentMiddleware(RequestDelegate next) => nextDelegate = next;

    public async Task Invoke(HttpContext httpContext) {
      if (httpContext.Request.Path.ToString().ToLower() == "/middleware")
{
        await httpContext.Response.WriteAsync(
          "This is from the content middleware", Encoding.UTF8);
      } else {
        await nextDelegate.Invoke(httpContext);
      }
    }
  }
}
```

Компоненты промежуточного ПО не реализуют какой-либо интерфейс и не являются унаследованными от какого-то общего базового класса. Взамен они определяют

конструктор, принимающий объект RequestDelegate, и метод Invoke(). Объект RequestDelegate представляет следующий компонент промежуточного ПО в цепочке, а метод Invoke() вызывается, когда ASP.NET получает HTTP-запрос.

Информация о запросе и ответе HTTP, которая будет возвращена клиенту, предоставляется методу Invoke() через аргумент HttpContext. Класс HttpContext и его свойства рассматриваются в главе 17, а пока достаточно знать, что метод Invoke() в листинге 14.17 инспектирует HTTP-запрос и проверяет, был ли он послан на URL вида /middleware. Если это так, тогда клиенту отправляется простой текстовый ответ; если использовался другой URL, то запрос перенаправляется следующему компоненту в цепочке.

Конвейер запросов настраивается внутри метода Configure() класса Startup. В листинге 14.18 из примера приложения удалены методы MVC и применен класс ContentMiddleware в качестве единственного компонента в конвейере.

Листинг 14.18. Использование специального компонента промежуточного ПО в файле Startup.cs из папки ConfiguringApps

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseMiddleware<ContentMiddleware>();
    }
  }
}
```

Специальные компоненты промежуточного ПО регистрируются с помощью расширяющего метода UseMiddleware() внутри метода Configure(). Метод UseMiddleware() применяет параметр типа для указания класса промежуточного ПО. Именно так инфраструктура ASP.NET Core может построить список всех компонентов промежуточного ПО, которые собирается использовать, и затем создать их экземпляры для формирования цепочки. На рис. 14.6 показан результат запуска приложения и запроса URL вида /middleware.

На рис. 14.7 изображен конвейер промежуточного ПО, созданный с применением класса ContentMiddleware. Когда инфраструктура ASP.NET Core получает HTTP-запрос, она передает его единственному компоненту промежуточного ПО, зарегистрированному в классе Startup. Если URL является /middleware, то компонент генерирует результат, который возвращается ASP.NET Core и отправляется клиенту.

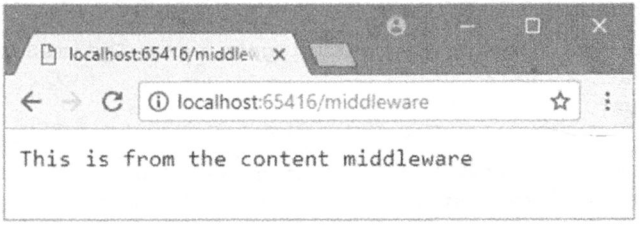


Рис. 14.6. Генерация содержимого из специального компонента промежуточного ПО

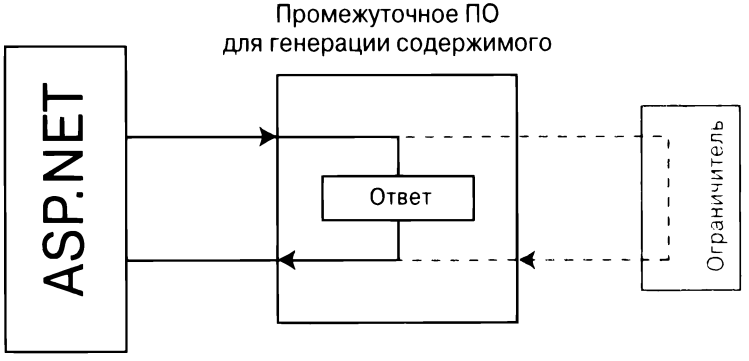


Рис. 14.7. Пример конвейера промежуточного ПО

Если URL отличается от `/middleware`, тогда класс `ContentMiddleware` передает запрос следующему компоненту в цепочке. Поскольку других компонентов нет, запрос достигает ограничительного обработчика, предоставленного инфраструктурой ASP.NET Core при создании конвейера, который посылает запрос обратно через конвейер в противоположном направлении (после того, как вы ознакомитесь с работой других типов промежуточного ПО, данный процесс обретет больший смысл).

Использование служб в промежуточном ПО

Применять службы, настроенные в методе `ConfigureServices()`, могут не только контроллеры. Инфраструктура ASP.NET Core инспектирует конструкторы классов промежуточного ПО и использует службы для предоставления значений любым аргументам, которые были определены. В листинге 14.19 к конструктору класса `ContentMiddleware` добавлен аргумент, который сообщает ASP.NET Core о необходимости предоставления объекта `UptimeService`.

Листинг 14.19. Применение службы в файле `ContentMiddleware.cs` из файла `Infrastructure`

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

  public class ContentMiddleware {
    private RequestDelegate nextDelegate;
```

```
    private UptimeService uptime;
    public ContentMiddleware(RequestDelegate next, UptimeService up) {
      nextDelegate = next;
      uptime = up;
    }
    public async Task Invoke(HttpContext httpContext) {
      if (httpContext.Request.Path.ToString().ToLower() == "/middleware")
      {
        await httpContext.Response.WriteAsync(
          "This is from the content middleware "+
            $"(uptime: {uptime.Uptime}ms)", Encoding.UTF8);
      } else {
        await nextDelegate.Invoke(httpContext);
      }
    }
  }
}
```

Наличие возможности использования служб означает, что компоненты промежуточного ПО могут разделять общую функциональность и избегать дублирования кода.

Запуск приложения и запрос URL вида `/middleware` приводит к получению вывода, представленного на рис. 14.8.

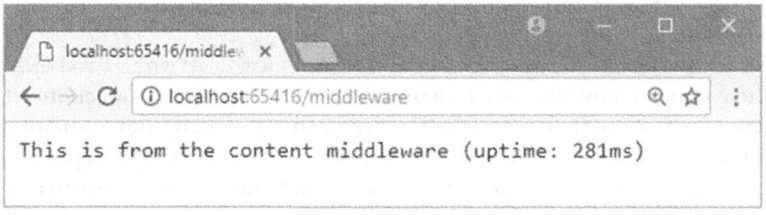


Рис. 14.8. Применение службы в специальном промежуточном ПО

Создание промежуточного ПО для обхода

Следующий тип промежуточного ПО перехватывает запросы до того, как они достигнут компонентов для генерации содержимого, чтобы *обойти* процесс прохождения конвейера, часто в целях, связанных с производительностью. В листинге 14.20 приведено содержимое файла класса по имени `ShortCircuitMiddleware.cs`, добавленного в папку `Infrastructure`.

Листинг 14.20. Содержимое файла `ShortCircuitMiddleware.cs` из папки `Infrastructure`

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

  public class ShortCircuitMiddleware {
```

```
    private RequestDelegate nextDelegate;

    public ShortCircuitMiddleware(RequestDelegate next) =>
        nextDelegate = next;

    public async Task Invoke(HttpContext httpContext) {
      if (httpContext.Request.Headers["User-Agent"]
            .Any(h => h.ToLower().Contains("edge"))) {
        httpContext.Response.StatusCode = 403;
      } else {
        await nextDelegate.Invoke(httpContext);
      }
    }
  }
}
```

Этот компонент промежуточного ПО проверяет заголовок `User-Agent` запроса, который браузеры применяют для идентификации самих себя. Использование заголовка `User-Agent` для опознания специфических браузеров не вполне надежно для реализации в реальном приложении, но достаточно для рассматриваемого примера.

Понятие "обход" применяется из-за того, что такой тип промежуточного ПО не всегда направляет запросы следующему компоненту в цепочке. В данном случае, если заголовок `User-Agent` содержит элемент `edge`, тогда компонент устанавливает код состояния в `403 - Forbidden` (403 — запрещено) и не направляет запрос следующему компоненту. Так как запрос отклонен, нет никакого смысла давать возможность запросу обрабатываться остальными компонентами, что привело бы к ненужному потреблению системных ресурсов. Взамен обработка запроса прекращается раньше, а клиенту посылается ответ 403.

Компоненты промежуточного ПО получают запросы в порядке, в котором они были настроены в классе `Startup`, что означает необходимость настройки промежуточного ПО для обхода перед промежуточным ПО для генерации содержимого (листинг 14.21).

Листинг 14.21. Регистрация компонента промежуточного ПО для обхода в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseMiddleware<ShortCircuitMiddleware>();
      app.UseMiddleware<ContentMiddleware>();
    }
  }
}
```

В результате запуска приложения и запрашивания любого URL с использованием браузера Microsoft Edge появится ошибка 403. Запросы из других браузеров компонент `ShortCircuitMiddleware` игнорирует и передает следующему компоненту в цепочке, т.е. ответ сгенерируется, когда запрошенным URL будет `/middleware`. На рис. 14.9 иллюстрируется добавление в конвейер компонента промежуточного ПО для обхода.

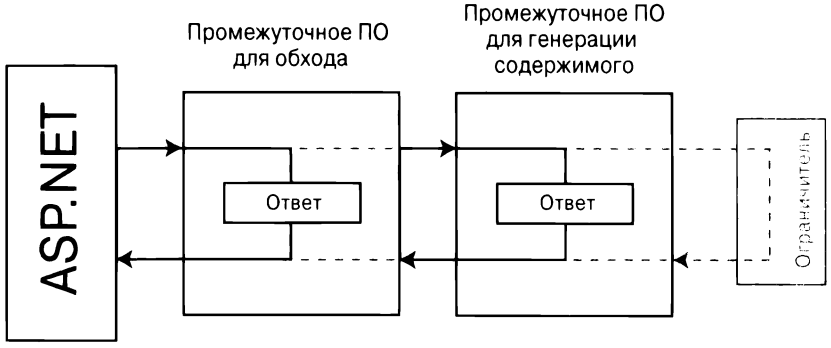


Рис. 14.9. Добавление в конвейер компонента промежуточного ПО для обхода

Создание промежуточного ПО для редактирования запросов

Следующий тип компонента промежуточного ПО, который мы исследуем, ответ не генерирует. Взамен компонент изменяет запросы перед тем, как они достигнут других компонентов, находящихся дальше в цепочке. Такой вид промежуточного ПО применяется преимущественно в целях интеграции с платформой, чтобы обогатить представление ASP.NET Core запроса HTTP средствами, специфическими для платформы. Он также может использоваться при подготовке запросов, чтобы их легче было обрабатывать последующими компонентами. В качестве демонстрации добавим в папку `Infrastructure` файл `BrowserTypeMiddleware.cs` с определением компонента промежуточного ПО, приведенным в листинге 14.22.

Листинг 14.22. Содержимое файла `BrowserTypeMiddleware.cs` из папки `Infrastructure`

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {
```

```
  public class BrowserTypeMiddleware {
    private RequestDelegate nextDelegate;

    public BrowserTypeMiddleware(RequestDelegate next) => nextDelegate = next;

    public async Task Invoke(HttpContext httpContext) {
      httpContext.Items["EdgeBrowser"]
        = httpContext.Request.Headers["User-Agent"]
          .Any(v => v.ToLower().Contains("edge"));
      await nextDelegate.Invoke(httpContext);
    }
  }
}
```

Новый компонент инспектирует заголовок `User-Agent` запроса и ищет в нем элемент `edge`, что предполагает возможность выдачи запроса браузером Edge. Объект `HttpContext` предоставляет через свойство `Items` словарь, который применяется для передачи данных между компонентами, и результат поиска в заголовке сохраняется с ключом `EdgeBrowser`.

Чтобы продемонстрировать, как компоненты промежуточного ПО могут взаимодействовать друг с другом, в листинге 14.23 показан класс `ShortCircuitMiddleware`, который отклоняет запросы, когда они поступают от браузера Edge, принимая решение на основе сгенерированных компонентом `BrowserTypeMiddleware` данных.

Листинг 14.23. Взаимодействие с другим компонентом в файле ShortCircuitMiddleware.cs из папки Infrastructure

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

  public class ShortCircuitMiddleware {
    private RequestDelegate nextDelegate;

    public ShortCircuitMiddleware(RequestDelegate next) => nextDelegate = next;

    public async Task Invoke(HttpContext httpContext) {
      if (httpContext.Items["EdgeBrowser"] as bool? == true) {
        httpContext.Response.StatusCode = 403;
      } else {
        await nextDelegate.Invoke(httpContext);
      }
    }
  }
}
```

Из-за присущей им природы компоненты промежуточного ПО, редактирующие запросы, необходимо помещать перед компонентами, с которыми они взаимодействуют или которые полагаются на вносимые ими изменения. В листинге 14.24 класс `BrowserTypeMiddleware` зарегистрирован как первый компонент в конвейере.

Листинг 14.24. Регистрация компонента промежуточного ПО для редактирования запросов в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseMiddleware<BrowserTypeMiddleware>();
      app.UseMiddleware<ShortCircuitMiddleware>();
      app.UseMiddleware<ContentMiddleware>();
    }
  }
}
```

Размещение компонента в начале конвейера гарантирует, что до попадания в другие компоненты запрос уже будет модифицирован (рис. 14.10).

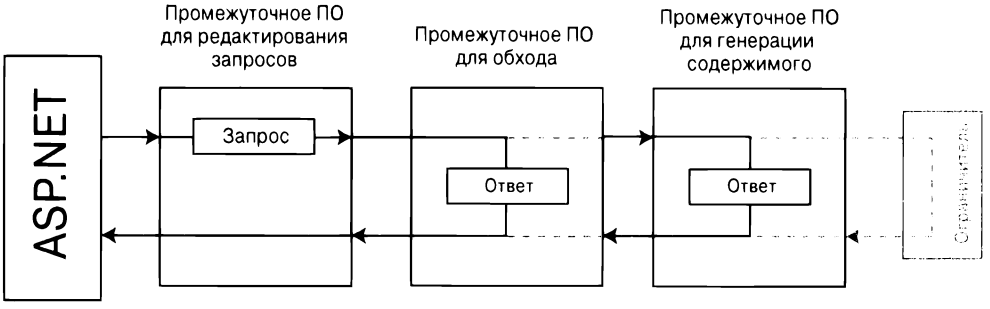


Рис. 14.10. Добавление в конвейер компонента промежуточного ПО для редактирования запросов

Создание промежуточного ПО для редактирования ответов

Последний тип промежуточного ПО оперирует на ответах, генерируемых другими компонентами в конвейере. Это полезно для регистрации в журнале деталей запросов и их ответов или для обработки ошибок. В листинге 14.25 представлено содержимое файла `ErrorMiddleware.cs`, который добавлен в папку `Infrastructure` для демонстрации данного вида компонента промежуточного ПО.

Листинг 14.25. Содержимое файла ErrorMiddleware.cs из папки Infrastructure

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace ConfiguringApps.Infrastructure {

  public class ErrorMiddleware {
    private RequestDelegate nextDelegate;

    public ErrorMiddleware(RequestDelegate next) {
      nextDelegate = next;
    }

    public async Task Invoke(HttpContext httpContext) {
      await nextDelegate.Invoke(httpContext);

      if (httpContext.Response.StatusCode == 403) {
        await httpContext.Response
          .WriteAsync("Edge not supported", Encoding.UTF8);
      } else if (httpContext.Response.StatusCode == 404) {
        await httpContext.Response
          .WriteAsync("No content middleware response", Encoding.UTF8);
      }
    }
  }
}
```

Компонент не заинтересован в запросе до тех пор, пока он не пройдет свой путь через конвейер промежуточного ПО и не будет сгенерирован ответ. Если кодом состояния ответа является 403 или 404, тогда компонент добавляет к ответу описательное сообщение. Все другие ответы игнорируются. В листинге 14.26 класс компонента регистрируется в классе Startup.

Совет. У вас может возникнуть вопрос, откуда поступил код состояния 404 - Not Found (404 — не найдено), поскольку он не устанавливается ни одним из трех созданных компонентов промежуточного ПО. Дело в том, что именно так инфраструктура ASP.NET конфигурирует ответ, когда запрос входит в конвейер, и этот ответ будет тем результатом, который возвращается клиенту, если компоненты промежуточного ПО его не изменяли.

Листинг 14.26. Регистрация компонента промежуточного ПО для редактирования ответов в файле Startup.cs из папки ConfiguringApps

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;
```

```
namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseMiddleware<ErrorMiddleware>();
      app.UseMiddleware<BrowserTypeMiddleware>();
      app.UseMiddleware<ShortCircuitMiddleware>();
      app.UseMiddleware<ContentMiddleware>();
    }
  }
}
```

Класс `ErrorMiddleware` зарегистрирован в первой позиции конвейера, что может выглядеть странным для компонента, который заинтересован только в ответах, но регистрация компонента в начале цепочки обеспечивает ему возможность инспектировать ответы, сгенерированные любыми другими компонентами (рис. 14.11). В случае помещения компонента дальше в конвейере он сможет инспектировать ответы, генерируемые только компонентами, находящимися после него.

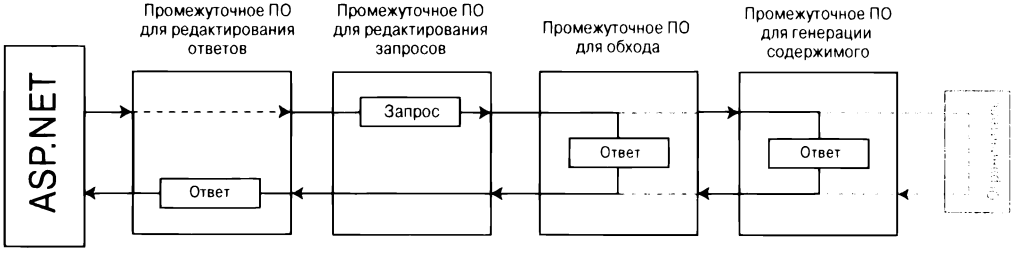


Рис. 14.11. Добавление в конвейер компонента промежуточного ПО для редактирования ответов

Эффект от нового промежуточного ПО можно увидеть, запустив приложение и запросив любой URL кроме `/middleware`. Результатом будет сообщение об ошибке, показанное на рис. 14.12.

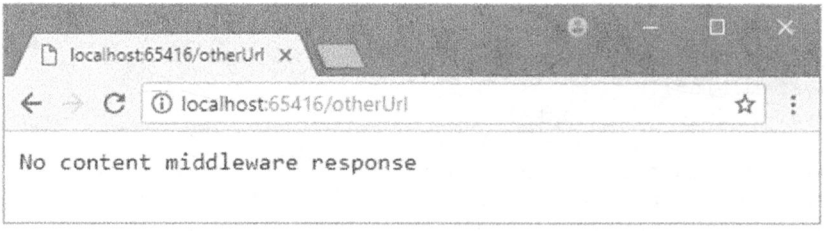


Рис. 14.12. Редактирование ответов от других компонентов промежуточного ПО

Особенности вызова метода Configure()

Платформа ASP.NET Core исследует метод Configure() перед его вызовом и получает список его аргументов, которые он предоставляет с использованием служб, настроенных в методе ConfigureServices(), или специальных служб, описанных в табл. 14.5.

Таблица 14.5. Специальные службы, доступные как аргументы метода Configure()

Тип	Описание
IApplicationBuilder	Этот интерфейс определяет функциональность, требующуюся для настройки конвейера промежуточного ПО приложения
IHostingEnvironment	Этот интерфейс определяет функциональность, необходимую для различения типов сред, таких как среда разработки и производственная среда

Использование интерфейса IApplicationBuilder

Хотя определять любые аргументы для метода Configure() не обязательно, в большинстве классов Startup будет применяться, по меньшей мере, интерфейс IApplicationBuilder, потому что он позволяет создать конвейер промежуточного ПО, как демонстрировалось ранее в главе. В случае специальных компонентов промежуточного ПО для регистрации классов используется расширяющий метод UseMiddleware(). Сложные пакеты промежуточного ПО для генерации содержимого предоставляют единственный метод, который настраивает все их компоненты промежуточного ПО за один шаг — точно так же, как они предоставляют единственный метод для определения применяемых ими служб. Для MVC доступны два расширяющих метода, описанные в табл. 14.6.

Таблица 14.6. Расширяющие методы IApplicationBuilder для MVC

Имя	Описание
UseMvcWithDefaultRoute()	Этот метод настраивает компоненты промежуточного ПО для MVC с использованием стандартного маршрута
UseMvc()	Этот метод настраивает компоненты промежуточного ПО для MVC с применением специальной конфигурации маршрутизации, указанной посредством лямбда-выражения

Маршрутизация — это процесс, посредством которого URL запросов сопоставляются с контроллерами и действиями, определенными в приложении; маршрутизация подробно рассматривается в главах 15 и 16. Метод UseMvcWithDefaultRoute() полезен при изучении разработки приложений MVC, но в большинстве приложений будет вызываться метод UseMvc(), даже если в результате явно определяется точно такая же конфигурация маршрутизации, как и создаваемая методом UseMvcWithDefaultRoute(). В листинге 14.27 приведен пример. Такой подход делает конфигурацию маршрутизации, используемую в приложении, очевидной для других разработчиков и облегчает дальнейшее добавление новых маршрутов (что в какой-то момент требуется практически во всех приложениях).

Листинг 14.27. Настройка компонентов промежуточного ПО для MVC в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseMiddleware<ErrorMiddleware>();
      app.UseMiddleware<BrowserTypeMiddleware>();
      app.UseMiddleware<ShortCircuitMiddleware>();
      app.UseMiddleware<ContentMiddleware>();

      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Поскольку MVC настраивает компоненты промежуточного ПО для генерации содержимого, метод `UseMvc()` вызывается после регистрации всех других компонентов промежуточного ПО. Чтобы подготовить службы, от которых зависит MVC, метод `AddMvc()` должен вызываться в методе `ConfigureServices()`.

Использование интерфейса `IHostingEnvironment`

Интерфейс `IHostingEnvironment` предоставляет базовую — но важную — информацию о среде размещения, в которой функционирует приложение, с применением свойств, описанных в табл. 14.7.

Таблица 14.7. Свойства `IHostingEnvironment`

Имя	Описание
`ApplicationName`	Это свойство возвращает имя приложения, установленное размещающей платформой
`EnvironmentName`	Это свойство возвращает строку, которая описывает текущую среду, как объясняется далее в главе

Окончание табл. 14.7

Имя	Описание
`ContentRootPath`	Это свойство возвращает путь, по которому находятся файлы содержимого и конфигурации приложения
`WebRootPath`	Это свойство возвращает строку, указывающую каталог, в котором находится статическое содержимое для приложения. Обычно им будет папка `wwwroot`
`ContentRootFileProvider`	Это свойство возвращает объект, который реализует интерфейс `Microsoft.AspNetCore.FileProviders.IFileProvider` и может использоваться для чтения файлов из папки, указанной свойством `ContentRootPath`
`WebRootFileProvider`	Это свойство возвращает объект, который реализует интерфейс `Microsoft.AspNetCore.FileProviders.IFileProvider` и может применяться для чтения файлов из папки, указанной свойством `WebRootPath`

Свойства `ContentRootPath` и `WebRootPath` интересны, но в большинстве приложений не нужны, потому что имеется встроенный компонент промежуточного ПО, который можно использовать для доставки статического содержимого, как описано в разделе "Включение статического содержимого" далее в главе.

Важным свойством является `EnvironmentName`, которое позволяет модифицировать конфигурацию приложения на основе среды, где оно выполняется. По соглашению различают три среды (*среда разработки* (`Development`), *подготовительная среда* (`Staging`) и *производственная среда* (`Production`)).

Текущая среда размещения устанавливается с применением переменной среды по имени `ASPNETCORE_ENVIRONMENT`. Чтобы установить переменную среды, выберем пункт ConfiguringApps Properties (Свойства `ConfiguringApps`) из меню Project (Проект) в Visual Studio и в открывшемся диалоговом окне перейдем на вкладку Debug (Отладка). Дважды щелкнем на поле Value (Значение) для переменной среды, по умолчанию установленной в `Development`, и изменим ее на `Staging` (рис. 14.13). Сохраним изменения, чтобы новое имя среды вступило в силу.

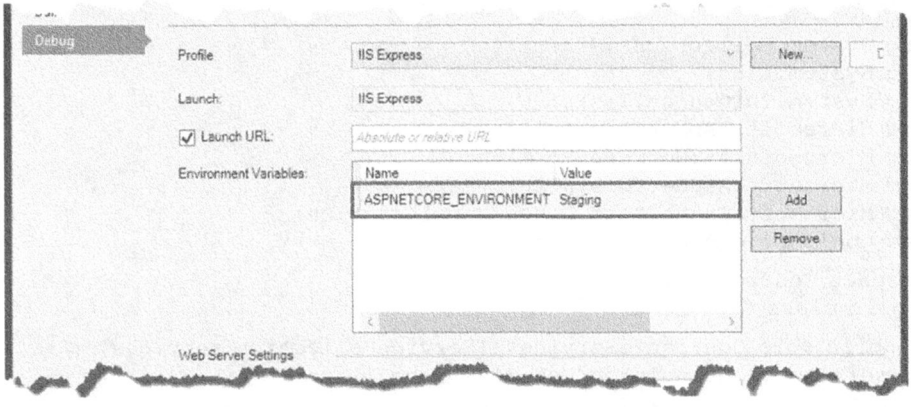


Рис. 14.13. Установка имени среды размещения

Совет. Имена сред не чувствительны к регистру символов, так что `Staging` и `staging` трактуются как одна и та же среда. Хотя `Development`, `Staging` и `Production` являются общепринятыми именами сред, допускается использовать любое желаемое имя. Это может быть полезно, например, когда над проектом трудятся несколько разработчиков, каждый из которых требует разных конфигурационных настроек. В разделе "Работа со сложными конфигурациями" далее в главе объясняется, как учитывать сложные различия между конфигурациями сред.

Выяснить внутри метода `Configure()`, какая среда размещения применяется, можно через свойство `IHostingEnvironment.EnvironmentName` или путем вызова одного из расширяющих методов, которые оперируют с объектами реализации `IHostingEnvironment` и описаны в табл. 14.8.

Таблица 14.8. Расширяющие методы `IHostingEnvironment`

Имя	Описание
`IsDevelopment()`	Этот метод возвращает `true`, если именем среды размещения является `Development`
`IsStaging()`	Этот метод возвращает `true`, если именем среды размещения является `Staging`
`IsProduction()`	Этот метод возвращает `true`, если именем среды размещения является `Production`
`IsEnvironment(env)`	Этот метод возвращает `true`, если имя среды размещения совпадает со значением аргумента `env`

Расширяющие методы применяются для изменения набора компонентов промежуточного ПО в конвейере, чтобы приспособить поведение приложения к различным средам размещения. В листинге 14.28 с помощью одного из расширяющих методов гарантируется, что специальные компоненты промежуточного ПО, созданные ранее в главе, присутствуют только в среде размещения `Development`.

Листинг 14.28. Использование среды размещения в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      if (env.IsDevelopment()) {
        app.UseMiddleware<ErrorMiddleware>();
        app.UseMiddleware<BrowserTypeMiddleware>();
        app.UseMiddleware<ShortCircuitMiddleware>();
        app.UseMiddleware<ContentMiddleware>();
      }
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Специальные компоненты промежуточного ПО не добавляются в конвейер при текущей конфигурации, в которой переменная среда была установлена в `Staging`. В результате запуска приложения и запроса URL вида `/middleware` будет получена ошибка `404 – Not Found`, т.к. доступными компонентами промежуточного ПО являются лишь те, которые настроены методом `UseMvc()`, а они не имеют контроллера для обработки указанного URL.

На заметку! Протестировав влияние от изменения среды размещения, не забудьте изменить ее снова на `Development`, иначе оставшиеся примеры в главе не будут работать должным образом.

Добавление оставшихся компонентов промежуточного программного обеспечения

Существует набор наиболее часто используемых компонентов промежуточного ПО, которые полезны в большинстве проектов MVC и применяются в примерах, рассматриваемых в настоящей книге. По мере добавления таких компонентов к конвейеру запросов в последующих разделах будет объясняться их работа.

Обеспечение возможности обработки исключений

Даже самое тщательно написанное приложение будет сталкиваться с исключениями, поэтому важно их надлежащим образом обрабатывать. В листинге 14.29 демонстрируется добавление к конвейеру запросов в классе `Startup` компонентов промежуточного ПО, которые имеют дело с исключениями. Кроме того, удалены специальные компоненты промежуточного ПО, чтобы можно было сосредоточиться на MVC.

Листинг 14.29. Добавление компонентов промежуточного ПО для обработки исключений в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

      if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
      } else {
        app.UseExceptionHandler("/Home/Error");
      }

      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Метод `UseStatusCodePages()` добавляет к ответам, не имеющим содержимого, описательные сообщения вроде `404 - Not Found` (404 — не найдено), что может быть удобно, т.к. не все браузеры отображают пользователю собственные сообщения.

Метод `UseDeveloperExceptionPage()` настраивает компонент промежуточного ПО для обработки ошибок, который отображает детали исключения в ответе, в том числе связанную трассировочную информацию. Информация подобного рода не должна быть видимой пользователям, поэтому вызов `UseDeveloperExceptionPage()` делается только в среде разработки, которая определяется с помощью объекта реализации `IHostingEnvironmment`.

В подготовительной или производственной среде взамен используется метод `UseExceptionHandler()`. Он настраивает обработку ошибок, позволяющую отображать специальное сообщение об ошибке, которое не раскрывает особенности внутренней работы приложения. Аргументом метода `UseExceptionHandler()` является URL, на который должен быть перенаправлен клиент, чтобы получить сообщение об ошибке. Это может быть любой URL, предоставляемый приложением, но по соглашению применяется `/Home/Error`.

В листинге 14.30 добавлена возможность генерации исключений по требованию действия `Index` контроллера `Home` и определено действие `Error`, так что запросы, генерируемые `UseExceptionHandler()`, могут быть обработаны.

Листинг 14.30. Генерация и обработка исключений в файле `HomeController.cs` из папки `Controllers`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;
namespace ConfiguringApps.Controllers {
  public class HomeController : Controller {
    private UptimeService uptime;
    public HomeController(UptimeService up) => uptime = up;
    public ViewResult Index(bool throwException = false) {
      if (throwException) {
        throw new System.NullReferenceException();
      }
      return View(new Dictionary<string, string> {
        ["Message"] = "This is the Index action",
        ["Uptime"] = $"{uptime.Uptime}ms"
      });
    }
    public ViewResult Error() => View(nameof(Index),
      new Dictionary<string, string> {
        ["Message"] = "This is the Error action"});
      }
}
```

Изменения, внесенные в действие `Index`, с помощью средства привязки моделей, которое обсуждается в главе 26, обеспечивают получение значения `throwException` из запроса. Действие генерирует исключение `NullReferenceException`, если значение `throwException` равно `true`, и выполняется нормально, если значение `throwException` равно `false`.

Действие `Error` использует представление `Index` для отображения простого сообщения. Запустив приложение и запросив URL вида `/Home/Index?throwException=true`, можно просмотреть результаты функционирования разных компонентов промежуточного ПО для обработки исключений. Строка запроса предоставляет значение для аргумента действия `Index`, а наблюдаемый ответ будет зависеть от имени среды размещения. На рис. 14.14 показан вывод, порождаемый методом `UseDeveloperExceptionPage()` (для среды размещения `Development`) и методом `UseExceptionHandler()` (для всех остальных сред размещения).

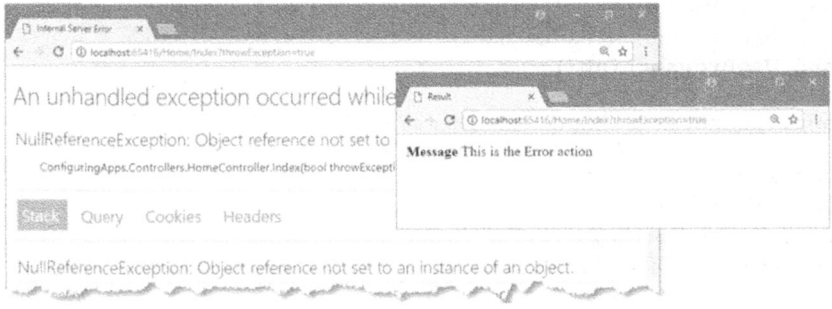


Рис. 14.14. Обработка исключений в среде разработки и подготовительной/производственной среде

Страница исключения для разработчика предлагает детали исключения, а также возможность исследовать его информацию трассировки стека и запрос, который привел к исключению. В противоположность этому страница исключения для пользователя должна применяться просто для указания о том, что что-то пошло не так.

Включение средства Browser Link

Средство Browser Link (Ссылка на браузер), описанное в главе 6, предназначено для управления браузерами на стадии разработки. Серверная часть средства Browser Link реализована в виде компонента промежуточного ПО, который должен быть добавлен в классе `Startup` как часть конфигурации приложения, потому что без такого компонента интеграция с Visual Studio работать не будет. Средство Browser Link полезно только на стадии разработки и не должно использоваться в подготовительной или производственной среде, поскольку оно модифицирует ответы, генерируемые другими компонентами промежуточного ПО, с целью вставки кода JavaScript, который открывает HTTP-подключения с серверной стороной, чтобы она могла получать уведомления о перезагрузке страницы. В листинге 14.31 показано, как вызывать метод `UseBrowserLink()`, который регистрирует компонент промежуточного ПО только для среды размещения `Development`.

Листинг 14.31. Включение средства Browser Link в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;
namespace ConfiguringApps {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseBrowserLink();
      } else {
        app.UseExceptionHandler("/Home/Error");
      }
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Включение статического содержимого

Финальный компонент промежуточного ПО, полезный для большинства проектов, предоставляет доступ к файлам в папке wwwroot, что позволяет приложениям включать изображения, файлы JavaScript и таблицы стилей CSS. Метод UseStaticFiles() добавляет компонент, который обходит конвейер запросов для статических файлов (листинг 14.32).

Листинг 14.32. Включение статического содержимого в файле Startup.cs из папки ConfiguringApps

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

      if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseBrowserLink();
      } else {
        app.UseExceptionHandler("/Home/Error");
      }
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Статическое содержимое обычно требуется вне зависимости от среды размещения, из-за чего метод UseStaticFiles() вызывается для всех сред. Это означает, что элемент link в представлении Index будет надлежаще работать и позволит браузеру загрузить таблицу стилей CSS из Bootstrap. Запустив приложение, можно увидеть результат, представленный на рис. 14.15.

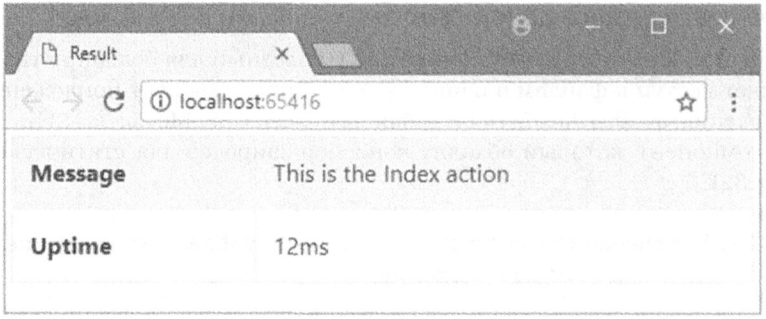


Рис. 14.15. Включение статического содержимого

Конфигурирование приложения

Некоторые данные конфигурации часто изменяются, например, когда приложение переносится из среды разработки в производственную среду и потому для серверов баз данных требуются другие параметры. Вместо того чтобы жестко кодировать подобную информацию в классе `Startup`, платформа ASP.NET Core разрешает поставлять данные конфигурации из целого диапазона более легко изменяемых источников, таких как переменные среды, аргументы командной строки и файлы в формате JSON (JavaScript Object Notation — система обозначений для объектов JavaScript).

Данные конфигурации обычно обрабатываются автоматически, но из-за того, что мы заменили стандартные настройки в классе `Program`, понадобится явно добавить код, который будет получать данные и делать их доступными остальной части приложения (листинг 14.33).

Листинг 14.33. Загрузка данных конфигурации в файле `Program.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
  public class Program {

    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) {

      return new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
```

```
        .ConfigureAppConfiguration((hostingContext, config) => {
          config.AddJsonFile("appsettings.json",
            optional: true, reloadOnChange: true);
          config.AddEnvironmentVariables();
          if (args != null) {
            config.AddCommandLine(args);
          }
        })
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();
    }
  }
}
```

Метод `ConfigureAppConfiguration()` применяется для обработки данных конфигурации; его аргументами являются объект `WebHostBuilderContext` и объект, реализующий интерфейс `IConfigurationBuilder`. В классе `WebBostBuilderContext` определены свойства, описанные в табл. 14.9.

Таблица 14.9. Свойства, определенные в классе `WebBostBuilderContext`

Имя	Описание
`HostingEnvironment`	Это свойство возвращает объект, который реализует интерфейс `IHostingEnvironment` и предоставляет информацию о среде размещения, где функционирует приложение. Подробные сведения были даны в разделе "Использование интерфейса `IHostingEnvironment`" ранее в главе
`Configuration`	Это свойство возвращает объект, который реализует интерфейс `IConfiguration`, предоставляющий доступ только по чтению к данным конфигурации в приложении

Интерфейс `IConfigurationBuilder` применяется при подготовке данных конфигурации для остальной части приложения, которая обычно выполняется с использованием расширяющих методов. Три метода, задействованные в листинге 14.33 для добавления данных конфигурации, описаны в табл. 14.10.

Таблица 14.10. Расширяющие методы интерфейса `IConfigurationBuilder`, предназначенные для добавления данных конфигурации

Имя	Описание
`AddJsonFile()`	Этот метод применяется для загрузки данных конфигурации из файла JSON, такого как `appsettings.json`
`AddEnvironmentVariables()`	Этот метод используется для загрузки данных конфигурации из переменных среды
`AddCommandLine()`	Этот метод применяется для загрузки данных конфигурации из аргументов командной строки, указываемых при запуске приложения

Из трех методов, которые использовались для загрузки данных конфигурации в листинге 14.33, самым интересным считается `AddJsonFile()`. В аргументах метода `AddJsonFile()` указываются имя файла, является ли файл необязательным, и должны ли данные конфигурации загружаться повторно, если файл изменяется:

```
...
config.AddJsonFile("appsettings.json",
                    optional: true, reloadOnChange: true);
...
```

В значениях аргументов указан файл по имени `appsettings.json`, которое представляет собой общепринятое имя для конфигурационного файла JSON. Файл необязателен, т.е. если он не существует, то никакого исключения не генерируется, и он будет отслеживаться на предмет изменений, так что данные конфигурации могут автоматически обновляться.

Повторная загрузка данных конфигурации

Система конфигурации ASP.NET Core поддерживает повторную загрузку данных, когда конфигурационные файлы изменяются. Некоторые встроенные компоненты промежуточного ПО, такие как система регистрации в журнале, поддерживают такую функциональность, что означает возможность изменения уровней регистрации во время выполнения без необходимости в перезапуске приложения. Аналогичные средства можно также предусмотреть при разработке специальных компонентов промежуточного ПО.

Однако просто тот факт, что средство делает что-то возможным, вовсе не означает, что оно является целесообразным. Внесение изменений в файлы конфигурации производственных систем — верный способ вызвать простой из-за отказа. Слишком легко допустить ошибку при модификации и получить неправильно работающую конфигурацию. Даже при успешном изменении могут возникать непредвиденные последствия, такие как переполнение дисков журнальными данными или резкое снижение производительности.

Рекомендуется избегать динамического редактирования и удостоверяться, что все изменения прошли через стандартные процедуры разработки и тестирования в подготовительной среде, прежде чем развертывать их в производственной среде. Может возникнуть соблазн заняться модификацией активной системы, чтобы установить причину проблемы, но это редко заканчивается хорошо. Если вы обнаруживаете, что занимаетесь редактированием файлов конфигурации производственной системы, то должны задаться вопросом, готовы ли вы превратить небольшую проблему в гораздо более крупную.

Создание конфигурационного файла JSON

Самое распространенное применение файла `appsettings.json` — хранение строк подключения к базам данных и настроек регистрации в журналах, но в нем можно держать любые данные, в которых нуждается приложение.

Чтобы посмотреть, как работает система конфигурации, добавим в корневой каталог проекта новый файл JSON по имени `appsettings.json` с содержимым, приведенным в листинге 14.34.

Листинг 14.34. Содержимое файла `appsettings.json` из папки `ConfiguringApps`

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
}
```

Формат JSON делает возможной структуру, предназначенную для настроек конфигурации. Содержимое JSON в листинге 14.34 определяет категорию конфигурации под названием `ShortCircuitMiddleware`, содержащую свойство по имени `EnableBrowserShortCircuit`, которое установлено в `true`.

Формат JSON: кавычки и запятые

Если ранее вы не имели дела с форматом JSON, тогда полезно уделить некоторое время чтению спецификации на веб-сайте `www.json.org`. С форматом JSON легко работать, к тому же большинство платформ предлагают хорошую поддержку для генерации и разбора данных JSON, в числе которых и приложения MVC (примеры ищите в главах 20 и 21), а на уровне клиента используется простой API-интерфейс JavaScript. В действительности большая часть разработчиков приложений MVC вообще не будут взаимодействовать с JSON напрямую, и написание кода JSON вручную требуется только в файлах конфигурации.

Существуют две ловушки, в которые попадают многие разработчики, не имеющие опыта работы с JSON. Хотя вы по-прежнему должны найти время на чтение спецификации, знание наиболее распространенных проблем предоставит определенную отправную точку в ситуации, когда Visual Studio или ASP.NET Core не смогут провести разбор ваших файлов JSON. Ниже показано добавление к стандартному содержимому файла `global.json`, в котором присутствуют две самые распространенные проблемы:

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
  mysetting : [ fast, slow ]
}
```

Во-первых, практически все в JSON помещается в кавычки. Очень легко забыть, что вы пишете вовсе не код C#, и посчитать, что имена свойств и значения должны быть восприняты без кавычек. Тем не менее, в JSON все кроме булевских значений и чисел должно помещаться в кавычки:

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  }
  "mysetting" : [ "fast", "slow"]
}
```

Во-вторых, при добавлении нового свойства к JSON-описанию объекта вы должны помнить о необходимости добавления запятой после предшествующего символа фигурной скобки:

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  },
  "mysetting" : [ "fast", "slow"]
}
```

Заметить отличие довольно трудно даже когда оно выделено полужирным (именно потому такая ошибка является распространенной), но здесь помещена запятая после символа `}`, закрывающего раздел `ShortCircuitMiddleware`. Однако будьте внимательны, потому что замыкающая запятая, после которой отсутствует раздел, также приводит к ошибке. Если внесенные вами изменения в файл JSON вызвали проблемы, то в первую очередь нужно выполнить проверку на предмет двух указанных выше ошибок.

Использование данных конфигурации

Класс Startup может получить доступ к данным конфигурации путем определения конструктора с аргументом типа IConfiguration. Когда в классе Program вызывается метод UseStartup(), данные конфигурации, подготовленные методом ConfigureAppConfiguration, применяются для создания объекта Startup. В листинге 14.35 демонстрируется добавление к классу Startup конструктора и доступ к данным конфигурации.

Листинг 14.35. Получение и использование данных конфигурации в файле Startup.cs из папки ConfiguringApps

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;
using Microsoft.Extensions.Configuration;

namespace ConfiguringApps {
  public class Startup {
    public Startup(IConfiguration configuration) {
      Configuration = configuration;
    }
    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      if ((Configuration.GetSection("ShortCircuitMiddleware")?
           .GetValue<bool>("EnableBrowserShortCircuit")).Value) {
        app.UseMiddleware<BrowserTypeMiddleware>();
        app.UseMiddleware<ShortCircuitMiddleware>();
      }
      if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseBrowserLink();
      } else {
        app.UseExceptionHandler("/Home/Error");
      }
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Конструктор получает объект реализации IConfiguration и присваивает его свойству по имени Configuration, которое затем может применяться для доступа к данным конфигурации, загруженным из переменных среды, командной строки и файла appsettings.json.

Чтобы получить значение, понадобится пройти по структуре данных до требуемого раздела конфигурации, представленного с помощью еще одного объекта, который реализует интерфейс IConfiguration, предоставляющий подмножество доступных для IConfigurationRoot членов (табл. 14.11).

Таблица 14.11. Члены, определяемые интерфейсом IConfiguration

Имя	Описание
[key]	Этот индексатор используется для получения строкового значения, соответствующего указанному ключу key
GetSection(name)	Этот метод возвращает объект реализации IConfiguration, который представляет раздел данных конфигурации
GetChildren()	Этот метод возвращает перечисление объектов реализации IConfiguration, которые представляют подразделы текущего объекта конфигурации

Расширяющие методы, описанные в табл. 14.12, можно применять для оперирования объектами реализации IConfiguration с целью получения значений и их преобразования из строк в другие типы.

Таблица 14.12. Расширяющие методы для интерфейса IConfiguration

Имя	Описание
GetValue<T>(keyName)	Этот метод получает значение, ассоциированное с указанным ключом keyName, и пытается преобразовать его в тип T
GetValue<T>(keyName, defaultValue)	Этот метод получает значение, ассоциированное с указанным ключом keyName, и пытается преобразовать его в тип T. Если значение для ключа в данных конфигурации отсутствует, тогда будет использоваться стандартное значение defaultValue

Важно не предполагать, что конфигурационное значение будет указано. В листинге 14.35 применяется null-условная операция, чтобы удостовериться в получении раздела ShortCircuitMiddleware до того, как пытаться извлечь значение EnableBrowserShortCircuit. В результате специальный компонент промежуточного ПО добавится к конвейеру запросов, только если значение EnableBrowserShortCircuit из раздела ShortCircuitMiddleware было определено и установлено в true.

Конфигурирование регистрации в журнале

Платформа ASP.NET Core включает поддержку сбора и обработки журнальных данных, а многие встроенные компоненты промежуточного ПО написаны так, чтобы генерировать журнальные сообщения. В большинстве проектов регистрация в журнале настраивается автоматически, но поскольку в классе Program используются индивидуальные конфигурационные операторы, для настройки средства регистрации в журнале необходимо добавить операторы, показанные в листинге 14.36.

Листинг 14.36. Конфигурирование регистрации в журнале в файле `Program.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
  public class Program {

    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) {

      return new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) => {
          config.AddJsonFile("appsettings.json",
            optional: true, reloadOnChange: true);
          config.AddEnvironmentVariables();
          if (args != null) {
            config.AddCommandLine(args);
          }
        })
        .ConfigureLogging((hostingContext, logging) => {
          logging.AddConfiguration(
            hostingContext.Configuration.GetSection("Logging"));
          logging.AddConsole();
          logging.AddDebug();
        })
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();
    }
  }
}
```

Метод `ConfigureLogging()` настраивает систему регистрации в журнале с применением лямбда-функции, которая получает объект `WebHostBuilderContext` (упомянутый ранее в главе) и объект, реализующий интерфейс `ILoggingBuilder`. Для интерфейса `ILoggingBuilder` предусмотрен набор расширяющих методов, позволяющих конфигурировать систему регистрации в журнале (табл. 14.13).

Таблица 14.13. Расширяющие методы для интерфейса `ILoggingBuilder`

Имя	Описание
`AddConfiguration()`	Этот метод используется для конфигурирования системы регистрации в журнале с применением данных конфигурации, которые были загружены из файла `appsettings.json`, из командной строки либо из переменных среды
`AddConsole()`	Этот метод отправляет журнальные сообщения консоли, что удобно в случае запуска приложения с использованием команды `dotnet run`
`AddDebug()`	Этот метод отправляет журнальные сообщения в окно Output (Вывод), когда выполняется отладчик Visual Studio
`AddEventLog()`	Этот метод отправляет журнальные сообщения в журнал событий Windows, что удобно в ситуации, когда приложение ASP.NET Core MVC развернуто на сервере Windows и нужно, чтобы сообщения из него смешивались с сообщениями от приложений других типов

Данные конфигурации системы регистрации в журнале

Метод `AddConfiguration()` применяется для конфигурирования системы регистрации в журнале, используя данные конфигурации, которые обычно определены в файле `appsettings.json`. В листинге 14.37 в файл `appsettings.json` добавляется раздел конфигурации по имени `Logging`, которое соответствует имени, применяемому для метода `AddConfiguration()` из листинга 14.36.

Листинг 14.37. Добавление раздела конфигурации в файле `appsettings.json` из папки `ConfiguringApps`

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": true
  },
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

В конфигурации системы регистрации в журнале указан уровень сообщений, которые должны отображаться из различных источников журнальных данных. Система регистрации в журнале поддерживает шесть уровней отладочной информации, которые описаны в табл. 14.14 в порядке их важности.

Элемент `Default` в листинге 14.37 устанавливает порог для отображения журнальных сообщений в `Debug`, т.е. отображаться будут только сообщения с уровнем `Debug` и выше. Остальные элементы переопределяют стандартные настройки для журнальных сообщений из специфических пространств имен, чтобы журнальные сообщения, которые поступают из пространств имен `System` и `Microsoft`, отображались только в случае, если они относятся к уровню `Information` и выше.

Таблица 14.14. Уровни отладочной информации ASP.NET

Уровень	Описание
Trace	Этот уровень используется для сообщений, которые полезны во время разработки, но не требуются в производственной среде
Debug	Этот уровень применяется для детализированных сообщений, требующихся разработчикам при отладке с целью выявления проблем
Information	Этот уровень используется для сообщений, которые описывают общее функционирование приложения
Warning	Этот уровень применяется для сообщений, описывающих события, которые не предвиделись, но и не приводят к прерыванию работы приложения
Error	Этот уровень используется для сообщений, которые описывают ошибки, прерывающие работу приложения
Critical	Этот уровень применяется для сообщений, описывающих катастрофические отказы
None	Этот уровень используется для отключения регистрации журнальных сообщений

Для просмотра результатов включения регистрации в журнале запустим приложение с применением отладчика Visual Studio, выбрав пункт меню Debug⇨Start Debugging (Отладка⇨Запустить отладку). Заглянув в окно Output, можно увидеть журнальные сообщения, которые показывают, как обрабатывался каждый HTTP-запрос:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:65417/
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method ConfiguringApps.Controllers.HomeController.Index
      (ConfiguringApps) with arguments (False) - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor[1]
      Executing ViewResult, running view at path /Views/Home/Index.cshtml.
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action ConfiguringApps.Controllers.HomeController.Index
      (ConfiguringApps) in 1597.3535ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 1695.6314ms 200 text/html; charset=utf-8
```

Создание специальных журнальных сообщений

Журнальные сообщения в предыдущем разделе генерировались компонентами ASP.NET Core и MVC, которые обрабатывают HTTP-запрос и производят ответ. Сообщения подобного рода способны предоставлять полезную информацию, но можно также генерировать специальные журнальные сообщения, которые являются специфичными для конкретного приложения, как демонстрируется в листинге 14.38.

Листинг 14.38. Специальная регистрация в журнале в файле `HomeController.cs` из папки `Controllers`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ConfiguringApps.Infrastructure;
using Microsoft.Extensions.Logging;
```

```
namespace ConfiguringApps.Controllers {
  public class HomeController : Controller {
    private UptimeService uptime;
    private ILogger<HomeController> logger;

    public HomeController(UptimeService up, ILogger<HomeController> log)
{
      uptime = up;
      logger = log;
    }

    public ViewResult Index(bool throwException = false) {
      logger.LogDebug($"Handled {Request.Path} at uptime {uptime.Uptime}");

      if (throwException) {
        throw new System.NullReferenceException();
      }
      return View(new Dictionary<string, string> {
        ["Message"] = "This is the Index action",
        ["Uptime"] = $"{uptime.Uptime}ms"
      });
    }

    public ViewResult Error() => View(nameof(Index),
      new Dictionary<string, string> {
        ["Message"] = "This is the Error action"});
      }
}
```

Интерфейс `ILogger` определяет функциональность, требующуюся для создания журнальных записей и получения объекта, который реализует этот интерфейс, а класс `HomeController` имеет конструктор, принимающий аргумент типа `ILogger<HomeController>`. Параметр типа позволяет системе регистрации в журнале использовать имя класса в журнальных сообщениях, и значение для аргумента конструктора предоставляется автоматически через средство внедрения зависимостей, которое будет описано в главе 18.

Располагая объектом реализации `ILogger`, журнальные сообщения можно создавать с применением расширяющих методов, определенных в пространстве имен `Microsoft.Extensions.Logging`. Предусмотрены методы для каждого уровня отладочной информации из табл. 14.14. Класс `HomeController` использует метод `LogDebug()` для создания сообщения на уровне `Debug`. Чтобы увидеть результат, понадобится запустить приложение с применением отладчика Visual Studio и поискать в окне Output журнальное сообщение вроде такого:

```
dbug: ConfiguringApps.Controllers.HomeController[0]
  Handled / at uptime 12
```

При запуске приложения отображается много сообщений, что может затруднить выявление отдельных сообщений. Одиночные сообщения легче найти, если щелкнуть на кнопке Clear All (Очистить все) в верхней части окна Output и затем перезагрузить страницу в браузере, что обеспечит отображение журнальных сообщений, относящихся к единственному запросу.

Конфигурирование внедрения зависимостей

Стандартная конфигурация для приложений ASP.NET Core включает подготовку поставщика служб, который используется средством внедрения зависимостей, детально описанным в главе 18. В листинге 14.39 показано добавление операторов конфигурирования в класс `Program`.

Листинг 14.39. Конфигурирование служб в файле `Program.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
  public class Program {

    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) {
      return new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) => {
          config.AddJsonFile("appsettings.json",
            optional: true, reloadOnChange: true);
          config.AddEnvironmentVariables();
          if (args != null) {
            config.AddCommandLine(args);
          }
        })
        .ConfigureLogging((hostingContext, logging) => {
          logging.AddConfiguration(
            hostingContext.Configuration.GetSection("Logging"));
          logging.AddConsole();
          logging.AddDebug();
        })
        .UseIISIntegration()
        .UseDefaultServiceProvider((context, options) => {
          options.ValidateScopes =
            context.HostingEnvironment.IsDevelopment();
        })
        .UseStartup<Startup>()
        .Build();
    }
  }
}
```

Метод `UseDefaultServiceProvider()` применяет встроенный поставщик служб ASP.NET Core. Хотя доступны альтернативные поставщики служб, в большинстве проектов подойдут встроенные средства, а использовать сторонний компонент рекомендуется только при решении специфической задачи и наличии хорошего понимания концепции внедрения зависимостей, которая будет обсуждаться в главе 18.

Метод `UseDefaultServiceProvider()` принимает лямбда-функцию, которая получает объекты `WebHostBuilderContext` и `ServiceProviderOptions`, применяемые для конфигурирования встроенного поставщика служб. Единственное свойство конфигурации называется `ValidateScopes`, и его установка в `false` является обязательной при работе с Entity Framework Core, как объяснялось в главе 8.

Конфигурирование служб MVC

Когда метод `AddMvc()` вызывается внутри метода `ConfigureServices()` класса `Startup`, он настраивает все службы, которые требуются приложениям MVC. Преимущество такого подхода связано с удобством, поскольку все службы MVC регистрируются за один шаг, но если необходимо изменить стандартное поведение, тогда понадобится выполнить дополнительную работу по реконфигурированию служб.

Метод `AddMvc()` возвращает объект, реализующий интерфейс `IMvcBuilder`, а инфраструктура MVC предлагает набор расширяющих методов, предназначенных для расширенного конфигурирования, наиболее полезные из которых описаны в табл. 14.15. Многие методы конфигурирования связаны со средствами, рассматриваемыми в последующих главах.

Таблица 14.15. Полезные расширяющие методы `IMvcBuilder`

Имя	Описание
`AddMvcOptions()`	Этот метод конфигурирует службы, используемые инфраструктурой MVC, как объясняется ниже
`AddFormatterMappings()`	Этот метод применяется для конфигурирования средства, которое позволяет клиентам указывать формат получаемых данных (глава 20)
`AddJsonOptions()`	Этот метод используется для конфигурирования способа создания данных JSON (глава 20)
`AddRazorOptions()`	Этот метод применяется для конфигурирования механизма визуализации Razor (глава 21)
`AddViewOptions()`	Этот метод используется для конфигурирования способа обработки представлений инфраструктурой MVC, в том числе применяемые механизмы визуализации (глава 21)

Метод `AddMvcOptions()` конфигурирует самые важные службы MVC. Он принимает функцию, получающую объект `MvcOptions`, который предоставляет набор конфигурационных свойств; наиболее полезные свойства приведены в табл. 14.16.

Перечисленные в табл. 14.16 методы конфигурирования используются для точной настройки способа функционирования инфраструктуры MVC, и в указанных главах вы найдете подробные описания связанных с ними средств. Тем не менее, в качестве небольшой демонстрации в листинге 14.40 показано, как можно применять метод `AddMvcOptions()` для изменения параметра конфигурации.

Таблица 14.16. Избранные свойства MvcOptions

Имя	Описание
Conventions	Это свойство возвращает список соглашений модели, которые используются для настройки способа создания контроллеров и действий инфраструктурой MVC (глава 31)
Filters	Это свойство возвращает список глобальных фильтров (глава 19)
FormatterMappings	Это свойство возвращает сопоставления, применяемые для того, чтобы позволить клиентам указывать формат получаемых ими данных (глава 20)
InputFormatters	Это свойство возвращает список объектов, используемых для разбора данных запросов (глава 20)
ModelBinders	Это свойство возвращает список связывателей модели, которые применяются для разбора запросов (глава 26)
ModelValidatorProviders	Это свойство возвращает список объектов, используемых для проверки достоверности данных (глава 27)
OutputFormatters	Это свойство возвращает список классов, которые форматируют данные, отправляемые из контроллеров API (глава 20)
RespectBrowserAcceptHeader	Это свойство указывает, должен ли учитываться заголовок Accept, когда принимается решение о том, какой формат данных задействовать для ответа (глава 20)

Листинг 14.40. Изменение параметра конфигурации в файле Startup.cs из папки ConfiguringApps

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddSingleton<UptimeService>();
  services.AddMvc().AddMvcOptions(options => {
    options.RespectBrowserAcceptHeader = true;
  });
}
...
```

Лямбда-выражение, передаваемое методу AddMvcOptions(), принимает объект MvcOptions, который используется для установки свойства RespectBrowserAcceptHeader в true. Это изменение позволяет клиентам оказывать большее влияние на формат данных, выбираемый процессом согласования содержимого, как будет объясняться в главе 20.

Работа со сложными конфигурациями

Если вы нуждаетесь в поддержке большого количества сред размещения или между средами размещения имеется много отличий, тогда применение операторов if для ветвления конфигураций в классе Startup может в результате дать конфигурацию, которую трудно читать и редактировать, не вызывая неожиданных изменений.

В последующих разделах будут описаны разнообразные способы использования класса `Startup` для сложных конфигураций.

Создание разных внешних конфигурационных файлов

Стандартная конфигурация для приложения, обеспечиваемая классом `Program`, ищет конфигурационные файлы JSON, которые специфичны к среде размещения, где запускается приложение. Таким образом, файл по имени `appsettings.production.json` можно применять для хранения настроек, специфичных для производственной среды. В листинге 14.41 восстановлен оператор, который загружает файл JSON в класс `Program`, удаленный в начале главы.

Листинг 14.41. Загрузка файлов среды в файле `Program.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
  public class Program {

    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) {

      return new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) => {
          var env = hostingContext.HostingEnvironment;
          config.AddJsonFile("appsettings.json",
            optional: true, reloadOnChange: true)
              .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                optional: true, reloadOnChange: true);
          config.AddEnvironmentVariables();
          if (args != null) {
            config.AddCommandLine(args);
          }
        })
        .ConfigureLogging((hostingContext, logging) => {
          logging.AddConfiguration(
            hostingContext.Configuration.GetSection("Logging"));
          logging.AddConsole();
          logging.AddDebug();
        })
        .UseIISIntegration()
```

```
        .UseDefaultServiceProvider((context, options) => {
          options.ValidateScopes =
            context.HostingEnvironment.IsDevelopment();
        })
        .UseStartup<Startup>()
        .Build();
    }
  }
}
```

При загрузке данных конфигурации из файла, специфичного для платформы, конфигурационные настройки, которые содержатся в нем, переопределяют любые существующие данные с теми же самыми именами. Например, в листинге 14.42 приведено содержимое файла по имени `appsettings.development.json`, созданного с применением шаблона элемента ASP.NET Configuration File. Данные конфигурации в этом файле устанавливают значение `EnableBrowserShortCircuit` в `false`.

Совет. Может показаться, что файл `appsettings.development.json` исчезает сразу же после его создания. Щелкнув на стрелке слева от записи `appsettings.json` в окне Solution Explorer, легко увидеть, что Visual Studio группирует вместе элементы с похожими именами.

Листинг 14.42. Содержимое файла `appsettings.development.json` из папки `ConfiguringApps`

```
{
  "ShortCircuitMiddleware": {
    "EnableBrowserShortCircuit": false
  }
}
```

Файл `appsettings.json` загрузится при запуске приложения, а за ним файл `appsettings.development.json`, если приложение выполняется в среде разработки. В результате свойство `EnableBrowserShortCircuit` получает значение `false`, когда приложение функционирует в среде разработки, и `true` — когда в подготовительной или производственной среде.

Создание разных методов конфигурирования

Выбор разных файлов с конфигурационными данными может быть удобен, но не предоставлять полного решения для сложных конфигураций, т.к. файлы данных не содержат операторы C#. Если желательно варьировать операторы конфигурирования, используемые для создания служб или регистрации компонентов промежуточного ПО, тогда допускается применять разные методы, причем имя метода должно включать название среды размещения (листинг 14.43).

Листинг 14.43. Использование разных имен методов в файле `Startup.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;
using Microsoft.Extensions.Configuration;

namespace ConfiguringApps {
  public class Startup {

    public Startup(IConfiguration configuration) {
      Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc().AddMvcOptions(options => {
      options.RespectBrowserAcceptHeader = true;
    });
  }

    public void ConfigureDevelopmentServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseExceptionHandler("/Home/Error");
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }

    public void ConfigureDevelopment(IApplicationBuilder app,
        IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseBrowserLink();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Когда инфраструктура ASP.NET Core ищет методы `ConfigureService()` и `Configure()` в классе `Startup`, она сначала выясняет, имеются ли методы, имена которых включают название среды размещения. В листинге 14.43 был добавлен метод `ConfigureDevelopmentServices()`, который в среде `Development` будет применяться вместо метода `ConfigureServices()`, и метод `ConfigureDevelopment()`, который в среде `Development` будет использоваться вместо метода `Configure()`.

Можно определить отдельные методы для каждой среды, которую необходимо поддерживать, и полагаться на стандартные методы, вызываемые в случае недоступности методов, специфичных для той или иной среды. В рассматриваемом примере это означает, что для подготовительной и производственной сред будут применяться методы `ConfigureServices()` и `Configure()`.

Внимание! Если определены методы, специфичные для среды, то стандартные методы не вызываются. Например, в листинге 14.43 инфраструктура ASP.NET Core не будет вызывать метод `Configure()` в среде `Development`, потому что имеется метод `ConfigureDevelopment()`. Другими словами, каждый метод несет ответственность за полную конфигурацию, требуемую для среды, на которую он ориентирован.

Создание разных классов конфигурирования

Использование разных методов означает, что вы не обязаны применять операторы `if` для проверки имени среды размещения, но в результате могут быть получены крупные классы, которые сами по себе являются проблемой. В случае особенно сложных конфигураций последним усовершенствованием будет создание собственного конфигурационного класса для каждой среды размещения. Когда инфраструктура ASP.NET Core ищет класс `Startup`, она первым делом проверяет, есть ли класс с именем, включающим название текущей среды размещения. С этой целью добавим в проект файл класса по имени `StartupDevelopment.cs`, содержимое которого показано в листинге 14.44.

Листинг 14.44. Содержимое файла `StartupDevelopment.cs` из папки `ConfiguringApps`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using ConfiguringApps.Infrastructure;

namespace ConfiguringApps {
  public class StartupDevelopment {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UptimeService>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseBrowserLink();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Класс `StartupDevelopment` содержит методы `ConfigureServices()` и `Configure()`, которые специфичны для среды размещения `Development`. Чтобы

предоставить инфраструктуре ASP.NET Core возможность найти класс `Startup`, специфичный для среды, в класс `Program` потребуется внести изменение, показанное в листинге 14.45.

Листинг 14.45. Включение класса `Startup`, специфичного для среды, в файле `Program.cs` из папки `ConfiguringApps`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using System.Reflection;

namespace ConfiguringApps {
  public class Program {
    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }
    public static IWebHost BuildWebHost(string[] args) {
      return new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) => {
          var env = hostingContext.HostingEnvironment;
          config.AddJsonFile("appsettings.json",
            optional: true, reloadOnChange: true)
              .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
            optional: true, reloadOnChange: true);
          config.AddEnvironmentVariables();
          if (args != null) {
            config.AddCommandLine(args);
          }
        })
        .ConfigureLogging((hostingContext, logging) => {
          logging.AddConfiguration(
            hostingContext.Configuration.GetSection("Logging"));
          logging.AddConsole();
          logging.AddDebug();
        })
        .UseIISIntegration()
        .UseDefaultServiceProvider((context, options) => {
          options.ValidateScopes =
            context.HostingEnvironment.IsDevelopment();
        })
        .UseStartup(nameof(ConfiguringApps))
        .Build();
    }
  }
}
```

Вместо указания какого-то класса методу `UseStartup()` передается имя сборки, которая должна использоваться. Когда приложение запускается, инфраструктура ASP.NET Core будет искать класс, имя которого включает название среды размещения, такой как `StartupDevelopment` или `StartupProduction`, и возвращаться к применению класса `Startup`, если специфичный для среды класс отсутствует.

Резюме

В главе рассматривались способы конфигурирования приложений MVC. Была описана роль классов `Startup` и `Program`, а также стандартных настроек конфигурации, которые они обеспечивают. Кроме того, обсуждалась обработка запросов с применением конвейера и использование различных типов промежуточного ПО для управления потоком запросов и результирующих ответов. В следующей главе будет представлена система маршрутизации, посредством которой инфраструктура MVC имеет дело с отображением URL запросов на контроллеры и действия.

ГЛАВА 15

Маршрутизация URL

В ранних версиях ASP.NET предполагалось наличие прямой связи между запрашиваемыми URL и файлами на жестком диске сервера. Работа сервера заключалась в получении запроса от браузера и доставке вывода из соответствующего файла. Подобный подход успешно работал для инфраструктуры Web Forms, в которой каждая страница ASPX представляла собой и файл, и самодостаточный ответ на запрос.

Это *совершенно не имеет* смысла в приложении MVC, где запросы обрабатываются методами действий из классов контроллеров, и однозначное соответствие между запросами и дисковыми файлами отсутствует.

При обработке URL в MVC платформа ASP.NET применяет *систему маршрутизации*, которая была модернизирована для версии ASP.NET Core. В настоящей главе вы узнаете, как использовать систему маршрутизации для обеспечения мощной и гибкой обработки URL в своих проектах. Вы увидите, что система маршрутизации позволяет создавать любой желаемый шаблон URL и выражать его в ясной и лаконичной манере. Система маршрутизации выполняет две функции.

- Исследование *входящих URL* и выбор контроллера и действия для обработки запроса.
- Генерация *исходящих URL*. Такие URL встречаются в HTML-разметке, визуализированной из представлений, так что специфическое действие может быть инициировано, когда пользователь щелкает на ссылке (после чего ссылка снова становится входящим URL).

В главе внимание будет сосредоточено на определении маршрутов и их применении для обработки входящих URL, чтобы пользователь мог достичь контроллеров и действий. Есть два способа создания маршрутов в приложении MVC: *маршрутизация на основе соглашений* и *маршрутизация с помощью атрибутов*. Мы рассмотрим здесь оба подхода.

Затем в следующей главе будет показано, как использовать те же самые маршруты для генерации исходящих URL, которые необходимо включать в представления, а также объяснены способы настройки системы маршрутизации и продемонстрировано применение связанного средства под названием *области*. В табл. 15.1 приведена сводка, позволяющая поместить маршрутизацию в контекст.

В табл. 15.2 представлена сводка по главе.

Таблица 15.1. Помещение маршрутизации в контекст

Вопрос	Ответ
Что это такое?	Система маршрутизации несет ответственность за обработку входящих запросов, а также выбор контроллеров и методов действий для их обработки. Вдобавок система маршрутизации используется для генерации маршрутов в представлениях, известных как исходящие URL
Чем она полезна?	Система маршрутизации обеспечивает гибкую обработку запросов без привязки URL к структуре классов в проекте Visual Studio
Как она используется?	Сопоставление URL с контроллерами и методами действий определяется в файле `Startup.cs` или путем применения атрибута `Route` к контроллерам
Существуют ли какие-то скрытые ловушки или ограничения?	Конфигурация маршрутизации для сложного приложения может стать трудной в управлении
Существуют ли альтернативы?	Нет. Система маршрутизации — неотъемлемая часть ASP.NET Core

Таблица 15.2. Сводка по главе

Задача	Решение	Листинг
Отображение между URL и методами действий	Определите маршрут	15.9
Возможность не указывать сегменты URL	Определите стандартные значения для сегментов маршрута	15.10–15.12
Сопоставление сегментов URL, которые не имеют соответствующих переменных маршрутизации	Определите статические сегменты	15.13–15.16
Передача сегментов URL методам действий	Определите специальные переменные сегментов	15.17–15.19
Возможность не указывать сегменты URL, для которых не предусмотрены стандартные значения	Определите необязательные сегменты	15.20, 15.21
Определение маршрутов, которые соответствуют любому количеству сегментов URL	Используйте сегмент общего захвата	15.22, 15.23
Ограничение URL, которым может соответствовать маршрут	Применяйте ограничения маршрута	15.24–15.33
Определение маршрута внутри контроллера	Используйте маршрутизацию с помощью атрибутов	15.34–15.38

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `UrlsAndRoutes` с применением шаблона ASP.NET Core Web Application (.NET Core) (Веб-приложение ASP.NET Core (.NET Core)). Добавим в класс `Startup` операторы, показанные в листинге 15.1, чтобы обеспечить поддержку MVC Framework, страниц ошибок для разработчиков и статических файлов.

Листинг 15.1. Конфигурирование приложения в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc();
    }
  }
}
```

Создание класса модели

Все усилия в текущей главе сконцентрированы на сопоставлении URL запросов с действиями. Единственному имеющемуся классу модели передаются детали о контроллере и методе действия, которые были выбраны для обработки запроса. Создадим папку `Models` и добавим в нее файл класса по имени `Result.cs` с определением, приведенным в листинге 15.2.

Листинг 15.2. Содержимое файла `Result.cs` из папки `Models`

```
using System.Collections.Generic;
namespace UrlsAndRoutes.Models {
  public class Result {
    public string Controller { get; set; }
    public string Action { get; set; }
    public IDictionary<string, object> Data { get; }
      = new Dictionary<string, object>();
  }
}
```

Свойства `Controller` и `Action` будут использоваться для указания того, как запрос был обработан, а словарь `Data` — для хранения других деталей о запросе, выдаваемых системой маршрутизации.

Создание примеров контроллеров

Для демонстрации работы маршрутизации нам нужны простые контроллеры. Создадим папку `Controllers` и добавим в нее файл класса по имени `HomeController.cs` с содержимым из листинга 15.3.

Листинг 15.3. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(HomeController),
        Action = nameof(Index)
      });
  }
}
```

Метод действия `Index()`, определенный в контроллере `Home`, вызывает метод `View()` для визуализации представления по имени `Result` (которое определяется в следующем разделе) и передает экземпляр `Result` в качестве объекта модели. Свойства объекта модели устанавливаются с применением операции `nameof`; они будут использоваться для указания, какие контроллер и метод действия были задействованы для обслуживания запроса.

Аналогичным образом добавим в папку `Controllers` файл класса `CustomerController.cs` с определением контроллера `Customer` (листинг 15.4).

Листинг 15.4. Содержимое файла `CustomerController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class CustomerController : Controller {

    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(CustomerController),
        Action = nameof(Index)
      });
    public ViewResult List() => View("Result",
      new Result {
        Controller = nameof(CustomerController),
        Action = nameof(List)
      });
  }
}
```

Третий (и последний) контроллер определяется в файле по имени AdminController.cs внутри папки Controllers (листинг 15.5). Он следует тому же подходу, как и ранее.

Листинг 15.5. Содержимое файла AdminController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class AdminController : Controller {

    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(AdminController),
        Action = nameof(Index)
      });
  }
}
```

Создание представления

Во всех методах действий, определенных в предыдущем разделе, указано представление Result, что позволяет создать одно представление, которое будет разделяться всеми контроллерами. Создадим папку Views/Shared и добавим в нее новый файл представления по имени Result.cshtml с содержимым, показанным в листинге 15.6.

Листинг 15.6. Содержимое файла Result.cshtml из папки Views/Shared

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.
min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
</body>
</html>
```

Представление содержит стилизованную с помощью Bootstrap таблицу, которая отображает свойства объекта модели. Чтобы добавить Bootstrap в проект, создадим файл bower.json, используя шаблон элемента Bower Configuration File (Файл конфигурации Bower), и укажем пакет Bootstrap в разделе dependencies (листинг 15.7).

Листинг 15.7. Добавление пакета Bootstrap в файле `bower.json` из папки `UrlsAndRoutes`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Финальный подготовительный шаг предусматривает создание в папке `Views` файла `_ViewImports.cshtml`, внутри которого настраиваются встроенные вспомогательные функции дескрипторов для применения в представлениях Razor и импортируется пространство имен модели (листинг 15.8).

Листинг 15.8. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@using UrlsAndRoutes.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Конфигурация в классе `Startup` не содержит никаких инструкций относительно того, как инфраструктура MVC должна сопоставлять HTTP-запросы с контроллерами и действиями. После запуска приложения любой запрашиваемый URL будет давать в результате ответ `404 - Not Found` (404 — не найдено), как видно на рис. 15.1.

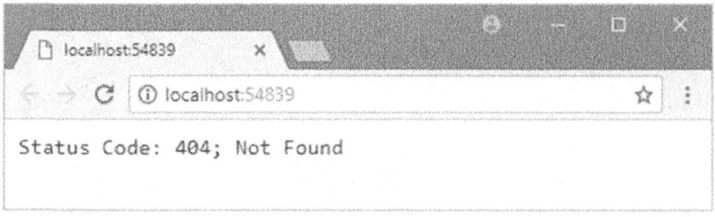


Рис. 15.1. Запуск примера приложения

Введение в шаблоны URL

Система маршрутизации работает с использованием множества *маршрутов*. Вместе такие маршруты образуют *схему* URL для приложения, представляющую собой набор URL, которые приложение будет распознавать и реагировать на них.

Набирать вручную все индивидуальные URL, которые планируется поддерживать в приложении, не придется. Взамен каждый маршрут содержит *шаблон URL*, с которым сравниваются входящие URL. Если входящий URL соответствует шаблону, тогда он применяется системой маршрутизации для обработки этого URL. Начнем с простого URL:

```
http://mysite.com/Admin/Index
```

URL могут быть разбиты на *сегменты* — части URL кроме имени хоста и строки запроса, которые отделяются друг от друга символом /. В приведенном выше примере URL есть два сегмента, как показано на рис. 15.2.



Рис. 15.2. Сегменты в примере URL

Первый сегмент содержит слово `Admin`, а второй — слово `Index`. В глазах человека совершенно очевидно, что первый сегмент имеет отношение к контроллеру, а второй — к действию. Однако, естественно, такое отношение должно быть выражено с использованием шаблона URL, который будет в состоянии воспринимать система маршрутизации. Вот шаблон URL, соответствующий примеру URL:

```
{controller}/{action}
```

При обработке входящего HTTP-запроса задача системы маршрутизации заключается в сопоставлении запрошенного URL с шаблоном и извлечении из URL значений для *переменных сегментов*, определенных в шаблоне.

Переменные сегментов выражаются с применением фигурных скобок (символов `{` и `}`). В приведенном выше примере шаблона присутствуют две переменные сегментов с именами `controller` и `action`, так что значением переменной сегмента `controller` будет `Admin`, а значением переменной сегмента `action` — `Index`.

Приложение MVC обычно будет иметь несколько маршрутов, и система маршрутизации будет сравнивать входящий URL с шаблоном URL каждого маршрута до тех пор, пока не найдет совпадение. По умолчанию шаблон URL будет соответствовать любому URL, который имеет подходящее количество сегментов. Например, шаблон `{controller}/{action}` будет соответствовать любому URL с двумя сегментами, как описано в табл. 15.3.

Таблица 15.3. Сопоставление URL

URL запроса	Переменные сегментов
`http://mysite.com/`**`Admin/Index`**	`controller=Admin, action=Index`
`http://mysite.com/`**`Admin`**	Соответствия нет — сегментов слишком мало
`http://mysite.com/`**`Admin/Index/Soccer`**	Соответствия нет — сегментов слишком много

В табл. 15.3 отражены две ключевые линии поведения шаблонов URL.

- Шаблоны URL *консервативны* в отношении количества сопоставляемых сегментов. Совпадение будет происходить только для тех URL, которые имеют то же самое количество сегментов, что и шаблон. Это можно наблюдать во втором и третьем примерах в таблице.
- Шаблоны URL *либеральны* в отношении содержимого сопоставляемых сегментов. Если URL имеет правильное количество сегментов, то шаблон извлечет значение каждого сегмента для переменной сегмента, каким бы оно ни было.

Описанные две стандартные линии поведения являются ключом к пониманию функционирования шаблонов URL. Позже в главе будет показано, как изменить такие стандартные линии поведения.

Создание и регистрация простого маршрута

Имеющийся шаблон URL можно использовать для определения маршрута. Маршруты определяются в файле `Startup.cs` и передаются в качестве аргументов методу `UseMvc()`, который применяется для настройки MVC в методе `Configure()`. В листинге 15.9 показан базовый маршрут, отображающий запросы на контроллеры в примере приложения.

Листинг 15.9. Определение базового маршрута в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(name: "default", template: "{controller}/{action}");
      });
    }
  }
}
```

Маршруты создаются с использованием лямбда-выражения, передаваемого в виде аргумента конфигурационному методу `UseMvc()`. Лямбда-выражение получает объект, который реализует интерфейс `IRouteBuilder` из пространства имен `Microsoft.AspNetCore.Routing`, а маршруты определяются с применением расширяющего метода `MapRoute()`. Чтобы облегчить понимание маршрутов, по соглашению при вызове метода `MapRoute()` указываются имена аргументов, что объясняет наличие в листинге явно именованных аргументов `name` и `template`. В аргументе `name` указывается имя маршрута, а в аргументе `template` определяется шаблон.

Совет. Назначать маршрутам имена не обязательно, и существует философский аргумент относительно того, что именование приносит в жертву четкое разделение обязанностей, которое в противном случае обеспечила бы маршрутизация. В разделе "Генерирование URL из специфического маршрута" главы 16 объясняется, почему именование может стать проблемой.

Запустив пример приложения, можно увидеть эффект от изменений, внесенных в маршрутизацию. При первом запуске приложения ничего не изменится — по-прежнему возникает ошибка 404, но если перейти на URL, который соответствует шаблону `{controller}/{action}`, тогда получится результат, подобный приведенному на рис. 15.3, где иллюстрируется переход на `/Admin/Index`.

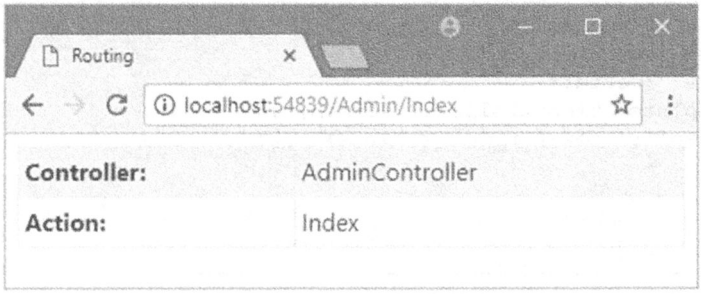


Рис. 15.3. Навигация с использованием простого маршрута

Корневой URL для приложения не работает из-за того, что маршрут, добавленный в файле `Startup.cs`, не сообщает инфраструктуре MVC о том, как выбирать класс контроллера и метод действия, когда URL запроса не содержит сегментов. Мы исправим это в следующем разделе.

Определение стандартных значений

Пример приложения возвращает ошибку 404, когда запрашивается стандартный URL, поскольку он не соответствует шаблону маршрута, определенному в классе `Startup`. Так как в стандартном URL отсутствуют сегменты, которые могли бы соответствовать переменным `controller` и `action`, определенным в шаблоне маршрута, система маршрутизации не дает совпадения.

Ранее объяснялось, что шаблоны URL будут соответствовать только URL с указанным количеством сегментов. Один из способов изменить такое поведение предусматривает применение *стандартных значений*. Стандартное значение используется, когда URL не содержит сегмент, который можно было бы сопоставить с шаблоном маршрута. В листинге 15.10 определен маршрут, в котором применяется стандартное значение.

Листинг 15.10. Предоставление стандартного значения в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {
```

```
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller}/{action}",
          defaults: new { action = "Index" });
      });
    }
  }
}
```

Стандартные значения задаются как свойства анонимного типа, передаваемого методу `MapRoute()` в аргументе `defaults`. В листинге 15.10 для переменной `action` предоставлено стандартное значение `Index`.

Как и ранее, этот маршрут будет соответствовать всем двухсегментным URL. Например, в случае запроса URL вида `http://mydomain.com/Home/Index` маршрут извлечет `Home` в качестве значения для `controller` и `Index` — для `action`.

Но теперь, когда имеется стандартное значение для сегмента `action`, маршрут будет *также* соответствовать и односегментным URL. При обработке односегментного URL система маршрутизации извлечет значение для переменной `controller` из URL и будет применять стандартное значение для переменной `action`. Таким образом, запрос пользователем `/Home` приведет к тому, что MVC вызовет метод действия `Index()` контроллера `Home` (рис. 15.4).

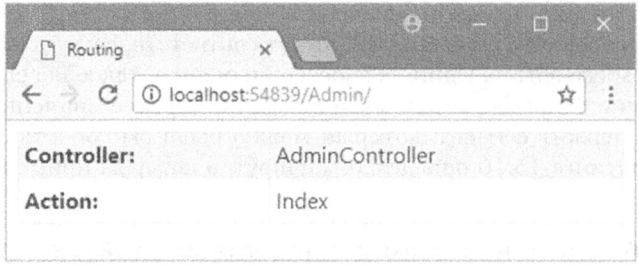


Рис. 15.4. Использование стандартного действия

Определение встраиваемых стандартных значений

Стандартные значения могут также быть выражены как часть шаблона URL, что дает более лаконичный способ определения маршрутов, как показано в листинге 15.11. Встраиваемый синтаксис может применяться только для предоставления стандартных значений переменным, которые являются частью шаблона URL, но, как вскоре вы узнаете, зачастую удобно иметь возможность устанавливать стандартные значения за пределами шаблона. По указанной причине важно понимать оба способа выражения стандартных значений.

Листинг 15.11. Определение встраиваемых стандартных значений в файле Startup.cs из папки UrlsAndRoutes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller}/{action=Index}");
      });
    }
  }
}
```

Мы можем пойти еще дальше и сопоставлять с URL, которые вообще не содержат переменных сегментов, полагаясь при идентификации действия и контроллера только на стандартные значения. Для примера в листинге 15.12 демонстрируется отображение корневого URL приложения за счет предоставления стандартных значений для обоих сегментов.

Листинг 15.12. Предоставление стандартных значений для переменных controller и action в файле Startup.cs из папки UrlsAndRoutes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}");
      });
    }
  }
}
```

За счет предоставления стандартных значений для переменных `controller` и `action` маршрут будет соответствовать URL с нулем, одним или двумя сегментами (табл. 15.4).

Таблица 15.4. Соответствие URL

Количество сегментов	Пример	На что отображается
0	/	`controller=Home, action=Index`
1	/Customer	`controller=Customer, action=Index`
2	/Customer/List	`controller=Customer, action=List`
3	/Customer/List/All	Соответствия нет — сегментов слишком много

Чем меньше сегментов получено во входящем URL, тем больше маршрут полагается на стандартные значения, вплоть до момента, когда URL без сегментов сопоставляется с использованием только стандартных значений.

Запустив пример приложения, можно увидеть результат применения стандартных значений. Когда браузер запрашивает корневой URL приложения, для переменных сегментов `controller` и `action` будут использоваться стандартные значения, которые приведут к тому, что инфраструктура MVC вызовет метод действия `Index()` контроллера `Home` (рис. 15.5).

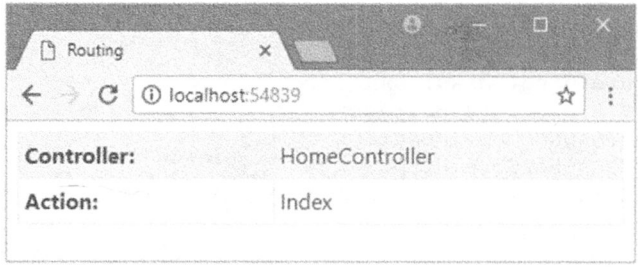


Рис. 15.5. Применение стандартных значений для расширения области действия маршрута

Использование статических сегментов URL

Не все сегменты в шаблоне URL должны быть переменными. Допускается также создавать шаблоны, имеющие *статические сегменты*. Предположим, что приложение нуждается в сопоставлении с URL, которые снабжены префиксом `Public`, например:

```
http://mydomain.com/Public/Home/Index
```

Это можно сделать с применением шаблона URL, подобного показанному в листинге 15.13.

Листинг 15.13. Использование статических сегментов в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}");

        routes.MapRoute(name: "",
          template: "Public/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

Новый шаблон будет соответствовать только URL, содержащим три сегмента, первым из которых *должен* быть `Public`. Остальные два сегмента могут содержать любые значения, и они будут использоваться для переменных `controller` и `action`. Если последние два сегмента не указаны, тогда будут применяться стандартные значения.

Можно также создавать шаблоны URL, которые имеют сегменты, содержащие как статические, так и переменные элементы, вроде шаблона в листинге 15.14.

Листинг 15.14. Смешивание сегментов в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute("", "X{controller}/{action}");

        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}");

        routes.MapRoute(name: "",
          template: "Public/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

Шаблон в показанном маршруте соответствует любому двухсегментному URL, в котором первый сегмент начинается с буквы `X`. Значение для `controller` берется из первого сегмента, исключая `X`. Значение для `action` получается из второго сегмента. Запустив приложение и перейдя на URL вида `/XHome/Index`, можно увидеть эффект от добавления нового маршрута (рис. 15.6).

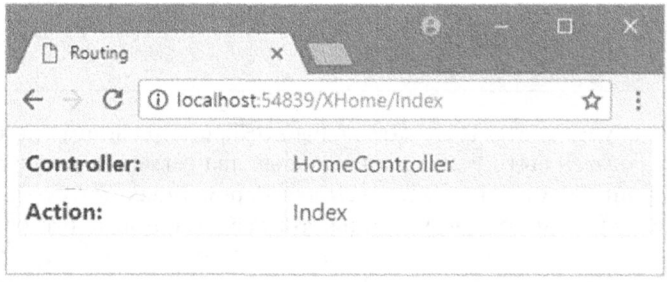


Рис. 15.6. Смешивание в одном сегменте статических и переменных элементов

Упорядочение маршрутов

В листинге 15.14 новый маршрут был определен и помещен перед другими маршрутами. Я поступил так потому, что маршруты применяются в порядке, в котором они определены. Метод `MapRoute()` добавляет маршрут в конец конфигурации маршрутизации, а это означает, что обычно маршруты применяются в порядке их определения. Формулировка "обычно" объясняется тем, что есть методы, которые позволяют вставлять маршруты в специфические позиции. Я стараюсь не использовать такие методы, поскольку применение маршрутов в порядке их определения упрощает понимание маршрутизации в приложении.

Система маршрутизации пытается сопоставить входящий URL с шаблоном URL маршрута, который был определен первым, и продолжает сопоставление со следующим маршрутом только в случае, если не произошло совпадение. Маршруты проверяются последовательно, пока не обнаружится соответствие или не исчерпается набор маршрутов. По указанной причине самые специфичные маршруты должны быть определены первыми. Маршрут, добавленный в листинге 15.14, является более специфичным, чем следующие за ним маршруты. Предположим, что порядок следования маршрутов изменен на обратный:

```
...
routes.MapRoute("MyRoute", "{controller=Home}/{action=Index}");
routes.MapRoute("", "X{controller}/{action}");
...
```

Тогда первый маршрут, который соответствует *любому* URL с нулем, одним или двумя сегментами, будет использоваться всегда. Более специфичный маршрут, который теперь второй в списке, никогда не будет достигнут. Новый маршрут исключает ведущую букву `X` из URL, но это не будет сделано из-за совпадения со старым маршрутом. Таким образом, URL вида:

```
http://mydomain.com/XHome/Index
```

будет нацелен на контроллер по имени `XHome`, предполагая существование в приложении класса `XHomeController` и наличие в нем метода действия `Index()`.

Статические сегменты URL и стандартные значения можно комбинировать с целью создания псевдонима для специфического URL. После развертывания приложения применяемая схема URL формирует контракт с пользователями, и если в будущем производится рефакторинг приложения, то предыдущий формат URL необходимо предохранить, чтобы любые URL, добавленные в закладки, макросы или сценарии, написанные пользователем, продолжили работать.

Представим, что у нас использовался контроллер по имени `Shop`, который теперь заменен контроллером `Home`. В листинге 15.15 показано, как создать маршрут для предохранения старой схемы URL.

Листинг 15.15. Сегменты и стандартные значения в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "ShopSchema",
          template: "Shop/{action}",
          defaults: new { controller = "Home" });

        routes.MapRoute("", "X{controller}/{action}");

        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}");

        routes.MapRoute(name: "",
          template: "Public/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

Добавленный маршрут соответствует любому двухсегментному URL, в котором первым сегментом является `Shop`. Значение `action` берется из второго сегмента URL. Шаблон URL не содержит переменного сегмента для `controller`, поэтому выбирается стандартное значение. Аргумент `defaults` предоставляет значение `controller`, потому что отсутствует какой-либо сегмент, к которому можно было бы применить значение как часть шаблона URL.

В результате запрос действия из контроллера `Shop` транслируется в запрос для контроллера `Home`. Запустив приложение и перейдя на URL вида `/Shop/Index`, можно оценить эффект от добавления такого маршрута. Как видно на рис. 15.7, новый маршрут приводит к тому, что инфраструктура MVC вызывает метод действия `Index()` из контроллера `Home`.

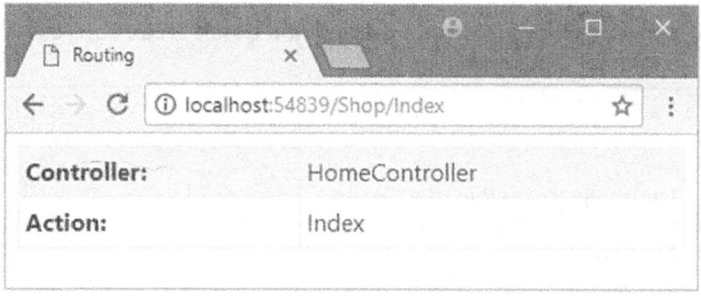


Рис. 15.7. Создание псевдонима для предохранения схемы URL

Можно продвинуться на шаг дальше и создать псевдонимы также для методов действий, которые в результате рефакторинга перестали существовать в контроллере. Чтобы сделать это, понадобится просто создать статический URL и предоставить стандартные значения для `controller` и `action` (листинг 15.16).

Листинг 15.16. Создание псевдонима для контроллера и действия в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {

        routes.MapRoute(
          name: "ShopSchema2",
          template: "Shop/OldAction",
          defaults: new { controller = "Home", action = "Index" });

        routes.MapRoute(
          name: "ShopSchema",
          template: "Shop/{action}",
          defaults: new { controller = "Home" });

        routes.MapRoute("", "X{controller}/{action}");

        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}");

        routes.MapRoute(name: "",
          template: "Public/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

Обратите внимание, что новый маршрут определен первым, поскольку он является более специфичным, чем маршруты, следующие после него. Если запрос для `Shop/OldAction` обработается, скажем, следующим определенным маршрутом, тогда может быть получен результат, отличающийся от того, который желательно получить при наличии контроллера с методом действия `OldAction()`.

Определение специальных переменных сегментов

Переменные сегментов `controller` и `action` имеют специальное предназначение в приложениях MVC и соответствуют контроллеру и методу действия, которые будут использоваться для обслуживания запроса. Они являются всего лишь встроенными переменными сегментов, и допускается также определять специальные переменные сегментов, как показано в листинге 15.17. (Маршруты, определенные в предыдущем разделе, удалены, чтобы можно было начать сначала.)

Листинг 15.17. Определение дополнительных переменных внутри шаблона URL в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(name: "MyRoute",
          template: "{controller=Home}/{action=Index}/{id=DefaultId}");
      });
    }
  }
}
```

Шаблон URL определяет стандартные переменные `controller` и `action` плюс специальную переменную по имени `id`. Маршрут будет соответствовать URL с количеством сегментов от нуля до трех. Содержимое третьего сегмента присваивается переменной `id`, а при отсутствии третьего сегмента применяется стандартное значение.

Внимание! Некоторые имена зарезервированы и не доступны для использования в качестве имен специальных переменных сегментов. К ним относятся `controller`, `action` и `area`. Назначение первых двух очевидно, а *области* (area) обсуждаются в следующей главе.

В классе `Controller`, который является базовым для контроллеров, определено свойство `RouteData`. Оно возвращает объект `Microsoft.AspNetCore.Routing.RouteData`, предоставляющий детали о системе маршрутизации и о способе, ко-

торым маршрутизировался текущий запрос. Внутри контроллера можно получать доступ к любой переменной сегмента в методе действия с применением свойства `RouteData.Values`, которое возвращает словарь, содержащий переменные сегмента. В целях демонстрации в контроллер `Home` добавлен метод действия по имени `CustomVariable()`, приведенный в листинге 15.18.

Листинг 15.18. Доступ к специальной переменной сегмента внутри метода действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(HomeController),
        Action = nameof(Index)
      });
    public ViewResult CustomVariable() {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(CustomVariable),
      };
      r.Data["Id"] = RouteData.Values["id"];
      return View("Result", r);
    }
  }
}
```

Метод действия `CustomVariable()` получает значение специальной переменной `id` в шаблоне URL маршрута, используя свойство `RouteData.Values`, которое возвращает словарь переменных, порожденных системой маршрутизации. Специальная переменная добавляется к объекту модели представления и может быть просмотрена путем запуска приложения и запроса следующего URL:

```
/Home/CustomVariable/Hello
```

Шаблон маршрута распознает третий сегмент в данном URL как значение для переменной `id`, давая приведенный на рис. 15.8 результат.

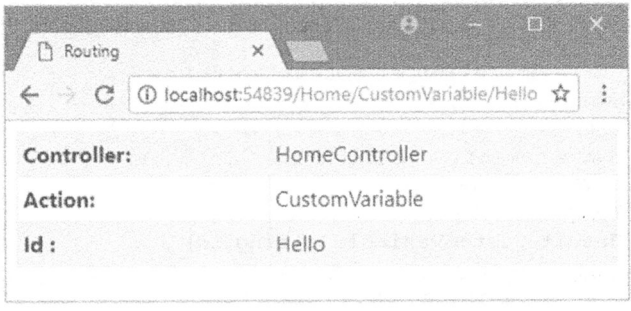


Рис. 15.8. Отображение значения специальной переменной сегмента

Шаблон URL в листинге 15.17 определяет стандартное значение для сегмента id, т.е. маршрут может также соответствовать URL, которые имеют два сегмента. Увидеть применение стандартного значения можно, запросив такой URL:

```
/Home/CustomVariable
```

Система маршрутизации использует стандартное значение для специальной переменной, как показано на рис. 15.9.

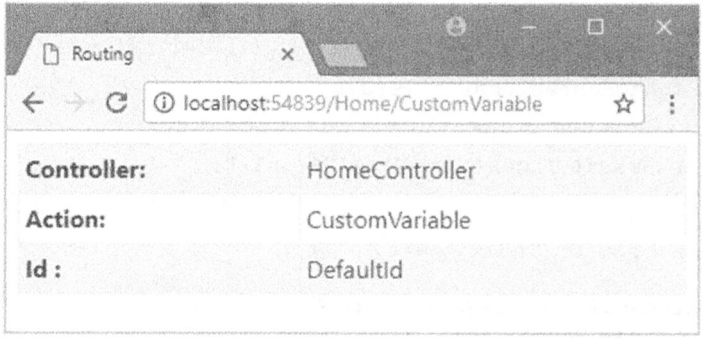


Рис. 15.9. Стандартное значение для специальной переменной сегмента

Использование специальных переменных в качестве параметров метода действия

Работа с коллекцией RouteData.Values — лишь один способ доступа к специальным переменным маршрута; другой способ может оказаться намного более элегантным. Если метод действия определяет параметры с именами, которые совпадают с именами переменных шаблона URL, то инфраструктура MVC будет автоматически передавать методу действия значения, извлеченные из URL, в виде аргументов.

Специальная переменная, определенная в маршруте из листинга 15.17, имела имя id. Модифицируем метод действия CustomVariable() в контроллере Home, чтобы он принимал параметр с тем же самым именем (листинг 15.19).

Листинг 15.19. Добавление параметра к действию в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class HomeController : Controller {
    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(HomeController),
        Action = nameof(Index)
    });
    public ViewResult CustomVariable(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(CustomVariable),
      };
```

```
            r.Data["Id"] = id;
            return View("Result", r);
        }
    }
}
```

Когда система маршрутизации обнаруживает соответствие какого-то URL маршруту, который был определен в листинге 15.17, значение третьего сегмента URL присваивается специальной переменной `id`. Инфраструктура MVC сравнивает список переменных сегментов со списком параметров метода действия, и в случае совпадения имен передает значения из URL данному методу.

Типом параметра `id` является `string`, но MVC будет пытаться преобразовать значение из URL в любой тип, указанный для параметра. Если параметр `id` метода действия объявлен как `int` или `DateTime`, тогда полученное значение из URL окажется преобразованным в экземпляр этого типа. Такое элегантное и удобное средство избавляет от необходимости обрабатывать преобразования самостоятельно. Запустив приложение и запросив URL вида `/Home/CustomVariable/Hello`, можно увидеть эффект от параметра метода действия (рис. 15.10). Если опустить третий сегмент, тогда методу действия будет предоставлено стандартное значение для переменной сегмента, которое также видно на рис. 15.10.

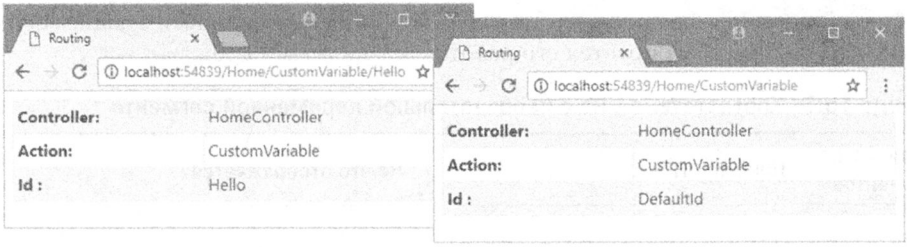


Рис. 15.10. Доступ к переменным сегментов с применением параметров метода действия

На заметку! Для преобразования значений, содержащихся внутри URL, в типы .NET инфраструктура MVC использует средство *привязки моделей*, которое может обрабатывать намного более сложные ситуации, чем продемонстрированная в приведенном выше примере. Привязка моделей будет описана в главе 26.

Определение необязательных сегментов URL

Необязательным является такой сегмент URL, который пользователь может не указывать и для которого не предусмотрено стандартного значения. Необязательный сегмент обозначается знаком вопроса (символом ?) после имени сегмента, как показано в листинге 15.20.

Листинг 15.20. Определение необязательного сегмента URL в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(name: "MyRoute",
        template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Данный маршрут будет соответствовать URL, имеющим или не имеющим сегмента `id`. В табл. 15.5 иллюстрируется его работа с различными URL.

Таблица 15.5. Соответствие URL с необязательной переменной сегмента

Количество сегментов	Пример URL	На что отображается
0	/	controller=Home, action=Index
1	/Customer	controller=Customer, action=Index
2	/Customer/List	controller=Customer, action=List
3	/Customer/List/All	controller=Customer, action=List id=All
4	/Customer/List/All/Delete	Соответствия нет — сегментов слишком много

В табл. 15.5 видно, что переменная `id` добавляется к набору переменных, только когда во входящем URL присутствует соответствующий сегмент. Такая возможность удобна, когда необходимо узнать, предоставил ли пользователь значение для переменной сегмента. Если для необязательной переменной сегмента значение не было задано, тогда значением соответствующего параметра будет `null`. В листинге 15.21 контроллер `Home` обновлен с целью учета ситуации, когда для переменной сегмента `id` значение не указано.

Листинг 15.21. Проверка необязательной переменной сегмента в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
```

```
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(HomeController),
        Action = nameof(Index)
    });

    public ViewResult CustomVariable(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(CustomVariable),
      };
      r.Data["Id"] = id ?? "<no value>";
      return View("Result", r);
    }
  }
}
```

На рис. 15.11 приведен результат запуска приложения и перехода по URL вида `/Home/CustomVariable`, в котором не включено значение для переменной сегмента `id`.

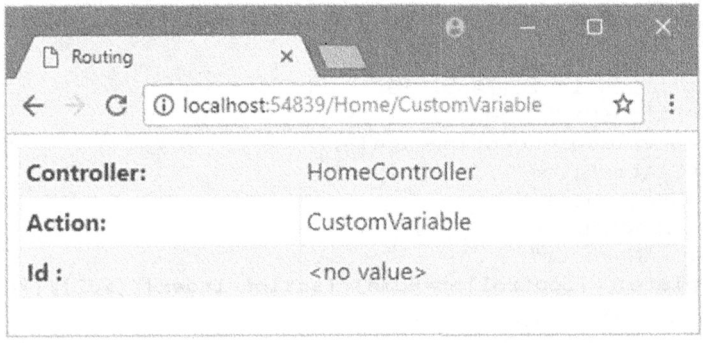


Рис. 15.11. Обнаружение ситуации, когда URL не содержит значения для необязательной переменной сегмента

Стандартная конфигурация маршрутизации

Инфраструктура MVC добавляется в классе `Startup` с применением метода `UseMvcWithDefaultRoute()`, который является просто удобным методом для настройки наиболее распространенной конфигурации маршрутизации и эквивалентен следующему коду:

```
...
app.UseMvc(routes => {
  routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
});
...
```

Такая стандартная конфигурация обеспечивает сопоставление с URL, которые нацелены на классы контроллеров и методы действий по их именам, а также могут содержать необязательный сегмент `id`. Если сегмент `controller` или `action` опущен, тогда используются стандартные значения, чтобы направить запрос контроллеру `Home` и методу действия `Index()`.

Определение маршрутов переменной длины

Еще один способ изменения консервативного стандартного поведения шаблонов URL предусматривает прием переменного количества сегментов URL. Это позволяет маршрутизировать URL произвольной длины в единственном маршруте. Поддержка переменного числа сегментов определяется путем назначения одной из переменных сегментов в качестве переменной *общего захвата*, для чего она предваряется символом звездочки (листинг 15.22).

Листинг 15.22. Назначение переменной общего захвата в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller=Home}/{action=Index}/{id?}/{*catchall}");
      });
    }
  }
}
```

Здесь был расширен маршрут из предыдущего примера за счет добавления переменной общего захвата по имени `catchall`. Теперь такой маршрут будет соответствовать *любому* URL вне зависимости от количества содержащихся в нем сегментов либо их значений. Первые три сегмента применяются для установки значений переменных `controller`, `action` и `id`. Если URL содержит дополнительные сегменты, то все они присваиваются переменной `catchall`, как показано в табл. 15.6.

В листинге 15.23 приведен обновленный контроллер `Customer`, в котором действие `List` передает представлению значение переменной `catchall` через объект модели.

Таблица 15.6. Сопоставление URL с переменной общего захвата

Количество сегментов	Пример URL	На что отображается
0	/	controller=Home, action=Index
1	/Customer	controller=Customer, action=Index
2	/Customer/List	controller=Customer, action=List
3	/Customer/List/All	controller=Customer, action=List, id=All
4	/Customer/List/All/Delete	controller=Customer, action=List, id=All, catchall=Delete
5	/Customer/List/All/Delete/Perm	controller=Customer, action=List, id=All, catchall=Delete/Perm

Листинг 15.23. Модификация метода действия в файле `CustomerController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class CustomerController : Controller {

    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(CustomerController),
        Action = nameof(Index)
    });

    public ViewResult List(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(List),
      };
      r.Data["Id"] = id ?? "<no value>";
      r.Data["catchall"] = RouteData.Values["catchall"];
      return View("Result", r);
    }
  }
}
```

Чтобы протестировать сегмент общего захвата, понадобится запустить приложение и запросить следующий URL:

```
/Customer/List/Hello/1/2/3
```

Верхнего предела количества сегментов, для которого шаблон URL в этом маршруте будет давать совпадение, не существует. На рис. 15.12 показан эффект от опре-

деления сегмента общего захвата. Обратите внимание, что сегменты, попадающие в переменную общего захвата, представлены в форме *сегмент/сегмент/сегмент*, и мы сами отвечаем за обработку строки для ее разбиения на отдельные сегменты.

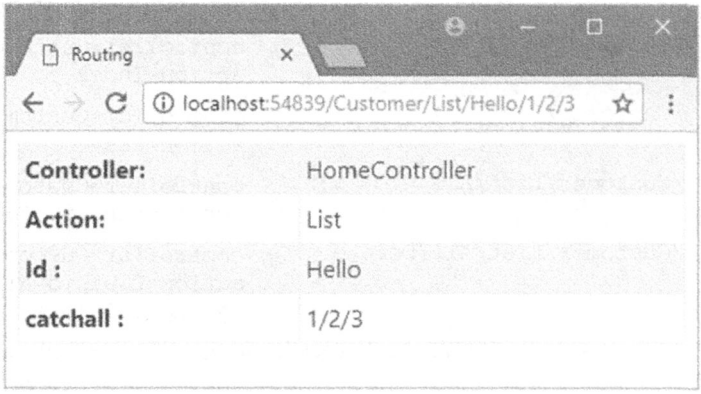


Рис. 15.12. Использование сегмента общего захвата

Ограничение маршрутов

В начале главы упоминалось, что шаблоны URL консервативны в отношении количества сопоставляемых сегментов и либеральны в отношении содержимого сопоставляемых сегментов. В нескольких предшествующих разделах объяснялись различные приемы управления степенью консервативности: увеличение или уменьшение числа сегментов с применением стандартных значений, необязательных переменных и т.д.

Теперь наступило время взглянуть на то, каким образом управлять либеральностью при сопоставлении с *содержимым* сегментов URL, а именно — как ограничить набор URL, с которыми будет сопоставляться маршрут. В листинге 15.24 демонстрируется использование простого ограничения URL, с которыми будет сопоставляться маршрут.

Листинг 15.24. Ограничение маршрута в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller=Home}/{action=Index}/{id:int?}");
      });
    }
  }
}
```

Ограничения отделяются от имени переменной сегмента двоеточием (символ :). Ограничением в листинге является `int`, и оно применяется к сегменту `id`. Это пример *встраиваемого ограничения*, которое определено как часть шаблона URL и применено к одиночному сегменту:

```
...
template: "{controller}/{action}/{id:int?}",
...
```

Ограничение `int` разрешает сопоставление шаблона URL только с сегментами, значение которых может быть преобразовано в тип `int`. Сегмент `id` необязателен, так что маршрут будет сопоставляться с URL без сегмента `id`, но если он присутствует, то должен быть целочисленным значением (в табл. 15.7 приведены примеры).

Таблица 15.7. Сопоставление URL с ограничением

Пример URL	На что отображается
/	`controller=Home, action=Index, id=null`
/Home/CustomVariable/Hello	Соответствия нет — сегмент `id` не может быть преобразован в значение `int`
/Home/CustomVariable/1	`controller=Home, action=CustomVariable, id=1`
/Home/CustomVariable/1/2	Соответствия нет — сегментов слишком много

Ограничения также могут быть указаны за пределами шаблона URL с использованием аргумента `constraints` метода `MapRoute()` при определении маршрута. Такой прием удобен, если предпочтение отдается хранению шаблона URL отдельно от ограничений или соблюдению стиля маршрутизации, принятого в более ранних версиях MVC, где встраиваемые ограничения не поддерживались. В листинге 15.25 показано то же самое ограничение `int` на переменной сегмента `id`, выраженное с применением отдельного ограничения. При использовании такого формата стандартные значения также выражаются внешне.

Листинг 15.25. Выражение ограничения за пределами шаблона URL в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
namespace UrlsAndRoutes {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller}/{action}/{id?}",
          defaults: new { controller = "Home", action = "Index" },
          constraints: new { id = new IntRouteConstraint() });
      });
    }
  }
}
```

Аргумент `constraints` метода `MapRoute()` определен с применением анонимного типа, имена свойств которого соответствуют ограничиваемым переменным сегментов. Пространство имен `Microsoft.AspNetCore.Routing.Constraints` содержит набор классов, которые можно использовать для определения индивидуальных ограничений. В листинге 15.25 аргумент `constraints` настроен на применение объекта `IntRouteConstraint` для сегмента `id`, что дает такой же результат, как встраиваемое ограничение из листинга 15.24.

В табл. 15.8 описан полный набор классов ограничений в пространстве имен `Microsoft.AspNetCore.Routing.Constraints` вместе с их встраиваемыми эквивалентами для ограничений, которые могут применяться к одиночным элементам в шаблоне URL; часть из них будет рассматриваться в последующих разделах.

Таблица 15.8. Ограничения маршрутов на уровне сегментов

Встраиваемое ограничение	Описание	Конструктор класса
`alpha`	Соответствует алфавитным символам независимо от регистра (A–Z, a–z)	`AlphaRouteConstraint()`
`bool`	Соответствует значению, которое может быть преобразовано в тип `bool`	`BoolRouteConstraint()`
`datetime`	Соответствует значению, которое может быть преобразовано в тип `DateTime`	`DateTimeRouteConstraint()`

Окончание табл. 15.8

Встраиваемое ограничение	Описание	Конструктор класса
`decimal`	Соответствует значению, которое может быть преобразовано в тип `decimal`	`DecimalRouteConstraint()`
`double`	Соответствует значению, которое может быть преобразовано в тип `double`	`DoubleRouteConstraint()`
`float`	Соответствует значению, которое может быть преобразовано в тип `float`	`FloatRouteConstraint()`
`guid`	Соответствует значению глобально уникального идентификатора (globally unique identifier — GUID)	`GuidRouteConstraint()`
`int`	Соответствует значению, которое может быть преобразовано в тип `int`	`IntRouteConstraint()`
`length(len)` `length(min,max)`	Соответствует строковому значению, которое содержит указанное количество символов или число символов между `min` и `max` (включительно)	`LengthRouteConstraint(len)` `LengthRouteConstraint(min, max)`
`long`	Соответствует значению, которое может быть преобразовано в тип `long`	`LongRouteConstraint()`
`maxlength(len)`	Соответствует строковому значению с количеством символов не более `len`	`MaxLengthRouteConstraint(len)`
`max(val)`	Соответствует значению `int`, если оно меньше `val`	`MaxRouteConstraint(val)`
`minlength(len)`	Соответствует строковому значению, которое имеет, по крайней мере, `len` символов	`MinLengthRouteConstraint(len)`
`min(val)`	Соответствует значению `int`, если оно больше `val`	`MinRouteConstraint(val)`
`range(min,max)`	Соответствует значению `int`, если оно находится между `min` и `max` (включительно)	`RangeRouteConstraint(min, max)`
`regex(expr)`	Соответствует регулярному выражению	`RegexRouteConstraint(expr)`

Совет. Можно ограничить доступ к методам действий со стороны запросов, которые сделаны с помощью специфических HTTP-команд, таких как `GET` или `POST`, используя набор атрибутов MVC наподобие `HttpGet` и `HttpPost`. В главе 17 описано применение этих атрибутов для обработки форм в контроллерах, а в главе 20 приведен полный список доступных атрибутов.

Ограничение маршрута с использованием регулярного выражения

Наибольшую гибкость предлагает ограничение `regex`, которое сопоставляет сегмент с применением регулярного выражения. В листинге 15.26 сегмент `controller` ограничивается диапазоном URL, с которыми он будет сопоставляться.

Листинг 15.26. Использование регулярного выражения для ограничения маршрута в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller:regex(^H.*)=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Примененное к маршруту ограничение обеспечивает его соответствие только таким URL, в которых сегмент `controller` начинается с буквы `H`.

На заметку! Стандартные значения используются перед проверкой ограничений. Таким образом, например, при запросе URL вида `/` для переменной сегмента `controller` применяется стандартное значение `Home`. Затем проверяются ограничения, и поскольку значение `controller` начинается с `H`, стандартный URL будет соответствовать маршруту.

С помощью регулярных выражений можно ограничивать маршрут так, что к совпадению будут приводить только специфические значения для какого-то сегмента URL. Это делается с использованием символа |, как показано в листинге 15.27. (Шаблон URL разбит на две части, чтобы уместиться на печатной странице, о чем заботиться в реальном проекте нет нужды.)

Листинг 15.27. Ограничение маршрута специфическим набором значений переменной сегмента в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller:regex(^H.*)=Home}/"
            + "{action:regex(^Index$|^About$)=Index}/{id?}");
      });
    }
  }
}
```

Такое ограничение разрешает маршруту соответствовать только URL, которые имеют сегмент `action` со значением `Index` или `About`. Ограничения применяются совместно, так что ограничения, наложенные на значение переменной `action`, комбинируются с ограничениями, наложенными на значение переменной `controller`. Другими словами, маршрут в листинге 15.27 будет соответствовать только таким URL, в которых значение переменной `controller` начинается с буквы `H`, а значением переменной `action` является `Index` или `About`.

Использование ограничений на основе типов и значений

Большинство ограничений применяются для ограничения маршрутов, чтобы они соответствовали только URL с сегментами, значения которых могут быть преобразованы в указанный тип либо имеют специфический формат. Хорошим примером может служить ограничение `int`, использованное в начале этого раздела: маршруты будут

соответствовать, только когда значение ограниченного сегмента может быть преобразовано в значение .NET-типа `int`. В листинге 15.28 демонстрируется применение ограничения `range`, которое ограничивает маршрут так, что он соответствует URL, только если значение сегмента может быть преобразовано в `int` и находится между указанными значениями.

Листинг 15.28. Ограничение на основе типа и значения в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller=Home}/{action=Index}/
{id:range(10,20)?}");
      });
    }
  }
}
```

Ограничение в рассмотренном примере применялось к необязательному сегменту `id`. Ограничение будет проигнорировано, если URL запроса не содержит, по крайней мере, три сегмента. Если сегмент `id` присутствует, то маршрут будет соответствовать URL, только когда значение сегмента может быть преобразовано в `int` и попадает в диапазон между 10 и 20. Ограничение `range` является включающим, т.е. значения 10 и 20 считаются входящими в диапазон.

Объединение ограничений

Если необходимо применить к одиночному элементу сразу несколько ограничений, тогда их можно соединить в цепочку, отделяя каждое ограничение от других двоеточием (листинг 15.29).

Листинг 15.29. Объединение встраиваемых ограничений в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller=Home}/{action=Index}"
            + "/{id:alpha:minlength(6)?}");
      });
    }
  }
}
```

В листинге 15.29 к сегменту `id` были применены ограничения `alpha` и `minlength`. Знак вопроса, обозначающий необязательный сегмент, учитывается после всех ограничений. В результате объединения ограничений маршрут будет соответствовать URL, только когда сегмент `id` опущен (поскольку он необязателен) или когда он присутствует и содержит, по крайней мере, шесть алфавитных символов.

Если вы не используете встраиваемые ограничения, то должны применять класс `Microsoft.AspNetCore.Routing.CompositeRouteConstraint`, который позволяет ассоциировать несколько ограничений с одиночным свойством в анонимно типизированном объекте. В листинге 15.30 показана комбинация ограничений, которые использовались в листинге 15.29.

Листинг 15.30. Объединение отдельных ограничений в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller}/{action}/{id?}",
          defaults: new { controller = "Home", action = "Index" },
          constraints: new {
            id = new CompositeRouteConstraint(
              new IRouteConstraint[] {
                new AlphaRouteConstraint(),
                new MinLengthRouteConstraint(6)
              })
          });
      });
    }
  }
}
```

Конструктор класса `CompositeRouteConstraint` принимает перечисление объектов, которые реализуют интерфейс `IRouteConstraint`, определяющий ограничения маршрутов. Система маршрутизации разрешит соответствие маршрута URL, только если все ограничения удовлетворены.

Определение специального ограничения

Если для обеспечения текущих потребностей стандартных ограничений не хватает, тогда можно определить собственные ограничения, реализовав интерфейс `IRouteConstraint`, который находится в пространстве имен `Microsoft.AspNetCore.Routing`. Чтобы проверить такую возможность, добавим в пример проекта папку `Infrastructure` и поместим в нее новый файл класса по имени `WeekDayConstraint.cs` с содержимым из листинга 15.31.

Листинг 15.31. Содержимое файла `WeekDayConstraint.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System.Linq;

namespace UrlsAndRoutes.Infrastructure {
```

```
  public class WeekDayConstraint : IRouteConstraint {
    private static string[] Days = new[] { "mon", "tue", "wed", "thu",
                                           "fri", "sat", "sun" };

    public bool Match(HttpContext httpContext, IRouter route,
          string routeKey, RouteValueDictionary values,
          RouteDirection routeDirection) {
      return Days.Contains(values[routeKey]?.ToString().ToLowerInvariant());
    }
  }
}
```

В интерфейсе `IRouteConstraint` определен метод `Match()`, который вызывается, чтобы позволить ограничению решить, должен ли запрос соответствовать маршруту. Параметры метода `Match()` предоставляют доступ к запросу, поступившему от клиента, маршруту, имени ограничиваемого сегмента, переменным сегментов, извлеченным из URL, и признаку того, для какого URL проверяется запрос — входящего или исходящего (исходящие URL рассматриваются в главе 16).

В этом примере с помощью параметра `routeKey` из параметра `values` извлекается значение переменной сегмента, к которому было применено ограничение, преобразуется в строку нижнего регистра и проверяется на предмет соответствия одному из дней недели, определенных в статическом поле `Days`. В листинге 15.32 новое ограничение применяется к маршруту с использованием синтаксиса отдельно определяемых ограничений.

Листинг 15.32. Применение специального ограничения в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller}/{action}/{id?}",
```

```
            defaults: new { controller = "Home", action = "Index" },
            constraints: new { id = new WeekDayConstraint() });
        });
    }
  }
}
```

В итоге маршрут будет соответствовать URL, только если сегмент `id` отсутствует (например, `/Customer/List`) или совпадает с одним из дней недели, которые определены в классе ограничения (скажем, `/Customer/List/Fri`).

Определение встраиваемого специального ограничения

Настройка специального ограничения так, чтобы оно могло использоваться встраиваемым образом, требует дополнительного шага по конфигурированию (листинг 15.33).

Листинг 15.33. Применение специального ограничения встраиваемым образом в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options =>
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "MyRoute",
          template: "{controller=Home}/{action=Index}/{id:weekday?}");
      });
    }
  }
}
```

В методе ConfigureService() конфигурируется объект RouteOptions, который управляет рядом линий поведения системы маршрутизации. Свойство ConstraintMap возвращает словарь, используемый для трансляции имен встраиваемых ограничений в классы реализации IRouteConstraint, которые предоставляют логику ограничений. В словарь было добавлено новое отображение, поэтому на класс WeekDayConstraint можно ссылаться встраиваемым образом как на weekday:

```
...
template: "{controller=Home}/{action=Index}/{id:weekday?}",
...
```

Результат будет тем же, но настройка отображения позволяет применять специальный класс ограничения встраиваемым образом.

Использование маршрутизации с помощью атрибутов

Во всех примерах, приведенных до сих пор в главе, маршруты определялись посредством приема, который называется *маршрутизацией на основе соглашений*. Инфраструктура MVC также поддерживает прием, известный как *маршрутизация с помощью атрибутов*, когда маршруты определяются через атрибуты C#, которые применяются напрямую к классам контроллеров. В последующих разделах будет показано, как создавать и конфигурировать маршруты с использованием атрибутов, которые могут свободно смешиваться с маршрутами на основе соглашений, продемонстрированными в предшествующих примерах.

Подготовка для маршрутизации с помощью атрибутов

Маршрутизация с помощью атрибутов включается при вызове метода UseMvc() в файле Startup.cs. Инфраструктура MVC исследует классы контроллеров в приложении, находит любые из них, имеющие атрибуты маршрутизации, и создает нужные маршруты.

Для целей текущего раздела мы вернем стандартную конфигурацию маршрутизации в примере приложения, которая была описана во врезке "Стандартная конфигурация маршрутизации" ранее в главе (листинг 15.34).

Листинг 15.34. Применение стандартной конфигурации в файле Startup.cs из папки UrlsAndRoutes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
  public class Startup {
```

```
    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options =>
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Стандартный маршрут будет сопоставляться с URL, используя следующий шаблон:

```
{controller}/{action}/{id?}
```

Применение маршрутизации с помощью атрибутов

Атрибут `Route` используется при указании маршрутов для индивидуальных контроллеров и действий. В листинге 15.35 атрибут `Route` применяется к классу `CustomerController`.

Листинг 15.35. Применение атрибута `Route` в файле `CustomerController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {
  public class CustomerController : Controller {
    [Route("myroute")]
    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(CustomerController),
        Action = nameof(Index)
      });
    public ViewResult List(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(List),
      };
      r.Data["Id"] = id ?? "<no value>";
      r.Data["catchall"] = RouteData.Values["catchall"];
      return View("Result", r);
    }
  }
}
```

Атрибут `Route` определяет маршрут к методу действия или контроллеру, к которому он применен. В листинге данный атрибут применен к методу действия `Index()` с указанием `myroute` в качестве маршрута, который должен использоваться. Результатом будет изменение набора маршрутов, которые применяются для достижения методов действий, определенных контроллером `Customer` (табл. 15.9).

Таблица 15.9. Маршруты для контроллера `Customer`

Маршрут	Описание
`/Customer/List`	Этот URL нацелен на метод действия `List()`, полагаясь на стандартный маршрут в файле `Startup.cs`
`/myroute`	Этот URL нацелен на метод действия `Index()`

Следует отметить два важных момента. Во-первых, когда используется атрибут `Route`, предоставленное для его конфигурирования значение применяется при определении полного маршрута, так что `myroute` становится завершенным URL для достижения метода действия `Index()`. Во-вторых, присутствие атрибута `Route` предотвращает использование стандартной конфигурации маршрутизации, поэтому метод действия `Index()` больше не будет достижимым с применением URL вида `/Customer/Index`.

Изменение имени метода действия

Определение уникального маршрута для одиночного метода действия в большинстве приложений не особенно полезно, но атрибут `Route` можно использовать более гибким образом. В листинге 15.36 внутри маршрута применяется специальный маркер `[controller]` для ссылки на контроллер и настройки базовой части маршрута.

Совет. Изменить имя действия можно также с помощью атрибута `ActionName`, который рассматривается в главе 31.

Листинг 15.36. Переименование действия в файле `CustomerController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class CustomerController : Controller {

    [Route("[controller]/MyAction")]
    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(CustomerController),
        Action = nameof(Index)
    });

    public ViewResult List(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(List),
      };
      r.Data["Id"] = id ?? "<no value>";
      r.Data["catchall"] = RouteData.Values["catchall"];
      return View("Result", r);
    }
  }
}
```

Применение маркера [controller] в аргументе для атрибута Route довольно похоже на использование операции nameof и позволяет указывать маршрут к контроллеру без жесткого кодирования имени класса. В табл. 15.10 описан эффект от наличия атрибута в листинге 15.36.

Таблица 15.10. Маршруты для контроллера Customer

Маршрут	Описание
/Customer/List	Этот URL нацелен на метод действия List()
/Customer/MyAction	Этот URL нацелен на метод действия Index()

Создание более сложного маршрута

Атрибут Route можно также применять к классу контроллера, позволяя определять структуру маршрута, как показано в листинге 15.37.

Листинг 15.37. Применение атрибута Route к контроллеру в файле CustomerController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  [Route("app/[controller]/actions/[action]/{id?}")]
  public class CustomerController : Controller {

    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(CustomerController),
        Action = nameof(Index)
      });

    public ViewResult List(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(List),
      };
      r.Data["Id"] = id ?? "<no value>";
      r.Data["catchall"] = RouteData.Values["catchall"];
      return View("Result", r);
    }
  }
}
```

В маршруте определена смесь статических и переменных сегментов, а также используются маркеры [controller] и [action] для ссылки на имена класса контроллера и методов действий. В табл. 15.11 описан эффект от создания такого маршрута.

Таблица 15.11. Маршруты для контроллера `Customer`

Маршрут	Описание
`app/customer/actions/index`	Этот URL нацелен на метод действия `Index()`
`app/customer/actions/index/myid`	Этот URL нацелен на метод действия `Index()` с необязательным сегментом `id`, установленным в `myid`
`app/customer/actions/list`	Этот URL нацелен на метод действия `List()`
`app/customer/actions/list/myid`	Этот URL нацелен на метод действия `List()` с необязательным сегментом `id`, установленным в `myid`

Применение ограничений к маршрутам

Маршруты, определенные посредством атрибутов, могут ограничиваться подобно маршрутам, которые определены в файле `Startup.cs`, с помощью того же самого встроенного подхода, используемого для маршрутов на основе соглашений. В листинге 15.38 созданное ранее в главе специальное ограничение применяется к необязательному сегменту `id`, определенному в атрибуте `Route`.

Листинг 15.38. Ограничение маршрута в файле `CustomerController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  [Route("app/[controller]/actions/[action]/{id:weekday?}")]
  public class CustomerController : Controller {
    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(CustomerController),
        Action = nameof(Index)
      });
    public ViewResult List(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(List),
      };
      r.Data["Id"] = id ?? "<no value>";
      r.Data["catchall"] = RouteData.Values["catchall"];
      return View("Result", r);
    }
  }
}
```

Можно использовать все ограничения, описанные в табл. 15.8, или, как продемонстрировано в листинге 15.38, применять специальные ограничения, которые были зарегистрированы с помощью службы `RouteOptions`. Чтобы применить сразу несколько ограничений, их понадобится соединить в цепочку, отделяя друг от друга двоеточиями.

Резюме

В главе подробно обсуждалась система маршрутизации. Вы увидели, как определяются маршруты по соглашениям и с помощью атрибутов, каким образом сопоставляются и обрабатываются входящие URL, и как настраиваются маршруты за счет изменения способа, которым они сопоставляются с сегментами URL, а также за счет использования стандартных значений и необязательных сегментов. Кроме того, было показано, каким образом ограничивать маршруты, чтобы сузить диапазон запросов, для которых они будут давать совпадение, с применением встроенных ограничений и специальных классов ограничений.

В следующей главе мы рассмотрим генерацию исходящих URL на основе маршрутов внутри представлений и выясним, как использовать средство *областей*, которое полагается на систему маршрутизации и может применяться для управления крупными и сложными приложениями MVC.

ГЛАВА 16

Дополнительные возможности маршрутизации

В предыдущей главе было показано, как использовать систему маршрутизации для обработки входящих URL, но это только часть общей картины. Мы также должны быть в состоянии применять схему URL для генерации *исходящих URL*, которые могут встраиваться в представления, чтобы пользователи имели возможность щелкать на ссылках и отправлять формы обратно приложению, нацеливаясь на корректный контроллер и действие.

В настоящей главе будут рассматриваться разнообразные приемы для генерации исходящих URL, демонстрироваться способы настройки системы маршрутизации за счет замены стандартных классов, реализующих маршрутизацию MVC, и объясняться, как использовать средство *областей* MVC, которое позволяет разбивать крупное и сложное приложение MVC на управляемые фрагменты. Глава завершится рядом советов по применению схем URL в приложениях MVC. В табл. 16.1 приведена сводка, позволяющая поместить дополнительные возможности маршрутизации в контекст.

Таблица 16.1. Помещение дополнительных возможностей маршрутизации в контекст

Вопрос	Ответ
Что это такое?	Система маршрутизации предоставляет возможности, которые выходят за рамки сопоставления с URL для HTTP-запросов. Имеется также поддержка для генерации URL в представлениях, замещения встроенной функциональности маршрутизации специальными классами и структуризации приложения в виде изолированных разделов
Чем они полезны?	Каждое средство полезно по своим соображениям. Наличие возможности генерирования URL облегчает изменение схемы URL без необходимости в обновлении всех представлений. Существование возможности использования специальных классов позволяет подстраивать систему маршрутизации под собственные требования. Наличие возможности структуризации приложения упрощает построение сложных проектов
Как они используются?	Ищите детальные сведения в разделах главы

Окончание табл. 16.1

Вопрос	Ответ
Существуют ли какие-то скрытые ловушки или ограничения?	Конфигурация маршрутизации для сложного приложения может стать трудной в управлении
Существуют ли альтернативы?	Нет. Система маршрутизации — неотъемлемая часть ASP.NET Core

В табл. 16.2 приведена сводка по главе.

Таблица 16.2. Сводка по главе

Задача	Решение	Листинг
Генерирование якорного элемента с URL	Используйте атрибуты `asp-action` и `asp-controller`	16.2–16.5
Предоставление значений для сегментов маршрутизации	Применяйте атрибуты с префиксом `asp-route-`	16.6, 16.7
Генерирование полностью заданных URL	Используйте атрибуты `asp-protocol`, `asp-host` и `asp-fragment`	16.8
Выбор маршрута для генерации URL	Применяйте атрибут `asp-route`	16.9, 16.10
Генерирование URL без HTML-элемента	Используйте вспомогательный метод `Url.Action()` в представлении или в методе действия	16.11, 16.12
Настройка системы маршрутизации	Применяйте метод `Configure()` в классе `Startup`	16.13
Создание специального класса маршрутизации	Реализуйте интерфейс `IRouter`	16.14–16.21
Разбиение приложения на функциональные разделы	Создайте области и используйте атрибут `Area`	16.22–16.28

Подготовка проекта для примера

Мы продолжим работать с проектом `UrlsAndRoutes` из предыдущей главы. Единственное изменение потребуется внести в класс `Startup`, где вызов метода `UseMvcWithDefaultRoute()` заменяется явным маршрутом с тем же самым результатом, как показано в листинге 16.1.

Листинг 16.1. Изменение конфигурации маршрутизации в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;
```

```
namespace UrlsAndRoutes {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options =>
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

После запуска приложения браузер запросит стандартный URL, который будет направлен на действие `Index` контроллера `Home` (рис. 16.1).

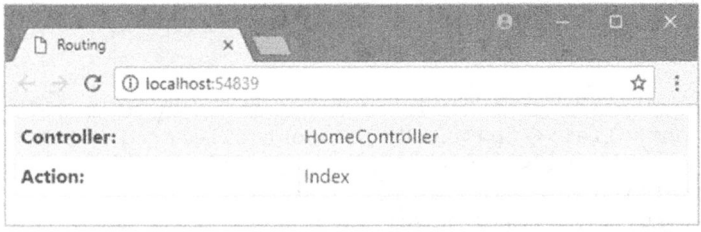


Рис. 16.1. Выполнение примера приложения

Генерирование исходящих URL в представлениях

Почти в каждом приложении MVC пользователи должны располагать возможностью перехода от одного представления к другому, что обычно предполагает включение в первое представление ссылки, нацеленной на метод действия, который генерирует второе представление. Было бы заманчиво просто добавить статический элемент `a` (известный как *якорный элемент*), указав в его атрибуте `href` нужный метод действия, например:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

Предполагая, что в приложении применяется стандартная конфигурация маршрутизации, такой HTML-элемент создает ссылку, которая будет нацелена на метод действия `CustomVariable()` в контроллере `Home`. Вручную определяемые URL подобного рода создавать быстро и просто. Но вместе с тем они чрезвычайно опасны: изменив схему URL для приложения, вы нарушите работу всех жестко закодированных URL. Затем придется пройтись по всем представлениям в приложении и обновить все ссылки на контроллеры и методы действий — утомительный, чреватый ошибками и

сложный в тестировании процесс. Более удачная альтернатива предусматривает использование для генерации исходящих URL системы маршрутизации, что обеспечивает применение схемы URL для динамического формирования URL способом, который гарантирует учет схемы URL приложения.

Генерирование исходящих ссылок

Сгенерировать исходящий URL в представлении проще всего с использованием вспомогательной функции дескриптора якоря, которая создаст атрибут `href` в HTML-элементе `a`, как иллюстрируется в листинге 16.2, в котором приведено добавление к представлению `/Views/Shared/Result.cshtml`.

Совет. Работа вспомогательных функций дескрипторов объясняется в главе 23.

Листинг 16.2. Использование вспомогательной функции дескриптора якоря в файле `Result.cshtml` из папки `Views/Shared`

```
@model Result
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
  <a asp-action="CustomVariable">This is an outgoing URL</a>
</body>
</html>
```

Атрибут `asp-action` применяется для указания имени метода действия, на который должен быть нацелен URL в атрибуте `href`. Результат запуска приложения можно увидеть на рис. 16.2.

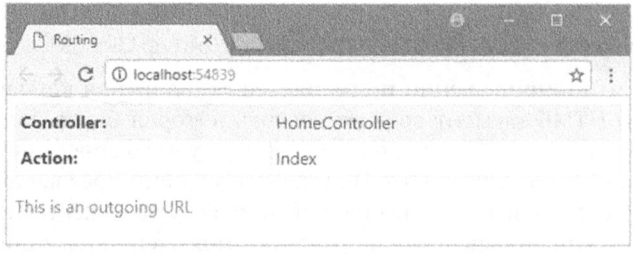


Рис. 16.2. Использование вспомогательной функции дескриптора для генерации ссылки

Вспомогательная функция дескриптора устанавливает атрибут `href` элемента `a` с применением текущей конфигурации маршрутизации. Просмотрев HTML-разметку, отправленную браузеру, легко заметить, что она содержит следующий элемент:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

Может показаться, что для воссоздания вручную определенного URL, который был приведен ранее, пришлось затратить слишком много дополнительных усилий, но преимущество данного подхода в том, что он автоматически реагирует на изменения в конфигурации маршрутизации. Чтобы продемонстрировать сказанное, добавим в файл `Startup.cs` новый маршрут (листинг 16.3).

Листинг 16.3. Добавление нового маршрута в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;
namespace UrlsAndRoutes {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options =>
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "NewRoute",
          template: "App/Do{action}",
          defaults: new { controller = "Home" });
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Новый маршрут изменяет схему URL для запросов, которые нацелены на контроллер `Home`. Запустив приложение, можно заметить, что это изменение отразилось в HTML-разметке, сгенерированной вспомогательной функцией дескриптора:

```
<a href="/App/DoCustomVariable">This is an outgoing URL</a>
```

Генерирование ссылок с использованием вспомогательной функции дескриптора решает важную проблему сопровождения. Мы имеем возможность изменять схему маршрутизации, а исходящие ссылки в представлениях автоматически отразят изменения без необходимости вручную редактировать представления в приложении.

Когда производится щелчок на ссылке, исходящий URL применяется для создания входящего HTTP-запроса, и тот же самый маршрут затем используется для нацеливания на метод действия и контроллер, которые будут обрабатывать запрос (рис. 16.3).

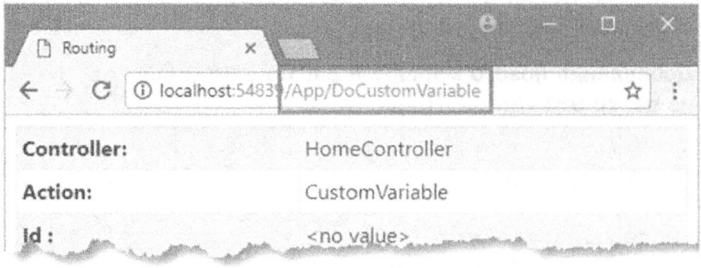


Рис. 16.3. Результатом щелчка на ссылке является применение исходящего URL для создания входящего запроса

Сопоставление исходящих URL с маршрутами

Вы уже видели, что изменение маршрутов, определяющих схему URL, изменяет способ генерации исходящих URL. В приложениях обычно определено множество маршрутов, поэтому важно понимать, как они выбираются для генерации URL. Система маршрутизации обрабатывает маршруты в порядке их определения, и каждый маршрут проверяется по очереди на предмет соответствия, что требует удовлетворения следующих трех условий.

- Для любой переменной сегмента, определенной в шаблоне URL, должно быть доступно значение. Чтобы найти значения для каждой переменной сегмента, система маршрутизации просматривает сначала предоставленные (посредством свойств анонимного типа) значения, затем значения переменных для текущего запроса и, наконец, стандартные значения, определенные в маршруте. (Позже в главе мы еще возвратимся ко второму источнику значений из числа упомянутых.)
- Ни одно из значений, предоставленных для переменных сегментов, не должно конфликтовать с определенными в маршруте переменными, которые имеют только стандартные значения. Это переменные, для которых были предоставлены стандартные значения, но которые не встречаются в шаблоне URL. Например, в показанном ниже определении маршрута `myVar` является переменной, имеющей только стандартное значение:

  ```
  routes.MapRoute("MyRoute", "{controller}/{action}",
    new { myVar = "true" });
  ```

  Чтобы соответствовать такому маршруту, значение для переменной `myVar` либо не должно предоставляться, либо должно совпадать со стандартным значением.
- Значения для всех переменных сегментов должны удовлетворять ограничениям маршрута. Примеры разных видов ограничений приводились в разделе "Ограничение маршрутов" предыдущей главы.

Следует уяснить, что система маршрутизации не пытается найти маршрут, который дает *наилучшее* совпадение. Она находит только *первое* совпадение и использует данный маршрут для генерирования URL; любые дальнейшие маршруты игнорируются. По указанной

причине наиболее специфичные маршруты должны определяться первыми. Важно проверить генерацию исходящих URL. Попытка сгенерировать URL, для которого не удается найти соответствующий маршрут, приведет к созданию ссылки с пустым атрибутом `href`:

```
<a href="">This is an outgoing URL</a>
```

Такая ссылка корректно визуализируется в представлении, но не будет функционировать ожидаемым образом, когда пользователь выполнит на ней щелчок. Если вы генерируете только URL (как будет показано позже в главе), тогда результирующим значением будет `null`, которое визуализируется в виде пустой строки в представлениях. С помощью именованных маршрутов можно получить определенный контроль над сопоставлением маршрутов. За более подробными сведениями обратитесь в раздел "Генерирование URL из специфического маршрута" далее в главе.

Направление на другие контроллеры

В случае указания атрибута `asp-action` внутри элемента `a` вспомогательная функция дескриптора предполагает, что нужно установить в качестве цели действие в том же контроллере, который вызвал визуализацию представления. Чтобы создать исходящий URL, который направляет на *другой* контроллер, можно применить атрибут `asp-controller` (листинг 16.4).

Листинг 16.4. Направление на другой контроллер в файле `Result.cshtml` из папки `Views/Shared`

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
  <a asp-controller="Admin" asp-action="Index">
    This targets another controller
  </a>
</body>
</html>
```

В результате визуализации представления сгенерируется следующая HTML-разметка:

```
<a href="/Admin">This targets another controller</a>
```

Запрос к URL, который нацелен на метод действия `Index()` контроллера `Admin`, был выражен вспомогательной функцией дескриптора как `/Admin`. Системе маршрутизации известно, что маршрут, определенный в приложении, по умолчанию будет использовать метод действия `Index()`, позволяя опускать ненужные сегменты.

При выяснении, каким образом нацеливаться на заданный метод действия, система маршрутизации включает маршруты, которые были определены с применением атрибута `Route`. В листинге 16.5 атрибут `asp-controller` нацелен на действие `Index` в контроллере `Customer`, к которому был применен атрибут `Route` в главе 15.

Листинг 16.5. Направление на действие, декорированное атрибутом `Route`, в файле `Result.cshtml` из папки `Views/Shared`

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="panel-body">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
  <a asp-controller="Customer" asp-action="Index">This is an outgoing URL</a>
</body>
</html>
```

Вот какая ссылка сгенерируется:

```
<a href="/app/Customer/actions/Index">This is an outgoing URL</a>
```

Результат соответствует атрибуту `Route`, который применялся к контроллеру `Customer` в главе 15:

```
...
[Route("app/[controller]/actions/[action]/{id:weekday?}")]
public class CustomerController : Controller {
...
```

Передача дополнительных значений

Системе маршрутизации можно передавать значения для переменных сегментов за счет определения атрибутов с именами, начинающимися со строки `asp-route-`, за которой следует имя сегмента. Таким образом, атрибут `asp-route-id` используется для установки значения сегмента `id` (листинг 16.6).

Листинг 16.6. Предоставление значений для переменных сегментов в файле `Result.cshtml` из папки `Views/Shared`

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
  <a asp-controller="Home" asp-action="Index" asp-route-id="Hello">
    This is an outgoing URL
  </a>
</body>
</html>
```

Здесь предоставляется значение для переменной сегмента по имени `id`. Если приложение применяет маршрут, показанный в листинге 16.3, тогда при визуализации представления будет получена следующая HTML-разметка:

```
<a href="/App/DoIndex?id=Hello">This is an outgoing URL</a>
```

Обратите внимание, что значение сегмента было добавлено как часть строки запроса, чтобы вписываться в шаблон URL, определенный маршрутом. Причина связана с отсутствием переменной сегмента, которая бы соответствовала `id` в этом маршруте. Чтобы решить проблему, необходимо отредактировать файл `Startup.cs`, оставив только маршрут, который *имеет* сегмент `id` (листинг 16.7).

Листинг 16.7. Редактирование маршрутов в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;
```

```
namespace UrlsAndRoutes {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options =>
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        // routes.MapRoute(
        //   name: "NewRoute",
        //   template: "App/Do{action}",
        //   defaults: new { controller = "Home" });
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
      });
    }
  }
}
```

Запустив приложение еще раз, можно заметить, что вспомогательная функция дескриптора выдала следующий HTML-элемент, в котором значение свойства `id` включено как сегмент URL:

```
<a href="/Home/Index/Hello">This is an outgoing URL</a>
```

Повторное использование переменных сегментов

При описании способа сопоставления маршрутов для исходящих URL было указано, что когда система маршрутизации пытается найти значения для каждой переменной сегмента в шаблоне URL маршрута, она просматривает текущий запрос. Такое поведение сбивает с толку многих программистов и может вылиться в длительный сеанс отладки.

Предположим, что приложение имеет единственный маршрут:

```
...
app.UseMvc(routes => {
  routes.MapRoute(name: "MyRoute",
    template: "{controller}/{action}/{color}/{page}");
});
...
```

Пусть в текущий момент пользователь запросил URL вида `/Home/Index/Red/100` и визуализируется показанная ниже ссылка:

```
...
<a asp-controller="Home" asp-action="Index" asp-route-page="789">
  This is an outgoing URL
</a>
...
```

Можно было бы ожидать, что система маршрутизации не сумеет выполнить сопоставление с маршрутом, т.к. не было предоставлено значение для переменной сегмента `color`, к тому же для нее не определено стандартное значение. Однако дело обстоит не так. Система мар-

шрутизации *будет* сопоставлять с определенным ранее маршрутом. В результате сгенерируется следующая HTML-разметка:

```
...
<a href="/Home/Index/Red/789">This is an outgoing URL</a>
...
```

Система маршрутизации достаточно интеллектуальна, чтобы провести сопоставление с маршрутом, поскольку при генерировании *исходящего* URL она будет повторно использовать значения переменных сегментов из *входящего* URL. В рассматриваемом случае переменная `color` получит значение `Red`, т.к. его включал URL, с которого стартовал пользователь.

Это *не* поведение, предназначенное для крайних случаев. Система маршрутизации будет применять такой прием как часть обычной оценки маршрутов, даже если имеется последующий маршрут, который обеспечит совпадение, не требуя повторного использования значений из текущего запроса.

Я настоятельно рекомендую не полагаться на описанное поведение и предоставлять значения для всех переменных сегментов в шаблоне URL. Если опираться на данное поведение, то код станет гораздо труднее в восприятии, к тому же в коде будут делаться допущения относительно порядка, в котором пользователи выдают запросы, что обернется проблемами при сопровождении приложения.

Генерирование полностью заданных URL

Все ссылки, которые генерировались до сих пор, содержали относительные URL, но вспомогательная функция дескриптора `a` способна также генерировать полностью заданные URL, как показано в листинге 16.8.

Листинг 16.8. Генерирование полностью заданного URL в файле `Result.cshtml` из папки `Views/Shared`

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
  <a asp-controller="Home" asp-action="Index" asp-route-id="Hello"
     asp-protocol="https" asp-host="myserver.mydomain.com"
     asp-fragment="myFragment">
    This is an outgoing URL
  </a>
</body>
</html>
```

Атрибуты `asp-protocol`, `asp-host` и `asp-fragment` применяются для указания протокола (`https` в листинге), имени сервера (`myserver.mydomain.com`) и фрагмента URL (`myFragment`). Эти значения объединяются с выводом из системы маршрутизации для создания полностью заданного URL, который можно увидеть, запустив приложение и просмотрев отправленную браузеру HTML-разметку:

```
<a href="https://myserver.mydomain.com/Home/Index/Hello#myFragment">
  This is an outgoing URL
</a>
```

Будьте осторожны при использовании полностью заданных URL, поскольку они создают зависимости от инфраструктуры приложения и, когда инфраструктура изменяется, вы должны не забыть внести соответствующие изменения в представления MVC.

Генерирование URL из специфического маршрута

В предшествующих примерах система маршрутизации выбирала маршрут для применения при генерировании URL. Если важно генерировать URL в специфическом формате, тогда можно указать маршрут, подлежащий использованию для генерации исходящего URL. Чтобы продемонстрировать прием в работе, добавим в файл `Startup.cs` новый маршрут, так что в примере приложения окажется два маршрута (листинг 16.9).

Листинг 16.9. Добавление маршрута в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options =>
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint)));
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {

        // routes.MapRoute(
        //   name: "NewRoute",
        //   template: "App/Do{action}",
        //   defaults: new { controller = "Home" });
```

```
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");

        routes.MapRoute(
          name: "out",
          template: "outbound/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

Представление, приведенное в листинге 16.10, содержит два якорных элемента, каждый из которых указывает на те же самые контроллер и действие. Разница в том, что второй элемент применяет атрибут `asp-route` вспомогательной функции дескриптора для указания на необходимость использования маршрута `out` при генерации URL, относящегося к атрибуту `href`.

Листинг 16.10. Генерирование URL в файле `Result.cshtml` из папки `Views/Shared`

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.
min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
  <a asp-controller="Home" asp-action="CustomVariable">This is an outgoing URL
</a>
  <a asp-route="out">This is an outgoing URL</a>
</body>
</html>
```

Атрибут `asp-route` может применяться только в ситуации, когда отсутствуют атрибуты `asp-controller` и `asp-action`, т.е. выбирать можно лишь специфический маршрут для контроллера и действия, которые привели к визуализации представления. В результате запуска приложения и запрашивания URL вида `/Home/CustomVariable` получаются два разных URL, сгенерированные маршрутами:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
<a href="/outbound">This is an outgoing URL</a>
```

Доводы против именованных маршрутов

Проблема с зависимостью от имен маршрутов при генерации исходящих URL связана с тем, что это нарушает принцип разделения обязанностей, который занимает центральное место в паттерне проектирования MVC. При генерации ссылки либо URL в представлении или методе действия необходимо сосредоточиться на действии и контроллере, куда пользователь будет направлен, а не на формате URL, который будет использоваться. Привнося знание о различных маршрутах в представления или контроллеры, мы создаем зависимости, которых можно было бы избежать. В своих проектах я стараюсь не именовать маршруты (указывая `null` в аргументе `name`) и предпочитаю применять комментарии в коде, чтобы не забыть о том, для чего предназначен каждый маршрут.

Генерирование URL (без ссылок)

Ограничение вспомогательных функций дескрипторов заключается в том, что они трансформируют HTML-элементы и не могут без труда быть переориентированы в случае, если необходимо генерировать URL для приложения без окружающей HTML-разметки.

Инфраструктура MVC предлагает вспомогательный класс, который можно использовать для создания URL напрямую, доступный через метод `Url.Action()`, как демонстрируется в листинге 16.11.

Листинг 16.11. Генерирование URL в файле `Result.cshtml` из папки `Views/Shared`

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>
  <p>URL: @Url.Action("CustomVariable", "Home", new { id = 100 })</p>
</body>
</html>
```

В аргументах метода `Url.Action()` указываются метод действия, контроллер и значения для любых переменных сегментов. Результатом добавления разметки в листинге 16.11 будет генерация следующего вывода:

```
<p>URL: /Home/CustomVariable/100</p>
```

Генерирование URL в методах действий

Метод `Url.Action()` может применяться также и в методах действий для создания URL внутри кода C#. В листинге 16.12 модифицирован один из методов действий контроллера `Home`, чтобы генерировать URL с использованием `Url.Action()`.

Листинг 16.12. Генерирование URL внутри метода действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Models;

namespace UrlsAndRoutes.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("Result",
      new Result {
        Controller = nameof(HomeController),
        Action = nameof(Index)
    });

    public ViewResult CustomVariable(string id) {
      Result r = new Result {
        Controller = nameof(HomeController),
        Action = nameof(CustomVariable),
      };
      r.Data["Id"] = id ?? "<no value>";
      r.Data["Url"] = Url.Action("CustomVariable", "Home", new { id = 100 });
      return View("Result", r);
    }
  }
}
```

Запустив приложение и запросив URL вида `/Home/CustomVariable`, легко заметить, что в таблице появилась строка, которая отображает данный URL (рис. 16.4).

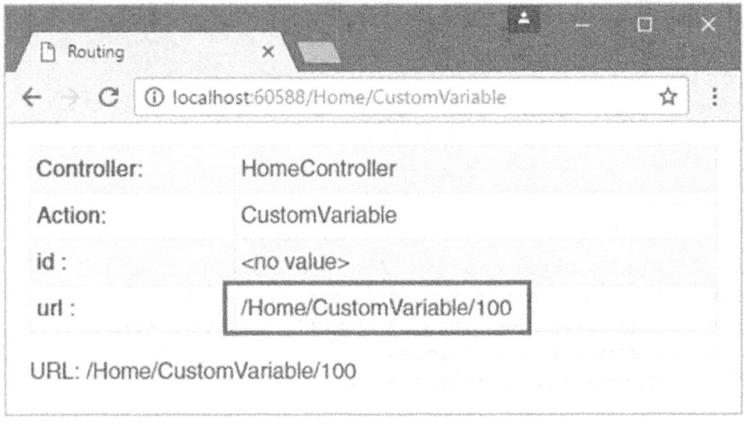


Рис. 16.4. Генерирование URL в методе действия

Настройка системы маршрутизации

Вы уже видели, насколько гибкой и конфигурируемой является система маршрутизации, но если она не удовлетворяет существующим требованиям, тогда ее поведение можно настроить. В этом разделе будут показаны два способа такой настройки.

Изменение конфигурации системы маршрутизации

В главе 15 вы узнали, как конфигурировать объект `RouteOptions` в файле `Startup.cs` для настройки специального ограничения маршрута. Объект `RouteOptions` также применяется для конфигурирования ряда средств маршрутизации с помощью свойств, описанных в табл. 16.3.

Таблица 16.3. Конфигурационные свойства `RouteOptions`

Имя	Описание
`AppendTrailingSlash`	Когда это свойство типа `bool` равно `true`, к генерируемым системой маршрутизации URL добавляется завершающий символ косой черты. Стандартным значением является `false`
`LowercaseUrls`	Когда это свойство типа `bool` равно `true`, результирующие URL преобразуются в нижний регистр, если контроллер, действие или значения сегментов содержат символы нижнего регистра. Стандартным значением является `false`

В листинге 16.13 приведено содержимое файла `Startup.cs` с добавленными операторами для настройки обоих конфигурационных свойств из табл. 16.3.

Листинг 16.13. Конфигурирование системы маршрутизации в файле `Startup.cs` из папки `UrlsAndRoutes`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options => {
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint));
        options.LowercaseUrls = true;
        options.AppendTrailingSlash = true;
      });
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
        routes.MapRoute(
          name: "out",
          template: "outbound/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

После запуска приложения и просмотра URL, которые сгенерировала система маршрутизации, выяснится, что изменение конфигурационных свойств обеспечило преобразование URL в нижний регистр и добавление к ним завершающей косой черты (рис. 16.5).

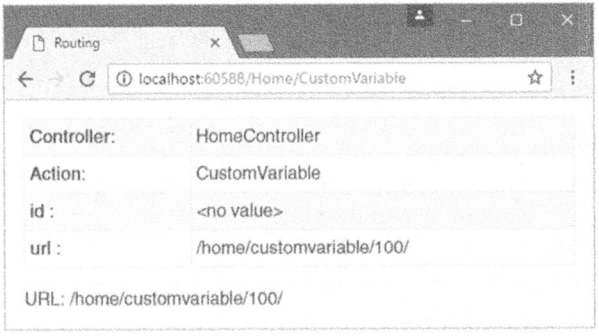


Рис. 16.5. Конфигурирование системы маршрутизации

Создание специального класса маршрута

Если вам не нравится способ, которым система маршрутизации производит сопоставление с URL, или для вашего приложения необходимо реализовать что-то специфическое, тогда можете создать собственные классы маршрутизации и использовать их для обработки URL. Инфраструктура ASP.NET предоставляет интерфейс `Microsoft.AspNetCore.Routing.IRouter`, который можно реализовать для создания специального маршрута. Вот как выглядит определение интерфейса `IRouter`:

```
using System.Threading.Tasks;
namespace Microsoft.AspNetCore.Routing {
  public interface IRouter {
    Task RouteAsync(RouteContext context);
    VirtualPathData GetVirtualPath(VirtualPathContext context);
  }
}
```

Чтобы создать специальный маршрут, понадобится реализовать метод `RouteAsync()` для обработки входящих запросов и метод `GetVirtualPath()`, если нужно генерировать исходящие URL.

В целях демонстрации мы создадим специальный класс маршрутизации, который будет обрабатывать запросы к унаследованным URL. Предположим, что мы перенесли какое-то существующее приложение в MVC, но есть пользователи, которые поместили старые URL в закладки или жестко закодировали их в сценариях. Мы хотим по-прежнему поддерживать такие старые URL. Мы могли бы обработать это с применением обычной системы маршрутизации, но решение данной задачи будет хорошим примером для целей настоящего раздела.

Маршрутизация входящих URL

Чтобы понять, как работают специальные маршруты, мы начнем с создания одного такого маршрута, который будет обрабатывать каждый аспект запроса самостоятельно, не используя контроллер и представление. Создадим в папке `Infrastructure` файл класса по имени `LegacyRoute.cs` с реализацией интерфейса `IRouter`, приведенной в листинге 16.14.

Листинг 16.14. Содержимое файла `LegacyRoute.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UrlsAndRoutes.Infrastructure {
  public class LegacyRoute : IRouter {
    private string[] urls;

    public LegacyRoute(params string[] targetUrls) {
      this.urls = targetUrls;
    }

    public Task RouteAsync(RouteContext context) {

      string requestedUrl = context.HttpContext.Request.Path
        .Value.TrimEnd('/');

      if (urls.Contains(requestedUrl, StringComparer.OrdinalIgnoreCase)) {
        context.Handler = async ctx => {
          HttpResponse response = ctx.Response;
          byte[] bytes = Encoding.ASCII.GetBytes($"URL: {requestedUrl}");
          await response.Body.WriteAsync(bytes, 0, bytes.Length);
        };
      }
      return Task.CompletedTask;
    }

    public VirtualPathData GetVirtualPath(VirtualPathContext context) {
      return null;
    }
  }
}
```

Класс LecagyRoute реализует интерфейс IRouter, но определяет код только для метода RouteAsync(), который применяется для обработки входящих запросов; вскоре мы добавим поддержку и для исходящих URL.

В методе RouteAsync() находится совсем немного операторов, но в своей работе они полагаются на несколько важных типов ASP.NET. Лучше всего начать исследование с сигнатуры метода:

```
...
public async Task RouteAsync(RouteContext context) {
...
```

Метод RouteAsync() отвечает за оценку, может ли запрос быть обработан, и если может, то за управление процессом, посредством которого генерируется ответ, отправляемый обратно клиенту. Такой процесс выполняется асинхронно, поэтому метод RouteAsync() возвращает объект Task.

Метод RouteAsync() вызывается с аргументом RouteContext, который предоставляет доступ ко всему, что известно о запросе, и предлагает средства, требуемые для отправки ответа клиенту. Класс RouteContext находится в пространстве имен Microsoft.AspNetCore.Routing и определяет три свойства, описанные в табл. 16.4.

Таблица 16.4. Свойства, определенные в классе RouteContext

Имя	Описание
RouteData	Это свойство возвращает объект Microsoft.AspNetCore.Routing.RouteData. При написании специального маршрута, который полагается на средства MVC (как объясняется в следующем разделе), данный объект используется для определения контроллера, метода действия и аргументов, применяемых для обработки запроса
HttpContext	Это свойство возвращает объект Microsoft.AspNetCore.Http.HttpContext, который предоставляет доступ к деталям HTTP-запроса и предназначен для выпуска HTTP-ответа
Handler	Это свойство используется для предоставления системе маршрутизации объекта RequestDelegate, который будет обрабатывать запрос. Если метод RouteAsync не устанавливает данное свойство, тогда система маршрутизации продолжит работу своим способом посредством набора маршрутов в конфигурации приложения

Система маршрутизации вызывает метод RouteAsync() каждого маршрута в приложении и после каждого вызова проверяет значение свойства Handler. Если свойство Handler было установлено в RequestDelegate, тогда маршрут предоставляет системе маршрутизации делегат, который способен обработать запрос, и этот делегат вызывается для генерации ответа. Ниже показана сигнатура делегата RequestDelegate, определенного в пространстве имен Microsoft.AspNetCore.Http:

```
using System.Threading.Tasks;
namespace Microsoft.AspNetCore.Http {
  public delegate Task RequestDelegate(HttpContext context);
}
```

Делегат RequestDelegate принимает объект HttpContext и возвращает объект Task, который будет генерировать ответ. Если свойство Handler не установлено ни в одном из маршрутов, то системе маршрутизации известно, что приложение не может обработать запрос, и она сгенерирует ответ 404 - Not Found (404 — не найдено).

С этой целью реализация метода RouteAsync() должна выяснить, может ли она обработать запрос, для чего обычно требуется объект HttpContext. В рассматриваемом примере применяется свойство HttpContext.Request, которое возвращает объект Microsoft.AspNetCore.Http.HttpRequest, описывающий запрос. Объект HttpRequest предоставляет доступ ко всей информации о запросе, включая заголовки, тело и детали того, откуда запрос произошел, но мы заинтересованы в свойстве Path, поскольку оно дает подробные сведения относительно URL, запрошенного клиентом. Свойство Path возвращает объект PathString, предлагающий удобные методы для объединения и сравнения путей URL. Мы используем свойство Value, т.к. оно дает раздел полного пути из URL в виде строки, которую можно сравнить с набором поддерживаемых URL, полученным конструктором LegacyRoute.

```
...
string requestedUrl = context.HttpContext.Request.Path.Value.TrimEnd('/');
if (urls.Contains(requestedUrl, StringComparer.OrdinalIgnoreCase)) {
...
```

Здесь с помощью метода TrimEnd() из URL удаляется завершающая косая черта (если она есть), которая могла быть добавлена либо пользователем, либо в результате установки конфигурационного свойства AppendTrailingSlash, описанного в разделе "Изменение конфигурации системы маршрутизации" ранее в главе.

Если запрошенный путь входит в число сконфигурированных для поддержки классом LegacyRoute, тогда мы устанавливаем свойство Handler с применением лямбда-функции, которая будет генерировать ответ:

```
...
context.Handler = async ctx => {
  HttpResponse response = ctx.Response;
  byte[] bytes = Encoding.ASCII.GetBytes($"URL: {requestedUrl}");
  await response.Body.WriteAsync(bytes, 0, bytes.Length);
};
...
```

Свойство HttpContext.Response возвращает объект HttpResponse, который может использоваться для создания ответа клиенту и предоставляет доступ к заголовкам и содержимому, подлежащему отправке клиенту. Метод HttpResponse.Body.WriteAsync() применяется для асинхронной записи в качестве ответа простой строки ASCII. Такой прием не будет предприниматься в реальном проекте, но он позволяет выпустить ответ, не выбирая и не визуализируя представления (хотя в следующем разделе будет показано, как это делать).

Когда свойство Handler установлено, системе маршрутизации известно, что ее поиск маршрута закончен и можно вызывать делегат для генерации ответа клиенту.

Применение специального класса маршрута

Расширяющий метод MapRoute(), который до сих пор использовался для создания маршрутов, не поддерживает применение специальных классов маршрутов. Чтобы воспользоваться классом LegacyRoute, придется задействовать другой подход (листинг 16.15).

Листинг 16.15. Применение специального класса маршрутизации в файле Startup.cs из папки UrlsAndRoutes

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options => {
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint));
        options.LowercaseUrls = true;
        options.AppendTrailingSlash = true;
      });
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.Routes.Add(new LegacyRoute(
          "/articles/Windows_3.1_Overview.html",
          "/old/.NET_1.0_Class_Library"));

        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");

        routes.MapRoute(
          name: "out",
          template: "outbound/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

При использовании специальных классов потребуется вызвать метод `Add()` на коллекции маршрутов, чтобы зарегистрировать класс реализации `IRouter`. В текущем примере аргументами конструктора `LegacyRoute` являются унаследованные URL, которые должен поддерживать специальный маршрут. Запустив приложение и запросив `/articles/Windows_3.1_Overview.html`, можно увидеть результат — специальный маршрут отобразит запрошенный URL (рис. 16.6).

Маршрутизация на контроллеры MVC

Существует крупная брешь между сопоставлением с простыми строками URL и применением системы контроллеров, действий и представлений Razor инфраструктуры MVC.

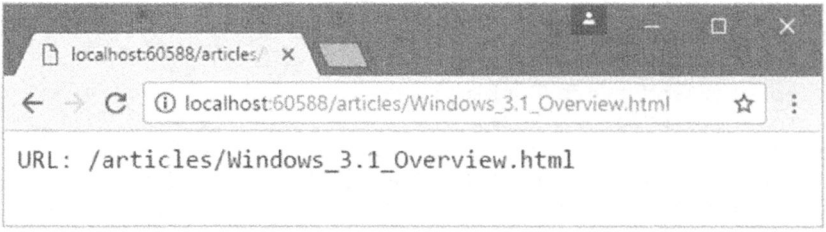


Рис. 16.6. Использование специального маршрута

К счастью, при создании специальных маршрутов реализовывать такую функциональность самостоятельно не придется, потому что для выполнения всей тяжелой работы может быть задействован класс, который MVC использует "за кулисами". Чтобы подготовиться к применению инфраструктуры MVC, добавим в папку `Controllers` файл класса по имени `LegacyController.cs` с содержимым, приведенным в листинге 16.16.

Листинг 16.16. Содержимое файла `LegacyController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace UrlsAndRoutes.Controllers {

  public class LegacyController : Controller {

    public ViewResult GetLegacyUrl(string legacyUrl)
      => View((object)legacyUrl);
  }
}
```

Метод действия `GetLegacyURL()` в таком простом контроллере принимает параметр, который содержит унаследованный URL, запрошенный клиентом. Если бы данный контроллер был реализован в реальном проекте, то упомянутый метод использовался бы для извлечения запрошенных файлов, но здесь просто отображается URL в представлении.

Совет. Обратите внимание, что в листинге 16.16 аргумент метода `View()` приводится к типу `object`. Одна из перегруженных версий метода `View()` принимает параметр типа `string`, указывающий имя представления для визуализации, и без приведения компилятор C# выберет именно ее. Во избежание этого мы выполняем приведение к `object`, чтобы однозначно вызывалась перегруженная версия, которая принимает модель представления и применяет стандартное представление. Проблему можно было бы также решить за счет использования перегруженной версии, принимающей как имя представления, так и модель представления, но предпочтительнее по возможности не делать явных ассоциаций между методами действий и представлениями. За дополнительными сведениями обращайтесь в главу 17.

Создадим папку `Views/Legacy` и поместим в нее файл представления по имени `GetLegacyUrl.cshtml` с содержимым из листинга 16.17. Новое представление отображает значение модели, которое покажет запрошенный клиентом URL.

Листинг 16.17. Содержимое файла `GetLegacyUrl.cshtml` из папки `Views/Legacy`

```
@model string
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <h2>GetLegacyURL</h2>
  The URL requested was: @Model
</body>
</html>
```

В листинге 16.18 класс `LegacyRoute` модифицирован так, чтобы обрабатываемые им URL маршрутизировались на действие `GetLegacyUrl` контроллера `Legacy`.

Листинг 16.18. Маршрутизация на контроллер в файле `LegacyRoute.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Internal;
using Microsoft.Extensions.DependencyInjection;
namespace UrlsAndRoutes.Infrastructure {
  public class LegacyRoute : IRouter {
    private string[] urls;
    private IRouter mvcRoute;
    public LegacyRoute(IServiceProvider services, params string[] targetUrls) {
      this.urls = targetUrls;
      mvcRoute = services.GetRequiredService<MvcRouteHandler>();
    }
    public async Task RouteAsync(RouteContext context) {
      string requestedUrl = context.HttpContext.Request.Path
        .Value.TrimEnd('/');
      if (urls.Contains(requestedUrl, StringComparer.OrdinalIgnoreCase)) {
        context.RouteData.Values["controller"] = "Legacy";
        context.RouteData.Values["action"] = "GetLegacyUrl";
        context.RouteData.Values["legacyUrl"] = requestedUrl;
        await mvcRoute.RouteAsync(context);
      }
    }
    public VirtualPathData GetVirtualPath(VirtualPathContext context) {
      return null;
    }
  }
}
```

Класс Microsoft.AspNetCore.Mvc.Internal.MvcRouteHandler предоставляет механизм, посредством которого переменные сегментов controller и action применяются для нахождения класса контроллера, выполнения метода действия и возвращения результата клиенту. Этот класс был написан таким образом, что к нему можно обращаться из специальной реализации IRouter, которая предоставляет значения controller и action, а также любые другие требующиеся значения вроде аргументов метода действия.

В листинге 16.18 создается новый экземпляр класса MvcRouteHandler, которому делегируется задача нахождения класса контроллера. Для этого необходимо предоставить данные маршрутизации:

```
...
context.RouteData.Values["controller"] = "Legacy";
context.RouteData.Values["action"] = "GetLegacyUrl";
context.RouteData.Values["legacyUrl"] = requestedUrl;
...
```

Свойство RouteContext.RouteData.Vales возвращает словарь, который используется для предоставления значений данных классу MvcRouteHandler. В стандартной системе маршрутизации значения данных создаются путем применения шаблона URL к запросу, но в специальном классе маршрута значения жестко закодированы, так что всегда производится направление на действие GetLegacyUrl контроллера Legacy. Между запросами изменяется только значение данных legacyUrl, которое устанавливается в URL запроса; оно будет применяться в качестве аргумента с таким же именем, получаемого методом действия.

Последнее изменение в листинге 16.18 делегирует ответственность за поиск и использование класса контроллера для обработки запроса.

```
...
await mvcRoute.RouteAsync(context);
...
```

Объект RouteContext, теперь содержащий значения controller, action и legacyUrl, передается методу RouteAsync() объекта MvcRouteHandler, который берет на себя обязанность за любую дальнейшую обработку запроса, включая установку свойства Handler. В результате класс LegacyRoute может сосредоточиться на решении о том, какие URL он будет обрабатывать, не имея дела с деталями работы с контроллерами напрямую.

Объект MvcRouteHandler, выполняющий работу в рассматриваемом примере, должен запрашиваться как служба, что объясняется в главе 18. Чтобы предоставить конструктор LegacyRoute с объектом реализации IServiceProvider, который необходим ему для создания объекта MvcRouteHandler, в классе Startup обновлен оператор, определяющий маршрут, с целью снабжения его возможностью доступа к службам приложения (листинг 16.19).

Листинг 16.19. Обеспечение доступа к службам приложения в файле Startup.cs из папки UrlsAndRoutes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options => {
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint));
        options.LowercaseUrls = true;
        options.AppendTrailingSlash = true;
      });
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.Routes.Add(new LegacyRoute(
          app.ApplicationServices,
          "/articles/Windows_3.1_Overview.html",
          "/old/.NET_1.0_Class_Library"));

        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");

        routes.MapRoute(
          name: "out",
          template: "outbound/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

Запустив приложение и запросив `/articles/Windows_3.1_Overview.html` еще раз, можно увидеть, что простой текстовый ответ был заменен выводом из представления (рис. 16.7).



Рис. 16.7. Делегирование работы с контроллерами и представлениями

Генерирование исходящих URL

Для поддержки генерации исходящих URL мы должны реализовать в классе LegacyRoute метод GetVirtualPath(), как показано в листинге 16.20.

Листинг 16.20. Генерирование исходящих URL в файле LegacyRoute.cs из папки Infrastructure

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Internal;
using Microsoft.Extensions.DependencyInjection;

namespace UrlsAndRoutes.Infrastructure {
  public class LegacyRoute : IRouter {
    private string[] urls;
    private IRouter mvcRoute;

    public LegacyRoute(IServiceProvider services, params string[] targetUrls) {
      this.urls = targetUrls;
      mvcRoute = services.GetRequiredService<MvcRouteHandler>();
    }

    public async Task RouteAsync(RouteContext context) {

      string requestedUrl = context.HttpContext.Request.Path
        .Value.TrimEnd('/');

      if (urls.Contains(requestedUrl, StringComparer.OrdinalIgnoreCase)) {
        context.RouteData.Values["controller"] = "Legacy";
        context.RouteData.Values["action"] = "GetLegacyUrl";
        context.RouteData.Values["legacyUrl"] = requestedUrl;
        await mvcRoute.RouteAsync(context);
      }
    }

    public VirtualPathData GetVirtualPath(VirtualPathContext context) {
      if (context.Values.ContainsKey("legacyUrl")) {
        string url = context.Values["legacyUrl"] as string;
        if (urls.Contains(url)) {
          return new VirtualPathData(this, url);
        }
      }
      return null;
    }
  }
}
```

Система маршрутизации вызывает метод GetVirtualPath() каждого маршрута, который был определен в классе Startup, давая каждому шанс сгенерировать исходящий URL, требующийся приложению. Аргумент метода GetVirtualPath() — это объект VirtualPathContext, который предоставляет необходимую информацию относительно URL. В табл. 16.5 описаны свойства класса VirtualPathContext.

Таблица 16.5. Свойства, определенные в классе `VirtualPathContext`

Имя	Описание
`RouteName`	Это свойство возвращает имя маршрута
`Values`	Это свойство возвращает индексированный по имени словарь значений, которые могут применяться для переменных сегментов
`AmbientValues`	Это свойство возвращает словарь значений, которые полезны при генерации URL, но не будут встраиваться в результат. В случае реализации собственного класса маршрутизации такой словарь обычно пуст
`HttpContext`	Это свойство возвращает объект `HttpContext`, который предоставляет информацию о запросе и ответе, находящемся в процессе подготовки

В рассматриваемом примере свойство `Values` используется для получения значения по имени `legacyUrl`, и если оно соответствует одному из URL, которые были сконфигурированы для поддержки маршрутом, тогда возвращается объект `VirtualPathData`, снабжающий систему маршрутизации деталями URL. Аргументами конструктора класса `VirtualPathData` являются объект реализации `IRouter`, который генерирует URL, и сам URL.

```
...
return new VirtualPathData(this, url);
...
```

В листинге 16.21 показано измененное представление `Result.cshtml` для запрашивания исходящих URL, которые нацелены на специальное представление.

Листинг 16.21. Генерирование исходящих URL из специального класса маршрута в файле `Result.cshtml` из папки `Views/Shared`

```
@model Result
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Routing</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>

<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
    @foreach (string key in Model.Data.Keys) {
      <tr><th>@key :</th><td>@Model.Data[key]</td></tr>
    }
  </table>

  <a asp-route-legacyurl="/articles/Windows_3.1_Overview.html"
     class="btn btn-primary">
    This is an outgoing URL
  </a>
```

```
  <p>
    URL: @Url.Action(null, null,
      new { legacyurl = "/articles/Windows_3.1_Overview.html" })
  </p>
</body>
</html>
```

В этом примере нет необходимости указывать вспомогательной функции дескриптора контроллер и действие для исходящего маршрута, т.к. при генерации URL они не применяются, а потому атрибуты `asp-controller` и `asp-action` дескриптора `a` опущены. Когда генерируется только URL, первые два аргумента для вспомогательного метода `Url.Action()` устанавливаются в `null` по той же причине.

Если запустить приложение и просмотреть HTML-разметку, полученную в ответ на запрос стандартного URL, то легко заметить, что для создания URL использовался специальный класс маршрута:

```
<a class="btn btn-primary" href="/articles/windows_3.1_overview.html/">
  This is an outgoing URL
</a>
<p>URL: /articles/windows_3.1_overview.html/</p>
```

Завершающие косые черты, добавленные к URL, являются результатом установки в `true` конфигурационного свойства `AppendTrailingSlash` внутри файла `Startup.cs`, и важно, чтобы сопоставление входящего маршрута было способно соответствовать URL с добавленным символом косой черты.

Совет. Если URL, который вы видите в HTML-ответе, имеет отличающийся формат, такой как `/?legacyurl=%2Farticles%2FWindows_3.1_Overview.html`, то значит, что для генерации URL специальный маршрут не применялся, а вместо него был вызван какой-то другой маршрут в приложении. Поскольку никакого контроллера или действия не было указано, произошло нацеливание на действие `Index` контроллера `Home`, и к строке запроса URL добавилось значение `legacyUrl`. В таком случае удостоверьтесь в установке свойства `IsBound` в `true` внутри метода `GetVirtualPath()`, и проверьте, что в конфигурации внутри файла `Startup.cs` указаны корректные URL для конструктора класса `LegacyRoute`, а специальный маршрут определен перед всеми остальными маршрутами.

Работа с областями

Инфраструктура ASP.NET Core MVC поддерживает организацию веб-приложения в виде *областей*, где каждая область представляет функциональный сегмент приложения, такой как администрирование, выписка счетов-фактур, поддержка пользователей и т.д. Прием удобен в крупных проектах, в которых наличие единственного набора папок для всех контроллеров, представлений и моделей может привести к сложностям в управлении.

Каждая область MVC имеет собственную структуру папок, позволяя все хранить отдельно. Такой подход делает более очевидным то, какие элементы проекта относятся к каждой части функциональности приложения, и помогает множеству разработчиков трудиться над проектом, не конфликтуя друг с другом. Области в значительной степени поддерживаются системой маршрутизации и потому они рассматриваются вместе с URL и маршрутами. В настоящем разделе будет показано, как настраивать и пользоваться областями в проектах MVC.

Создание области

Создание области требует добавления папок в проект. Папка верхнего уровня называется Areas. Внутри нее для каждой необходимой области предусмотрена своя папка, содержащая собственные подпапки Controllers, Views и Models. Мы планируем создать область по имени Admin, что означает создание набора папок, описанных в табл. 16.6. Для подготовки примера проекта создадим все папки, перечисленные в табл. 16.6.

Таблица 16.6. Папки, требуемые для подготовки областей

Имя	Описание
Areas	Эта папка будет содержать все области в приложении MVC
Areas/Admin	Эта папка будет содержать классы и представления для области Admin
Areas/Admin/Controllers	Эта папка будет содержать контроллеры для области Admin
Areas/Admin/Views	Эта папка будет содержать представления для области Admin
Areas/Admin/Views/Home	Эта папка будет содержать представления для контроллера Home в области Admin
Areas/Admin/Models	Эта папка будет содержать модели для области Admin

Хотя каждая область применяется по отдельности, многие средства MVC полагаются на стандартные функциональные возможности C# или .NET, такие как пространства имен. Чтобы облегчить использование области, первым делом нужно добавить файл импортирования представлений, который позволит работать с моделями из области внутри представлений, не включая пространства имен и получая преимущество от применения вспомогательных функций дескрипторов. Создадим в папке Areas/Admin/Views файл импортирования представлений по имени _ViewImports.cshtml и поместим в него операторы, приведенные в листинге 16.22.

Листинг 16.22. Содержимое файла _ViewImports.cshtml из папки Areas/Admin/Views

```
@using UrlsAndRoutes.Areas.Admin.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Создание маршрута для области

Чтобы задействовать области, потребуется добавить в файл Startup.cs маршрут, который содержит переменную сегмента area (листинг 16.23).

Листинг 16.23. Добавление маршрута для областей в файле Startup.cs из папки UrlsAndRoutes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
```

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing.Constraints;
using Microsoft.AspNetCore.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.Configure<RouteOptions>(options => {
        options.ConstraintMap.Add("weekday", typeof(WeekDayConstraint));
        options.LowercaseUrls = true;
        options.AppendTrailingSlash = true;
      });
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "areas",
          template: "{area:exists}/{controller=Home}/{action=Index}");
        routes.Routes.Add(new LegacyRoute(
          app.ApplicationServices,
          "/articles/Windows_3.1_Overview.html",
          "/old/.NET_1.0_Class_Library"));
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
        routes.MapRoute(
          name: "out",
          template: "outbound/{controller=Home}/{action=Index}");
      });
    }
  }
}
```

Переменная сегмента `area` используется для сопоставления с URL, которые нацелены на контроллеры в специфических областях. В листинге задействован стандартный шаблон URL, но сегмент `area` можно добавлять в любой требующийся шаблон. Маршрут, который добавляет поддержку областей, должен находиться перед менее специфическими маршрутами для обеспечения корректного сопоставления с URL. Ограничение `exists` применяется для гарантии того, что запросы соответствуют только областям, которые были определены в приложении.

Заполнение области

Контроллеры, представления и модели в области можно создавать точно так же, как в главной части приложения MVC. Чтобы создать модель, щелкнем правой кнопкой мыши на папке `Areas/Admin/Models`, выберем в контекстном меню пункт

Add⇨Class (Добавить⇨Класс) и создадим файл класса по имени Person.cs, содержимое которого показано в листинге 16.24.

Листинг 16.24. Содержимое файла Person.cs из папки Areas/Admin/Models

```
namespace UrlsAndRoutes.Areas.Admin.Models {
  public class Person {
    public string Name { get; set; }
    public string City { get; set; }
  }
}
```

Для создания контроллера щелкнем правой кнопкой мыши на папке Areas/Admin/Controllers, выберем в контекстном меню пункт Add⇨Class и создадим файл класса по имени HomeController.cs с определением контроллера из листинга 16.25.

Листинг 16.25. Содержимое файла HomeController.cs из папки Areas/Admin/Controllers

```
using Microsoft.AspNetCore.Mvc;
using UrlsAndRoutes.Areas.Admin.Models;

namespace UrlsAndRoutes.Areas.Admin.Controllers {

  [Area("Admin")]
  public class HomeController : Controller {
    private Person[] data = new Person[] {
      new Person { Name = "Alice", City = "London" },
      new Person { Name = "Bob", City = "Paris" },
      new Person { Name = "Joe", City = "New York" }
    };
    public ViewResult Index() => View(data);
  }
}
```

Новый контроллер в целом похож на стандартный кроме одного аспекта. Чтобы ассоциировать контроллер с областью, к классу должен быть применен атрибут Area:

```
...
[Area("Admin")]
public class HomeController : Controller {
...
```

Без атрибута Area контроллеры не будут принадлежать области, даже если они не определены в главной части приложения. Отсутствие атрибута Area может привести к странным результатам. Это первое, что следует проверять, если при работе с областями получаются не те результаты, которые ожидались.

Совет. Если для настройки маршрутов используются атрибуты, как было описано в главе 15, тогда с применением маркера [area] в аргументе атрибута Route можно ссылаться на область, указанную с помощью атрибута Area: [Route("[area]/app/[controller]/actions/[action]/{id:weekday?}")].

Наконец, добавим в папку Areas/Admin/Views/Home файл представления Razor по имени Index.cshtml, содержимое которого показано в листинге 16.26.

Листинг 16.26. Содержимое файла `Index.cshtml` из папки `Areas/Admin/Views/Home`

```
@model Person[]
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Areas</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Name</th><th>City</th></tr>
    @foreach (Person p in Model) {
      <tr><td>@p.Name</td><td>@p.City</td></tr>
    }
  </table>
</body>
</html>
```

Моделью для данного представления является массив объектов `Person`. Благодаря файлу импортирования представлений, содержимое которого было приведено в листинге 16.22, есть возможность ссылаться на тип `Person` без указания пространства имен. Запуск приложения и запрос URL вида `/Admin` даст результат, показанный на рис. 16.8.

Влияние области на приложение MVC

Важно понимать влияние, которое области оказывают на остальные части приложения. Мы создали область по имени `Admin`, но в главной части приложения имеется также контроллер `Admin`. До создания области запрос `/Admin` направлялся бы действию `Index` контроллера `Admin` в главной части приложения; теперь он будет нацелен на действие `Index` контроллера `Home` в области `Admin` (корень области предоставляет стандартные значения для переменных сегментов `controller` и `action`). Изменение такого вида может стать причиной неожиданного поведения, и наилучший способ применения областей предусматривает встраивание их использования в первоначальную схему именования контроллеров для проекта. Если возникла потребность добавить области в готовое приложение, то вы должны тщательно обдумать их влияние на имеющиеся маршруты.

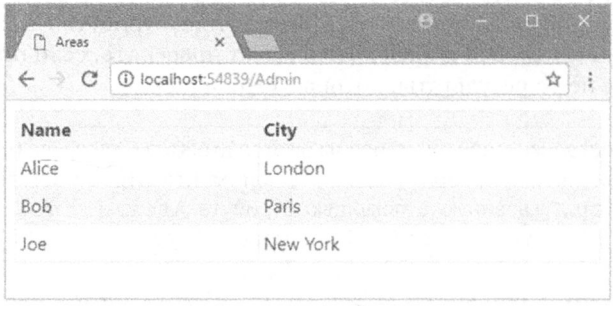


Рис. 16.8. Использование области

Генерирование ссылок на действия в областях

Для создания ссылок, указывающих на действия в той же области, к которой относится текущий запрос, никаких дополнительных шагов предпринимать не придется. Инфраструктура MVC обнаруживает, что текущий запрос относится к конкретной области, и гарантирует, что средство генерации исходящих URL будет искать соответствие только среди маршрутов, определенных для данной области. Например, в листинге 16.27 демонстрируется добавление элемента `a` в файле `Index.cshtml` из папки `Areas/Admin/Views/Home`.

Листинг 16.27. Добавление элемента a в файле Index.cshtml из папки Areas/Admin/Views/Home

```
@model Person[]
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Areas</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Name</th><th>City</th></tr>
    @foreach (Person p in Model) {
      <tr><td>@p.Name</td><td>@p.City</td></tr>
    }
  </table>
  <a asp-action="Index" asp-controller="Home">Link</a>
</body>
</html>
```

Запустив приложение и запросив URL вида `/admin`, можно заметить, что ответ содержит следующий элемент:

```
<a href="/admin/">Link</a>
```

При генерации исходящей ссылки система маршрутизации выбрала маршрут для области и учла стандартные значения, доступные для переменных сегментов `controller` и `action`.

Чтобы создать ссылку на действие в *другой* области или в главной части приложения, систему маршрутизации необходимо снабдить значением для сегмента `area` (листинг 16.28).

Листинг 16.28. Нацеливание на другую область в файле Index.cshtml из папки Areas/Admin/Views/Home

```
@model Person[]
@{ Layout = null; }

<!DOCTYPE html>
<html>
```

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Areas</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-striped table-sm">
    <tr><th>Name</th><th>City</th></tr>
    @foreach (Person p in Model) {
      <tr><td>@p.Name</td><td>@p.City</td></tr>
    }
  </table>
  <a asp-action="Index" asp-controller="Home">Link</a>
  <a asp-action="Index" asp-controller="Home" asp-route-area="">Link</a>
</body>
</html>
```

Атрибут `asp-route-area` устанавливает значение для переменной сегмента `area`. В данном случае эта переменная устанавливается в пустую строку, что указывает на главную часть приложения и дает следующий HTML-элемент:

```
<a href="/">Link</a>
```

Если вы располагаете в своих контроллерах множеством областей и хотите маршрутизировать на них, тогда вместо пустой строки используйте имя нужной области.

Полезные советы относительно схемы URL

После всего сказанного у вас может все же остаться вопрос: с чего начинать проектирование собственной схемы URL? Вы можете просто принять стандартную схему, но организация собственной схемы обеспечивает ряд преимуществ. В последние годы проектное решение с URL приложения применяется все более широко, и появилось несколько важных принципов проектирования. Следуя этим принципам проектирования, вы улучшите удобство использования, совместимость и поисковый рейтинг своих приложений.

Делайте URL чистыми и понятными человеку

Пользователи обращают внимание на URL в приложениях. Просто вспомните, когда вы в последний раз пытались отправить кому-то URL веб-сайта Amazon. Вот как выглядит URL для предыдущего англоязычного издания настоящей книги:

```
https://www.amazon.com/Pro-ASP-NET-Core-ADAM-FREEMAN/dp/1484203984
```

Отправить по электронной почте такой URL еще реально, но попробуйте продиктовать его по телефону. Когда подобной работой приходится заниматься часто, в итоге проще сообщить номер ISBN книги и предложить самостоятельно найти ее страницу на веб-сайте Amazon. Было бы неплохо получать доступ к странице книги с помощью URL следующего вида:

```
http://www.amazon.com/books/pro-asp-net-core
```

Такой URL вполне *можно* было бы диктовать по телефону, и он не выглядит так, будто при наборе почтового сообщения на клавиатуру что-то свалилось.

На заметку! В качестве пояснения: я глубоко уважаю компанию Amazon, которая продает моих книг больше, чем все остальные вместе взятые. Мне известен факт, что все сотрудники Amazon — замечательно умные и приятные люди. Никто из них не окажется до такой степени мелочным, что прекратит продажи моих книг из-за настолько несущественного, как критика применяемого в Amazon формата URL. Я люблю Amazon. Я преклоняюсь перед Amazon. Я просто хочу, чтобы там привели в порядок свои URL.

Ниже даются руководящие принципы построения дружественных к пользователям URL.

- Проектируйте URL так, чтобы они описывали предоставляемое содержимое, а не детали реализации приложения. Используйте `/Articles/AnnualReport`, а не `/Website_v2/CachedContentServer/FromCache/AnnualReport`.
- Отдавайте предпочтение заголовкам содержимого перед идентификационными номерами. Применяйте `/Articles/AnnualReport`, а не `/Articles/2392`. Если вы должны использовать идентификационные номера (для различения элементов с одинаковыми заголовками или во избежание дополнительных запросов к базе данных, необходимых для поиска элемента по его заголовку), тогда указывайте и номер, и заголовок (например, `/Articles/2392/AnnualReport`). Такой URL дольше набирать, но он имеет больше смысла для человека и улучшает поисковый рейтинг. Приложение может просто игнорировать заголовок и отображать элемент, соответствующий идентификатору.
- *Не* применяйте расширения имен файлов для HTML-страниц (скажем, `.aspx` или `.mvc`), но *используйте* их для специализированных типов файлов (таких как `.jpg`, `.pdf` и `.zip`). Веб-браузеры не придают значения расширениям имен файлов, если тип MIME установлен корректно, но люди ожидают, что имена, скажем, файлов PDF заканчиваются на `.pdf`.
- Создавайте осмысленную иерархию (например, `/Products/Menswear/Shirts/Red`), чтобы посетитель смог догадаться, как выглядит URL родительской категории.
- Поддерживайте независимость от регистра символов. (Кто-то может пожелать ввести URL, указанный на печатной странице.) По умолчанию система маршрутизации ASP.NET Core не чувствительна к регистру символов.
- Избегайте применения знаков, кодов и символьных последовательностей. Если вам нужен разделитель слов, то используйте дефис (как в `/my-great-article`). Подчеркивания не являются дружественными к пользователю, а URL-кодированные пробелы выглядят либо странно (`/my+great+article`), либо плохо (`/my%20great%20article`).
- Не изменяйте URL. Неработающие ссылки равнозначны потерям в бизнесе. После изменения URL продолжайте поддерживать старую схему URL насколько возможно долго посредством перенаправления.
- Поддерживайте согласованность. Примите один формат URL в масштабах всего приложения.

URL должны быть короткими, легкими для набора, редактируемыми человеком и постоянными, к тому же они обязаны отражать структуру сайта. Якоб Нильсен, гуру по удобству использования, развивает эту тему по адресу `https://www.nngroup.com/articles/url-as-ui/`. Тим Бернерс-Ли, изобретатель веб-сети, предлагает аналогичные советы (`https://www.w3.org/Provider/Style/URI`).

GET и POST: выбор правильного запроса

Эмпирическое правило гласит, что запросы GET должны применяться для извлечения информации, предназначенной только для чтения, а запросы POST — для любой операции, которая изменяет состояние приложения. В терминах соответствия стандартам запросы GET предназначены для *безопасных* взаимодействий (не имеющих побочных эффектов помимо извлечения информации), а запросы POST — для *небезопасных* взаимодействий (принятие решения либо изменение чего-нибудь). Такие соглашения установлены консорциумом W3C (World Wide Web Consortium) и описаны по адресу:

```
http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
```

Запросы GET являются *адресуемыми*: вся информация содержится в URL, а потому подобные адреса можно добавлять в закладки и ссылаться на них в будущем.

Не используйте запросы GET для операций, которые изменяют состояние. Многие разработчики веб-приложений научились этому на горьком опыте в 2005 году, когда было выпущено приложение Google Web Accelerator. Оно с упреждением выбирало все содержимое, на которое ссылалась каждая страница, что в рамках протокола HTTP вполне законно, поскольку запросы GET должны быть безопасными. К сожалению, многие разработчики веб-приложений проигнорировали соглашения HTTP и размещали в своих приложениях простые ссылки на операции типа "удалить элемент" или "добавить в корзину". В результате получился хаос.

В одной компании были уверены, что их система управления содержимым стала целью неоднократных атак со стороны злоумышленников, т.к. все содержимое постоянно удалялось. Позже в компании обнаружили, что поисковый агент наткнулся на URL административной страницы и прошелся по всем ссылкам на удаление. Аутентификация может защитить от таких действий со стороны пользователей, но не способна обеспечить защиту от веб-акселераторов.

Резюме

В главе рассматривались дополнительные возможности системы маршрутизации. Вы узнали, как генерировать исходящие ссылки и URL, а также научились настраивать систему маршрутизации. По ходу дела вы ознакомились с концепцией областей и с рекомендациями относительно того, как должна создаваться удобная и содержательная схема URL. В следующей главе мы перейдем к исследованию контроллеров и действий, которые являются центральной частью инфраструктуры ASP.NET Core MVC. Будет подробно описана их работа и показано, как их применять для получения наилучших результатов в приложении.

ГЛАВА 17

Контроллеры и действия

Каждый запрос, который поступает в приложение, обрабатывается контроллером. В инфраструктуре ASP.NET Core MVC контроллеры являются классами .NET, которые содержат логику, требующуюся для обработки запроса. В главе 3 объяснялось, что роль контроллера заключается в инкапсуляции логики приложения. Это значит, что контроллеры отвечают за обработку входящих запросов, выполнение операций над моделью предметной области и выбор представлений для визуализации пользователю.

Контроллер волен обрабатывать запрос любым способом, какой посчитает подходящим, до тех пор, пока он не забирается в области ответственности, которые относятся к модели и представлению. Таким образом, контроллеры не содержат и не сохраняют данные, а также не генерируют пользовательские интерфейсы.

В настоящей главе мы рассмотрим реализацию контроллеров, а также различные способы применения контроллеров для получения и генерации вывода. В табл. 17.1 приведена сводка, позволяющая поместить контроллеры в контекст.

Таблица 17.1. Помещение контроллеров в контекст

Вопрос	Ответ
Что это такое?	Контроллеры содержат логику для получения запросов, обновления состояния или модели приложения и выбора ответа, который будет отправлен клиенту
Чем они полезны?	Контроллеры являются центральной частью проектов MVC и содержат логику предметной области для веб-приложения
Как они используются?	Контроллеры — это классы C#, открытые методы которых вызываются для обработки HTTP-запроса. Методы могут брать на себя ответственность за выдачу ответа клиенту напрямую, но более распространенный подход предусматривает возвращение результата действия, который сообщает MVC, как ответ должен быть подготовлен
Существуют ли какие-то скрытые ловушки или ограничения?	Если вы — новичок в области разработки приложений MVC, то можете легко создавать контроллеры, содержащие функциональность, которая больше подходит для модели или представления. Более специфичная проблема связана с тем, что любые открытые классы с именами, заканчивающимися на `Controller`, инфраструктура MVC считает контроллерами; это означает возможность непредумышленной обработки HTTP-запросов в классах, которые не планировались быть контроллерами
Существуют ли альтернативы?	Нет, контроллеры — основная часть приложений MVC

В табл. 17.2 приведена сводка по главе.

Таблица 17.2. Сводка по главе

Задача	Решение	Листинг
Определение контроллера	Создайте открытый класс с именем, заканчивающимся на `Controller`, или унаследуйте его от класса `Controller`	17.7–17.9
Получение деталей HTTP-запроса	Используйте объекты контекста или определите параметры методов действий	17.10–17.13
Выдача результата из метода действия	Работайте напрямую с объектом контекста результата или создайте объект результата действия	17.14–17.16
Выдача HTML-результата	Создайте результат представления	17.17–17.24
Перенаправление клиента	Создайте результат перенаправления	17.25–17.30
Возвращение содержимого клиенту	Создайте результат содержимого	17.31–17.35
Возвращение кода состояния HTTP	Создайте HTTP-результат	17.36, 17.37

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `ControllersAndActions` с применением шаблона ASP.NET Core Web Application (.NET Core) (Веб-приложение ASP.NET Core (.NET Core)). Добавим в класс `Startup` операторы, чтобы включить инфраструктуру MVC и другие компоненты промежуточного ПО (листинг 17.1).

На заметку! Данная глава содержит модульные тесты для основных средств. Для краткости проекты модульного тестирования в инструкции по созданию примера проекта не включены. Вы можете создавать тестовые проекты, следуя процессу, который описан в главе 7, или загрузить готовый проект SportsStore из хранилища GitHub для настоящей книги (`https://github.com/apress/pro-asp.net-core-mvc-2`).

Листинг 17.1. Добавление инфраструктуры MVC и других компонентов промежуточного ПО в файле `Startup.cs` из папки `ControllersAndActions`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace ControllersAndActions {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.AddMemoryCache();
      services.AddSession();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseSession();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Методы `AddMemoryCache()` и `AddSession()` создают службы, необходимые для управления сеансом. Метод `UseSession()` добавляет в конвейер компонент промежуточного ПО, который ассоциирует данные сеанса с запросами, и добавляет cookie-наборы к ответам, гарантируя возможность идентификации будущих запросов. Метод `UseSession()` должен вызываться перед вызовом метода `UseMvc()`,чтобы компонент сеанса мог перехватывать запросы, прежде чем они достигнут промежуточного ПО MVC, и модифицировать ответы после того, как они были сгенерированы. Другие методы настраивают стандартные пакеты, которые рассматривались в главе 14.

Подготовка представлений

Основное внимание в главе уделяется контроллерам и их методам действий, поэтому классы контроллеров будут определяться в главе повсеместно. В качестве подготовки будут определены представления, которые помогут демонстрировать работу контроллеров и методов действий. Созданные в текущем разделе представления находятся в папке `Views/Shared`, что позволит их использовать в любых контроллерах, которые будут определяться позже в главе. Создадим папку `Views/Shared`, добавим в нее файл представления Razor по имени `Result.cshtml` и поместим в файл разметку, показанную в листинге 17.2.

Листинг 17.2. Содержимое файла `Result.cshtml` из папки `Views/Shared`

```
@model string
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Controllers and Actions</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  Model Data: @Model
</body>
</html>
```

Моделью для данного представления является объект типа `string`, который позволит отображать простые сообщения. Далее создадим в папке `Views/Shared` файл по имени `DictionaryResult.cshtml` и поместим в него разметку, приведенную в листинге 17.3. В качестве модели для этого представления выбран словарь, который даст возможность отображать более сложные данные, чем в предыдущем представлении.

Листинг 17.3. Содержимое файла `DictionaryResult.cshtml` из папки `Views/Shared`

```
@model IDictionary<string, string>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Controllers and Actions</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <table class="table table-bordered table-sm table-striped">
    <tr><th>Name</th><th>Value</th></tr>
    @foreach (string key in Model.Keys) {
      <tr><td>@key</td><td>@Model[key]</td></tr>
    }
  </table>
</body>
</html>
```

Затем создадим в папке `Views/Shared` файл по имени `SimpleForm.cshtml` и определим в нем представление, показанное в листинге 17.4. Как следует из названия, представление содержит простую HTML-форму, которая будет получать данные от пользователя.

Листинг 17.4. Содержимое файла `SimpleForm.cshtml` из папки `Views/Shared`

```
@{ Layout = null; }
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Controllers and Actions</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <form method="post" asp-action="ReceiveForm">
    <div class="form-group">
      <label for="name">Name:</label>
      <input class="form-control" name="name" />
    </div>
    <div class="form-group">
      <label for="name">City:</label>
      <input class="form-control" name="city" />
    </div>
```

```
    <button class="btn btn-primary center-block" type="submit">Submit
    </button>
  </form>
</body>
</html>
```

В представлениях используются встроенные вспомогательные функции дескрипторов для генерации URL из системы маршрутизации. Чтобы включить вспомогательные функции дескрипторов, создадим в папке `Views` файл импортирования представлений по имени `_ViewImports.cshtml` и добавим в него выражение, приведенное в листинге 17.5.

Листинг 17.5. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Все представления, созданные в папке `Views/Shared`, зависят от пакета CSS по имени Bootstrap. Для добавления Bootstrap в проект создадим файл `bower.json` с применением шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap (листинг 17.6).

Листинг 17.6. Добавление пакета в файле `bower.json` из папки `ControllersAndActions`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Понятие контроллеров

Контроллеры — это классы C#, открытые методы которых (известные как *методы действий* или просто *действия*) несут ответственность за обработку HTTP-запроса и подготовку ответа, возвращаемого клиенту. Для определения, какой класс контроллера и метод действия должны обработать запрос, инфраструктура MVC использует систему маршрутизации, описанную в главах 15 и 16. Затем она создает новый экземпляр класса контроллера, вызывает метод действия и применяет результат метода для выпуска ответа клиенту.

Инфраструктура MVC снабжает методы действий *данными контекста*, так что они могут выяснить, каким образом обрабатывать запрос. Доступен широкий спектр данных контекста, который описывает все, что касается текущего запроса, подготавливаемого ответа, извлеченных системой маршрутизации данных и деталей идентичности пользователя.

Когда MVC вызывает метод действия, ответ из метода описывает ответ, который должен быть послан клиенту. Самый распространенный вид ответа создается путем визуализации представления Razor, поэтому метод действия использует свой ответ, чтобы сообщить MVC, какое представление задействовать и какие данные модели представления ему предоставить. Но имеются также другие виды ответов, и методы

действий могут делать все, что угодно, от требования у MVC отправки клиенту перенаправления HTTP до отправки сложных объектов данных.

Таким образом, существуют три области функциональности, которые важны для понимания контроллеров. Во-первых, нужно понимать особенности определения контроллеров, чтобы инфраструктура MVC могла применять их для обработки запросов. Контроллеры представляют собой просто классы C#, но создавать их можно разными способами, и важно уяснить отличия между ними. Определение контроллеров рассматривается в разделе "Создание контроллеров" далее в главе.

Во-вторых, следует понимать, каким образом MVC снабжает методы действий данными контекста. Получение необходимых данных контекста важно для эффективной разработки веб-приложений, а инфраструктура MVC облегчает его, определяя набор классов, которые используются с целью описания всего, что требуется тому или иному методу действия. В разделе "Получение данных контекста" позже в главе объясняется, как MVC описывает запросы и ответы.

Наконец, в-третьих, важно понимать, каким образом методы действий выпускают ответ. Методы действий редко нуждаются в самостоятельной генерации HTTP-ответа, и вы должны знать, каким образом инструктировать MVC о необходимости выпуска требуемых ответов, что будет показано в разделе "Генерирование ответа" далее в главе.

Создание контроллеров

Вы сталкивались с применением контроллеров почти во всех предшествующих главах. Наступило время заглянуть "за кулисы" и посмотреть, как они определяются. В последующих разделах будут описаны различные способы создания контроллеров и приведены объяснения отличий между ними.

Создание контроллеров POCO

Инфраструктура MVC поддерживает соглашение по конфигурации, а это значит, что контроллеры в приложении MVC обнаруживаются автоматически вместо того, чтобы определяться в конфигурационном файле. Базовый процесс обнаружения прост: любой класс `public` с именем, заканчивающимся на `Controller`, является контроллером, а любой определенный в нем метод `public` — действием. Чтобы выяснить, как все работает, добавим в проект папку `Controllers` и поместим в нее файл класса по имени `PocoController.cs`, содержимое которого приведено в листинге 17.7.

Совет. Хотя соглашение предусматривает помещение контроллеров в папку `Controllers`, они могут находиться где угодно в проекте и MVC по-прежнему отыщет их.

Листинг 17.7. Содержимое файла `PocoController.cs` из папки `Controllers`

```
namespace ControllersAndActions.Controllers {

  public class PocoController {

    public string Index() => "This is a POCO controller";
  }
}
```

Класс `PocoController` удовлетворяет простому критерию, который MVC ожидает обнаружить в контроллере. Он определяет единственный метод `public` по имени `Index()`, который будет использоваться как метод действия и который возвращает объект `string`.

Класс `PocoController` является примером *контроллера POCO*, где POCO означает "plain old CLR object" ("простой старый объект CLR") и опирается на тот факт, что контроллер реализован с применением стандартных средств .NET без какой-либо прямой зависимости от API-интерфейса, предоставляемого инфраструктурой ASP.NET Core MVC.

Чтобы протестировать контроллер POCO, запустим приложение и запросим URL вида `/Poco/Index/`. Система маршрутизации сопоставит запрос со стандартным шаблоном URL и направит запрос методу `Index()` класса `PocoController` (рис. 17.1).

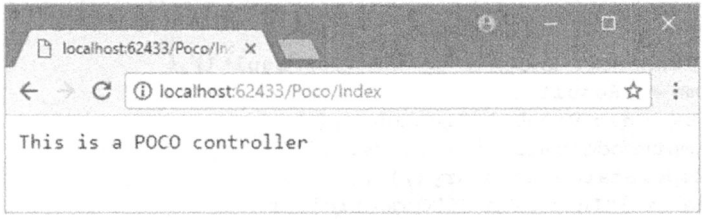


Рис. 17.1. Использование контроллера POCO

Применение атрибутов для корректировки идентификации контроллеров

Поддержка контроллеров POCO не всегда работает желательным образом. Общая проблема заключается в том, что MVC будет идентифицировать в качестве контроллеров фиктивные классы, созданные для модульного тестирования. Избежать указанной проблемы проще всего, уделяя внимание именам классов и не допуская имен вроде `FakeController`. Если это невозможно, тогда к классу необходимо применять атрибут `NonController`, определенный в пространстве имен `Microsoft.AspNetCore.Mvc`, чтобы сообщить MVC о том, что класс не является контроллером. Имеется также атрибут `NonAction`, который может применяться к методам, чтобы предотвратить их использование как методов действий.

В ряде проектов может отсутствовать возможность следования соглашению об именовании для класса, который должен выступать в качестве контроллера POCO. Инфраструктуре MVC можно указать, что класс является контроллером, даже когда он не удовлетворяет критерию выбора POCO, применив атрибут `Controller`, который также определен в пространстве имен `Microsoft.AspNetCore.Mvc`.

Использование API-интерфейса контроллеров инфраструктуры MVC

Класс `PocoController` — удобная демонстрация способа идентификации контроллеров инфраструктурой MVC и возможной реализации простых контроллеров. Но *чистые* контроллеры POCO, не имеющие зависимостей от пространств имен `Microsoft.AspnetCore`, не особенно полезны, поскольку у них отсутствует доступ к средствам, которые MVC предлагает для обработки запросов.

Доступ к определенным частям API-интерфейса MVC может осуществляться за счет создания новых экземпляров классов из пространств имен `Microsoft.AspnetCore`. В качестве простого примера класс POCO может затребовать у MVC визуализацию представления Razor, возвращая объект `ViewResult` из своих методов действий, как показано в листинге 17.8. (Класс `ViewResult` будет рассматриваться в разделе "Генерирование ответа" далее в главе.)

Листинг 17.8. Использование API-интерфейса MVC в файле `PocoController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace ControllersAndActions.Controllers {

  public class PocoController {

    public ViewResult Index() => new ViewResult() {
      ViewName = "Result",
      ViewData = new ViewDataDictionary(
        new EmptyModelMetadataProvider(),
        new ModelStateDictionary()) {
          Model = $"This is a POCO controller"
        }
      };
  }
}
```

Класс `PocoController` больше не является чистым контроллером POCO, т.к. он имеет прямые зависимости от API-интерфейса MVC. Однако, оставив в стороне чистоту, данный класс намного более полезен, чем класс из предыдущего примера, потому что он запрашивает у MVC визуализацию представления Razor. К сожалению, код стал сложным. Чтобы создать объект `ViewResult`, необходимо создать объекты `ViewDataDictionary`, `EmptyModelMetadataProvider` и `ModelStateDictionary`, которые требуют доступа к трем разным пространствам имен. (Средства, к которым относятся упомянутые типы, будут описаны в последующих главах.) Суть рассматриваемого примера — демонстрация того, что к средствам, предлагаемым MVC, можно обращаться напрямую, даже если в результате получается некоторая путаница.

Выделенные полужирным изменения в листинге 17.8 обеспечивают визуализацию представления `Result.cshtml` с применением типа `string` в качестве модели представления. В результате запуска приложения и запрашивания URL вида `/Poco/Index` будет получен ответ, приведенный на рис. 17.2.

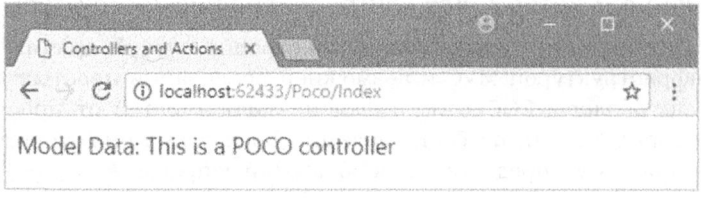


Рис. 17.2. Работа с API-интерфейсом MVC напрямую

Использование базового класса Controller

Предшествующие примеры проиллюстрировали, каким образом начать с контроллера POCO и обеспечить ему доступ к средствам MVC. Такой подход проливает свет на то, как работает MVC; это полезное знание на тот случай, если вы обнаружите, что невольно создаете контроллеры, но контроллеры POCO неудобны в написании, восприятии и сопровождении.

Более легкий способ создания контроллеров предусматривает наследование классов от класса `Microsoft.AspNetCore.Mvc.Controller`, в котором определены методы и свойства, предоставляющие доступ к средствам MVC в сжатой и удобной манере. Добавим в папку `Controllers` файл класса по имени `DerivedController.cs` с определением контроллера, показанным в листинге 17.9.

Листинг 17.9. Наследование от класса `Controller` в файле `DerivedController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

  public class DerivedController : Controller {

    public ViewResult Index() =>
      View("Result", $"This is a derived controller");
  }
}
```

На рис. 17.3 приведен результат запуска приложения и запрашивания URL вида `/Derived/Index`.

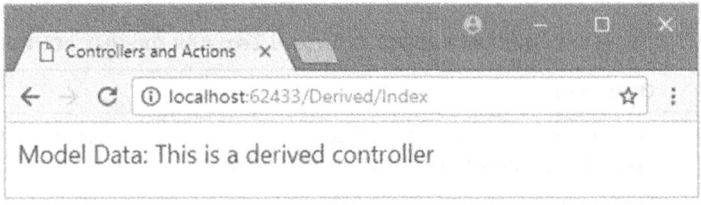


Рис. 17.3. Использование базового класса `Controller`

Контроллер в листинге 17.9 делает то же самое, что и контроллер в листинге 17.8 (он требует у инфраструктуры MVC визуализировать представление с моделью представления `string`), но применение базового класса `Controller` означает возможность получения результата более простым путем.

Основное изменение касается того, что объект `ViewResult`, требующийся для визуализации представления Razor, можно создать с использованием метода `View()` вместо явного создания экземпляра этого класса (и необходимых ему других типов) прямо в методе действия. Метод `View()` унаследован от базового класса `Controller`, а объект `ViewResult` по-прежнему создается тем же способом, просто не загромождая метод действия. Наследование от класса `Controller` не изменяет манеру работы контроллеров; оно всего лишь упрощает код, который пишется для решения общих задач.

На заметку! Инфраструктура MVC создает новый экземпляр класса контроллера для каждого запроса, который ей предлагается обработать. В итоге не придется синхронизировать доступ к методам действий или свойствам и полям экземпляра. Разделяемые объекты, в том числе базы данных и службы-одиночки, которые описаны в главе 18, могут применяться параллельно и должны быть соответствующим образом реализованы.

Получение данных контекста

Независимо от того, как определяются контроллеры, они редко будут существовать в изоляции и обычно нуждаются в доступе к данным из входящего запроса, таким как значения строки запроса, значения формы и параметры, извлеченные из URL системой маршрутизации, что все вместе называют *данными контекста*. Есть три основных способа доступа к данным контекста:

- извлечение данных из набора *объектов контекста*;
- получение данных как *параметра* метода действия;
- явное обращение к средству *привязки моделей* инфраструктуры.

Здесь мы рассмотрим подходы к получению входных данных для методов действий, сосредоточившись на объектах контекста и параметрах методов действий. Привязка моделей раскрывается в главе 26.

Получение данных из объектов контекста

Одно из главных преимуществ использования базового класса `Controller` для создания контроллеров заключается в удобном доступе к набору объектов контекста, которые описывают текущий запрос, подготавливаемый ответ и состояние приложения. В табл. 17.3 описаны наиболее полезные свойства контекста `Controller`.

Таблица 17.3. Полезные свойства класса `Controller` для данных контекста

Имя	Описание
`Request`	Это свойство возвращает объект `HttpRequest`, который описывает запрос, полученный от клиента (табл. 17.4)
`Response`	Это свойство возвращает объект `HttpResponse`, который применяется для создания ответа клиенту (табл. 17.7)
`HttpContext`	Это свойство возвращает объект `HttpContext`, который является источником многих объектов, возвращаемых другими свойствами, такими как `Request` и `Response`. Оно также предоставляет информацию о доступных возможностях HTTP и доступ к низкоуровневым средствам наподобие веб-сокетов
`RouteData`	Это свойство возвращает объект `RouteData`, выпускаемый системой маршрутизации, когда имеется совпадение с запросом, как описано в главах 15 и 16
`ModelState`	Это свойство возвращает объект `ModelStateDictionary`, который используется для проверки достоверности данных, отправленных клиентом (глава 27)
`User`	Это свойство возвращает объект `ClaimsPrincipal`, который описывает пользователя, сделавшего запрос, как объясняется в главах 29 и 30

Многие контроллеры пишутся без применения свойств из табл. 17.3, потому что данные контекста доступны через средства, рассматриваемые в последующих главах,

которые больше гармонируют со стилем разработки приложений MVC. Например, большинство контроллеров не нуждаются в использовании свойства Request для получения деталей обрабатываемого HTTP-запроса, поскольку та же самая информация доступна через процесс привязки модели, который обсуждается в главе 26.

Но свойства, перечисленные в табл. 17.3, все-таки могут быть полезны для понимания и работы с объектами контекста, а также для целей отладки. В листинге 17.10 свойство Request применяется для доступа к заголовкам в HTTP-запросе.

Листинг 17.10. Использование данных контекста в файле DerivedController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
namespace ControllersAndActions.Controllers {
  public class DerivedController : Controller {
    public ViewResult Index() =>
      View("Result", $"This is a derived controller");
    public ViewResult Headers() => View("DictionaryResult",
      Request.Headers.ToDictionary(kvp => kvp.Key,
        kvp => kvp.Value.First()));
  }
}
```

Работа с объектами контекста означает перемещение по диапазону различных типов и пространств имен. Свойство Controller.Request, которое применяется для получения данных контекста о HTTP-запросе в листинге, возвращает объект HttpRequest. В табл. 17.4 описаны свойства HttpRequest, которые наиболее часто используются при написании классов контроллеров.

Таблица 17.4. Распространенные свойства HttpRequest

Имя	Описание
Path	Это свойство возвращает раздел пути URL запроса
QueryString	Это свойство возвращает раздел строки запроса URL запроса
Headers	Это свойство возвращает словарь заголовков запроса, индексированный по именам
Body	Это свойство возвращает поток, который может применяться для чтения тела запроса
Form	Это свойство возвращает словарь данных формы в запросе, индексированный по именам
Cookies	Это свойство возвращает словарь cookie-наборов запроса, индексированный по именам

Свойство Request.Headers предназначено для получения словаря заголовков, который обрабатывается с помощью LINQ:

```
...
View("DictionaryResult", Request.Headers.ToDictionary(kvp => kvp.Key,
  kvp => kvp.Value.First()));
...
```

Словарь, возвращаемый свойством `Request.Headers`, хранит значения всех заголовков с использованием структуры `StringValues`, которая применяется в ASP.NET для представления последовательности строковых значений. Клиент HTTP может отправить для заголовков HTTP несколько значений, но нам нужно отобразить только первое значение. С помощью LINQ-метода `ToDictionary()` для каждого заголовка получается объект `KeyValuePair<string, StringValues>` и выбирается первое значение. Результатом будет словарь, содержащий значения `string`, которые могут отображаться представлением `DictionaryResult`. В результате запуска приложения и запрашивания URL вида `/Derived/Headers` получается вывод, подобный показанному на рис. 17.4. (Набор заголовков и их значений будет отличаться в зависимости от используемого браузера.)

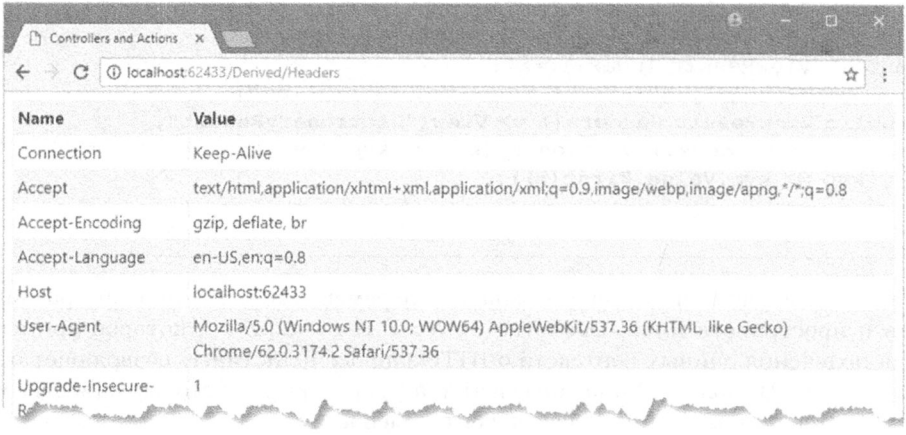


Рис. 17.4. Отображение данных контекста

Получение данных контекста в контроллере POCO

Хотя контроллеры POCO не особенно полезны в обычных проектах, они позволяют заглянуть "за кулисы" и посмотреть, как функционирует MVC. Получение данных контекста в контроллере POCO проблематично, поскольку нельзя просто создать собственные объекты `HttpRequest` или `HttpResponse`; необходимы объекты, которые были созданы ASP.NET и обновлены всеми компонентами промежуточного ПО, заполняющими поля данных этих объектов по мере обработки запроса.

Чтобы получить данные контекста, контроллер POCO должен запросить их предоставление у инфраструктуры MVC. В листинге 17.11 класс `PocoController` модифицирован для добавления метода действия, который отображает заголовки HTTP-запроса.

Листинг 17.11. Отображение данных контекста в файле `PocoController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using System.Linq;
```

```
namespace ControllersAndActions.Controllers {
  public class PocoController {
    [ControllerContext]
    public ControllerContext ControllerContext { get; set; }
    public ViewResult Index() => new ViewResult() {
      ViewName = "Result",
      ViewData = new ViewDataDictionary(new EmptyModelMetadataProvider(),
        new ModelStateDictionary()) {
          Model = $"This is a POCO controller"
        }
      };
    public ViewResult Headers() =>
      new ViewResult() {
        ViewName = "DictionaryResult",
        ViewData = new ViewDataDictionary(
          new EmptyModelMetadataProvider(),
          new ModelStateDictionary()) {
            Model = ControllerContext.HttpContext.Request.Headers
              .ToDictionary(kvp => kvp.Key, kvp => kvp.Value.First())
          }
      };
  }
}
```

Для получения данных контекста определяется свойство по имени `ControllerContext` типа `ControllerContext`, которое декорировано атрибутом, также имеющим имя `ControllerContext`.

Полезно объяснить три разных случая применения термина `ControllerContext`. Первое использование — класс `ControllerContext` из пространства имен `Microsoft.AspNetCore.Mvc`, представляющий собой класс, который собирает вместе все объекты контекста, требующиеся методу действия контроллера, с применением свойств, описанных в табл. 17.5.

Таблица 17.5. Самые важные свойства `ControllerContext`

Имя	Описание
`ActionDescriptor`	Это свойство возвращает объект `ActionDescriptor`, который описывает метод действия
`HttpContext`	Это свойство возвращает объект `HttpContext`, который предоставляет детали HTTP-запроса и отправляемого взамен HTTP-ответа (табл. 17.6)
`ModelState`	Это свойство возвращает объект `ModelStateDictionary`, который используется для проверки достоверности данных, отправляемых клиенту (глава 27)
`RouteData`	Это свойство возвращает объект `RouteData`, описывающий способ, которым система маршрутизации обработала запрос (глава 15)

Доступ к данным, связанным с HTTP, осуществляется через свойство `ControllerContext.HttpContext`, которое возвращает объект `Microsoft.AspNetCore.Http.HttpContext`.

Класс `HttpContext` объединяет несколько объектов, описывающих различные аспекты запроса, которые доступны через свойства, перечисленные в табл. 17.6.

Таблица 17.6. Распространенные свойства `HttpContext`

Имя	Описание
`Connection`	Это свойство возвращает объект `ConnectionInfo`, который описывает низкоуровневое подключение к клиенту
`Request`	Это свойство возвращает объект `HttpRequest`, который описывает HTTP-запрос, полученный от клиента, как объяснялось ранее в главе
`Response`	Это свойство возвращает объект `HttpResponse`, применяемый для создания ответа, который будет возвращен клиенту, как описано в разделе "Генерирование ответа" далее в главе
`Session`	Это свойство возвращает объект `ISession`, описывающий сеанс, с которым ассоциирован запрос
`User`	Это свойство возвращает объект `ClaimsPrincipal`, который описывает пользователя, ассоциированного с запросом (глава 28)

Второе использование атрибута `ControllerContext` продемонстрировано в листинге 17.11: свойство декорируется атрибутом `ControllerContext`, чтобы сообщить инфраструктуре MVC о необходимости установки значения свойства в объект `ControllerContext`, который описывает текущий запрос. В этом случае применяется прием, называемый *внедрением зависимостей*, который рассматривается в главе 18. Инфраструктура MVC будет использовать такое свойство для предоставления контроллеру данных контекста перед вызовом какого-то метода действия, чтобы обработать запрос.

Наконец, третье применение `ControllerContext` связано с именем самого свойства. В контроллерах POCO можно использовать любые допустимые имена свойств C#, но имя `ControllerContext` было выбрано из-за того, что оно применяется классом `Controller`. "За кулисами" класс `Controller` для своих данных контекста полагается на тот же самый класс `ControllerContext`, который декорирован тем же самым атрибутом `ControllerContext`. Все свойства класса `Controller`, описанные в табл. 17.3, являются просто более удобной и краткой альтернативой использованию свойств класса `ControllerContext` напрямую, что в действительности происходит внутри свойств, предоставляемых классом `Controller`. В качестве примера ниже приведено определение свойства `HttpContext` из класса `Controller`:

```
...
public HttpContext HttpContext {
  get {
    return ControllerContext.HttpContext;
  }
}
...
```

Свойство `HttpContext` — лишь удобный способ получить значение свойства `ControllerContext.HttpContext`. Никакой "магии" в базовом классе `Controller` нет: он дает в результате более простые и ясные контроллеры по той причине, что объединяет общие задачи в удобные методы и свойства, которые при необходимости можно воссоздать самостоятельно в контроллере POCO. Если углубиться в детали, то выяснится, что масса функциональности в инфраструктуре ASP.NET Core MVC удивительно проста; здесь отсутствует какая-либо специальная изюминка — только продуманная функциональность, предлагаемая тщательно спроектированным набором пакетов NuGet. При наличии времени рекомендуется удостовериться в этом, загрузив исходный код MVC из `http://github.com/aspnet` и изучив его.

Использование параметров метода действия

Определенные данные контекста могут также получаться через параметры метода действия, что позволяет создавать более естественный и элегантный код. Распространенный пример касается ситуации, когда метод действия нуждается в получении значений данных формы, отправленной пользователем. Для сравнения ниже будет показано, как получить данные формы через объекты контекста и затем через параметры метода действия.

Доступ к значениям данных формы осуществляется посредством свойства `Request.Form` класса `Controller`. В целях демонстрации добавим в папку `Controllers` файл класса по имени `HomeController.cs` и определим в нем производный от `Controller` класс контроллера, код которого приведен в листинге 17.12.

Листинг 17.12. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("SimpleForm");

    public ViewResult ReceiveForm() {
      var name = Request.Form["name"];
      var city = Request.Form["city"];
      return View("Result", $"{name} lives in {city}");
    }
  }
}
```

Метод действия `Index()` в классе контроллера `HomeController` визуализирует представление `SimpleForm`, которое было создано внутри папки `Views/Shared` в начале главы. Нас интересует метод `ReceiveForm()`, потому что он применяет объект контекста `HttpRequest` для получения значений данных формы из запроса.

Как было описано в табл. 17.4, свойство `Form`, определенное в классе `HttpRequest`, возвращает коллекцию значений данных формы, индексированную по именам ассоциированных HTML-элементов. В представлении `SimpleForm` присутствуют два элемента `input` (`name` и `city`), значения которых извлекаются из объекта контекста и используются для создания строки, передаваемой представлению `Result` в качестве его модели.

После запуска приложения и запрашивания URL вида `/Home` отобразится форма. Если заполнить поля и щелкнуть на кнопке Submit (Отправить), то браузер отправит данные формы как часть HTTP-запроса `POST`, который будет обработан методом `ReceiveForm()`, порождая показанный на рис. 17.5 результат.

Подход из листинга 17.12 работает великолепно, но существует более элегантная альтернатива. В методах действий можно определять параметры, которые MVC применяет для передачи контроллеру данных контекста, включая детали HTTP-запроса. Альтернативный подход более лаконичен, чем извлечение данных контекста непосредственно из объектов контекста, и дает методы действий, которые легче в восприятии. Чтобы получить данные формы, объявим в методе действия параметры, имена которых соответствуют значениям данных формы (листинг 17.13).

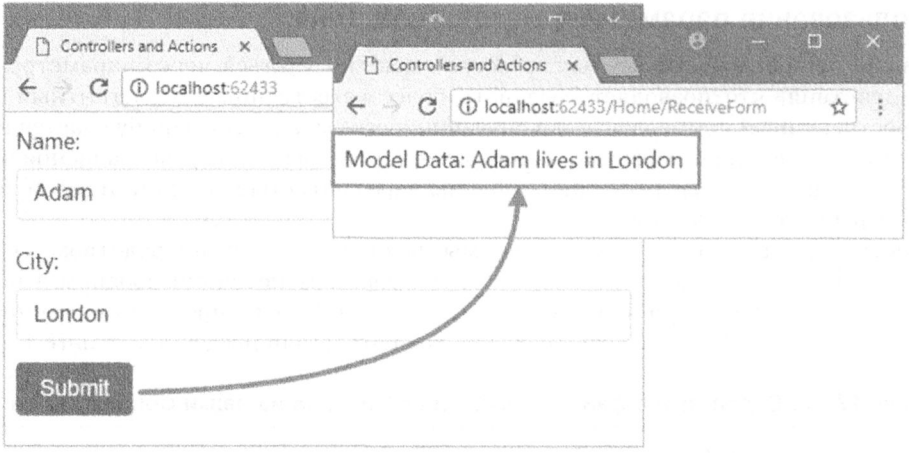


Рис. 17.5. Получение данных формы из объектов контекста

Листинг 17.13. Получение данных контекста как параметров в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("SimpleForm");

    public ViewResult ReceiveForm(string name, string city)
      => View("Result", $"{name} lives in {city}");
  }
}
```

Пересмотренный метод действия выпускает тот же самый результат, но он проще в восприятии. Инфраструктура MVC будет предоставлять значения для параметров метода действия, автоматически проверяя объекты контекста, в том числе `Request.QueryString`, `Request.Form` и `RouteData.Values`. Имена параметров трактуются как нечувствительные к регистру символов, так что параметр метода действия `city` может быть наполнен значением, скажем, из `Request.Form["City"]`. Такой подход также дает методы действий, легче поддающиеся модульному тестированию, поскольку значения, которыми оперирует метод действия, получаются как обычные параметры C# и не требуют имитации объектов контекста.

Генерирование ответа

После того как метод действия завершил обработку запроса, ему необходимо сгенерировать ответ. Для генерирования вывода из методов действий доступно много средств, которые будут описаны в последующих разделах.

Генерирование ответа с использованием объекта контекста

Самый низкоуровневый способ генерации вывода предусматривает применение объекта контекста HttpResponse — именно так инфраструктура ASP.NET Core предоставляет доступ к HTTP-ответу, который будет отправлен клиенту. В табл. 17.7 перечислены базовые свойства класса HttpResponse, определенного в пространстве имен Microsoft.AspNetCore.Http.

Таблица 17.7. Распространенные свойства HttpResponse

Имя	Описание
StatusCode	Это свойство используется для установки кода состояния HTTP, связанного с ответом
ContentType	Это свойство применяется для установки заголовка Content-Type ответа
Headers	Это свойство возвращает словарь заголовков HTTP, которые будут включены в ответ
Cookies	Это свойство возвращает коллекцию, которая используется для добавления cookie-наборов к ответу
Body	Это свойство возвращает объект System.IO.Stream, который применяется для записи данных тела запроса

В листинге 17.14 контроллер Home обновлен, чтобы его действие ReceivedForm генерировало ответ с использованием объекта HttpResponse, возвращаемого свойством Controller.Request.

Листинг 17.14. Выпуск ответа в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Text;

namespace ControllersAndActions.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("SimpleForm");

    public void ReceiveForm(string name, string city) {
      Response.StatusCode = 200;
      Response.ContentType = "text/html";
      byte[] content = Encoding.ASCII
        .GetBytes($"<html><body>{name} lives in {city}</body>");
      Response.Body.WriteAsync(content, 0, content.Length);
    }
  }
}
```

Это плохой способ генерации ответа из-за жесткого кодирования HTML-разметки в методе действия с применением строк C#, что чревато ошибками и трудно поддается модульному тестированию. Тем не менее, мы получаем отправную точку для выяснения, каким образом ответы создаются "за кулисами".

Существуют эффективные альтернативы, которые предусматривают работу непосредственно с объектом HttpResponse. Инфраструктура MVC строит низкоуровневый

ответ с помощью намного более удобного средства, находящегося в центральной части функциональности контроллеров — результата действия.

Понятие результатов действий

Результаты действий используются в MVC для отделения *заявлений о намерениях* от *исполнения намерений*. После освоения концепция покажется простой, но понимание лежащего в основе подхода может занять некоторое время по причине присущей ему косвенности.

Вместо того чтобы иметь дело напрямую с объектом `HttpResponse`, методы действий возвращают объект, который реализует интерфейс `IActionResult` из пространства имен `Microsoft.AspNetCore.Mvc`. Объект реализации `IActionResult`, известный как *результат действия*, описывает, каким должен быть ответ из контроллера, например, визуализация представления или перенаправление клиента на другой URL. Но — и здесь в игру вступает косвенность — ответ не генерируется напрямую. Взамен для выпуска результата инфраструктура MVC обрабатывает результат действия.

На заметку! Система результатов действий является примером *паттерна "Команда" (Command)*. Этот паттерн описывает сценарии, при которых сохраняются и передаются объекты, описывающие выполняемые операции. Дополнительные детали можно найти в книге Роберта Мартина *Гибкая разработка программ на Java и C++: принципы, паттерны и методики* (издательство "Диалектика", 2017 год).

Ниже приведено определение интерфейса `IActionResult` из исходного кода MVC:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc {
  public interface IActionResult {
    Task ExecuteResultAsync(ActionContext context);
  }
}
```

Интерфейс `IActionResult` может показаться простым, но причина в том, что MVC не навязывает виды ответов, которые может выпускать результат действия. Когда метод действия возвращает результат действия, инфраструктура MVC вызывает его метод `ExecuteResultAsync()`, который отвечает за генерирование ответа от имени метода действия. Аргумент `ActionContext` предоставляет данные контекста для генерации ответа, включая объект `HttpResponse`. (Класс `ActionContext` является суперклассом `ControllerContext` и определяет все свойства, описанные в табл. 17.5.)

Чтобы посмотреть, как работают результаты действий, добавим в проект папку `Infrastructure` и поместим в нее файл класса по имени `CustomHtmlResult.cs` с определением результата действия из листинга 17.15.

Листинг 17.15. Содержимое файла `CustomHtmlResult.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using System.Threading.Tasks;

namespace ControllersAndActions.Infrastructure {
```

```
  public class CustomHtmlResult : IActionResult {

    public string Content { get; set; }

    public Task ExecuteResultAsync(ActionContext context) {
      context.HttpContext.Response.StatusCode = 200;
      context.HttpContext.Response.ContentType = "text/html";
      byte[] content = Encoding.ASCII.GetBytes(Content);
      return context.HttpContext.Response.Body.WriteAsync(content,
        0, content.Length);
    }
  }
}
```

Класс `CustomHtmlResult` реализует интерфейс `IActionResult`, а его метод `ExecuteResultAsync()` применяет объект `HttpResponse` для записи HTML-ответа, который содержит значение свойства по имени `Content`. Метод `ExecuteResultAsync()` должен возвращать объект `Task`, так что ответ может строиться асинхронно. Это хорошо согласуется с его реализацией в классе `CustomHtmlResult`, которая полагается на метод `WriteAsync()` объекта `Stream`, представляющего тело ответа. Вызов `WriteAsync()` возвращает объект `Task`, который можно использовать как результат метода `ExecuteResultAsync()` в классе `CustomHtmlResult`.

В листинге 17.16 класс результата действия применяется к контроллеру `Home`, упрощая метод действия `ReceiveForm()` контроллера `Home`.

Листинг 17.16. Использование результата действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("SimpleForm");

    public IActionResult ReceiveForm(string name, string city)
      => new CustomHtmlResult {
        Content = $"{name} lives in {city}"
    };
  }
}
```

Код, отправляющий ответ, теперь определен отдельно от данных, которые содержатся в ответе, что делает метод действия проще и позволяет создавать ответ того же самого типа в других методах действий, не дублируя код.

Модульное тестирование контроллеров и действий

Многие части ASP.NET Core MVC спроектированы для упрощения модульного тестирования, и это особенно справедливо в отношении действий и контроллеров. Наличие такой поддержки объясняется несколькими причинами.

- Тестировать действия и контроллеры можно за пределами веб-сервера.
- Для тестирования результата метода действия проводить разбор какой-либо HTML-разметки не понадобится. Чтобы удостовериться в получении ожидаемых результатов, можно проинспектировать возвращаемый объект реализации `IActionResult`.
- Эмуляция клиентских запросов не нужна. Система привязки моделей MVC позволяет писать методы действий, которые получают входные данные в качестве своих параметров. Для тестирования метода действия необходимо просто вызвать его напрямую и предоставить интересующие значения параметров.

Далее в главе будет показано, как создавать модульные тесты для разных видов результатов действий. В главе 7 приводились инструкции для настройки проекта модульного тестирования; можно также загрузить готовые проекты из хранилища GitHub для настоящей книги (`https://github.com/apress/pro-asp.net-core-mvc-2`).

Генерирование HTML-ответа

В предыдущем разделе была возможность изъять из класса контроллера код, генерирующий ответ, с применением результата действия. Инфраструктура ASP.NET Core MVC укомплектована более гибким подходом к выпуску ответов — классом `ViewResult`.

Класс `ViewResult` — это результат действия, предоставляющий доступ к механизму визуализации Razor, который обрабатывает файлы `.cshtml` для внедрения данных модели и отправляет результат клиенту через механизм контекста `HttpResponse`. Работа механизмов визуализации объясняется в главе 21, а здесь мы сосредоточим внимание на использовании класса `ViewResult` как результата действия.

В листинге 17.17 специальный класс результата действия заменяется классом `ViewResult`, экземпляр которого создается посредством метода `View()`, предоставляемого базовым классом `Controller`.

Листинг 17.17. Применение класса `ViewResult` в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("SimpleForm");

    public ViewResult ReceiveForm(string name, string city)
      => View("Result", $"{name} lives in {city}");
  }
}
```

Объекты `ViewResult` можно создавать напрямую, как демонстрировалось в контроллере POCO в начале главы, но использование метода `View()` дает более простой и краткий код. Класс `Controller` предлагает несколько версий метода `View()`, которые позволяют выбирать представление, подлежащее визуализации, и снабжать его данными модели (табл. 17.8).

Таблица 17.8. Методы `View()` класса `Controller`

Метод	Описание
`View()`	Этот метод создает объект `ViewResult` для стандартного представления, ассоциированного с методом действия, так что вызов `View()` в методе по имени `MyAction()` будет визуализировать представление под названием `MyAction.cshtml`. Данные модели не применяются
`View(view)`	Этот метод создает объект `ViewResult`, который визуализирует указанное представление, так что вызов `View("MyView")` будет визуализировать представление по имени `MyView.cshtml`. Данные модели не используются
`View(model)`	Этот метод создает объект `ViewResult` для стандартного представления, ассоциированного с методом действия, и применяет указанный объект как данные модели
`View(view,model)`	Этот метод создает объект `ViewResult` для указанного представления и использует указанный объект как данные модели

После запуска приложения и отправки формы появится знакомый результат (рис. 17.6).

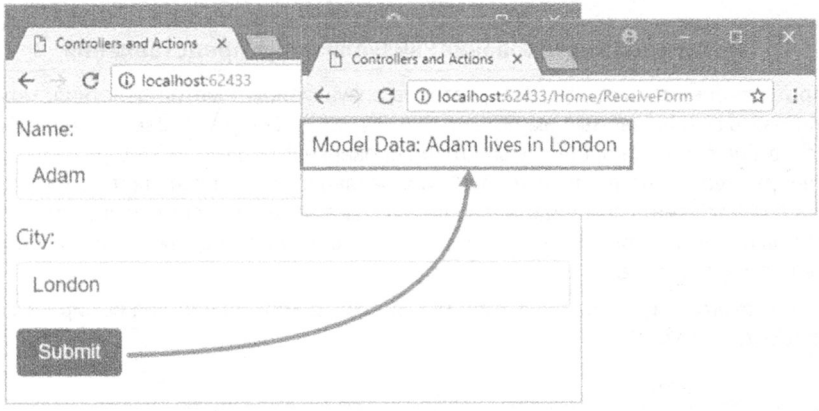


Рис. 17.6. Применение объекта `ViewResult` для генерации HTML-ответа

Поиск файла представления

Когда инфраструктура MVC вызывает метод `ExecuteResultAsync()` объекта `ViewResult`, начинается поиск представления, которое было указано. Последовательность каталогов, где MVC ищет представление, является примером соглашения по конфигурации. Регистрировать файлы представлений с помощью инфраструктуры не придется. Они просто помещаются в одно из набора известных местоположений, и инфраструктура найдет их. По умолчанию MVC будет искать представление в следующих местоположениях:

```
/Views/<ИмяКонтроллера>/<ИмяПредставления>.cshtml
/Views/Shared/<ИмяПредставления>.cshtml
```

Поиск начинается с папки, которая содержит представления, выделенные для текущего контроллера. В имени папки опускается часть `Controller` имени класса, так что папкой для класса `HomeController` будет `Views/Home`.

Если имя представления в объекте `ViewResult` не указано, тогда будет использоваться значение переменной `action` из данных маршрутизации. Для большинства контроллеров это означает применение имени метода действия, следовательно, стандартным файлом представления, ассоциированным с методом `Index()`, будет `Index.cshtml`. Однако если вы использовали атрибут `Route`, то имя представления, связанное с методом действия, может быть другим.

Если контроллер принадлежит области, как было описано в главе 16, тогда местоположения поиска отличаются:

```
/Areas/<ИмяОбласти>/Views/<ИмяКонтроллера>/<ИмяПредставления>.cshtml
/Areas/<ИмяОбласти>/Views/Shared/<ИмяПредставления>.cshtml
/Views/Shared/<ИмяПредставления>.cshtml
```

Инфраструктура MVC проверяет, существует ли каждый из перечисленных файлов, по очереди. Как только обнаруживается совпадение, она применяет найденное представление для визуализации результата метода действия. Области в примере проекта не используются, поэтому метод действия из листинга 17.17 заставит MVC начать поиск с файла `Views/Home/Result.cshtml`. Такого файла не существует, так что MVC продолжит поиск файла `Views/Shared/Result.cshtml`, который будет найден и применен для визуализации HTML-ответа.

Модульное тестирование: визуализация представления

Чтобы протестировать представление, которое визуализирует метод действия, можно проинспектировать возвращаемый им объект `ViewResult`. Это не совсем то же самое (в конце концов, вы не следуете процессу вплоть до проверки финальной HTML-разметки, которая была сгенерирована), но достаточно близко к реальности, т.к. позволяет удостовериться в корректной работе системы представлений MVC. Добавим в проект модульного тестирования новый файл по имени `ActionTests.cs` для хранения модульных тестов, задействованных в настоящей главе.

Первой тестируемой ситуацией является момент, когда метод действия выбирает специфическое представление:

```
...
public ViewResult ReceiveForm(string name, string city)
  => View("Result", $"{name} lives in {city}");
...
```

Прочитав свойство `ViewName` объекта `ViewResult`, как показано в следующем тестовом методе, можно определить, какое представление было выбрано:

```
using ControllersAndActions.Controllers;
using Microsoft.AspNetCore.Mvc;
using Xunit;

namespace ControllersAndActions.Tests {

  public class ActionTests {

    [Fact]
    public void ViewSelected() {
      // Организация
      HomeController controller = new HomeController();
```

```
        // Действие
        ViewResult result = controller.ReceiveForm("Adam", "London");

        // Утверждение
        Assert.Equal("Result", result.ViewName);
    }
  }
}
```

При тестировании метода действия, который выбирает стандартное представление, возникает небольшая вариация:

```
...
public ViewResult Result() => View();
...
```

В таких ситуациях необходимо удостовериться, что именем представления является `null`:

```
...
Assert.Null(result.ViewName);
...
```

С помощью значения `null` объект `ViewResult` сигнализирует MVC о том, что было выбрано стандартное представление, ассоциированное с методом действия.

Указание представления по его пути

Подход с соглашением об именовании для представлений удобен и прост, но он ограничивает представления, которые можно визуализировать. Если требуется визуализировать специфичное представление, то можно предоставить явный путь и пропустить стадию поиска. Ниже приведен пример:

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public ViewResult Index() {
      return View("/Views/Admin/Index");
    }
  }
}
```

При таком указании представления путь должен начинаться с `/` или `~/` и может включать расширение имени файла (которым будет `.cshtml`, если оно не указано).

Когда вы обнаруживаете, что пользуетесь этой возможностью, возьмите паузу и задайте себе вопрос: чего вы стараетесь достичь? Если вы пытаетесь визуализировать представление, принадлежащее другому контроллеру, тогда может быть лучше перенаправить пользователя на метод действия в другом контроллере (пример ищите в разделе "Перенаправление на метод действия" далее в главе). Если вы пытаетесь обойти схему именования файлов представлений, поскольку она не соответствует способу организации вашего проекта, тогда обратитесь в главу 21, в которой объясняется реализация специальной последовательности поиска.

Передача данных из метода действия в представление

Когда объект `ViewResult` используется для выбора представления, из метода действия можно передавать данные для применения при генерации HTML-содержимого.

Инфраструктура MVC предлагает различные способы передачи данных из метода действия в представление, которые описаны в последующих разделах. Эти средства естественным образом касаются темы представлений, которая более подробно раскрывается в главе 21. В настоящей главе мы обсудим только ту функциональность представлений, которой достаточно для демонстрации средств контроллеров.

Использование объекта модели представления

Отправить объект представлению можно путем его передачи в качестве параметра методу `View()`, что приведет к установке свойства `ViewData.Model` созданного объекта `ViewResult`. Ранее в главе указанное свойство устанавливалось напрямую, чтобы объяснить, как работают контроллеры POCO, но метод `View()` позаботится об этом более лаконичным образом. В листинге 17.18 показан добавленный в папку `Controllers` новый класс `ExampleController`, который передает объект модели представления методу `View()`.

Листинг 17.18. Содержимое файла `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public ViewResult Index() => View(DateTime.Now);
  }
}
```

Объект `DateTime` передается методу `View()` с целью применения в качестве модели представления. Для доступа к объекту изнутри представления используется ключевое слово `Model` механизма Razor. Далее создается папка `Views/Example`, куда помещается файл представления по имени `Index.cshtml`, содержимое которого приведено в листинге 17.19.

Листинг 17.19. Содержимое файла `Index.cshtml` из папки `Views/Example`

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Controllers and Actions</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.
min.css" />
</head>
<body class="m-1 p-1">
  Model: @(((DateTime)Model).DayOfWeek)
</body>
</html>
```

Представление в листинге 17.19 называется *нетипизированным* или *слабо типизированным*. Такому представлению ничего не известно об объекте модели представления, и оно трактует его как экземпляр `object`.

Чтобы получить значение свойства `DayOfWeek`, экземпляр `object` понадобится привести к типу `DateTime`:

```
...
Model: @((DateTime)Model).DayOfWeek)
...
```

Прием работает, но порождает запутанные представления. Навести порядок можно за счет создания *строго типизированных представлений*, когда представление включает детали типа объекта модели представления, как демонстрируется в листинге 17.20.

Листинг 17.20. Добавление строго типизированного представления в файле `Index.cshtml` из папки `Views/Example`

```
@model DateTime
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Controllers and Actions</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.
min.css" />
</head>
<body class="m-1 p-1">
  Model: @Model.DayOfWeek
</body>
</html>
```

Тип модели представления указывается с применением ключевого слова `model` синтаксиса Razor. Обратите внимание на использование буквы `m` нижнего регистра при указании типа модели и буквы `M` верхнего регистре при чтении значения.

Строгая типизация не только помогает привести в порядок представление, но также позволяет Visual Studio поддерживать средство IntelliSense (рис. 17.7).



Рис. 17.7. Поддержка средства IntelliSense для строго типизированных представлений

Модульное тестирование: объекты модели представления

Объекты модели представления присваиваются свойству `ViewResult.ViewData.Model`, а это означает возможность проверки того, что метод действия отправляет ожидаемые данные, когда вызывается метод `View()`. Вот тестовый метод, который проверяет тип модели для метода действия из листинга 17.18:

```
...
[Fact]
public void ModelObjectType() {
  // Организация
  ExampleController controller = new ExampleController();

  // Действие
  ViewResult result = controller.Index();

  // Утверждение
  Assert.IsType<System.DateTime>(result.ViewData.Model);
}
...
```

Метод `Assert.IsType()` применяется для проверки того, что объект модели представления является экземпляром `DateTime`.

С использованием метода `View()` связано одно затруднение, которое возникает, когда нужно задействовать стандартное представление, ассоциированное с действием, и снабдить его объектом модели `string` (листинг 17.21).

Листинг 17.21. Использование метода `View()` в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {
  public class ExampleController : Controller {
    public ViewResult Index() => View(DateTime.Now);
    public ViewResult Result() => View("Hello, World");
  }
}
```

В новом методе действия `Result()` необходимо применить метод `View()`, который визуализирует стандартное представление для действия, и указать данные модели, что соответствует третьей версии данного метода из табл. 17.8. Но после запуска приложения и запрашивания URL вида `/Example/Result` будет получено сообщение об ошибке вроде показанного ниже:

```
InvalidOperationException: The view 'Hello, World' was not found.
The following locations were searched:
/Views/Example/Hello, World.cshtml
/Views/Shared/Hello, World.cshtml

InvalidOperationException: Представление 'Hello, World' не найдено.
Поиск производился в следующих местоположениях:
/Views/Example/Hello, World.cshtml
/Views/Shared/Hello, World.cshtml
```

Проблема в том, что вызов метода `View()` с объектом `string` дал совпадение со второй версией данного метода из табл. 17.8, т.е. аргумент `string` был интерпретирован как имя представления, подлежащего визуализации, и потому инфраструктура MVC пыталась найти файл представления под названием `Hello, World.cshtml`, а не `Result.cshtml`. Это распространенная проблема, но ее легко исправить, выполнив приведение данных модели к типу `object` (листинг 17.22).

Листинг 17.22. Выбор корректного метода `View()` в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public ViewResult Index() => View(DateTime.Now);

    public ViewResult Result() => View((object)"Hello, World");
  }
}
```

Явное приведение данных модели к `object` гарантирует соответствие вызова правильной версии метода `View()` и обеспечит визуализацию представления из файла `Result.cshtml`.

Передача данных с помощью `ViewBag`

Объект `ViewBag` был описан в главе 2. Такое средство позволяет определять свойства в динамическом объекте и получать доступ к ним в представлении. Динамический объект доступен через свойство `ViewBag`, предоставляемое классом `Controller`, как демонстрируется в листинге 17.23.

Листинг 17.23. Использование объекта `ViewBag` в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {
  public class ExampleController : Controller {
    public ViewResult Index() {
      ViewBag.Message = "Hello";
      ViewBag.Date = DateTime.Now;
      return View();
    }
    public ViewResult Result() => View((object)"Hello World");
  }
}
```

Свойства `Message` и `Date` объекта `ViewBag` определяются путем присваивания им значений. До этого момента указанные свойства не существовали, и никакая подготовительная работа для их создания не выполнялась. Чтобы прочитать данные в представлении, понадобится извлечь значения из тех же самых свойств, которые устанавливались в методе действия (листинг 17.24).

Листинг 17.24. Чтение данных из объекта ViewBag в файле Index.cshtml из папки Views/Example

```
@model DateTime
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Controllers and Actions</title>
  <link rel="stylesheet"
        asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  <p>The day is: @ViewBag.Date.DayOfWeek</p>
  <p>The message is: @ViewBag.Message</p>
</body>
</html>
```

Преимущество применения объекта ViewBag перед объектом модели представления связано с легкостью отправки множества объектов представлению. Если бы инфраструктура MVC поддерживала только модели представлений, то для получения того же результата пришлось бы создавать новый тип с членами string и DateTime.

Внимание! Среда Visual Studio не способна предоставлять поддержку IntelliSense для любых динамических объектов, включая ViewBag, и ошибки не будут обнаружены до тех пор, пока не произойдет визуализация представления.

Модульное тестирование: объект ViewBag

Свойство ViewResult.ViewData возвращает словарь, ключами которого являются имена свойств ViewBag, определяемых методом действия. Вот тестовый метод для метода действия из листинга 17.23:

```
...
[Fact]
public void ModelObjectType() {
  // Организация
  ExampleController controller = new ExampleController();

  // Действие
  ViewResult result = controller.Index();

  // Утверждение
  Assert.IsType<string>(result.ViewData["Message"]);
  Assert.Equal("Hello", result.ViewData["Message"]);
  Assert.IsType<System.DateTime>(result.ViewData["Date"]);
}
...
```

Данный тестовый метод проверяет типы свойств Message и Date, используя метод Assert.IsType(), и проверяет значение свойства Message с применением метода Assert.Equal().

Выполнение перенаправления

Обычный результат из метода действия предназначен не для выпуска какого-либо вывода напрямую, а для перенаправления клиента на другой URL. Большую часть времени таким URL является другой метод действия внутри приложения, который генерирует вывод, подлежащий отображению пользователям. Когда выполняется перенаправление, браузеру отправляется один из следующих двух кодов HTTP.

- Код HTTP 302, который означает *временное* перенаправление. Это наиболее часто используемый тип перенаправления и именно его необходимо посылать в случае применения паттерна Post/Redirect/Get.
- Код HTTP 301, который означает *постоянное* перенаправление. Этот код должен использоваться с осторожностью, т.к. он инструктирует получателя не запрашивать снова исходный URL, а применять новый URL, включенный вместе с кодом перенаправления. В случае сомнений используйте временное перенаправление, т.е. отправляйте код HTTP 302.

Для выполнения перенаправления могут применяться и другие результаты действий, которые описаны в табл. 17.9.

Таблица 17.9. Результаты действий для перенаправления

Имя	Метод класса `Controller`	Описание
`RedirectResult`	`Redirect()` `RedirectPermanent()`	Этот результат действия посылает ответ с кодом состояния HTTP 301 или HTTP 302, выполняя перенаправление клиента на новый URL
`LocalRedirectResult`	`LocalRedirect()` `LocalRedirectPermanent()`	Этот результат действия выполняет перенаправление клиента на локальный URL
`RedirectToActionResult`	`RedirectToAction()` `RedirectionToActionPermanent()`	Этот результат действия выполняет перенаправление клиента на указанное действие и контроллер
`RedirectToRouteResult`	`RedirectToRoute()` `RedirectToRoutePermanent()`	Этот результат действия выполняет перенаправление клиента на URL, сгенерированный из специфического маршрута

Перенаправление на буквальный URL

Наиболее базовый способ перенаправления браузера предусматривает вызов метода `Redirect()`, предоставляемого классом `Controller`, который возвращает экземпляр класса `RedirectResult` (листинг 17.25).

Листинг 17.25. Перенаправление на буквальный URL в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {
  public class ExampleController : Controller {
    public ViewResult Index() {
      ViewBag.Message = "Hello";
      ViewBag.Date = DateTime.Now;
      return View();
    }
    public ViewResult Result() => View((object)"Hello World");
    public RedirectResult Redirect() => Redirect("/Example/Index");
  }
}
```

Здесь URL перенаправления выражается как аргумент `string` метода `Redirect()`, который обеспечивает временное перенаправление. Постоянное перенаправление можно выполнить с помощью метода `RedirectPermanent()`, как показано в листинге 17.26.

Совет. Объект `LocalRedirectionResult` — это альтернативный результат действия, который будет генерировать исключение, если контроллер попытается выполнить перенаправление на любой URL, не являющийся локальным. Он полезен в ситуации, когда при перенаправлении на URL, которые предоставлены пользователями, предпринимается попытка *атаки открытым перенаправлением* с целью перенаправления другого пользователя на ненадежный сайт. Такой вид результата действия может быть создан посредством метода `LocalRedirect()`, унаследованного от класса `Controller`.

Листинг 17.26. Постоянное перенаправление на буквальный URL в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;
namespace ControllersAndActions.Controllers {
  public class ExampleController : Controller {
    public ViewResult Index() {
      ViewBag.Message = "Hello";
      ViewBag.Date = DateTime.Now;
      return View();
    }
    public ViewResult Result() => View((object)"Hello World");
    public RedirectResult Redirect() => RedirectPermanent("/Example/Index");
  }
}
```

Модульное тестирование: перенаправление на буквальный URL

Перенаправление на буквальный URL тестировать легко. С помощью свойств `Url` и `Permanent` класса `RedirectResult` можно прочитать URL и вид перенаправления (постоянное или временное). Вот тестовый метод для постоянного перенаправления из листинга 17.26:

```
...
[Fact]
public void Redirection() {
  // Организация
  ExampleController controller = new ExampleController();

  // Действие
  RedirectResult result = controller.Redirect();

  // Утверждение
  Assert.Equal("/Example/Index", result.Url);
  Assert.True(result.Permanent);
}
...
```

Обратите внимание, что тест обновлен для получения объекта `RedirectResult` при вызове метода действия.

Перенаправление на URL системы маршрутизации

При перенаправлении пользователя на другую часть приложения необходимо удостовериться в том, что отправляемый URL допустим в рамках схемы URL. Проблема при использовании для перенаправления буквальных URL связана с тем, что любое изменение в схеме маршрутизации означает необходимость пересмотра кода и обновления таких URL. К счастью, можно задействовать систему маршрутизации и генерировать допустимые URL с помощью метода `RedirectToRoute()`, который создает экземпляр класса `RedirectToRouteResult` (листинг 17.27).

Совет. Если вы последовательно прорабатываете примеры из настоящей главы, тогда для нормального функционирования кода в листинге 17.27 может понадобиться очистить хронологию браузера. Причина в том, что браузер запоминает постоянное перенаправление в листинге 17.26, и будет транслировать запрос URL вида `/Example/Redirect` в запрос `/Example/Index`, не контактируя с сервером.

Листинг 17.27. Перенаправление на URL системы маршрутизации в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public ViewResult Index() {
      ViewBag.Message = "Hello";
      ViewBag.Date = DateTime.Now;
      return View();
    }
```

```
    public ViewResult Result() => View((object)"Hello World");
    public RedirectToRouteResult Redirect() =>
      RedirectToRoute(new { controller = "Example",
                            action = "Index",
                            ID = "MyID" });
  }
}
```

Метод `RedirectToRoute()` выпускает временное перенаправление. Для постоянного перенаправления должен применяться метод `RedirectToRoutePermanent()`. Оба метода принимают анонимный тип, свойства которого затем передаются системе маршрутизации для генерирования URL, как объяснялось в главе 16.

Модульное тестирование: перенаправление на URL системы маршрутизации

Ниже приведен модульный тест для метода действия из листинга 17.27:

```
...
[Fact]
public void Redirection() {
  // Организация
  ExampleController controller = new ExampleController();
  // Действие
  RedirectToRouteResult result = controller.Redirect();
  // Утверждение
  Assert.False(result.Permanent);
  Assert.Equal("Example", result.RouteValues["controller"]);
  Assert.Equal("Index", result.RouteValues["action"]);
  Assert.Equal("MyID", result.RouteValues["ID"]);
}
...
```

Результат проверяется косвенным образом за счет просмотра информации о маршруте, предоставляемой объектом `RedirectToRouteResult`, поэтому нет необходимости в разборе URL, что потребовало бы допущений в модульном тесте о схеме URL, используемой приложением.

Перенаправление на метод действия

Выполнить перенаправление на метод действия можно более элегантно с применением метода `RedirectToAction()` (для временного перенаправления) или метода `RedirectToActionPermanent()` (для постоянного перенаправления). Данные методы — просто оболочки вокруг метода `RedirectToRoute()`, которые позволяют указывать значения для метода действия и контроллера без необходимости в создании анонимного типа (листинг 17.28).

Листинг 17.28. Перенаправление с использованием метода `RedirectToAction()` в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {
```

```
public class ExampleController : Controller {
  public ViewResult Index() {
    ViewBag.Message = "Hello";
    ViewBag.Date = DateTime.Now;
    return View();
  }
  public RedirectToActionResult Redirect() => RedirectToAction(nameof(Index));
}
}
```

Если указан только метод действия, то предполагается, что он относится к текущему контроллеру. Для перенаправления на другой контроллер понадобится предоставить в параметре имя контроллера:

```
...
public RedirectToActionResult Redirect()
  => RedirectToAction(nameof(HomeController),
                      nameof(HomeController.Index));
...
```

Доступны и другие перегруженные версии, с помощью которых можно предоставлять дополнительные значения для генерации URL. Они выражаются с применением анонимного типа, который отнюдь не умаляет пользу от удобного метода, но может облегчить восприятие кода.

На заметку! Значения, предоставляемые методу действия и контроллеру, не проверяются перед их передачей системе маршрутизации. Вы сами отвечаете за то, что указываемые цели действительно существуют.

Модульное тестирование: перенаправления на метод действия

Вот модульный тест для метода действия из листинга 17.28:

```
...
[Fact]
public void Redirection() {
  // Организация
  ExampleController controller = new ExampleController();
  // Действие
  RedirectToActionResult result = controller.Redirect();
  // Утверждение
  Assert.False(result.Permanent);
  Assert.Equal("Index", result.ActionName);
}
...
```

В классе `RedirectToActionResult` определены свойства `ControllerName` и `ActionName`, которые упрощают инспектирование перенаправления, созданного контроллером, и не требуют разбора URL.

Использование паттерна Post/Redirect/Get

Перенаправление чаще всего применяется в методах действий, которые обрабатывают HTTP-запросы POST. Как объяснялось в предыдущей главе, запросы POST используются, когда нужно изменить состояние приложения. Если после обработки запроса POST просто возвращать HTML-ответ, то есть риск того, что пользователь щелкнет на кнопке перезагрузки страницы в браузере и отправит форму во второй раз, приводя к непредсказуемым и нежелательным результатам.

Наблюдать такую проблему можно в контроллере Home внутри примера приложения. Метод ReceiveForm() принимает параметры, значения которых получаются из данных формы, и применяет метод View() для возвращения объекта ViewResult:

```
...
public ViewResult ReceiveForm(string name, string city)
  => View("Result", $"{name} lives in {city}");
...
```

Чтобы взглянуть на проблему, запустим приложение и запросим URL вида /Home. Отправим форму и затем щелкнем на кнопке перезагрузки страницы в браузере. С помощью инструментов, доступных по нажатию клавиши <F12>, изучим HTTP-запросы, сделанные браузером; легко заметить, что серверу был отправлен новый запрос POST. В таком простом приложении он не оказывает какого-либо влияния, но если запросы POST в итоге удаляют данные, отправляют заказы или выполняют другие важные задачи, которые пользователь не намеревался предпринимать, то проблема может нанести ущерб.

Чтобы избежать возникновения описанной проблемы, можно следовать паттерну под названием Post/Redirect/Get, согласно которому мы получаем запрос POST, обрабатываем его и затем выполняем перенаправление браузера, так что он делает запрос GET для другого URL. Запросы GET не должны модифицировать состояние приложения, поэтому случайные повторные отправки данного запроса не вызовут никаких проблем. В листинге 17.29 добавлено перенаправление браузера на другой URL с помощью запроса GET.

Листинг 17.29. Реализация паттерна Post/Redirect/Get в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("SimpleForm");

    [HttpPost]
    public RedirectToActionResult ReceiveForm(string name, string city)
      => RedirectToAction(nameof(Data));

    public ViewResult Data() => View("Result");
  }
}
```

Метод RedirectToActionResult получает данные от пользователя через запрос POST и выполняет перенаправление клиента на метод действия Data(). Если пользо-

ватель перезагрузит страницу, тогда методу действия Data() посылается безвредный запрос GET. Атрибут HttpPost, который будет описан в главе 20, гарантирует возможность отправки действию ReceiveFor() только запросов POST.

Использование объекта TempData

Перенаправление заставляет браузер послать полностью новый HTTP-запрос, что означает отсутствие доступа к данным формы из исходного запроса. Таким образом, методу Data() ничего не известно о значениях name и city, которые должны быть отображены пользователю.

Для предохранения данных между запросами можно применять средство TempData. Объект TempData похож на данные сеанса, которые использовались в главе 9, за исключением того, что значения TempData помечаются для удаления, когда они прочитаны, и удаляются из хранилища данных после обработки запроса. Это идеальный вариант для краткосрочных данных, которые необходимы для того, чтобы перенаправление работало в паттерне Post/Redirect/Get. Объект TempData доступен через свойство TempData класса Controller (листинг 17.30).

На заметку! Объект TempData полагается на промежуточное ПО сеанса. В начале главы внутри класса Startup были включены компоненты промежуточного ПО, требующиеся для данного средства.

Листинг 17.30. Использование объекта TempData в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using ControllersAndActions.Infrastructure;

namespace ControllersAndActions.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() {
      return View("SimpleForm");
    }

    [HttpPost]
    public RedirectToActionResult ReceiveForm(string name, string city) {
      TempData["name"] = name;
      TempData["city"] = city;
      return RedirectToAction(nameof(Data));
    }

    public ViewResult Data() {
      string name = TempData["name"] as string;
      string city = TempData["city"] as string;
      return View("Result", $"{name} lives in {city}");
    }
  }
}
```

Метод ReceiveForm() применяет свойство TempData, возвращающее словарь, для сохранения значений name и city перед перенаправлением клиента на действие

Data. Метод Data() использует то же самое свойство TempData для извлечения значений данных и их применения при создании данных модели, которые будут отображаться представлением.

Совет. Словарь TempData также предоставляет метод Peek(), который позволяет получить значение данных, не помечая его для удаления, и метод Keep(), который можно использовать, чтобы предотвратить удаление ранее прочитанного значения. Метод Keep() не защищает значение навсегда. Если значение прочитано снова, то оно будет помечено для удаления еще раз. Если вы хотите хранить элементы, чтобы они не удалялись, когда запрос будет обработан, тогда применяйте данные сеанса.

Возвращение разных типов содержимого

HTML — не единственный вид ответа, который способны генерировать методы действий, и в табл. 17.10 описаны встроенные результаты действий, которые можно использовать для разных типов данных.

Таблица 17.10. Результаты действий для содержимого

Имя	Метод класса Controller	Описание
JsonResult	Json	Этот результат действия сериализирует объект в формат JSON и возвращает его клиенту
ContentResult	Content	Этот результат действия отправляет ответ, тело которого содержит указанный объект
ObjectResult	-	Этот результат действия будет применять согласование содержимого для отправки объекта клиенту
OkObjectResult	Ok	Этот результат действия будет использовать согласование содержимого для отправки объекта клиенту с кодом состояния HTTP 200, если согласование содержимого оказалось успешным
NotFoundObjectResult	NotFound	Этот результат действия будет применять согласование содержимого для отправки объекта клиенту с кодом состояния HTTP 404, если согласование содержимого оказалось успешным

Генерирование ответа JSON

Формат JSON (JavaScript Object Notation — система обозначений для объектов JavaScript) стал стандартным способом передачи данных между веб-приложением и его клиентами. Формат JSON почти полностью заместил XML в качестве формата обмена, поскольку с ним проще работать, особенно при написании кода JavaScript клиентской стороны, т.к. JSON тесно связан с синтаксисом, который JavaScript использует для определения литеральных значений данных. Тема формата JSON и его роли в веб-приложениях раскрывается в главе 20, а в листинге 17.31 демонстрируется применение метода Json() для создания объекта JsonResult.

Листинг 17.31. Генерирование ответа JSON в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public JsonResult Index() => Json(new[] { "Alice", "Bob", "Joe" });
  }
}
```

В результате запуска приложения и запрашивания URL вида `/Example` будет получен ответ, который выражает строковый массив C# из метода действия в формате JSON:

```
["Alice","Bob","Joe"]
```

Большинство браузеров отображают результаты JSON встроенным образом, но некоторые, включая Microsoft Internet Explorer, требуют сохранения данных в файл, прежде чем их можно будет инспектировать.

Модульное тестирование: результаты действий, отличающиеся от HTML

Важно помнить, что модульные тесты для метода действия должны быть сосредоточены на данных, возвращаемых с целью последующего форматирования, а не на самом форматировании, которое поддерживается инфраструктурой MVC и обычно выходит за рамки большинства проектов тестирования. В качестве примера ниже показан модульный тест для метода действия из листинга 17.31:

```
...
[Fact]
public void JsonActionMethod() {
  // Организация
  ExampleController controller = new ExampleController();

  // Действие
  JsonResult result = controller.Index();

  // Утверждение
  Assert.Equal(new[] { "Alice", "Bob", "Joe" }, result.Value);
}
...
```

Класс `JsonResult` предлагает свойство `Value`, возвращающее данные, которые будут преобразованы в формат JSON для формирования ответа клиенту. В модульном тесте производится сравнение свойства `Value` с данными, которые ожидается получить.

Использование объектов для генерации ответов

Многие приложения нуждаются только в ответах HTML и JSON от контроллеров, а при доставке других типов содержимого, таких как изображения, файлы JavaScript и таблицы стилей CSS, рассчитывают на поддержку для статических файлов. Тем не менее, могут возникать ситуации, когда необходимо возвращать в ответе содержимое специфического типа, и справиться с задачей помогут соответствующие результаты

действий. Простейшим является класс `ContentResult`, создаваемый посредством метода `Content()`, который применяется для отправки значения `string` с дополнительным типом содержимого MIME. В листинге 17.32 метод `Content()` используется для воссоздания вручную результата JSON из предыдущего раздела.

Листинг 17.32. Ручное создание результата JSON в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public ContentResult Index()
      => Content("[\"Alice\",\"Bob\",\"Joe\"]", "application/json");
  }
}
```

Такой тип результата действия полезен, когда имеется содержимое, которое удобно представлять в формате `string`, и известно, что клиент способен принимать указываемый тип MIME. Опасность этого подхода в том, что клиенту может быть отправлен ответ в формате, который он не в состоянии обработать. Более надежный подход полагается на согласование содержимого, которое выполняется классом `ObjectResult` (листинг 17.33).

Листинг 17.33. Применение согласования содержимого в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public ObjectResult Index() => Ok(new string[] { "Alice", "Bob", "Joe" });
  }
}
```

Термин *согласование содержимого* намекает на сложную систему выявления общего формата между браузером и приложением, но в действительности представляет собой простой процесс. Когда браузер делает HTTP-запрос, он включает в него заголовок `Accept`, который указывает, какие форматы он может обрабатывать. Вот как выглядит такой заголовок в версии браузера Google Chrome, используемой для тестирования примера:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
  image/webp,*/*;q=0.8
```

Поддерживаемые форматы выражаются как типы MIME. Инфраструктура MVC имеет набор форматов, которые она может применять для значений данных, и сравнивает их с форматами, поддерживаемыми браузером. Предпочтительным форматом, используемым инфраструктурой MVC, является JSON, и он будет применяться в большинстве случаев кроме ситуации, когда действие возвращает значение `string` и используется простой текст. Процесс согласования содержимого и особенности его реализации более подробно рассматриваются в главе 20.

Реагирование с помощью содержимого файлов

Большинство приложений при доставке содержимого файлов рассчитывают на промежуточное ПО статических файлов, но есть также набор результатов действий, которые можно применять для отправки файлов клиенту (табл. 17.11).

Внимание! Будьте осторожны при использовании этих результатов действий, чтобы не создать приложение, которое позволит запрашивать содержимое произвольных файлов. В частности, не получайте путь к подлежащему отправке файлу из любой части запроса либо из любого хранилища данных, которое пользователь может модифицировать через запрос.

Таблица 17.11. Результаты действий для файлов

Имя	Метод класса Controller	Описание
FileContentResult	File()	Этот результат действия посылает клиенту байтовый массив с указанным типом MIME
FileStreamResult	File()	Этот результат действия читает поток и отправляет содержимое клиенту
VirtualFileResult	File()	Этот результат действия читает поток из виртуального пути (относительного к каталогу, где размещается приложение)
PhysicalFileResult	PhysicalFile()	Этот результат действия читает содержимое файла из указанного пути и посылает его клиенту

В листинге 17.34 метод File(), унаследованный от класса Controller, применяется для возвращения CSS-файла Bootstrap в качестве результата метода действия Index() контроллера Example.

Листинг 17.34. Использование содержимого файла в качестве ответа в файле ExampleController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public VirtualFileResult Index()
      => File("/lib/bootstrap/dist/css/bootstrap.css", "text/css");
  }
}
```

Для применения метода действия Index() понадобится модифицировать элемент link в файле SimpleForm.cshtml, чтобы в нем использовался вспомогательный класс Url (листинг 17.35).

Листинг 17.35. Нацеливание на метод действия `Index()` в файле `SimplerForm.cshtml` из папки `Views/Shared`

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Controllers and Actions</title>
  <link rel="stylesheet" href="@Url.Action("Index", "Example")" />
</head>

<body class="m-1 p-1">
  <form method="post" asp-action="ReceiveForm">
    <div class="form-group">
      <label for="name">Name:</label>
      <input class="form-control" name="name" />
    </div>
    <div class="form-group">
      <label for="name">City:</label>
      <input class="form-control" name="city" />
    </div>
    <button class="btn btn-primary center-block" type="submit">Submit
    </button>
  </form>
</body>
</html>
```

После запуска приложения и запрашивания URL вида `/Home` отправленный браузеру HTML-ответ будет включать следующий элемент:

```
<link rel="stylesheet" href="/Example" />
```

Он заставит браузер послать HTTP-запрос, нацеленный на метод действия из листинга 17.35, который отправит файл CSS, требующийся для стилизации содержимого представления.

На заметку! В главе 25 будет показано, что вспомогательные функции дескрипторов являются намного более удобным инструментом для доставки файлов CSS.

Возвращение ошибок и кодов HTTP

Финальный набор встроенных классов `ActionResult` может применяться для отправки клиенту специфичных сообщений об ошибках и результирующих кодов HTTP (табл. 17.12). Большинство приложений не нуждаются в таких средствах, поскольку ASP.NET Core и MVC генерируют эти виды результатов автоматически. Однако они могут быть полезны в ситуациях, когда необходимо получить более прямой контроль над ответами, отправляемыми клиенту.

Таблица 17.12. Результаты действий для кодов состояния

Имя	Метод класса Controller	Описание
StatusCodeResult	StatusCode()	Этот результат действия отправляет клиенту указанный код состояния HTTP
OkResult	Ok()	Этот результат действия отправляет клиенту код состояния HTTP 200
CreatedResult	Created()	Этот результат действия отправляет клиенту код состояния HTTP 201
CreatedAtActionResult	CreatedAtAction()	Этот результат действия отправляет клиенту код состояния HTTP 201 вместе с URL в заголовке Location, который нацелен на действие и контроллер
CreatedAtRouteResult	CreatedAtRoute()	Этот результат действия отправляет клиенту код состояния HTTP 201 вместе с URL в заголовке Location, который сгенерирован из специфического маршрута
BadRequestResult	BadRequest()	Этот результат действия отправляет клиенту код состояния HTTP 400
UnauthorizedResult	Unauthorized()	Этот результат действия отправляет клиенту код состояния HTTP 401
NotFoundResult	NotFound()	Этот результат действия отправляет клиенту код состояния HTTP 404
UnsupportedMediaTypeResult	-	Этот результат действия отправляет клиенту код состояния HTTP 415

Отправка специфического результирующего кода HTTP

Отправить браузеру специфичный код состояния HTTP можно с использованием метода StatusCode(), который создает объект StatusCodeResult (листинг 17.36).

Листинг 17.36. Отправка специфического кода состояния в файле ExampleController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
```

```
namespace ControllersAndActions.Controllers {
  public class ExampleController : Controller {
    public StatusCodeResult Index()
      => StatusCode(StatusCodes.Status404NotFound);
  }
}
```

Метод `StatusCode()` принимает значение `int`, в котором можно напрямую указывать код состояния. Класс `StatusCodes` из пространства имен `Microsoft.AspNetCore.Http` определяет поля для всех кодов состояния, поддерживаемых HTTP. В листинге 17.36 применяется поле `Status404NotFound` для возвращения кода HTTP 404, который означает, что запрошенный ресурс не существует.

Отправка результата 404 с использованием удобного класса

Другие результаты действий, описанные в табл. 17.12, расширяют или основаны на классе `StatusCodeResult`, который предлагает более удобный способ отправки специфических кодов состояния. Того же самого результата, что и в листинге 17.36, можно достичь с применением более удобного класса `NotFoundResult`, который является производным от класса `StatusCodeResult` и может быть создан с использованием удобного метода `NotFound()` класса `Controller` (листинг 17.37).

Листинг 17.37. Генерирование результата 404 в файле `ExampleController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace ControllersAndActions.Controllers {
  public class ExampleController : Controller {
    public StatusCodeResult Index() => NotFound();
  }
}
```

Модульное тестирование: коды состояния HTTP

Класс `StatusCodeResult` следует шаблону, который вы уже видели для других типов результатов, и делает свое состояние доступным через набор свойств. В этом случае свойство `StatusCode` возвращает числовой код состояния HTTP, а свойство `StatusDescription` — связанную описательную строку. Приведенный ниже тестовый метод предназначен для проверки метода действия из листинга 17.37:

```
...
[Fact]
public void NotFoundActionMethod() {
  // Организация
  ExampleController controller = new ExampleController();
  // Действие
  StatusCodeResult result = controller.Index();
  // Утверждение
  Assert.Equal(404, result.StatusCode);
}
...
```

Другие классы результатов действий

Некоторые дополнительные классы результатов действий тесно связаны со средствами MVC, рассматриваемыми в других главах. В табл. 17.13 эти классы кратко описаны со ссылками на главы, где обсуждаются средства, к которым они имеют отношение.

Таблица 17.13. Другие классы результатов действий

Имя	Метод класса `Controller`	Описание
`PartialViewResult`	`PartialView()`	Этот результат действия применяется для выбора частичного представления, как объясняется в главе 21
`ViewComponentResult`	`ViewComponent()`	Этот результат действия используется для выбора компонента представления, как описано в главе 22
`EmptyResult`	-	Этот результат действия ничего не делает и производит пустой ответ для клиента
`ChallengeResult`	-	Этот результат действия применяется для обеспечения соблюдения политик безопасности в запросах. Подробные сведения приведены в главе 30

Резюме

Контроллеры являются одним из основных строительных блоков в паттерне проектирования MVC и находятся в центральной части разработки приложений MVC. В главе было показано, как создавать контроллеры POCO с использованием обычных классов C# и получать преимущество от удобства, предлагаемого базовым классом `Controller`. Вы узнали, какую роль результаты действий играют в контроллерах MVC, и выяснили, каким образом они облегчают модульное тестирование. Были продемонстрированы различные способы получения ввода и генерации вывода из метода действия, а также встроенные результаты действий, которые делают этот процесс простым и гибким. В следующей главе будет описано одно из средств, которое вызывает наибольшую путаницу у разработчиков, использующих ASP.NET, но очень важно для эффективной разработки приложений MVC — внедрение зависимостей.

ГЛАВА 18

Внедрение зависимостей

В настоящей главе рассматривается *внедрение зависимостей* (dependency injection — DI) — методика, которая помогает создавать гибкие приложения и содействует упрощению модульного тестирования. Внедрение зависимостей может оказаться трудной в понимании темой, как в плане того, почему оно может быть полезным, так и в отношении того, каким образом оно выполняется. Соответственно мы будем двигаться медленно, начав с традиционного способа построения компонентов приложения и постепенно разбираясь, как работает внедрение зависимостей и почему оно важно. В табл. 18.1 приведена сводка, позволяющая поместить внедрение зависимостей в контекст.

Таблица 18.1. Помещение внедрения зависимостей в контекст

Вопрос	Ответ
Что это такое?	Внедрение зависимостей облегчает создание слабо связанных компонентов, что обычно относится к компонентам, которые потребляют функциональность, определяемую интерфейсами, и не обладают непосредственным знанием о том, какие классы реализации используются
Чем оно полезно?	Внедрение зависимостей упрощает изменение поведения приложения путем изменения компонентов, которые реализуют интерфейсы, определяющие функциональные средства приложения. Оно также дает в результате компоненты, которые легче изолировать для модульного тестирования
Как оно используется?	С помощью класса `Startup` указывается, какие классы реализации применяются для доставки функциональности, определяемой интерфейсами, которые использует приложение. Когда для обработки запросов создаются новые объекты, такие как контроллеры, они автоматически снабжаются требуемыми экземплярами классов реализации
Существуют ли какие-то скрытые ловушки или ограничения?	Главное ограничение заключается в том, что классы объявляют о своем использовании служб в виде аргументов конструктора. Это может привести к появлению конструкторов, единственной ролью которых является получение зависимостей и их присваивание полям экземпляра
Существуют ли альтернативы?	Вы не обязаны применять внедрение зависимостей в собственном коде, но полезно знать, каким образом оно работает, поскольку внедрение зависимостей используется инфраструктурой MVC для предоставления функциональных средств разработчикам

В табл. 18.2 приведена сводка по главе.

Таблица 18.2. Сводка по главе

Задача	Решение	Листинг
Создание слабо связанных компонентов	Изолируйте классы посредством интерфейсов и соедините их вместе с использованием внешних отображений	18.9–18.16
Объявление зависимости в компоненте, таком как контроллер	Определите аргумент конструктора с типом, который требует компонент	18.17
Конфигурирование отображения службы	Добавьте отображение в класс `Startup`	18.18, 18.20–18.26
Модульное тестирование компонента с зависимостью	Создайте имитированную реализацию интерфейса службы и передайте ее как аргумент конструктора при создании компонента в модульном тесте	18.19
Указание способа для создания объектов реализации	Создайте отображение службы с применением метода жизненного цикла, который подходит управляемой службе	18.27–18.31
Получение зависимостей для отдельных методов действий в контроллере	Применяйте внедрение в действия	18.32
Запрос вручную объекта реализации в контроллере	Используйте свойство `HttpContext.RequestServices`	18.33

Подготовка проекта для примера

Для целей этой главы мы создадим новый проект типа Empty (Пустой) по имени `DependencyInjection` с применением шаблона ASP.NET Core Web Application (.NET Core) (Веб-приложение ASP.NET Core (.NET Core)). В листинге 18.1 приведен класс `Startup`, который конфигурирует службы и компоненты промежуточного ПО для проекта.

Листинг 18.1. Содержимое файла `Startup.cs` из папки `DependencyInjection`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace DependencyInjection {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание модели и хранилища

Для примеров, рассматриваемых в главе, необходима простая модель. Создадим папку `Models` и добавим в нее файл класса по имени `Product.cs` с определением, показанным в листинге 18.2.

Листинг 18.2. Содержимое файла `Product.cs` из папки `Models`

```
namespace DependencyInjection.Models {

  public class Product {

    public string Name { get; set; }
    public decimal Price { get; set; }
  }
}
```

Для управления моделью добавим в папку `Models` файл класса по имени `IRepository.cs` и определим в нем интерфейс, представленный в листинге 18.3.

Листинг 18.3. Содержимое файла `IRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DependencyInjection.Models {

  public interface IRepository {

    IEnumerable<Product> Products { get; }

    Product this[string name] { get; }

    void AddProduct(Product product);
    void DeleteProduct(Product product);
  }
}
```

В интерфейсе `IRepository` определены операции, которые могут выполняться над коллекцией объектов `Product`. Чтобы предоставить реализацию этого интерфейса, добавим в папку `Models` файл класса по имени `MemoryRepository.cs` и поместим в него содержимое из листинга 18.4.

Листинг 18.4. Содержимое файла `MemoryRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DependencyInjection.Models {
```

```
public class MemoryRepository : IRepository {
  private Dictionary<string, Product> products;

  public MemoryRepository() {
    products = new Dictionary<string, Product>();
      new List<Product> {
        new Product { Name = "Kayak", Price = 275M },
        new Product { Name = "Lifejacket", Price = 48.95M },
        new Product { Name = "Soccer ball", Price = 19.50M }
    }.ForEach(p => AddProduct(p));
  }

  public IEnumerable<Product> Products => products.Values;

  public Product this[string name] => products[name];

  public void AddProduct(Product product) =>
    products[product.Name] = product;

  public void DeleteProduct(Product product) =>
    products.Remove(product.Name);
 }
}
```

Класс `MemoryRepository` хранит свои объекты модели в памяти, используя словарь. Другими словами, постоянное хранилище отсутствует, и останов либо перезапуск приложения сбросит модель к примерам объектов данных, которые создаются в конструкторе. В реальном проекте такой подход нецелесообразен, но его будет достаточно для текущей главы, где внимание сосредоточено на другом аспекте работы приложения.

Создание контроллера и представления

Создадим папку `Controllers` и добавим в нее файл класса по имени `HomeController.cs` с содержимым, показанным в листинге 18.5.

Листинг 18.5. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace DependencyInjection.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View();
  }
}
```

Контроллер `Home` имеет только один метод действия, применяющий метод `View()` для создания объекта `ViewResult`, который будет визуализировать стандартное представление. Чтобы получить представление, ассоциированное с методом действия, создадим папку `Views/Home` и добавим в нее файл представления Razor по имени `Index.cshtml`. Разметка, помещаемая в представление, приведена в листинге 18.6.

Листинг 18.6. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>
@{ Layout = null; }
```

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Dependency Injection</title>
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
</head>
<body class="m-1 p-1">
  @if (ViewData.Count > 0) {
    <table class="table table-bordered table-sm table-striped">
      @foreach (var kvp in ViewData) {
        <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
      }
    </table>
  }
  <table class="table table-bordered table-sm table-striped">
    <thead>
      <tr><th>Name</th><th>Price</th></tr>
    </thead>
    <tbody>
      @if (Model == null) {
        <tr><td colspan="3" class="text-center">No Model Data</td></tr>
      } else {
        @foreach (var p in Model) {
          <tr>
            <td>@p.Name</td>
            <td>@string.Format("{0:C2}", p.Price)</td>
          </tr>
        }
      }
    </tbody>
  </table>
</body>
</html>
```

Представление строго типизировано с использованием перечисления объектов `Product`, и основным содержимым представления является HTML-таблица. Если контроллер не предоставляет какие-либо данные модели, тогда в таблице будет отображаться только подходящее сообщение. При наличии данных модели для каждого объекта `Product` из перечисления к таблице добавляется строка. Имеется также таблица, в которой будут перечислены ключи и значения из объекта `ViewBag`, если они есть; в противном случае она будет скрыта. Данная таблица будет задействована позже в главе.

При стилизации HTML-элементов представление полагается на пакет CSS из Bootstrap. Чтобы добавить Bootstrap в проект, создадим файл `bower.json` с применением шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в разделе `dependencies` (листинг 18.7).

Листинг 18.7. Добавление пакета Bootstrap в файле `bower.json` из папки `DependencyInjection`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Последний подготовительный шаг предусматривает создание в папке `Views` файла `_ViewImports.cshtml`, который устанавливает встроенные вспомогательные функции дескрипторов для использования в представлениях Razor и импортирует пространство имен модели (листинг 18.8).

Листинг 18.8. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@using DependencyInjection.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Создание проекта модульного тестирования

Создадим с помощью шаблона xUnit Test Project (.NET Core) (Проект тестирования xUnit (.NET Core)) проект по имени `DependencyInjection.Tests`, следуя инструкциям из главы 7. Удалим файл `UnitTest1.cs`, чтобы в проекте не осталось тестов.

На рис. 18.1 показан результат запуска приложения.

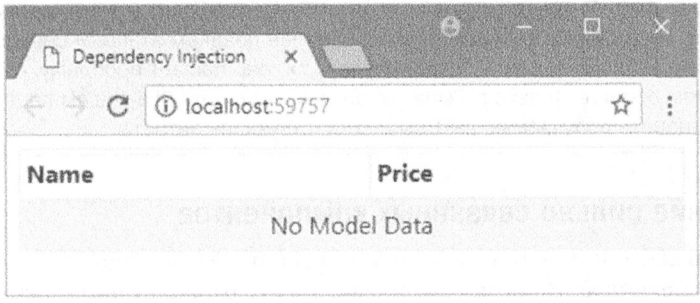


Рис. 18.1. Запуск примера приложения

Создание слабо связанных компонентов

Как видно на рис. 18.1, данные модели отсутствуют. Причина в том, что не установлено отношение между классом `HomeController`, который должен передавать данные модели своему представлению, и классом `MemoryRepository`, содержащим данные модели. Целью соединения вместе компонентов в приложении MVC является обеспечение легкой замены компонента альтернативной реализацией той же самой функциональности.

Наличие возможности замены компонентов позволяет эффективно проводить модульное тестирование, облегчает изменение поведения приложения в разных средах размещения (таких как сервер разработки и производственный сервер) и упрощает долгосрочное сопровождение приложения.

В последующих разделах начнется объяснение альтернативного подхода и проблем, которые он привносит. Это может выглядеть как непрямой способ объяснения средства внедрения зависимостей, но одна из сложностей, присущих DI, заключается в том, что DI решает проблему, которая не всегда очевидна при написании кода и проявляется только на более поздней стадии цикла разработки.

Мнение о внедрении зависимостей

Внедрение зависимостей — одна из тем, по поводу которых читатели связываются со мной наиболее часто. Примерно в половине сообщений электронной почты выражается недовольство относительно того, что я "навязываю" DI. Как ни странно, во второй половине сообщений сетуют на то, что я недостаточно сильно делаю акцент на преимуществах DI, и другие читатели могут не осознать, насколько полезным способно быть данное средство.

Внедрение зависимостей может оказаться сложной для понимания темой, и ценность его спорна. Средство DI может быть удобным инструментом, но далеко не всем оно нравится и не всем требуется.

Средство DI приносит лишь ограниченную пользу, если вы не проводите модульное тестирование или выполняете небольшой, самодостаточный и устойчивый проект. Понимать, как работает DI, по-прежнему полезно, поскольку DI используется для доступа к ряду важных средств MVC, но вы вовсе не обязаны всегда применять DI в своих контроллерах и других классах.

Я использую DI в собственных проектах главным образом потому, что проекты часто двигаются в неожиданных направлениях, а наличие возможности легкой замены компонента новой реализацией может уберечь от внесения большого числа утомительных и чреватых ошибками изменений. Я предпочитаю приложить определенные усилия в начале проекта, чем иметь дело со сложным набором правок на более позднем этапе. Я совершенно не категоричен в отношении внедрения зависимостей, т.к. оно решает проблему, которая возникает не в каждом проекте. Только вы можете решить, нужно ли DI в вашем проекте, и только вы способны оценить получаемые выгоды и сопутствующие затраты.

Исследование сильно связанных компонентов

Выбор наиболее прямого пути к решению задачи — естественная склонность большинства разработчиков. В примере приложения это означает применение ключевого слова `new` для создания объекта хранилища, который требуется контроллеру, чтобы получить данные модели (листинг 18.9).

Листинг 18.9. Создание объекта хранилища в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;

namespace DependencyInjection.Controllers {
  public class HomeController : Controller {
    public ViewResult Index() => View(new MemoryRepository().Products);
  }
}
```

Есть хорошая новость относительно приведенного кода — он работает. Запустив приложение, вы увидите детали объектов модели, отображаемые в браузере (рис. 18.2).

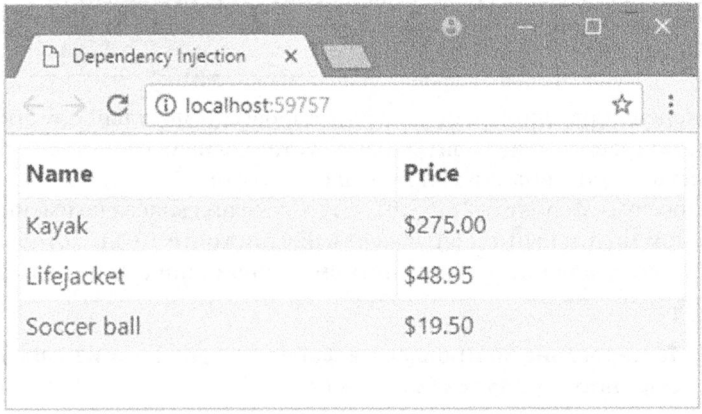


Рис. 18.2. Отображение данных модели

Плохая новость в том, что контроллер `Home` и класс `MemoryRepository` теперь *сильно связаны*, т.е. заменить хранилище, не изменяя класс `HomeController`, не удастся. Как объяснялось в главе 7, выполнение эффективного модульного тестирования означает возможность изоляции одиночного компонента, но протестировать метод действия `Index()` в листинге 18.9 невозможно без неявного тестирования также и класса хранилища. Если модульный тест не пройдет, то нельзя будет утверждать, где конкретно присутствует проблема — в контроллере, в хранилище или в каком-то другом компоненте, от которого зависит хранилище. Во всех практических смыслах контроллер `Home` и класс `MemoryRepository` формируют единственный индивидуальный модуль, как иллюстрируется на рис. 18.3.

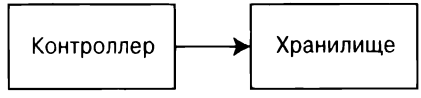


Рис. 18.3. Эффект от сильно связанных компонентов

Развязывание компонентов для модульного тестирования

В главе 7 использовалось свойство для хранения ссылки на класс хранилища через реализуемый им интерфейс, что позволило создавать имитированное хранилище в целях модульного тестирования. В листинге 18.10 такой подход применяется к контроллеру в текущем примере приложения.

Листинг 18.10. Использование свойства для хранилища в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
```

```
namespace DependencyInjection.Controllers {
  public class HomeController : Controller {
    public IRepository Repository { get; set; } = new MemoryRepository();
    public ViewResult Index() => View(Repository.Products);
  }
}
```

Прием прекрасно подходит, если вы хотите провести модульное тестирование, т.к. он позволяет изолировать класс контроллера, устанавливая свойство `Repository` перед вызовом метода действия внутри модульного теста.

Добавим в проект `DependencyInjection.Tests` файл класса по имени `DITests.cs` и определим в нем модульный тест, показанный в листинге 18.11, который применяет свойство `Repository` для настройки фиктивного хранилища перед взаимодействием с контроллером.

Листинг 18.11. Тестирование контроллера в файле `DITests.cs` из папки `DependencyInjection.Tests`

```
using DependencyInjection.Controllers;
using DependencyInjection.Models;
using Microsoft.AspNetCore.Mvc;
using Moq;
using Xunit;

namespace Tests {
  public class DITests {
    [Fact]
    public void ControllerTest() {
      // Организация
      var data = new[] { new Product { Name = "Test", Price = 100 } };
      var mock = new Mock<IRepository>();
      mock.SetupGet(m => m.Products).Returns(data);
      HomeController controller = new HomeController {
        Repository = mock.Object
      };
      // Действие
      ViewResult result = controller.Index();
      // Утверждение
      Assert.Equal(data, result.ViewData.Model);
    }
  }
}
```

Свойство `Repository` позволяет изолировать контроллер и предоставить тестовые данные, которые можно инспектировать в объекте `ViewResult`, созданном методом действия. В итоге мы обеспечиваем только частичное решение проблемы сильно связанных компонентов, потому что установить свойство `Repository`, когда приложение функционирует, невозможно. В главе 17 было объяснялось, что инфраструктура MVC отвечает за создание экземпляров контроллеров для обработки запросов, но ей ничего не известно об особой важности, придаваемой свойству `Repository`.

Результатом такого приема является то, что контроллер и хранилище слабо связаны при проведении модульного тестирования и сильно связаны во время выполнения приложения (рис. 18.4).

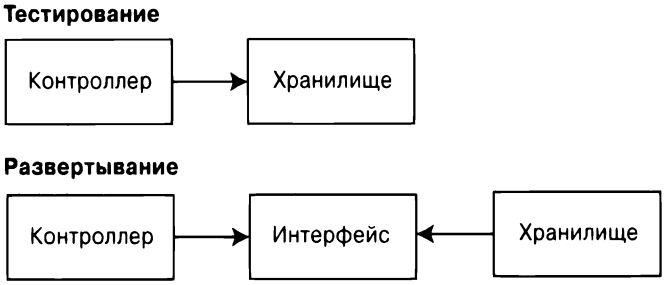


Рис. 18.4. Эффект от добавления свойства `Repository`

Использование брокера типов

Следующий логический шаг предусматривает вынесение кода, в котором решается, какую реализацию интерфейса хранилища применять, за пределы класса контроллера и его помещение куда-то в другое место приложения. Чтобы посмотреть, как это может выглядеть, добавим в проект приложения папку `Infrastructure` и создадим в ней файл класса по имени `TypeBroker.cs` с содержимым из листинга 18.12.

Листинг 18.12. Содержимое файла `TypeBroker.cs` из папки `Infrastructure`

```
using DependencyInjection.Models;
using System;

namespace DependencyInjection.Infrastructure {
  public static class TypeBroker {
    private static Type repoType = typeof(MemoryRepository);
    private static IRepository testRepo;
    public static IRepository Repository =>
      testRepo ?? Activator.CreateInstance(repoType) as IRepository;
    public static void SetRepositoryType<T>() where T : IRepository =>
      repoType = typeof(T);
    public static void SetTestObject(IRepository repo) {
      testRepo = repo;
    }
  }
}
```

В классе `TypeBroker` определено свойство `Repository`, которое возвращает новые объекты, реализующие интерфейс `IRepository`. Класс реализации, используемый свойством `Repository`, определяется значением поля `repoType`, которое по умолчанию установлено в `MemoryRepository`, но может быть изменено вызовом метода `SetRepositoryType()`.

Для поддержки модульного тестирования метод `SetTestObject()` позволяет применять специфический объект. В листинге 18.13 контроллер обновлен, чтобы получать объект хранилища от брокера типов.

Листинг 18.13. Использование брокера типов в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;
namespace DependencyInjection.Controllers {
  public class HomeController : Controller {
    public IRepository Repository { get; } = TypeBroker.Repository;
    public ViewResult Index() => View(Repository.Products);
  }
}
```

Теперь в примере приложения имеется более сложный набор отношений, как продемонстрировано на рис. 18.5. Важно отметить, что между классом контроллера и классом хранилища никакого прямого отношения нет — все решается при посредстве интерфейса и брокера. Это значит, что класс хранилища можно изменять без необходимости во внесении изменений в класс контроллера.



Рис. 18.5. Эффект от добавления брокера типов

Чтобы посмотреть, как применяется брокер типов, добавим в папку `Models` файл класса по имени `AlternateRepository.cs` с еще одной реализацией интерфейса `IRepository`, приведенной в листинге 18.14.

Листинг 18.14. Содержимое файла `AlternateRepository.cs` из папки `Models`

```
using System.Collections.Generic;
namespace DependencyInjection.Models {
  public class AlternateRepository : IRepository {
    private Dictionary<string, Product> products;
    public AlternateRepository() {
      products = new Dictionary<string, Product>();
      new List<Product> {
        new Product { Name = "Corner Flags", Price = 34.95M },
        new Product { Name = "Stadium", Price = 79500M }
      }.ForEach(p => AddProduct(p));
    }
    public IEnumerable<Product> Products => products.Values;
    public Product this[string name] => products[name];
    public void AddProduct(Product product) =>
      products[product.Name] = product;
    public void DeleteProduct(Product product) =>
      products.Remove(product.Name);
  }
}
```

В реальном приложении альтернативное хранилище могло бы хранить данные в другом формате либо использовать постоянство другого вида. В текущем примере отличие между классами `AlternateRepository` и `MemoryRepository` касается данных модели, которые они создают при создании их экземпляров. Для применения класса `AlternateRepository` сконфигурируем брокер типов в методе `ConfigureServices()` класса `Startup`, как показано в листинге 18.15.

Листинг 18.15. Конфигурирование брокера типов в файле `Startup.cs` из папки `DependencyInjection`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;
namespace DependencyInjection {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      TypeBroker.SetRepositoryType<AlternateRepository>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Результат изменения можно увидеть, запустив приложение, которое отобразит данные, предоставляемые новым классом хранилища (рис. 18.6).

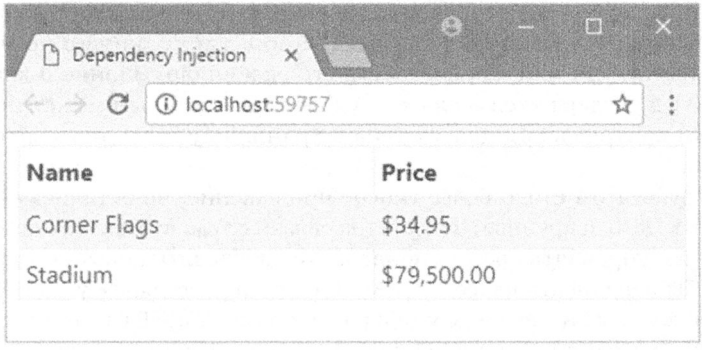


Рис. 18.6. Изменение класса хранилища

Брокер типов позволяет использовать в качестве хранилища специфический объект, что делает возможным написание модульных тестов, подобных представленному в листинге 18.16.

Листинг 18.16. Тестирование контроллера через брокер в файле `DITests.cs` из папки `DependencyInjection.Tests`

```
using DependencyInjection.Controllers;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;
using Microsoft.AspNetCore.Mvc;
using Moq;
using Xunit;

namespace Tests {

  public class DITests {

    [Fact]
    public void ControllerTest() {
      // Организация
      var data = new[] { new Product { Name = "Test", Price = 100 } };
      var mock = new Mock<IRepository>();
      mock.SetupGet(m => m.Products).Returns(data);
      TypeBroker.SetTestObject(mock.Object);
      HomeController controller = new HomeController();

      // Действие
      ViewResult result = controller.Index();

      // Утверждение
      Assert.Equal(data, result.ViewData.Model);
    }
  }
}
```

Введение в средство внедрения зависимостей ASP.NET

В предыдущем разделе вы ознакомились с процессом отделения класса контроллера от хранилища, которое поставляет данные модели. Теперь класс `HomeController` способен получать реализацию интерфейса `IRepository`, ничего не зная о том, какой класс применяется или как создается его экземпляр. Знание о классе реализации `IRepository` содержится в классе `TypeBroker`, который может использоваться любым другим контроллером для доступа к хранилищу и для применения тестового объекта.

Общим результатом стало более гибкое приложение, но остались еще некоторые шероховатости. Самый крупный недостаток связан с тем, что для каждого нового типа, которым должен управлять брокер, понадобится добавлять новые методы и свойства. Можно было бы переписать класс `TypeBroker`, сделав его более универсальным, но в этом нет необходимости, поскольку инфраструктура ASP.NET Core предлагает изящную версию той же самой функциональности, пакетированную таким способом, который облегчает ее применение и не требует никаких специальных классов.

Подготовка к внедрению зависимостей

Термин *внедрение зависимостей* (DI) описывает альтернативный подход к созданию слабо связанных компонентов, который интегрирован в ASP.NET Core и используется инфраструктурой MVC автоматически, в итоге чего контроллеры и другие компоненты не обязаны знать, каким образом создаются требующиеся им типы. В листинге 18.17 продемонстрирована подготовка контроллера `Home` к внедрению зависимостей.

Листинг 18.17. Подготовка к внедрению зависимостей в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) => repository = repo;

    public ViewResult Index() => View(repository.Products);
  }
}
```

Контроллер объявляет свои зависимости в виде аргументов конструктора. Таким образом, здесь учитывается вторая часть термина: *зависимости* во "внедрении зависимостей" — это объекты, которые требуются для создания нового экземпляра класса. В рассматриваемом случае класс контроллера объявляет зависимость от интерфейса `IRepository`.

В инфраструктуре ASP.NET Core компонент под названием *поставщик служб* отвечает за отображение интерфейсов на типы реализации, которые применяются для удовлетворения зависимостей.

Когда требуется новый контроллер, инфраструктура MVC запрашивает у поставщика служб создание нового экземпляра класса `HomeController`. Поставщик служб инспектирует конструктор `HomeController` для выяснения его зависимостей, создает требуемые объекты служб и внедряет их внутрь конструктора, чтобы создать новый контроллер, который может использоваться для обработки запроса. Это основной процесс внедрения зависимостей, а потому его стоит повторить для ясности.

1. Инфраструктура MVC получает входящий запрос к методу действия в контроллере `Home`.
2. Инфраструктура MVC требует у компонента поставщика служб ASP.NET новый экземпляр класса `HomeController`.
3. Поставщик служб инспектирует конструктор `HomeController` и обнаруживает, что он имеет зависимость от интерфейса `IRepository`.
4. Поставщик служб обращается к своим отображениям, чтобы найти класс реализации, который подлежит применению для зависимостей от интерфейса `IRepository`.
5. Поставщик служб создает новый экземпляр найденного класса реализации.

6. Поставщик служб создает новый объект `HomeController`, используя объект реализации в качестве аргумента конструктора.
7. Поставщик служб возвращает созданный объект `HomeController` инфраструктуре MVC, который она применяет для обработки входящего HTTP-запроса.

В целом результат будет таким же, как в случае использования специального класса брокера типов, но важное преимущество заключается в том, что процесс внедрения зависимостей интегрирован в MVC, т.е. компонент поставщика служб будет применяться всякий раз, когда создается экземпляр класса контроллера. Таким образом, классам контроллеров разрешено объявлять зависимости, не имея никакого понятия о том, как они будут распознаваться. Вы просто пишете классы контроллеров, которые объявляют свои зависимости как параметры конструкторов, и позволяете инфраструктуре вместе с компонентом поставщика служб позаботиться об остальном.

На заметку! Во всех примерах главы используется встроенная система внедрения зависимостей, являющаяся частью ASP.NET Core. Существуют сторонние пакеты, которые допускается применять в качестве замены встроенной функциональности и которые могут предлагать усовершенствования и дополнительные средства. В число популярных пакетов входят Autofac и StructureMap, хотя на время написания книги для их интеграции в ASP.NET Core требовались дополнительные пакеты. Подробные сведения доступны по адресу `https://github.com/aspnet/DependencyInjection/`.

Конфигурирование поставщика служб

Объявление зависимости через конструктор `HomeController` нарушает работу приложения, в чем легко удостовериться, запустив проект. Когда MVC пытается создать экземпляр класса `HomeController` для обслуживания запроса, возникает ошибка, показанная на рис. 18.7.

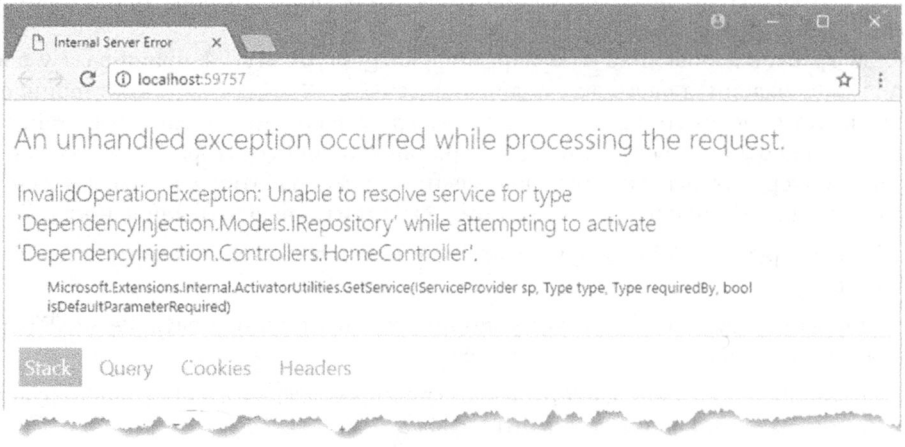


Рис. 18.7. Запуск примера приложения

Для распознавания зависимостей поставщик служб должен быть сконфигурирован так, чтобы он знал, каким образом распознавать зависимости от служб. В настоящий момент поставщик служб не располагает такой информацией и генерирует исключе-

ние, когда ему предлагается создать объект HomeController, потому что ему ничего не известно о том, как распознавать зависимость от интерфейса IRepository.

Конфигурация для поставщика служб определяется в классе Startup и в итоге к моменту запуска приложения служба будет на месте для получения запросов. В листинге 18.18 поставщик служб конфигурируется так, чтобы ему было известно, каким образом иметь дело с зависимостями от интерфейса IRepository.

Листинг 18.18. Конфигурирование поставщика служб в файле Startup.cs из папки DependencyInjection

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IRepository, MemoryRepository>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Внедрение зависимостей конфигурируется с использованием расширяющих методов, которые вызываются на объекте реализации IServiceCollection, полученном с помощью метода ConfigureServices(). Расширяющий метод AddTransient(), который применялся в листинге 18.18, сообщает поставщику служб, как обрабатывать зависимость (он будет описан более подробно позже в главе). Отображение выражается с использованием параметров типов, где первый тип является интерфейсом, а второй — классом реализации.

```
...
services.AddTransient<IRepository, MemoryRepository>();
...
```

Этот оператор указывает поставщику служб о том, что распознавать зависимости от интерфейса IRepository необходимо путем создания объекта MemoryRepository. Запустив приложение, вы увидите, что зависимость, объявленная конструктором HomeController, распознана и контроллеру предоставлен доступ к данным модели (рис. 18.8).

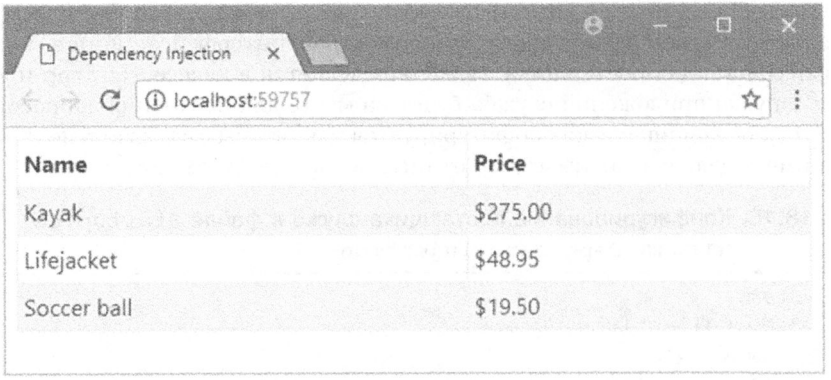


Рис. 18.8. Конфигурирование внедрения зависимостей

Модульное тестирование контроллера с зависимостью

Применение конструктора для получения зависимостей облегчает модульное тестирование контроллеров. В листинге 18.19 приведен модульный тест для контроллера из листинга 18.18.

Листинг 18.19. Тестирование контроллера в файле `DITests.cs` из папки `DependencyInjection.Tests`

```
using DependencyInjection.Controllers;
using DependencyInjection.Models;
using Microsoft.AspNetCore.Mvc;
using Moq;
using Xunit;

namespace Tests {

  public class DITests {

    [Fact]
    public void ControllerTest() {
      // Организация
      var data = new[] { new Product { Name = "Test", Price = 100 } };
      var mock = new Mock<IRepository>();
      mock.SetupGet(m => m.Products).Returns(data);
      HomeController controller = new HomeController(mock.Object);

      // Действие
      ViewResult result = controller.Index();

      // Утверждение
      Assert.Equal(data, result.ViewData.Model);
    }
  }
}
```

Контроллер ничего не знает, да и не заботится о том, какой вид объекта передается конструктору, до тех пор, пока он реализует корректный интерфейс. В результате появляется возможность использовать фиктивное хранилище, не полагаясь на какой-то внешний класс, такой как брокер типов, что может повлиять на исход теста.

Использование цепочек зависимостей

Когда поставщику служб необходимо распознать зависимость, он инспектирует тип, который был сконфигурирован для применения, чтобы выяснить, имеет ли он в свою очередь зависимости, подлежащие распознаванию. Результатом является возможность создания *цепочки* зависимостей, которые все распознаются во время выполнения и могут управляться посредством конфигурации в классе `Startup`. Чтобы посмотреть на образование цепочки зависимостей, добавим в папку `Models` файл класса по имени `IModelStorage.cs` с определением интерфейса, представленным в листинге 18.20.

Листинг 18.20. Содержимое файла `IModelStorage.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DependencyInjection.Models {

  public interface IModelStorage {
    IEnumerable<Product> Items { get; }
    Product this[string key] { get; set; }
    bool ContainsKey(string key);
    void RemoveItem(string key);
  }
}
```

Интерфейс `IModelStorage` определяет поведение простого механизма хранилища для объектов `Product`. Для реализации интерфейса `IModelStorage` добавим в папку `Models` файл класса по имени `DictionaryStorage.cs` с определением из листинга 18.21.

Листинг 18.21. Содержимое файла `DictionaryStorage.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DependencyInjection.Models {
  public class DictionaryStorage : IModelStorage {
    private Dictionary<string, Product> items
      = new Dictionary<string, Product>();

    public Product this[string key] {
      get { return items[key]; }
      set { items[key] = value; }
    }

    public IEnumerable<Product> Items => items.Values;
    public bool ContainsKey(string key) => items.ContainsKey(key);
    public void RemoveItem(string key) => items.Remove(key);
  }
}
```

Класс `DictionaryStorage` реализует интерфейс `IModelStorage`, используя строго типизированный словарь для хранения объектов модели. Такая функциональность в текущий момент содержится внутри класса `MemoryRepository`, и в реальном проекте ее отделение с применением интерфейса имеет малую ценность, но мы получаем полезный пример того, как внедрение зависимостей может использоваться, не привнося слишком много дополнительной сложности в приложение.

В листинге 18.22 класс MemoryRepository модифицирован так, что он объявляет зависимость от интерфейса IModelStorage, но без какого-либо знания о классе реализации, который будет применяться во время выполнения.

Листинг 18.22. Объявление зависимости в файле MemoryRepository.cs из папки Models

```
using System.Collections.Generic;

namespace DependencyInjection.Models {
  public class MemoryRepository : IRepository {
    private IModelStorage storage;

    public MemoryRepository(IModelStorage modelStore) {
      storage = modelStore;
      new List<Product> {
        new Product { Name = "Kayak", Price = 275M },
        new Product { Name = "Lifejacket", Price = 48.95M },
        new Product { Name = "Soccer ball", Price = 19.50M }
      }.ForEach(p => AddProduct(p));
    }

    public IEnumerable<Product> Products => storage.Items;

    public Product this[string name] => storage[name];

    public void AddProduct(Product product) =>
      storage[product.Name] = product;

    public void DeleteProduct(Product product) =>
      storage.RemoveItem(product.Name);
  }
}
```

Запустив приложение, можно заметить, что поставщик служб сгенерирует исключение со следующим сообщением:

```
InvalidOperationException: Unable to resolve service for type
'DependencyInjection.Models.IModelStorage' while attempting
to activate 'DependencyInjection.Models.MemoryRepository'.
```

InvalidOperationException: не удается распознать службу для типа DependencyInjection.Models.IModelStorage при попытке активации DependencyInjection.Models.MemoryRepository.

Здесь демонстрируется прохождение поставщика служб через цепочку зависимостей. Когда у него было затребовано создание нового объекта контроллера, он проинспектировал конструктор HomeController и нашел зависимость от интерфейса IRepository, которая, как ему известно, должна распознаваться посредством объекта MemoryRepository. Затем поставщик служб исследовал конструктор MemoryRepository, который имеет зависимость от интерфейса IModelStorage. Но в конфигурации не указано, каким образом должны распознаваться зависимости от IModelStorage, т.е. объект MemoryRepository не может быть создан, равно как не может быть создан и объект HomeController. Поставщик служб не в состоянии предоставить инфраструктуре MVC объект, необходимый для обработки запроса, и потому было сгенерировано исключение.

Необходимо добавить отображение типа, которое сообщит поставщику служб о том, как должны распознаваться зависимости от IModelStorage, что и делается в конфигурации приложения, показанной в листинге 18.23.

Листинг 18.23. Конфигурирование дополнительного отображения типа в файле Startup.cs из папки DependencyInjection

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IRepository, MemoryRepository>();
      services.AddTransient<IModelStorage, DictionaryStorage>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

С таким добавлением поставщик служб может удовлетворить обе зависимости в цепочке и способен создать набор объектов, требуемых для обслуживания запроса: объекта DictionaryStorage, внедряемого внутрь конструктора класса MemoryRepository, объект которого внедряется внутрь конструктора HomeController. Цепочки зависимостей — не просто искусный трюк; они позволяют формировать сложную функциональность, объединяя компоненты, которые могут быть легко изолированы с целью тестирования и очень просто изменены, чтобы удовлетворять развивающимся требованиям проекта по мере обретения им зрелости.

Использование внедрения зависимостей для конкретных типов

Внедрение зависимостей также может применяться для конкретных типов, доступ к которым не производится через интерфейсы. Несмотря на то что здесь не обеспечиваются преимущества слабой связанности, присущие использованию интерфейсов, сам по себе это удобный прием, поскольку он позволяет получать доступ к объектам где угодно в приложении и помещает конкретные типы под управление жизненным циклом, как описано далее в главе.

Добавим в папку Models файл класса по имени ProductTotalizer.cs с содержимым, приведенным в листинге 18.24.

Листинг 18.24. Содержимое файла ProductTotalizer.cs из папки Models

```
using System.Linq;

namespace DependencyInjection.Models {
  public class ProductTotalizer {

    public ProductTotalizer(IRepository repo) => Repository = repo;

    public IRepository Repository { get; set; }

    public decimal Total => Repository.Products.Sum(p => p.Price);
  }
}
```

Класс ProductTotalizer не делает ничего особо полезного, но имеет зависимость от интерфейса IRepository, а это значит, что применение внедрения зависимостей распознает данную зависимость, используя конфигурацию, которая также применяется к остальной части приложения. В листинге 18.25 показано объявление класса ProductTotalizer как зависимости для класса HomeController.

Листинг 18.25. Добавление зависимости в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
    private ProductTotalizer totalizer;

    public HomeController(IRepository repo, ProductTotalizer total) {
      repository = repo;
      totalizer = total;
    }

    public ViewResult Index() {
      ViewBag.Total = totalizer.Total;
      return View(repository.Products);
    }
  }
}
```

Действие Index добавляет свойство ViewBag с общей суммой, вырабатываемой классом ProductTotalizer, которая отобразится в таблице для значений ViewBag, добавленной к представлению Index.cshtml в начале главы. Финальный шаг заключается в сообщении поставщику служб о том, как поступать с запросами ProductTotalizer (листинг 18.26).

Листинг 18.26. Конфигурирование поставщика служб в файле `Startup.cs` из папки `DependencyInjection`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IRepository, MemoryRepository>();
      services.AddTransient<IModelStorage, DictionaryStorage>();
      services.AddTransient<ProductTotalizer>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

В данной ситуации отсутствует отображение между типом службы и типом реализации, поэтому используется перегруженная версия расширяющего метода `AddTransient()`, принимающая единственный параметр типа, который сообщает поставщику служб, что для распознавания зависимости указанного типа он должен создавать экземпляр класса `ProductTotalizer`.

Преимущества такого подхода — в противоположность простому созданию экземпляра конкретного класса внутри контроллера — связаны с тем, что поставщик служб будет распознавать любые зависимости, объявляемые конкретным классом, а также с тем, что появляется возможность изменять конфигурацию для распознавания зависимостей от конкретного класса с применением более специализированных подклассов. Конкретные классы управляются поставщиком служб и подвергаются воздействию со стороны средств жизненного цикла, которые рассматриваются в следующей главе. В результате запуска приложения отобразится общая сумма для объектов `Product` в модели (рис. 18.9).

Жизненные циклы служб

В предыдущем разделе для сообщения поставщику служб о том, как он должен обрабатывать зависимости от интерфейсов `IRepository` и `IModelStorage`, применялся расширяющий метод `AddTransient()`. Метод `AddTransient()` — один из четырех способов определения отображений типов.

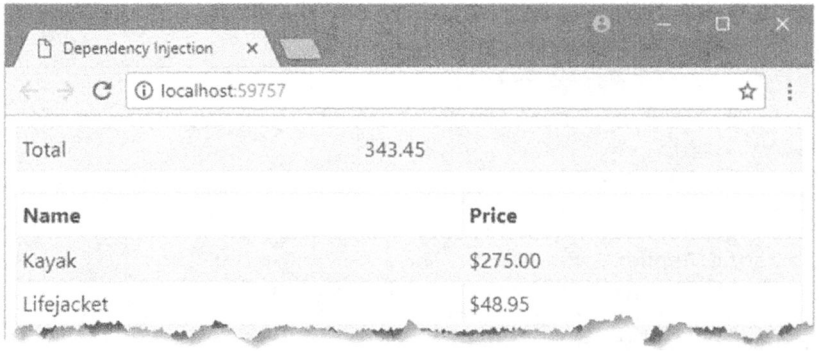


Рис. 18.9. Использование внедрения зависимостей для классов

В табл. 18.3 описаны расширяющие методы, которые указывают поставщику служб, каким образом распознавать зависимости. Все методы в табл. 18.3 используют параметры типов, но доступны также расширяющие методы, которые взамен принимают в качестве аргументов объекты `Type`, что может быть полезно, если необходимо генерировать отображения во время выполнения.

Таблица 18.3. Расширяющие методы, сообщающие поставщику служб о том, как распознавать зависимости

Имя	Описание
`AddTransient<service, implType>()`	Этот метод указывает поставщику служб на необходимость создания нового экземпляра типа реализации для каждой зависимости от типа службы, как описано в разделе "Использование переходного жизненного цикла" далее в главе
`AddTransient<service>()`	Этот метод применяется для регистрации одиночного типа, экземпляр которого будет создаваться для каждой зависимости, как объяснялось в разделе "Использование внедрения зависимостей для конкретных типов" р анее в главе
`AddTransient<service>(factoryFunc)`	Этот метод применяется для регистрации фабричной функции, которая будет вызываться с целью создания объекта реализации для каждой зависимости от типа службы, как показано в разделе "Использование фабричной функции" далее в главе

`AddScoped<service, implType>()`
`AddScoped<service>()`
`AddScoped<service>(factoryFunc)`
	Эти методы сообщают поставщику служб о том, что экземпляры типа реализации должны использоваться повторно. Таким образом, все запросы к службе со стороны компонентов, ассоциированных с общей областью действия, которой обычно является одиночный HTTP-запрос, разделяют один и тот же объект. Данные методы следуют тому же самому шаблону, что и соответствующие методы `AddTransient()`. См. раздел "Использование жизненного цикла, ограниченного областью действия" далее в главе

Окончание табл. 18.3

Имя	Описание

`AddSingleton<service, implType>()`
`AddSingleton<service>()`
`AddSingleton<service>(factoryFunc)`
	Эти методы указывают поставщику служб на необходимость создания нового экземпляра типа реализации для первого запроса к службе и затем его повторного использования для всех последующих запросов к службе. См. раздел "Использование жизненного цикла одиночки" далее в главе

`AddSingleton<service>(instance)`	Этот метод предоставляет поставщику служб объект, который должен применяться для обслуживания всех запросов к службе. Поставщик служб не будет создавать новые объекты

Использование переходного жизненного цикла

Простейший способ приступить к применению внедрения зависимостей предусматривает использование метода `AddTransient()`, который сообщает поставщику служб о том, что он должен создавать новый экземпляр типа реализации всякий раз, когда нужно распознать зависимость. Это конфигурация, которая уже присутствует в классе `Startup`:

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddTransient<IRepository, MemoryRepository>();
  services.AddTransient<IModelStorage, DictionaryStorage>();
  services.AddTransient<ProductTotalizer>();
  services.AddMvc();
}
...
```

Со всеми жизненными циклами, описанными в табл. 18.3, связаны компромиссы. Переходный жизненный цикл влечет за собой затраты по созданию нового экземпляра класса реализации каждый раз, когда распознается зависимость, но его преимущество в том, что не приходится беспокоиться об управлении параллельным доступом или обеспечении безопасного многократного применения объекта множеством запросов.

Для демонстрации переходного жизненного цикла в классе `MemoryRepository` переопределим метод `ToString()`, чтобы он генерировал глобально уникальный идентификатор (globally unique identifier — GUID), как показано в листинге 18.27.

Листинг 18.27. Переопределение метода `ToString()` в файле `MemoryRepository.cs` из папки `Models`

```
using System.Collections.Generic;
namespace DependencyInjection.Models {
  public class MemoryRepository : IRepository {
    private IModelStorage storage;
    private string guid = System.Guid.NewGuid().ToString();
    public MemoryRepository(IModelStorage modelStore) {
      storage = modelStore;
      new List<Product> {
        new Product { Name = "Kayak", Price = 275M },
        new Product { Name = "Lifejacket", Price = 48.95M },
        new Product { Name = "Soccer ball", Price = 19.50M }
      }.ForEach(p => AddProduct(p));
    }
```

```
    public IEnumerable<Product> Products => storage.Items;
    public Product this[string name] => storage[name];
    public void AddProduct(Product product) =>
      storage[product.Name] = product;
    public void DeleteProduct(Product product) =>
      storage.RemoveItem(product.Name);
    public override string ToString() {
      return guid;
    }
  }
}
```

Идентификатор GUID облегчит опознание специфического экземпляра класса `MemoryRepository` и покажет, как различные методы жизненного цикла изменяют поведение поставщика служб. В листинге 18.28 метод действия `Index()` контроллера `Home` модифицирован так, чтобы создавать свойство `Controller` в объекте `ViewBag` и устанавливать его в GUID из хранилища.

Листинг 18.28. Использование объекта `ViewBag` в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
    private ProductTotalizer totalizer;

    public HomeController(IRepository repo, ProductTotalizer total) {
      repository = repo;
      totalizer = total;
    }

    public ViewResult Index() {
      ViewBag.HomeController = repository.ToString();
      ViewBag.Totalizer = totalizer.Repository.ToString();
      return View(repository.Products);
    }
  }
}
```

Метод действия `Index()` добавляет в объект `ViewBag` значения, которые содержат идентификаторы GUID для объектов хранилища, получаемых самим конструктором `HomeController` и через конструктор класса `ProductTotalizer`, что можно увидеть, запустив приложение. Два идентификатора GUID отличаются друг от друга, поскольку поставщик служб был сконфигурирован с помощью метода `AddTransient()`, т.е. он создает новый объект `MemoryRepository` для распознавания зависимости класса `HomeController` и еще один для класса `ProductTotalizer` (рис. 18.10).

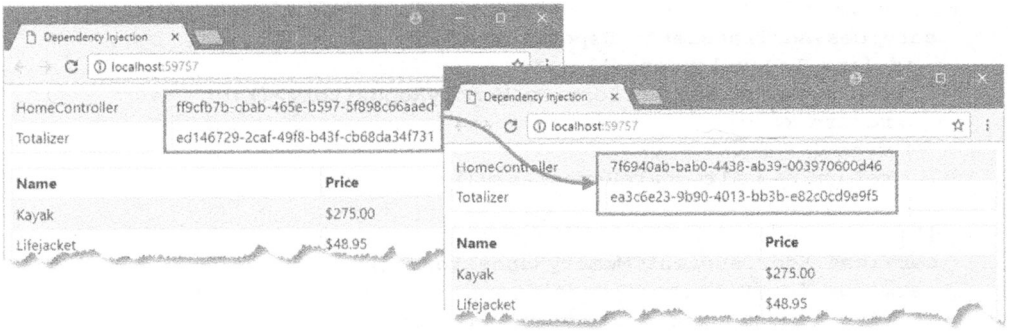


Рис. 18.10. Эффект от переходного жизненного цикла

При каждой перезагрузке веб-страницы в браузере новый HTTP-запрос приводит к тому, что инфраструктура MVC создает новый объект `HomeController`, вызывая создание двух новых объектов `MemoryRepository`, каждый с собственным идентификатором GUID.

Совет. Идентификаторы GUID уникальны (или настолько близки к уникальным, что фактическая разница отсутствует), поэтому при запуске приложения на своей машине вы увидите другие значения.

Использование фабричной функции

Одна из версий метода `AddTransient()` принимает фабричную функцию, которая вызывается каждый раз, когда встречается зависимость от типа службы. Это позволяет варьировать создаваемый объект, так что разные зависимости получают экземпляры разных типов или экземпляры, которые по-другому сконфигурированы. В листинге 18.29 фабричная функция применяется для выбора разных реализаций интерфейса `IRepository` на основе среды размещения, в которой выполняется приложение.

Листинг 18.29. Использование фабричной функции в файле `Startup.cs` из папки `DependencyInjection`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
  public class Startup {
    private IHostingEnvironment env;

    public Startup(IHostingEnvironment hostEnv) => env = hostEnv;
```

```
    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IRepository>(provider => {
        if (env.IsDevelopment()) {
          var x = provider.GetService<MemoryRepository>();
          return x;
        } else {
          return new AlternateRepository();
        }
      });
      services.AddTransient<MemoryRepository>();
      services.AddTransient<IModelStorage, DictionaryStorage>();
      services.AddTransient<ProductTotalizer>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

В главе 14 было описано, как инфраструктура ASP.NET снабжает класс `Startup` службами, содействуя в настройке приложения; в число этих служб входит реализация интерфейса `IHostingEnvironment`, предназначенная для определения среды размещения. Службы можно получать как аргументы метода `Configure()`, но не метода `ConfigureServices()`, а потому в класс `Startup` был добавлен конструктор, который предоставляет доступ к объекту реализации `IHostingEnvironment` и присваивает его полю по имени `env`.

Внутри метода `ConfigureServices()` с помощью метода `AddTransient()` определяется фабричная функция с указанием лямбда-выражения. Лямбда-выражение получает объект `System.IServiceProvider`, который может использоваться для создания экземпляров других типов, зарегистрированных посредством поставщика служб, с применением методов из табл. 18.4.

Таблица 18.4. Методы `IServiceProvider` и расширяющие методы

Имя	Описание
`GetService<service>()`	Этот метод использует поставщик служб для создания нового экземпляра типа службы. Он возвращает `null`, если для запрошенного типа отсутствует отображение
`GetRequiredService<service>()`	Этот метод применяет поставщик служб для создания нового экземпляра типа службы. Он генерирует исключение, если для запрошенного типа отсутствует отображение

Внутри фабричной функции объект реализации `IHostingEnvironment` используется для выяснения, функционирует ли приложение в среде разработки, и если данный факт подтверждается, тогда посредством метода `GetService()` создается экзем-

пляр класса MemoryRepository, который возвращается из фабричной функции как объект, подлежащий применению для зависимости от IRepository.

Метод GetService() используется для создания объекта из-за того, что класс MemoryRepository имеет собственную зависимость от интерфейса IModelStorage, а применение поставщика служб означает автоматическое управление обнаружением и распознаванием зависимости, но это также означает необходимость указания жизненного цикла, который должен использоваться для объектов MemoryRepository, например:

```
...
services.AddTransient<MemoryRepository>();
...
```

Без такого оператора поставщик служб не будет располагать информацией, которая ему нужна для создания и управления объектами MemoryRepository.

Если приложение выполняется не в среде разработки, тогда фабричная функция возвращает новый экземпляр класса AlternateRepository. Экземпляр AlternateRepository может быть создан напрямую с применением ключевого слова new, потому что в его конструкторе никаких зависимостей не объявлено.

Использование жизненного цикла, ограниченного областью действия

При таком жизненном цикле создается единственный объект класса реализации, который применяется для распознавания всех зависимостей, ассоциированных с одной областью действия, что обычно означает одиночный HTTP-запрос. (Вы можете создавать собственные области действия, но в большинстве приложений от них мало пользы.)

Поскольку стандартной областью действия является HTTP-запрос, рассматриваемый жизненный цикл позволяет единственному объекту разделяться всеми компонентами, которые обрабатывают запрос. Чаще всего это удобно при совместном использовании общих данных контекста, когда пишутся специальные классы, такие как маршруты. Жизненный цикл, ограниченный областью действия, создается путем применения расширяющего метода AddScoped() для конфигурирования поставщика служб (листинг 18.30).

Совет. Как было описано в табл. 18.4, доступны также версии метода AddScoped(), которые принимают фабричную функцию и могут использоваться для регистрации конкретного типа. Такие методы работают аналогично методу AddTransient(), который демонстрировался в предыдущем разделе, с вполне очевидной разницей в том, что жизненный цикл создаваемых ими объектов отличается.

Листинг 18.30. Использование жизненного цикла, ограниченного областью действия, в файле Startup.cs из папки DependencyInjection

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
  public class Startup {
    private IHostingEnvironment env;

    public Startup(IHostingEnvironment hostEnv) => env = hostEnv;

    public void ConfigureServices(IServiceCollection services) {
      services.AddScoped<IRepository, MemoryRepository>();
      services.AddTransient<IModelStorage, DictionaryStorage>();
      services.AddTransient<ProductTotalizer>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

В примере приложения экземпляры классов `HomeController` и `ProductTotalizer` создаются вместе, чтобы обрабатывать запрос, и оба они требуют у поставщика служб распознать зависимость от интерфейса `IRepository`. Применение метода `AddScoped()` гарантирует, что зависимости обоих объектов распознаются посредством единственного объекта `MemoryRepository`. Увидеть результат можно, запустив пример приложения; браузер отобразит два одинаковых идентификатора GUID (рис. 18.11). Перезагрузка страницы приводит к созданию нового HTTP-запроса, означая создание нового объекта `MemoryRepository`.

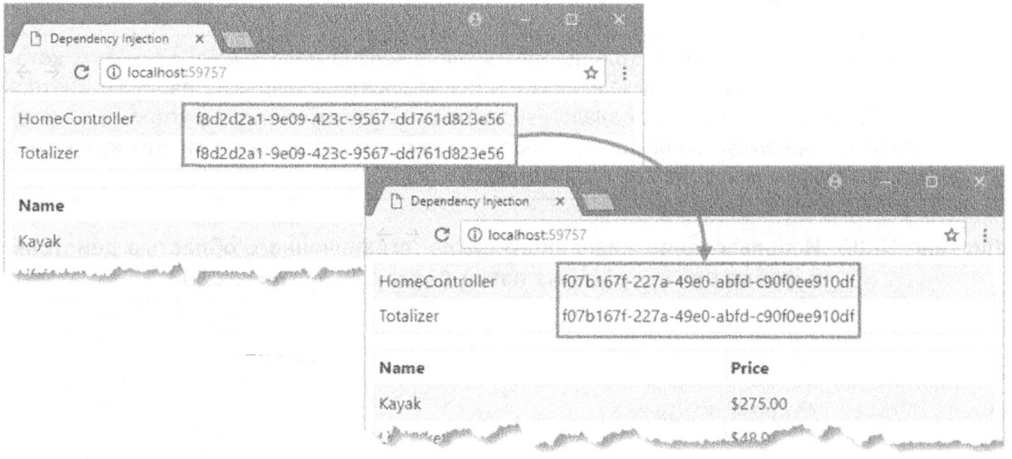


Рис. 18.11. Эффект от жизненного цикла, ограниченного областью действия

Использование жизненного цикла одиночки

Жизненный цикл одиночки гарантирует, что для распознавания всех зависимостей для заданного типа службы применяется единственный объект. При использовании такого жизненного цикла вы обязаны обеспечить безопасность параллельного доступа к классам реализации, применяемым для распознавания зависимостей. В листинге 18.31 изменена область действия для конфигурации IRepository.

Листинг 18.31. Использование жизненного цикла одиночки в файле Startup.cs из папки DependencyInjection

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using DependencyInjection.Infrastructure;
using DependencyInjection.Models;

namespace DependencyInjection {
  public class Startup {
    private IHostingEnvironment env;

    public Startup(IHostingEnvironment hostEnv) => env = hostEnv;

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IRepository, MemoryRepository>();
      services.AddTransient<IModelStorage, DictionaryStorage>();
      services.AddTransient<ProductTotalizer>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Метод AddSingleton() создает новый экземпляр класса MemoryRepository в первый раз, когда он должен распознавать зависимость от интерфейса IRepository, и затем повторно использует имеющийся экземпляр для всех последующих зависимостей, даже если они ассоциированы с другими HTTP-запросами (рис. 18.12).

Использование внедрения в действия

Стандартный способ объявления зависимости — через конструктор — представляет собой прием, который можно применять в любом классе и который полагается на средства внедрения зависимостей, являющиеся частью основной платформы ASP.NET.

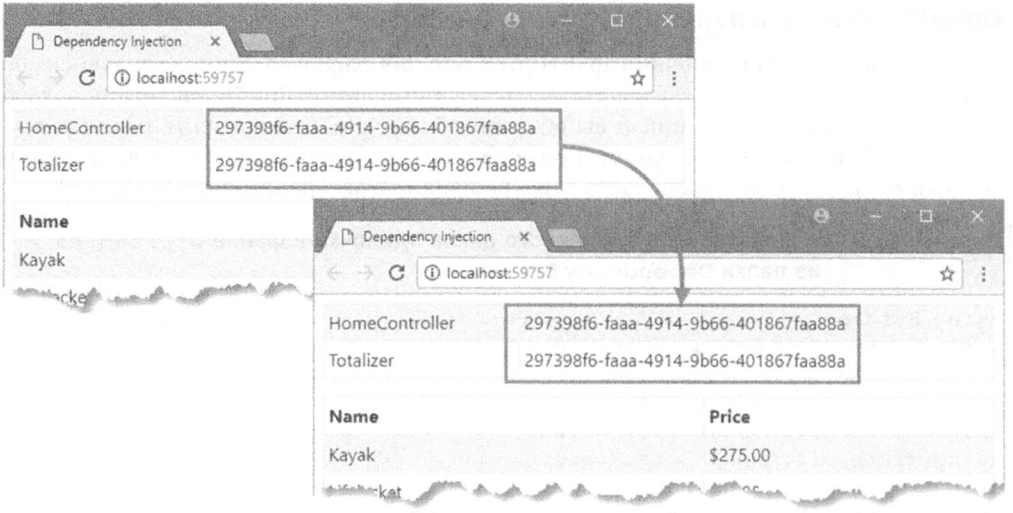


Рис. 18.12. Эффект от жизненного цикла одиночки

Инфраструктура MVC дополняет стандартную функциональность альтернативным подходом, называемым *внедрением в действия*, который позволяет объявлять зависимости через параметры методов действий. Строго говоря, внедрение в действия предоставляется системой привязки моделей, которая обсуждается в главе 26, но оно рассматривается в настоящей главе, т.к. позволяет использовать службы по-другому. Внедрение в действия выполняется с помощью атрибута `FromServices`, который применяется к параметру метода действия, как показано в листинге 18.32.

Листинг 18.32. Использование внедрения в действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;

namespace DependencyInjection.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
    public HomeController(IRepository repo) {
      repository = repo;
    }
    public ViewResult Index([FromServices]ProductTotalizer totalizer) {
      ViewBag.HomeController = repository.ToString();
      ViewBag.Totalizer = totalizer.Repository.ToString();
      return View(repository.Products);
    }
  }
}
```

Инфраструктура MVC с применением поставщика служб получает экземпляр класса `ProductTotalizer` и предоставляет его в качестве аргумента при вызове метода действия `Index()`. Использование внедрения в действия менее распространено, чем

стандартное внедрение в конструкторы, но может быть удобным в ситуации, когда имеется зависимость от объекта, создание которого сопряжено с высокими затратами, и он требуется только внутри одного из методов действий, определяемых контроллером. Применение внедрения в конструкторы приведет к распознаванию зависимости для всех методов действий, даже если метод действия, вызываемый для обработки запроса, не пользуется объектом реализации. Декорирование метода действия атрибутом `FromServices` сужает сферу влияния зависимости и гарантирует, что экземпляр типа реализации будет создаваться, только когда он необходим.

Использование атрибутов внедрения в свойства

В главе 17 объяснялось, как получать данные контекста в контроллере POCO за счет объявления свойства и его декорирования атрибутом `ControllerContext`. Теперь, читая настоящую главу, вы должны понимать, что это было специальной формой внедрения зависимостей. Она называется *внедрением в свойства*.

Инфраструктура MVC предлагает набор специализированных атрибутов, которые можно применять для получения специфических типов через внедрение в свойства внутри контроллеров и компонентов представлений (рассматриваемых в главе 22). В случае наследования контроллеров от базового класса `Controller` использовать такие атрибуты не обязательно, потому что информация контекста доступна через удобные свойства, но в табл. 18.5 приведен список атрибутов, предназначенных для применения в контроллерах POCO.

Таблица 18.5. Специализированные атрибуты для внедрения в свойства

Имя	Описание
`ControllerContext`	Этот атрибут устанавливает свойство `ControllerContext`, которое предоставляет надмножество функциональности класса `ActionContext`, как обсуждается в главе 31
`ActionContext`	Этот атрибут устанавливает свойство `ActionContext` для предоставления информации контекста методам действий. Классы `Controller` открывают доступ к информации контекста через свойство `ActionContext`, а также набор удобных свойств, описанных в главе 31
`ViewContext`	Этот атрибут устанавливает свойство `ViewContext`, чтобы предоставить данные контекста для операций с представлениями, включая вспомогательные функции дескрипторов (глава 23)
`ViewComponentContext`	Этот атрибут устанавливает свойство `ViewComponentContext` для компонентов представлений, которые рассматриваются в главе 22
`ViewDataDictionary`	Этот атрибут устанавливает свойство `ViewDataDictionary`, чтобы предоставить доступ к данным привязки модели, как описано в главе 26

Запрашивание объекта реализации вручную

Главное средство внедрения зависимостей ASP.NET и дополнительные атрибуты, которые инфраструктура MVC предлагает для внедрения в свойства и действия, предоставляют всю поддержку, требующуюся в большинстве приложений для создания

слабо связанных компонентов. Однако могут быть случаи, когда объект реализации для интерфейса удобно получать, не полагаясь на внедрение. В таких ситуациях можно работать напрямую с поставщиком служб, как демонстрируется в листинге 18.33.

Листинг 18.33. Использование поставщика служб напрямую в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using DependencyInjection.Models;
using DependencyInjection.Infrastructure;
using Microsoft.Extensions.DependencyInjection;

namespace DependencyInjection.Controllers {

  public class HomeController : Controller {

    public ViewResult Index([FromServices]ProductTotalizer totalizer) {

      IRepository repository =
        HttpContext.RequestServices.GetService<IRepository>();

      ViewBag.HomeController = repository.ToString();
      ViewBag.Totalizer = totalizer.Repository.ToString();
      return View(repository.Products);
    }
  }
}
```

Объект `HttpContext`, возвращаемый свойством с таким же именем, определяет свойство `RequestServices`, которое возвращает объект реализации `IServiceProvider`, разрешая вызов на нем методов из табл. 18.4.

В листинге 18.33 было удалено свойство `Repository`, которое устанавливалось с применением внедрения в свойства, а взамен задействовано свойство `HttpContext.RequestServices` для получения реализации интерфейса `IRepository`.

Это известно как *паттерн "Локатор служб"* (Service Locator), использования которого, по мнению ряда разработчиков, следует избегать. Марк Симен сделал хорошее описание проблем, к которым он может привести (`http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern`). Моя точка зрения отличается меньшей строгостью в том, что получение служб подобным способом совершенно обоснованно, когда обычный прием получения зависимостей через конструктор по некоторым причинам применять невозможно.

Резюме

В главе объяснялась роль, которую внедрение зависимостей играет в приложении MVC. Оно помогает создавать слабо связанные компоненты, которые легко заменять и просто изолировать в целях тестирования. Было продемонстрировано средство внедрения зависимостей ASP.NET, а также атрибуты, которые инфраструктура MVC предоставляет для внедрения зависимостей в свойства и методы действий. Кроме того, были описаны варианты жизненных циклов, доступные при конфигурировании поставщика служб, и показано, как они влияют на способ, которым создаются объекты. В следующей главе рассматриваются фильтры, добавляющие дополнительную логику в процесс обработки запросов.

ГЛАВА 19

Фильтры

Фильтры внедряют дополнительную логику в процесс обработки запросов MVC. Они предлагают простой и элегантный способ реализации *сквозной обязанности*; таким термином обозначается функциональность, которая используется по всему приложению и не может быть сосредоточена в одном месте, поскольку тогда нарушился бы принцип разделения обязанностей. Классическими примерами сквозной обязанности являются регистрация в журнале, авторизация и кеширование. В настоящей главе мы рассмотрим различные категории фильтров, поддерживаемые MVC, и покажем, как создавать и применять фильтры, а также каким образом управлять их выполнением. В табл. 19.1 приведена сводка, позволяющая поместить фильтры в контекст.

Таблица 19.1. Помещение фильтров в контекст

Вопрос	Ответ
Что это такое?	Фильтры используются для применения логики к методам действий, не добавляя код в класс контроллера
Чем они полезны?	Фильтры позволяют использовать код, который не является частью определения действия в классическом паттерне MVC. Результатом будут более простые классы контроллеров и многократно используемая функциональность, которая может применяться повсюду в приложении
Как они используются?	Существуют разнообразные типы фильтров, которые используются инфраструктурой MVC различными способами. Наиболее распространенный способ создания фильтра предусматривает создание класса, который унаследован от атрибута, предоставляемого MVC для требуемого типа фильтра
Существуют ли какие-то скрытые ловушки или ограничения?	Функциональность, предлагаемая разными типами фильтров, перекрывается, поэтому выяснение, какой тип необходим, может быть затруднено
Существуют ли альтернативы?	Нет, фильтры являются основным средством MVC и используются для реализации обычно требующейся функциональности, такой как авторизация

В табл. 19.2 приведена сводка по главе.

Таблица 19.2. Сводка по главе

Задача	Решение	Листинг
Внедрение дополнительной логики в обработку запросов	Примените фильтры к контроллерам либо к их методам действий	19.6–19.9
Ограничение доступа к действиям	Используйте фильтры авторизации	19.10, 19.11
Внедрение универсальной логики в процесс обработки запросов	Примените фильтры действий	19.12–19.14
Инспектирование либо изменение результатов, выпускаемых методами действий	Используйте фильтры результатов	19.15–19.19
Обработка ошибок	Примените фильтры исключений	19.20, 19.21
Использование служб в фильтрах	Объявите зависимости в конструкторе фильтра, зарегистрируйте службу в классе `Startup` и примените фильтр с помощью атрибута `TypeFilter`	19.22–19.26
Помещение фильтров под управление жизненным циклом	Используйте жизненные циклы внедрения зависимостей для регистрации фильтров в классе `Startup` и примените фильтры с использованием атрибута `ServiceFilter`	19.27–19.29
Применение фильтров к каждому методу действия в приложении	Используйте глобальный фильтр	19.30–19.32
Изменение порядка, в котором применяются фильтры	Используйте параметр `Order`	19.33–19.36

Подготовка проекта для примера

В этой главе при создании примера приложения будет применяться тот же самый подход, что и в предшествующих главах. Создадим новый проект типа Empty (Пустой) по имени `Filters` с использованием шаблона ASP.NET Core Web Application (.NET Core) (Веб-приложение ASP.NET Core (.NET Core)). В листинге 19.1 приведен класс `Startup`, который включает инфраструктуру MVC и другие компоненты промежуточного ПО для разработки.

Листинг 19.1. Содержимое файла `Startup.cs` из папки `Filters`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace Filters {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Включение SSL

Для ряда примеров в настоящей главе требуется применение протокола SSL, который по умолчанию отключен. Чтобы включить SSL, выберем пункт Filter Properties (Свойства фильтров) в меню Project (Проект) среды Visual Studio и отметим флажок Enable SSL (Включить SSL) на вкладке Debug (Отладка), как показано на рис. 19.1. Запомним назначенный номер порта, который в каждом проекте будет отличаться.

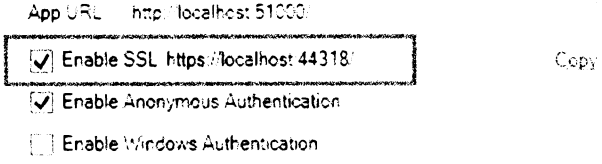


Рис. 19.1. Включение SSL

Создание контроллера и представления

Контроллеры в настоящей главе будут простыми, т.к. внимание сосредоточено на помещении логики в другое место внутри приложения. Создадим папку `Controllers`, добавим в нее файл класса по имени `HomeController.cs` и определим в нем контроллер, как продемонстрировано в листинге 19.2.

Листинг 19.2. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");
  }
}
```

Метод действия `Index()` визуализирует представление по имени `Message`, передавая ему строку в качестве данных представления. Создадим папку `Views/Shared` и поместим в нее файл представления Razor по имени `Message.cshtml` с разметкой из листинга 19.3.

Листинг 19.3. Содержимое файла `Message.cshtml` из папки `Views/Shared`

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Filters</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @if (Model is string) {
    @Model
  } else if (Model is IDictionary<string, string>) {
    var dict = Model as IDictionary<string, string>;
    <table class="table table-sm table-striped table-bordered">
      <thead><tr><th>Name</th><th>Value</th></tr></thead>
      <tbody>
        @foreach (var kvp in dict) {
          <tr><td>@kvp.Key</td><td>@kvp.Value</td></tr>
        }
      </tbody>
    </table>
  }
</body>
</html>
```

Представление является слабо типизированным и отображает либо значение `string`, либо объект `Dictionary<string, string>`; во втором случае выводится таблица.

При стилизации HTML-элементов представление полагается на пакет Bootstrap CSS. Для добавления Bootstrap в проект создадим файл `bower.json` с использованием шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел `dependencies` (листинг 19.4).

Листинг 19.4. Добавление пакета Bootstrap в файле `bower.json` из папки `Filters`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Последний подготовительный шаг предусматривает создание в папке `Views` файла `_ViewImports.cshtml`, который настраивает встроенные вспомогательные функции дескрипторов для применения в представлениях Razor (листинг 19.5).

Листинг 19.5. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Запустив приложение, вы увидите вывод, показанный на рис. 19.2.

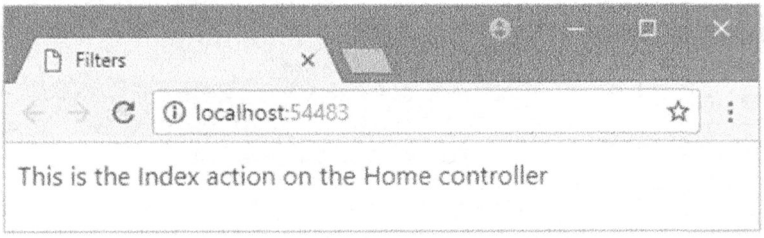


Рис. 19.2. Запуск примера приложения

Совет. Вам может быть предложено доверять сертификату, сгенерированному Visual Studio. Примите данный вариант, который относится к примерам главы, полагающимся на SSL.

Использование фильтров

Фильтры позволяют изъять логику, которая иначе применялась бы в методе действия, из контроллера и определить ее в многократно используемом классе. В качестве примера предположим, что нужно обеспечить возможность доступа к методам действий только по HTTPS, а не по обычному незашифрованному протоколу HTTP. Объект контекста `HttpRequest` предоставляет информацию, необходимую для выяснения, применяется ли HTTPS (листинг 19.6).

Листинг 19.6. Проверка на предмет использования HTTPS в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;

namespace Filters.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() {
      if (!Request.IsHttps) {
        return new StatusCodeResult(StatusCodes.Status403Forbidden);
      } else {
        return View("Message",
          "This is the Index action on the Home controller");
      }
    }
  }
}
```

Именно так решалась бы задача с HTTPS без фильтров. После запуска приложения браузер будет запрашивать стандартный URL для проекта, отличающийся от HTTPS, который метод действия `Index()` обработает путем возвращения объекта `StatusCodeResult`, отправляющего код состояния HTTP 403 в ответе (как описано в главе 17). Если запросить стандартный URL для HTTPS, например, `https://localhost:44318`, тогда метод действия `Index()` отреагирует визуализацией представления `Message` (прежде чем браузер отобразит результат, может понадобиться подтверждение в окне предупреждения безопасности). Оба исхода иллюстрируются на рис. 19.3.

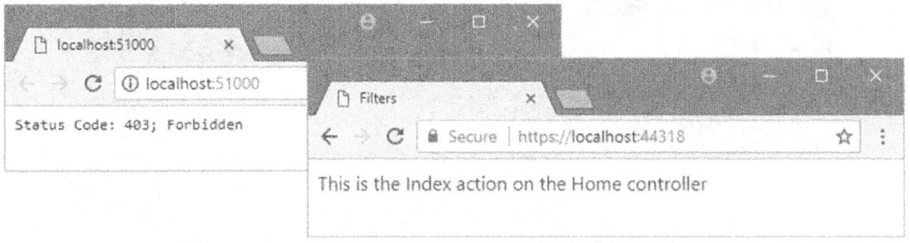


Рис. 19.3. Ограничение доступа только запросами HTTPS

Совет. Если вы не получаете ожидаемые результаты при выполнении примеров из данного раздела, тогда очистите хронологию в браузере. Браузеры часто отклоняют отправку серверу запросов, которые сгенерировали ошибки SSL, что является хорошей практикой защиты, но может мешать во время разработки.

Код в листинге 19.6 работает, но в нем присутствуют проблемы. Первая проблема в том, что метод действия содержит код, который больше относится к реализации политики безопасности, нежели к обработке запроса, обновлению модели и выбору ответа. Более серьезная проблема связана с тем, что решение с кодом обнаружения HTTP внутри метода действия плохо масштабируется и такой код должен быть продублирован в каждом методе действия, определяемом в контроллере (листинг 19.7).

Листинг 19.7. Добавление метода действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() {
      if (!Request.IsHttps) {
        return new StatusCodeResult(StatusCodes.Status403Forbidden);
      } else {
        return View("Message",
          "This is the Index action on the Home controller");
      }
    }
```

```
    public IActionResult SecondAction() {
      if (!Request.IsHttps) {
        return new StatusCodeResult(StatusCodes.Status403Forbidden);
      } else {
        return View("Message",
          "This is the SecondAction action on the Home controller");
      }
    }
  }
}
```

Нужно помнить о реализации той же самой проверки внутри каждого метода действия в каждом контроллере, для которого планируется требование протокола HTTPS. Код реализации политики безопасности занимает значимую часть довольно простого контроллера и затрудняет его понимание. Кроме того, забыть добавить его в новый метод действия и тем самым создать брешь в политике безопасности — дело только времени. Проблемы такого рода способны решать фильтры, как демонстрируется в листинге 19.8.

Листинг 19.8. Применение фильтра в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

  public class HomeController : Controller {
    [RequireHttps]
    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");

    [RequireHttps]
    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");
  }
}
```

Атрибут `RequireHttps` применяет к классу `HomeController` один из встроенных фильтров. Он ограничивает доступ к методам действий, чтобы поддерживались только запросы HTTPS, позволяя удалить из всех методов код, который относится к безопасности, и сосредоточиться на обработке успешных запросов.

На заметку! Фильтр `RequireHttps` работает не совсем так, как специальный код из листинга 19.6. Для запросов `GET` атрибут `RequireHttps` обеспечивает перенаправление клиента на первоначально запрошенный URL, но делает это с использованием схемы `https`, в результате чего запрос `http://localhost/Home/Index` будет перенаправлен на `https://localhost/Home/Index`. Такой подход имеет смысл в большинстве развернутых приложений, но не на стадии разработки, поскольку протоколы HTTP и HTTPS обслуживаются на разных локальных портах. В классе `RequireHttpsAttribute` определен защищенный метод по имени `HandleNonHttpsRequest()`, который можно переопределить для изменения поведения. В качестве альтернативы в разделе "Использование фильтров авторизации" далее в главе исходная функциональность воссоздается с нуля.

Разумеется, по-прежнему необходимо помнить о применении атрибута RequireHttps к каждому методу действия, о чем вполне можно забыть. Но фильтры поддерживают полезный трюк: применение атрибута к классу контроллера дает такой же эффект, как и его применение ко всем индивидуальным методам действий (листинг 19.9).

Листинг 19.9. Применение фильтра ко всем методам действий в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Filters.Controllers {

  [RequireHttps]
  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");

    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");
  }
}
```

Фильтры могут применяться с разными уровнями детализации. Если вы хотите ограничить доступ к одним действиям, но не ограничивать его к другим, тогда применяйте атрибут RequireHttps только к нужным методам. Чтобы защитить все методы действий, в том числе любые методы, которые будут добавлены в будущем, применяйте атрибут RequireHttps к классу. Если же вы желаете применить какой-нибудь фильтр к любому действию в приложении, то можете использовать *глобальные фильтры*, которые будут описаны далее в главе.

Понятие фильтров

Теперь, когда вы видели, как используются фильтры, наступило время объяснить, что происходит "за кулисами". Фильтры реализуют интерфейс IFilterMetadata, который находится в пространстве имен Microsoft.AspNetCore.Mvc.Filters. Вот его определение:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
  public interface IFilterMetadata { }
}
```

Интерфейс IFilterMetadata пуст и не требует от класса фильтра реализации какого-либо специфического поведения. Причина в том, что существует несколько отдельных типов фильтров, каждый из которых работает по-разному и служит для своих целей.

В табл. 19.3 описаны типы фильтров, определяющие их интерфейсы и то, что они делают. (Инфраструктура MVC поддерживает и другие типы фильтров, но они не используются напрямую. Взамен они интегрированы в средства, рассматриваемые в других главах, и применяются через специфические атрибуты, включая атрибуты Produces и Consumes, которые обсуждаются в главе 20.)

Таблица 19.3. Различные типы фильтров

Разновидность	Интерфейсы	Описание
Фильтры авторизации	`IAuthorizationFilter` `IAsyncAuthorizationFilter`	Этот тип фильтров используется для применения политики безопасности приложения, включая авторизацию пользователей
Фильтры действий	`IActionFilter` `IAsyncActionFilter`	Этот тип фильтров используется для выполнения работы немедленно до или после выполнения метода действия
Фильтры результатов	`IResultFilter` `IAsyncResultFilter`	Этот тип фильтров используется для выполнения работы немедленно до или после обработки результата, полученного из метода действия
Фильтры исключений	`IExceptionFilter` `IAsyncExceptionFilter`	Этот тип фильтров используется для обработки исключений

В табл. 19.3 приведены нечеткие описания, потому что фильтры можно использовать для широкого спектра задач, ограниченного только вашим воображением и проблемами, которые необходимо решить. По мере погружения в детали их работы все станет яснее, а пока следует понять два важных момента.

Во-первых, для каждого типа фильтра в табл. 19.3 предусмотрены два разных интерфейса. Фильтры могут выполнять свою работу синхронно или асинхронно, так что синхронный фильтр результатов, например, реализует интерфейс `IResultFilter`, тогда как асинхронный — интерфейс `IAsyncResultFilter`.

Во-вторых, фильтры применяются в особом порядке. Фильтры авторизации выполняются первыми, за ними идут фильтры действий, а затем фильтры результатов. Фильтры исключений выполняются только в случае генерации какого-либо исключения, которое нарушает нормальную последовательность.

Получение данных контекста

Фильтры обеспечиваются данными контекста в форме объекта `FilterContext`. Класс `FilterContext` является производным от `ActionContext`, который также представляет собой базовый класс для класса `ControllerContext`, рассмотренного в главе 17. Из соображений удобства в табл. 19.4 перечислены свойства, унаследованные от класса `ActionContext`, наряду с дополнительными свойствами, которые определены в `FilterContext`.

Таблица 19.4. Свойства `FilterContext`

Имя	Описание
`ActionDescriptor`	Это свойство возвращает объект `ActionDescriptor`, который описывает метод действия
`HttpContext`	Это свойство возвращает объект `HttpContext`, который предоставляет детали HTTP-запроса и отправляемого взамен HTTP-ответа
`ModelState`	Это свойство возвращает объект `ModelStateDictionary`, который используется для проверки достоверности данных, отправленных клиентом (глава 27)

Окончание табл. 19.4

Имя	Описание
RouteData	Это свойство возвращает объект RouteData, который описывает способ обработки запроса системой маршрутизации (глава 15)
Filters	Это свойство возвращает список фильтров, которые были применены к методу действия, выраженный как IList<IFilterMetadata>

Использование фильтров авторизации

Фильтры авторизации применяются для реализации политики безопасности приложения. Фильтры авторизации выполняются перед фильтрами других типов и до запуска метода действия. Вот как выглядит определение интерфейса IAuthorizationFilter:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
  public interface IAuthorizationFilter : IFilterMetadata {
    void OnAuthorization(AuthorizationFilterContext context);
  }
}
```

Метод OnAuthorization() вызывается для предоставления фильтру возможности авторизовать запрос. Ниже показано определение интерфейса IAsyncAuthorizationFilter, используемого асинхронными фильтрами авторизации:

```
using System.Threading.Tasks;
namespace Microsoft.AspNetCore.Mvc.Filters {
  public interface IAsyncAuthorizationFilter : IFilterMetadata {
    Task OnAuthorizationAsync(AuthorizationFilterContext context);
  }
}
```

Метод OnAuthorizationAsync() вызывается, чтобы фильтр мог авторизовать запрос. Какой бы интерфейс ни применялся, фильтр получает данные контекста, описывающие запрос, через экземпляр класса AuthorizationFilterContext, который является производным от класса FilterContext и добавляет одно важное свойство, приведенное в табл. 19.5.

Таблица 19.5. Свойство AuthorizationFilterContext

Имя	Описание
Result	Это свойство типа IActionResult устанавливается фильтрами авторизации, когда запрос не удовлетворяет политике авторизации приложения. Если оно установлено, тогда вместо вызова метода действия инфраструктура MVC визуализирует объект реализации IActionResult

Создание фильтра авторизации

Чтобы посмотреть, как работают фильтры авторизации, создадим в проекте папку Infrastructure, поместим в нее файл класса по имени HttpsOnlyAttribute.cs и определим в нем фильтр, как показано в листинге 19.10.

Листинг 19.10. Содержимое файла HttpsOnlyAttribute.cs из папки Infrastructure

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {
  public class HttpsOnlyAttribute : Attribute, IAuthorizationFilter {

    public void OnAuthorization(AuthorizationFilterContext context) {
      if (!context.HttpContext.Request.IsHttps) {
        context.Result =
          new StatusCodeResult(StatusCodes.Status403Forbidden);
      }
    }
  }
}
```

Если запрос удовлетворяет политике авторизации, то фильтр авторизации ничего не делает; такая пассивность позволяет инфраструктуре MVC перейти к следующему фильтру и в конечном итоге выполнить метод действия.

На заметку! Атрибут Authorize, который может использоваться для ограничения доступа специфическими пользователями и группами, был реализован в виде фильтра, но в ASP.NET Core MVC это больше не так. Атрибут Authorize по-прежнему применяется, но работает по-другому. Внутренне имеется глобальный фильтр (глобальные фильтры обсуждаются позже в главе), который обнаруживает атрибут Authorize и обеспечивает соблюдение политик, определенных системой ASP.NET Core Identity, однако атрибут Authorize не является фильтром и не реализует интерфейс IAuthorizationFilter. Использование системы ASP.NET Core Identity и атрибута Authorize рассматривается в главе 29.

При наличии проблемы фильтр устанавливает свойство Result объекта AuthorizationFilterContext, передаваемого методу OnAuthorization(), что препятствует дальнейшему выполнению и предоставляет инфраструктуре MVC результат для возвращения клиенту. В листинге 19.10 класс HttpsOnlyAttribute инспектирует свойство IsHttps объекта контекста HttpRequest и устанавливает свойство Result, чтобы прервать выполнение, если запрос был сделан без HTTPS. В листинге 19.11 новый фильтр применяется к контроллеру Home.

Листинг 19.11. Применение специального фильтра в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {

  [HttpsOnly]
  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");

    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");
  }
}
```

Этот фильтр воссоздает функциональность, которая была включена в методы действий из листинга 19.7. Такой прием менее полезен в реальных проектах, чем выполнение перенаправления подобно встроенному фильтру `RequireHttps`, поскольку пользователи могут не понять смысл кода состояния 403, но он демонстрирует хороший пример того, как работают фильтры авторизации.

Модульное тестирование фильтров

Большая часть работы при модульном тестировании фильтра связана с настройкой объекта контекста, который передается методам фильтра. Объем требующейся имитации зависит от информации контекста, используемой фильтром. В качестве примера ниже приведен модульный тест для фильтра `HttpsOnly` из листинга 19.10.

```
using System.Linq;
using Filters.Infrastructure;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Abstractions;
using Microsoft.AspNetCore.Mvc.Filters;
using Moq;
using Xunit;

namespace Tests {

  public class FilterTests {

    [Fact]
    public void TestHttpsFilter() {

      // Организация
      var httpRequest = new Mock<HttpRequest>();
      httpRequest.SetupSequence(m => m.IsHttps).Returns(true)
                                               .Returns(false);
      var httpContext = new Mock<HttpContext>();
      httpContext.SetupGet(m => m.Request).Returns(httpRequest.Object);
      var actionContext = new ActionContext(httpContext.Object,
          new Microsoft.AspNetCore.Routing.RouteData(),
          new ActionDescriptor());
```

```
        var authContext = new AuthorizationFilterContext(actionContext,
            Enumerable.Empty<IFilterMetadata>().ToList());
            HttpsOnlyAttribute filter = new HttpsOnlyAttribute();

        // Действие и утверждение
        filter.OnAuthorization(authContext);
        Assert.Null(authContext.Result);

        filter.OnAuthorization(authContext);
        Assert.IsType(typeof(StatusCodeResult), authContext.Result);
        Assert.Equal(StatusCodes.Status403Forbidden,
          (authContext.Result as StatusCodeResult).StatusCode);
      }
    }
}
```

Тест начинается с создания имитированных объектов контекста `HttpRequest` и `HttpContext`, которые позволяют представить запрос с или без HTTPS. Необходимо протестировать оба условия, что делается следующим образом:

```
...
httpRequest.SetupSequence(m => m.IsHttps).Returns(true).Returns(false);
...
```

Этот оператор настраивает свойство `HttpRequest.IsHttps` таким образом, что оно возвращает последовательность значений: свойство возвращает `true`, когда читается в первый раз, и `false` — во второй раз. Имеющийся объект `HttpContext` можно применить для создания объекта `ActionContext`, который позволит создать объект `AuthorizationContext`, необходимый модульным тестам. За счет инспектирования свойства `Result` объекта `AuthorizationFilterContext` производится проверка, как фильтр реагирует на запросы, отличные от HTTPS, и затем проверка того, что происходит с запросами HTTP. Для настройки объекта `AuthorizationFilterContext` требуется много типов, которые полагаются на многочисленные пространства имен ASP.NET Core и MVC, но после получения объекта контекста написание оставшейся части теста будет относительно простым.

Использование фильтров действий

Понять фильтры действий проще всего, взглянув на интерфейс, который их определяет. Вот как выглядит интерфейс `IActionFilter`:

```
namespace Microsoft.AspNetCore.Mvc.Filters {

  public interface IActionFilter : IFilterMetadata {
    void OnActionExecuting(ActionExecutingContext context);
    void OnActionExecuted(ActionExecutedContext context);
  }
}
```

Когда к методу действия применен фильтр действий, метод `OnActionExecuting()` вызывается прямо перед вызовом метода действия, а метод `OnActionExecuted()` — сразу после вызова метода действия. Фильтры действий снабжаются данными контекста через два класса контекста: `ActionExecutingContext` для метода `OnActionExecuting()` и `ActionExecutedContext` для метода `OnActionExecuted()`. Оба класса контекста расширяют класс `FilterContext`, свойства которого были перечислены в табл. 19.4.

В классе ActionExecutingContext, используемом для описания действия, которое планируется вызвать, определены дополнительные свойства, представленные в табл. 19.6.

Таблица 19.6. Свойства ActionExecutingContext

Имя	Описание
Controller	Это свойство возвращает объект контроллера, методы действий которого должны быть вызваны. (Детали метода действия доступны через свойство ActionDescriptor, унаследованное от базовых классов.)
ActionArguments	Это свойство возвращает индексированный по имени словарь аргументов, которые будут переданы методу действия. Фильтр может вставлять, удалять либо изменять аргументы
Result	Если фильтр присваивает этому свойству объект реализации IActionResult, тогда произойдет обход процесса обработки запросов, и результат действия будет применяться для генерации ответа клиенту без обращения к методу действия

Класс ActionExecutedContext используется для представления действия, которое было выполнено, и определяет свойства, описанные в табл. 19.7.

Таблица 19.7. Свойства ActionExecutedContext

Имя	Описание
Controller	Это свойство возвращает объект Controller, чей метод действия будет вызван
Canceled	Это свойство типа bool устанавливается в true, если другой фильтр действий предпринял обход процесса обработки запросов за счет присваивания результата действия свойству Result объекта ActionExecutingContext
Exception	Это свойство содержит любой объект Exception, который был сгенерирован методом действия
ExceptionDispatchInfo	Это свойство возвращает объект ExceptionDispatchInfo, который содержит информацию трассировки стека для любого исключения, сгенерированного методом действия
ExceptionHandled	Установка этого свойства в true указывает, что фильтр обработал исключение, которое дальше распространяться не будет
Result	Это свойство возвращает объект реализации IActionResult, возвращенный методом действия. При необходимости фильтр может изменить или полностью заменить результат действия

Создание фильтра действий

Фильтры действий являются универсальным инструментом и могут применяться для реализации любой сквозной обязанности в приложении. Фильтры действий можно использовать для прерывания обработки запроса перед вызовом действия и для изменения результата после выполнения действия. Простейший способ создания фильтра действий предусматривает наследование класса от класса ActionFilterAttribute,

который реализует интерфейс IActionFilter. В целях демонстрации добавим в папку Infrastructure файл класса по имени ProfileAttribute.cs и определим в нем фильтр (листинг 19.12).

Листинг 19.12. Содержимое файла ProfileAttribute.cs из папки Infrastructure

```
using System.Diagnostics;
using System.Text;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {

  public class ProfileAttribute : ActionFilterAttribute {
    private Stopwatch timer;

    public override void OnActionExecuting(ActionExecutingContext context) {
      timer = Stopwatch.StartNew();
    }

    public override void OnActionExecuted(ActionExecutedContext context) {
      timer.Stop();
      string result = "<div>Elapsed time: "
        + $"{timer.Elapsed.TotalMilliseconds} ms</div>";
      byte[] bytes = Encoding.ASCII.GetBytes(result);
      context.HttpContext.Response.Body.Write(bytes, 0, bytes.Length);
    }
  }
}
```

В листинге 19.12 для измерения количества миллисекунд, в течение которых выполнялся метод действия, применяется объект Stopwatch, запуская таймер в методе OnActionExecuting() и останавливая его в методе OnActionExecuted(). Чтобы пометить результат, с использованием объекта контекста получается ответ HttpResponse, в который включается простой фрагмент HTML-разметки.

В листинге 19.13 иллюстрируется применение атрибута Profile к контроллеру Home. (Кроме того, удален предыдущий фильтр, так что будут приниматься запросы по стандартному протоколу HTTP.)

Совет. Как ни странно, контроллеры являются также и фильтрами действий. Базовый класс Controller реализует интерфейсы IActionFilter и IAsyncActionFilter, а это значит, что можно переопределить методы, определяемые упомянутыми интерфейсами, и создать функциональность фильтра действий. Для контроллеров POCO инфраструктура MVC инспектирует классы и проверяет, реализуют ли они любой из интерфейсов фильтра действий, и если реализуют, то автоматически использует их в качестве фильтров действий.

Листинг 19.13. Применение фильтра в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {

  [Profile]
```

```
  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");

    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");
  }
}
```

В результате запуска приложения отобразится сообщение, подобное показанному на рис. 19.4. Количество миллисекунд будет варьироваться в зависимости от скорости машины разработки.

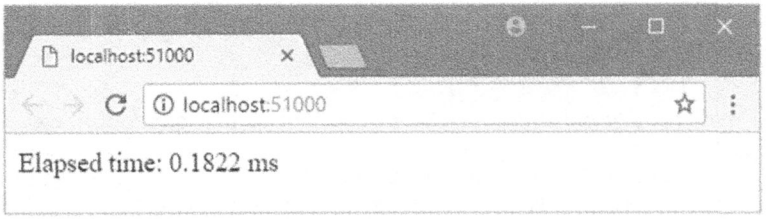


Рис. 19.4. Использование фильтра действий

На заметку! При записи фрагментов HTML-разметки прямо в ответ производится расчет на толерантность браузера к неправильно оформленным HTML-документам: элемент `div`, генерируемый в фильтре, находится в начале тела ответа перед элементами `DOCTYPE` и `html`, которые указывают на начало HTML-документа, выпускаемого представлением Razor. Такой прием работает и может быть удобным для отображения диагностической информации, но на него не следует полагаться при реализации производственных функций.

Создание асинхронного фильтра действий

Интерфейс `IAsyncActionFilter` применяется для определения фильтров действий, которые оперируют асинхронным образом. Ниже показано определение этого интерфейса:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.Filters {

  public interface IAsyncActionFilter : IFilterMetadata {

    Task OnActionExecutionAsync(ActionExecutingContext context,
      ActionExecutionDelegate next);
  }
}
```

В интерфейсе имеется единственный метод, который посредством продолжения задачи позволяет фильтру запускаться до и после выполнения метода действия. В листинге 19.14 используется метод `OnActionExecutionAsync()` из фильтра `Profile`.

Листинг 19.14. Создание асинхронного фильтра действий в файле ProfileAttribute.cs из папки Infrastructure

```
using System.Diagnostics;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {

  public class ProfileAttribute : ActionFilterAttribute {
    public override async Task OnActionExecutionAsync(
          ActionExecutingContext context,
          ActionExecutionDelegate next) {
      Stopwatch timer = Stopwatch.StartNew();
      await next();
      timer.Stop();
      string result = "<div>Elapsed time: "
        + $"{timer.Elapsed.TotalMilliseconds} ms</div>";
      byte[] bytes = Encoding.ASCII.GetBytes(result);
      await context.HttpContext.Response.Body.WriteAsync(bytes,
        0, bytes.Length);
    }
  }
}
```

Объект `ActionExecutingContext` снабжает фильтр данными контекста, а объект `ActionExectionDelegate` представляет метод действия (или следующий фильтр), подлежащий выполнению. Фильтр выполняет свою подготовительную работу перед вызовом делегата и затем завершает работу, когда делегат заканчивает функционирование. Делегат возвращает объект `Task`, поэтому в листинге применяется ключевое слово `await`.

Использование фильтров результатов

Фильтры результатов применяются до и после того, как инфраструктура MVC обработала результат, возвращаемый методом действия. Фильтры результатов способны изменять или заменять результат действия либо полностью аннулировать запрос (даже если метод действия уже был вызван). Вот интерфейс `IResultFilter`, который определяет фильтр результатов:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
  public interface IResultFilter : IFilterMetadata {
    void OnResultExecuting(ResultExecutingContext context);
    void OnResultExecuted(ResultExecutedContext context);
  }
}
```

Фильтры результатов следуют тому же самому шаблону, что и фильтры действий. Метод `OnResultExecuting()` вызывается до того, как результат действия, выпущенный методом действия, будет обработан, и снабжается информацией контекста через объект `ResultExecutingContext`. Класс `ResultExecutingContext` является производным от класса `FilterContext` и определяет дополнительные свойства, описанные в табл. 19.8.

Таблица 19.8. Свойства `ResultExecutingContext`

Имя	Описание
`Controller`	Это свойство возвращает объект контроллера, чей метод действия был выполнен
`Cancel`	Установка этого свойства типа `bool` в `true` остановит обработку результата действия для генерации ответа
`Result`	Это свойство возвращает объект реализации `IActionResult`, возвращаемый методом действия

Метод `OnResultExecuted()` вызывается после обработки инфраструктурой MVC результата действия и снабжается данными контекста через экземпляр класса `ResultExecutedContext`, который в дополнение к свойствам, унаследованным от `FilterContext`, определяет свойства, перечисленные в табл. 19.9.

Таблица 19.9. Свойства `ResultExecutedContext`

Имя	Описание
`Controller`	Это свойство возвращает объект контроллера, чей метод действия был выполнен
`Canceled`	Это свойство типа `bool` указывает, был ли аннулирован запрос
`Exception`	Это свойство содержит любой объект `Exception`, который был сгенерирован методом действия
`ExceptionDispatchInfo`	Это свойство возвращает объект `ExceptionDispatchInfo`, который содержит информацию трассировки стека для любого исключения, сгенерированного методом действия
`ExceptionHandled`	Установка этого свойства в `true` указывает, что фильтр обработал исключение, которое дальше распространяться не будет
`Result`	Это свойство возвращает объект реализации `IActionResult`, который использовался для генерации ответа клиенту

Создание фильтра результатов

Класс `ResultFilterAttribute` реализует интерфейсы фильтров результатов и предлагает самый легкий способ создания фильтра результатов, который может применяться как атрибут. Чтобы посмотреть на фильтр результатов в работе, добавим в папку `Infrastructure` файл класса по имени `ViewResultDetailsAttribute.cs` с определением фильтра из листинга 19.15.

Листинг 19.15. Содержимое файла `ViewResultDetailsAttribute.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
```

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace Filters.Infrastructure {

  public class ViewResultDetailsAttribute : ResultFilterAttribute {

    public override void OnResultExecuting(ResultExecutingContext context) {
      Dictionary<string, string> dict = new Dictionary<string, string> {
        ["Result Type"] = context.Result.GetType().Name
      };

      ViewResult vr;
      if ((vr = context.Result as ViewResult) != null) {
        dict["View Name"] = vr.ViewName;
        dict["Model Type"] = vr.ViewData.Model.GetType().Name;
        dict["Model Data"] = vr.ViewData.Model.ToString();
      }

      context.Result = new ViewResult {
        ViewName = "Message",
        ViewData = new ViewDataDictionary(
          new EmptyModelMetadataProvider(),
          new ModelStateDictionary()) { Model = dict }
      };
    }
  }
}
```

Класс `ViewResultDetailsAttribute` переопределяет единственный метод `OnResultExecuting()` и использует объект контекста для изменения результата действия, применяемого при генерации ответа клиенту. Фильтр создает объект `ViewResult`, который визуализирует представление `Message`, используя словарь с простой диагностической информацией в качестве модели представления.

Метод `OnResultExecuting()` вызывается после того, как метод действия выпустил результат действия, но перед его обработкой с целью генерации результата. Изменение значения свойства `Result` объекта контекста позволяет предоставлять другой тип результата из метода действия, к которому фильтр был применен. В листинге 19.16 фильтр результатов применяется к контроллеру `Home`.

Листинг 19.16. Применение фильтра результатов в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {

  [ViewResultDetails]
  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");

    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");
  }
}
```

Запустив приложение, можно увидеть эффект от фильтра результатов (рис. 19.5).

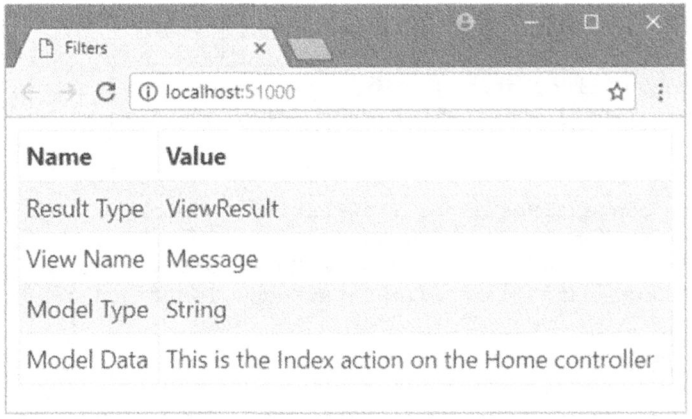


Рис. 19.5. Эффект от фильтра результатов

Создание асинхронного фильтра результатов

Интерфейс `IAsyncResultFilter` используется для создания асинхронных фильтров результатов. Вот его определение:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.Filters {

  public interface IAsyncResultFilter : IFilterMetadata {

    Task OnResultExecutionAsync(ResultExecutingContext context,
      ResultExecutionDelegate next);
  }
}
```

Интерфейс `IAsyncResultFilter` похож на интерфейс, предназначенный для асинхронных фильтров действий. В листинге 19.17 класс `ViewResultDetailsAttribute` переписан с целью реализации интерфейса `IAsyncResultFilter`.

Листинг 19.17. Создание асинхронного фильтра результатов в файле `ViewResultDetailsAttribute.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace Filters.Infrastructure {

  public class ViewResultDetailsAttribute : ResultFilterAttribute {

    public override async Task OnResultExecutionAsync(
        ResultExecutingContext context,
        ResultExecutionDelegate next) {

      Dictionary<string, string> dict = new Dictionary<string, string> {
```

```
      ["Result Type"] = context.Result.GetType().Name,
    };
    ViewResult vr;
    if ((vr = context.Result as ViewResult) != null) {
      dict["View Name"] = vr.ViewName;
      dict["Model Type"] = vr.ViewData.Model.GetType().Name;
      dict["Model Data"] = vr.ViewData.Model.ToString();
    }
    context.Result = new ViewResult {
      ViewName = "Message",
      ViewData = new ViewDataDictionary(
        new EmptyModelMetadataProvider(),
        new ModelStateDictionary()) {
          Model = dict
        }
    };
    await next();
  }
 }
}
```

Обратите внимание, что именно вы несете ответственность за вызов делегата, полученного в виде аргумента методом `OnResultExecutionAsync()`. Если не вызвать делегат, тогда конвейер обработки запросов не будет завершен и результат действия не визуализируется.

Создание гибридного фильтра действий/результатов

Проводить различие между стадиями действия и результата внутри процесса обработки запросов не всегда удобно. Так может случиться из-за того, что вы хотите трактовать обе стадии как один шаг, либо потому, что ваш фильтр реагирует на способ выполнения действия, но делает это, вмешиваясь в результат. Полезно располагать возможностью создания фильтра, который является фильтром одновременно действий и результатов и способен выполнять работу на каждой стадии.

Требование настолько распространенное, что класс `ActionFilterAttribute` реализует интерфейсы для обеих разновидностей фильтров, т.е. в единственном атрибуте можно смешивать и сочетать типы фильтров. Для демонстрации в листинге 19.18 приведен пересмотренный класс `ProfileAttribute`, в котором фильтр действий объединяется с фильтром результатов.

Листинг 19.18. Создание гибридного фильтра в файле `ProfileAttribute.cs` из папки `Infrastructure`

```
using System.Diagnostics;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {

  public class ProfileAttribute : ActionFilterAttribute {
    private Stopwatch timer;
```

```
    private double actionTime;
    public override async Task OnActionExecutionAsync(
          ActionExecutingContext context,
          ActionExecutionDelegate next) {
      timer = Stopwatch.StartNew();
      await next();
      actionTime = timer.Elapsed.TotalMilliseconds;
    }
    public override async Task OnResultExecutionAsync(
       ResultExecutingContext context,
       ResultExecutionDelegate next) {
      await next();
      timer.Stop();
      string result = "<div>Action time: "
        + $"{actionTime} ms</div><div>Total time: "
        + $"{timer.Elapsed.TotalMilliseconds} ms</div>";
      byte[] bytes = Encoding.ASCII.GetBytes(result);
      await context.HttpContext.Response.Body.WriteAsync(bytes,
        0, bytes.Length);
    }
  }
}
```

Здесь для фильтров обоих типов используются асинхронные методы, но получить требуемую функциональность можно за счет смешивания и сочетания, т.к. стандартные реализации этих методов вызывают свои синхронные аналоги. Внутри фильтра с помощью `Stopwatch` измеряется время выполнения действия и общее время, а результаты записываются в ответ. В листинге 19.19 комбинированный фильтр применяется к контроллеру `Home`.

Листинг 19.19. Применение гибридного фильтра в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {

  [Profile]
  [ViewResultDetails]
  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");

    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");
  }
}
```

В результате запуска приложения появится вывод, подобный показанному на рис. 19.6. Вывод отобразится после содержимого, предоставляемого атрибутом

ViewResultDetails, т.к. он записывается на стадии постобработки фильтра результатов, а не получается из метода фильтра действий, который использовался в предыдущей версии.

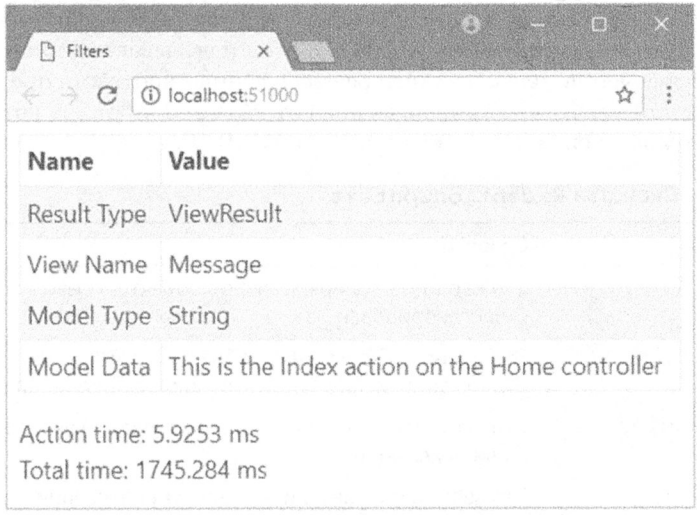


Рис. 19.6. Вывод из гибридного фильтра действий/результатов

Использование фильтров исключений

Фильтры исключений позволяют реагировать на исключения без необходимости в написании блоков try...catch в каждом методе действия. Фильтры исключений могут применяться к классам контроллеров или к методам действий. Они вызываются, когда исключение не обработано методом действия либо фильтрами действий или результатов, которые были применены к методу действия. (Фильтры действий и результатов могут иметь дело с необработанным исключением, устанавливая свойство ExceptionHandled своих объектов контекста в true.) Фильтры исключений реализуют интерфейс IExceptionFilter, который определен следующим образом:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
  public interface IExceptionFilter : IFilterMetadata {
    void OnException(ExceptionContext context);
  }
}
```

Метод OnException() вызывается при столкновении с необработанным исключением. Интерфейс IAsyncExceptionFilter может использоваться для создания асинхронных фильтров исключений, которые удобны, когда необходимо реагировать на исключения с применением асинхронного API-интерфейса. Вот определение интерфейса IAsyncExceptionFilter:

```
using System.Threading.Tasks;
namespace Microsoft.AspNetCore.Mvc.Filters {
```

```
    public interface IAsyncExceptionFilter : IFilterMetadata {
      Task OnExceptionAsync(ExceptionContext context);
    }
}
```

Метод `OnExceptionAsync()` является асинхронным аналогом метода `OnException()` из интерфейса `IExceptionFilter` и вызывается при наличии необработанного исключения. Для обоих интерфейсов данные контекста предоставляются через класс `ExceptionContext`, производный от `FilterContext`, в котором определены дополнительные свойства, описанные в табл. 19.10.

Таблица 19.10. Свойства `ExceptionContext`

Имя	Описание
`Exception`	Это свойство содержит любой объект `Exception`, который был сгенерирован
`ExceptionDispatchInfo`	Это свойство возвращает объект `ExceptionDispatchInfo`, содержащий информацию трассировки стека для исключения
`ExceptionHandled`	Это свойство типа `bool` используется для указания, обработано ли исключение
`Result`	Это свойство устанавливает объект реализации `IActionResult`, который будет применяться для генерации ответа

Создание фильтра исключений

Класс `ExceptionFilterAttribute` реализует оба интерфейса фильтров исключений и является самым легким способом создания фильтра, так что он может быть применен как атрибут. Наиболее распространенное использование фильтра исключений связано с отображением специальной страницы ошибки для специфического типа исключения, снабжая пользователя более полезной информацией, нежели способны обеспечить стандартные возможности обработки ошибок. В качестве демонстрации добавим в папку `Infrastructure` файл класса по имени `RangeExceptionAttribute.cs` и определим в нем фильтр (листинг 19.20).

Листинг 19.20. Содержимое файла `RangeExceptionAttribute.cs` из папки `Infrastructure`

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ViewFeatures;

namespace Filters.Infrastructure {

  public class RangeExceptionAttribute : ExceptionFilterAttribute {

    public override void OnException(ExceptionContext context) {
      if (context.Exception is ArgumentOutOfRangeException) {
        context.Result = new ViewResult() {
          ViewName = "Message",
          ViewData = new ViewDataDictionary(
```

```
            new EmptyModelMetadataProvider(),
            new ModelStateDictionary()) {
              Model = @"The data received by the
                      application cannot be processed"
            }
          };
        }
      }
    }
}
```

Этот фильтр применяет объект `ExceptionContext` для получения типа необработанного исключения. Если типом является `ArgumentOutOfRangeException`, то создается результат действия, который отображает сообщение пользователю. В листинге 19.21 в контроллер `Home` добавляется метод действия, а к контроллеру применяется фильтр исключений.

Листинг 19.21. Применение фильтра исключений в файле `HomeController.cs` из папки `Controllers`

```
using Filters.Infrastructure;
using Microsoft.AspNetCore.Mvc;
using System;

namespace Filters.Controllers {

  [Profile]
  [ViewResultDetails]
  [RangeException]
  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");

    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");

    public ViewResult GenerateException(int? id) {
      if (id == null) {
        throw new ArgumentNullException(nameof(id));
      } else if (id > 10) {
        throw new ArgumentOutOfRangeException(nameof(id));
      } else {
        return View("Message", $"The value is {id}");
      }
    }
  }
}
```

При получении из URL запроса значения `int`, допускающего `null`, метод действия `GenerateException()` полагается на стандартный шаблон маршрутизации. Этот метод действия генерирует исключение `ArgumentNullException`, если соответствующий сегмент URL отсутствует, и исключение `ArgumentOutOfRangeException`, если его значение больше 10. При наличии сегмента со значением менее 10 метод действия возвращает объект `ViewResult`.

Протестировать фильтр исключений можно, запустив приложение и запросив URL вида `/Home/GenerateException/100`. Значение в последнем сегменте выходит за пределы, ожидаемые методом действия, который сгенерирует исключение с типом, обрабатываемым фильтром, что даст результат, представленный на рис. 19.7. В случае запроса `/Home/GenerateException` метод действия сгенерирует исключение, которое не будет обработано фильтром, поэтому задействуется стандартная обработка ошибок.

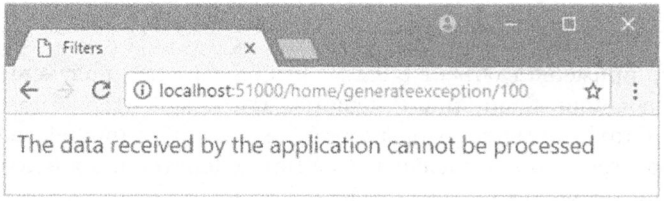


Рис. 19.7. Использование фильтра исключений

Использование внедрения зависимостей для фильтров

Когда фильтр делается производным от одного из удобных классов атрибутов, таких как `ExceptionFilterAttribute`, для обработки каждого запроса инфраструктура MVC создает новый экземпляр класса фильтра. Это рациональный подход, поскольку он позволяет избежать возможных проблем, связанных с многократным использованием или параллелизмом, и удовлетворяет нуждам большинства классов фильтров, которые требуются разработчикам.

Альтернативный подход предусматривает применение системы внедрения зависимостей с целью выбора другого жизненного цикла для фильтров. Использовать внедрение зависимостей в фильтрах можно двумя способами, которые будут описаны в последующих разделах.

Распознавание зависимостей в фильтрах

Первый подход заключается в применении внедрения зависимостей для управления данными контекста, которые предназначены фильтрам. Это позволяет фильтрам разных типов совместно использовать данные или одиночному фильтру разделять данные между своими экземплярами, применяемыми для обработки других запросов. Чтобы посмотреть, как все работает, добавим в папку `Infrastructure` файл класса по имени `FilterDiagnostics.cs` с определениями интерфейса и класса реализации из листинга 19.22.

Листинг 19.22. Содержимое файла `FilterDiagnostics.cs` из папки `Infrastructure`

```
using System.Collections.Generic;

namespace Filters.Infrastructure {

  public interface IFilterDiagnostics {
    IEnumerable<string> Messages { get; }
    void AddMessage(string message);
  }
```

```
  public class DefaultFilterDiagnostics : IFilterDiagnostics {
    private List<string> messages = new List<string>();

    public IEnumerable<string> Messages => messages;

    public void AddMessage(string message) =>
      messages.Add(message);
  }
}
```

Интерфейс `IFilterDiagnostics` определяет простую модель для сбора диагностических сообщений во время выполнения фильтра. Класс `DefaultFilterDiagnostics` является реализацией, которая будет использоваться. В листинге 19.23 показан обновленный класс `Startup`, конфигурирующий поставщик служб с целью применения нового интерфейса и его реализации.

Листинг 19.23. Конфигурирование поставщика служб в файле `Startup.cs` из папки `Filters`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Filters.Infrastructure;

namespace Filters {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddScoped<IFilterDiagnostics, DefaultFilterDiagnostics>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Поставщик служб конфигурируется с использованием расширяющего метода `AddScoped()`, т.е. все фильтры, создаваемые для работы с одиночным запросом, получат тот же самый объект `DefaultFilterDiagnostics`. Это является основой для разделения специальных данных контекста между фильтрами.

Создание фильтров с зависимостями

На следующем шаге создаются фильтры, которые объявляют зависимости от интерфейса `IFilterDiagnostics`. Добавим в папку `Infrastructure` файл класса по имени `TimeFilter.cs` с содержимым, приведенным в листинге 19.24.

Листинг 19.24. Содержимое файла `TimeFilter.cs` из папки `Infrastructure`

```
using System.Diagnostics;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {

  public class TimeFilter : IAsyncActionFilter, IAsyncResultFilter {
    private Stopwatch timer;
    private IFilterDiagnostics diagnostics;

    public TimeFilter(IFilterDiagnostics diags) {
      diagnostics = diags;
    }

    public async Task OnActionExecutionAsync(
          ActionExecutingContext context,
          ActionExecutionDelegate next) {

      timer = Stopwatch.StartNew();
      await next();
      diagnostics.AddMessage($@"Action time:
        {timer.Elapsed.TotalMilliseconds}");
    }

    public async Task OnResultExecutionAsync(
          ResultExecutingContext context,
          ResultExecutionDelegate next) {

      await next();
      timer.Stop();
      diagnostics.AddMessage($@"Result time:
        {timer.Elapsed.TotalMilliseconds}");
    }
  }
}
```

Класс `TimeFilter` представляет собой гибридный фильтр действий/результатов, который воссоздает функциональность таймера из предыдущего примера, но хранит информацию о времени с применением реализации интерфейса `IFilterDiagnostics`, которая объявлена как аргумент конструктора и будет предоставлена системой внедрения зависимостей при создании фильтра.

Обратите внимание, что класс `TimeFilter` явно реализует интерфейсы фильтров, а не наследуется от удобного класса атрибута. Вы увидите, что фильтры, которые опираются на внедрение зависимостей, применяются через другой атрибут и не используются для декорирования контроллеров или действий напрямую.

Чтобы выяснить, каким образом фильтры задействуют внедрение зависимостей для разделения данных контекста, добавим в папку `Infrastructure` файл класса по имени `DiagnosticsFilter.cs` и определим в нем фильтр, показанный в листинге 19.25.

Листинг 19.25. Содержимое файла `DiagnosticsFilter.cs` из папки `Infrastructure`

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;
```

```
namespace Filters.Infrastructure {
  public class DiagnosticsFilter : IAsyncResultFilter {
    private IFilterDiagnostics diagnostics;

    public DiagnosticsFilter(IFilterDiagnostics diags) {
      diagnostics = diags;
    }

    public async Task OnResultExecutionAsync(
          ResultExecutingContext context,
          ResultExecutionDelegate next) {

      await next();

      foreach (string message in diagnostics?.Messages) {
        byte[] bytes = Encoding.ASCII
          .GetBytes($"<div>{message}</div>");
        await context.HttpContext.Response.Body
          .WriteAsync(bytes, 0, bytes.Length);
      }
    }
  }
}
```

Класс `DiagnosticsFilter` — это фильтр результатов, который получает реализацию интерфейса `IFilterDiagnostics` в виде аргумента конструктора и записывает содержащиеся в ней сообщения в ответ.

Применение фильтров

На финальном шаге фильтры применяются к классу контроллера. Стандартные атрибуты C# не располагают всеобъемлющей поддержкой для распознавания зависимостей конструктора, из-за чего фильтры, которые были определены в предшествующих разделах, не являются атрибутами. Взамен применяется атрибут `TypeFilter`, сконфигурированный с необходимым типом фильтра (листинг 19.26).

Совет. Порядок применения фильтров в листинге 19.26 важен, как будет объясняться в разделе "Порядок применения фильтров и его изменение" далее в главе.

Листинг 19.26. Применение фильтров с зависимостями в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;
using System;

namespace Filters.Controllers {

  [TypeFilter(typeof(DiagnosticsFilter))]
  [TypeFilter(typeof(TimeFilter))]
  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");
    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");
```

```
    public ViewResult GenerateException(int? id) {
      if (id == null) {
        throw new ArgumentNullException(nameof(id));
      } else if (id > 10) {
        throw new ArgumentOutOfRangeException(nameof(id));
      } else {
        return View("Message", $"The value is {id}");
      }
    }
  }
}
```

Атрибут `TypeFilter` создает новый экземпляр класса фильтра для каждого запроса, но делает это с использованием средства внедрения зависимостей, которое позволяет создавать слабо связанные компоненты и помещать объекты, участвующие в распознавании зависимостей, под управление жизненным циклом.

В текущем примере сказанное означает, что оба фильтра, примененные в листинге 19.26, получат тот же самый объект реализации `IFilterDiagnostics`, а сообщения, сохраненные классом `TimeFilter`, будут записаны в ответ классом `DiagnosticsFilter`. Запустив приложение и запросив стандартный для него URL, можно увидеть результат (рис. 19.8).

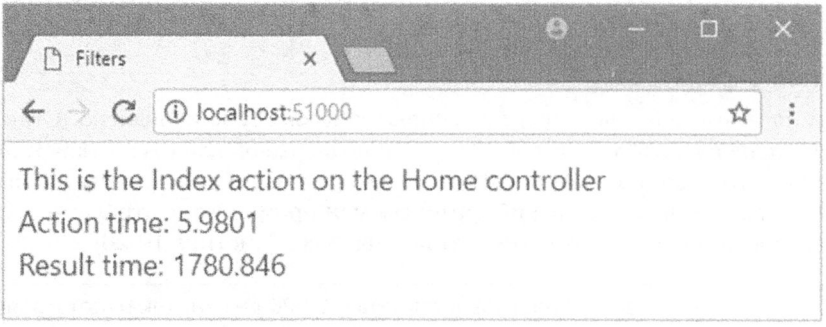


Рис. 19.8. Использование фильтров с зависимостями

Управление жизненными циклами фильтров

В случае применения атрибута `TypeFilter` новый экземпляр класса фильтра создается для каждого запроса. Такое поведение ничем не отличается от поведения при использовании фильтра напрямую как атрибута за исключением того, что атрибут `TypeFilter` позволяет классу фильтра объявлять зависимости, которые распознаются посредством поставщика служб.

Атрибут `ServiceFilter` продвигается на шаг дальше и применяет поставщик служб для создания объекта фильтра, что также дает возможность помещать объекты фильтров под управление жизненным циклом. Для демонстрации в листинге 19.27 приведен модифицированный класс `TimeFilter`, который сохраняет среднее арифметическое записанных значений времени.

Листинг 19.27. Сохранение средних арифметических значений в файле TimeFilter.cs из папки Infrastructure

```
using System.Collections.Concurrent;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {
  public class TimeFilter : IAsyncActionFilter, IAsyncResultFilter {
    private ConcurrentQueue<double> actionTimes = new ConcurrentQueue<double>();
    private ConcurrentQueue<double> resultTimes = new ConcurrentQueue<double>();
    private IFilterDiagnostics diagnostics;
    public TimeFilter(IFilterDiagnostics diags) {
      diagnostics = diags;
    }
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context, ActionExecutionDelegate next) {
      Stopwatch timer = Stopwatch.StartNew();
      await next();
      timer.Stop();
      actionTimes.Enqueue(timer.Elapsed.TotalMilliseconds);
      diagnostics.AddMessage($@"Action time:
        {timer.Elapsed.TotalMilliseconds}
        Average: {actionTimes.Average():F2}");
    }
    public async Task OnResultExecutionAsync(
        ResultExecutingContext context, ResultExecutionDelegate next) {
      Stopwatch timer = Stopwatch.StartNew();
      await next();
      timer.Stop();
      resultTimes.Enqueue(timer.Elapsed.TotalMilliseconds);
      diagnostics.AddMessage($@"Result time:
        {timer.Elapsed.TotalMilliseconds}
        Average: {resultTimes.Average():F2}");
    }
  }
}
```

Фильтр теперь использует безопасную к потокам коллекцию для хранения значений времени, которые он записывает для стадий действия и результата обработки запроса, и применяет отдельный объект Stopwatch каждый раз, когда поступает требование обработать запрос. В листинге 19.28 класс TimeFilter регистрируется как одиночка с помощью поставщика служб в классе Startup.

Листинг 19.28. Конфигурирование поставщика служб в файле Startup.cs из папки Filters

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Filters.Infrastructure;

namespace Filters {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
     services.AddSingleton<IFilterDiagnostics, DefaultFilterDiagnostics>();
     services.AddSingleton<TimeFilter>();
     services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Обратите внимание, что жизненный цикл для класса реализации `IFilterDiagnostics` также изменен, чтобы он стал одиночкой. Если бы сохранилось создание нового экземпляра для каждого запроса, то объект-одиночка `TimeFilter` получал бы разные объекты реализации `IFilterDiagnostics` от фильтра `DiagnosticsFilter`, который продолжил бы создаваться через атрибут `TypeFilter` и создавался бы для каждого запроса.

Применение фильтра

Последний шаг касается применения фильтра к контроллеру с использованием атрибута `ServiceType` (листинг 19.29).

Листинг 19.29. Применение фильтра в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;
using System;

namespace Filters.Controllers {

  [TypeFilter(typeof(DiagnosticsFilter))]
  [ServiceFilter(typeof(TimeFilter))]
  public class HomeController : Controller {

    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");

    public ViewResult SecondAction() => View("Message",
      "This is the SecondAction action on the Home controller");
```

```
    public ViewResult GenerateException(int? id) {
      if (id == null) {
        throw new ArgumentNullException(nameof(id));
      } else if (id > 10) {
        throw new ArgumentOutOfRangeException(nameof(id));
      } else {
        return View("Message", $"The value is {id}");
      }
    }
  }
}
```

Запустив приложение и запросив стандартный URL, можно увидеть результат. Поскольку для распознавания всех зависимостей используется единственный объект реализации интерфейса `IFilterDiagnostics`, набор отображаемых сообщений строится с каждым запросом (рис. 19.9).

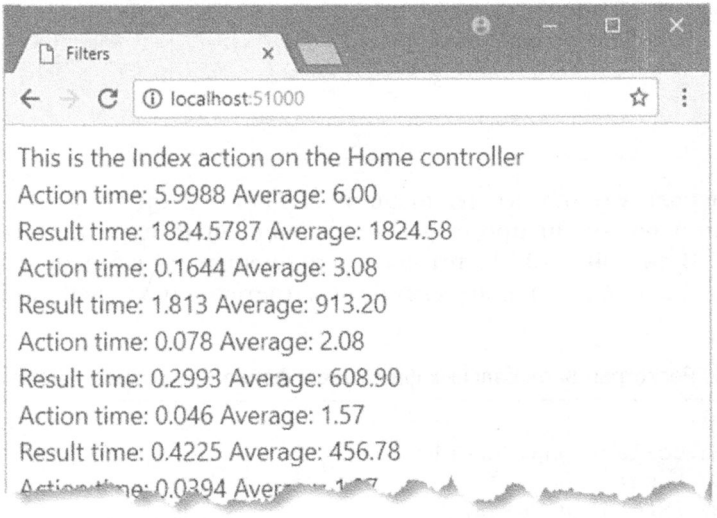


Рис. 19.9. Применение поставщика служб для управления жизненным циклом фильтра

Создание глобальных фильтров

В начале главы было указано, что вы можете применять фильтры к целому классу контроллера, чтобы не применять их к индивидуальным методам действий. *Глобальные фильтры* продвигаются еще дальше и применяются один раз в классе `Startup`, после чего, как и следует из их названия, автоматически применяются к каждому методу действия в каждом контроллере внутри приложения. В качестве глобального фильтра может использоваться любой фильтр; в целях демонстрации добавим в папку `Infrastructure` файл класса по имени `ViewResultDiagnostics.cs` с определением фильтра, показанным в листинге 19.30.

Листинг 19.30. Содержимое файла `ViewResultDiagnostics.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
namespace Filters.Infrastructure {
  public class ViewResultDiagnostics : IActionFilter {
    private IFilterDiagnostics diagnostics;
    public ViewResultDiagnostics(IFilterDiagnostics diags) {
      diagnostics = diags;
    }
    public void OnActionExecuting(ActionExecutingContext context) {
      // Ничего не делать - метод в этом фильтре не используется.
    }
    public void OnActionExecuted(ActionExecutedContext context) {
      ViewResult vr;
      if ((vr = context.Result as ViewResult) != null) {
        diagnostics.AddMessage($"View name: {vr.ViewName}");
        diagnostics.AddMessage($@"Model type:
          {vr.ViewData.Model.GetType().Name}");
      }
    }
  }
}
```

Фильтр использует объект реализации `IFilterDiagnostics` для сохранения сообщений об имени представления и типе модели результатов действий `ViewResult`. В листинге 19.31 этот фильтр применяется глобально наряду с классом `DiagnosticsFilter`, на который он полагается при записи диагностических сообщений.

Листинг 19.31. Регистрация глобальных фильтров в файле `Startup.cs` из папки `Filters`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Filters.Infrastructure;
namespace Filters {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddScoped<IFilterDiagnostics, DefaultFilterDiagnostics>();
      services.AddScoped<TimeFilter>();
      services.AddScoped<ViewResultDiagnostics>();
      services.AddScoped<DiagnosticsFilter>();
      services.AddMvc().AddMvcOptions(options => {
        options.Filters.AddService(typeof(ViewResultDiagnostics));
        options.Filters.AddService(typeof(DiagnosticsFilter));
      });
    }
```

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Глобальные фильтры настраиваются путем конфигурирования пакета служб MVC. В рассматриваемом примере для регистрации фильтров как глобальных используется метод `MvcOptions.Filters.AddService()`. Метод `AddService()` принимает тип .NET, экземпляр которого будет создан с применением правил жизненного цикла, указанных где-то в другом месте метода `ConfigureServices()`. Жизненный цикл других типов фильтров изменен на ограниченный областью действия, так что новые экземпляры создаются для каждого запроса. В результате новые экземпляры фильтров `ViewResultDiagnostics` и `DiagnosticsFilter` создаются и применяются к каждому запросу, адресованному каждому контроллеру.

Совет. Добавлять глобальные фильтры можно также с использованием вместо `AddService()` метода `Add()`, который позволяет регистрировать объект фильтра в качестве глобального фильтра, не полагаясь на внедрение зависимостей и поставщика служб. Метод `Add()` применяется в следующем разделе.

Добавим в папку `Controllers` файл класса по имени `GlobalController.cs` с определением контроллера из листинга 19.32.

Листинг 19.32. Содержимое файла `GlobalController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
namespace Filters.Controllers {
  public class GlobalController : Controller {
    public ViewResult Index() => View("Message",
      "This is the global controller");
  }
}
```

Никакие фильтры к контроллеру `Global` не применялись, но после запуска приложения и запрашивания URL вида `/global` отобразится вывод из двух глобальных фильтров (рис. 19.10).

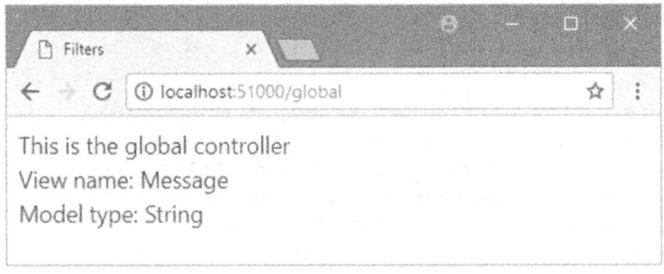


Рис. 19.10. Использование глобальных фильтров

Порядок применения фильтров и его изменение

Фильтры запускаются в специфической очередности: фильтры авторизации, фильтры действий и фильтры результатов. Но при наличии нескольких фильтров заданного типа порядок их применения управляется областью действия, через которую фильтры были применены. Добавим в папку `Infrastructure` файл класса по имени `MessageAttribute.cs` и определим в нем фильтр, приведенный в листинге 19.33.

Листинг 19.33. Содержимое файла `MessageAttribute.cs` из папки `Infrastructure`

```
using System.Text;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters.Infrastructure {
  public class MessageAttribute : ResultFilterAttribute {
    private string message;
    public MessageAttribute(string msg) {
      message = msg;
    }
    public override void OnResultExecuting(ResultExecutingContext context) {
      WriteMessage(context, $"<div>Before Result:{message}</div>");
    }
    public override void OnResultExecuted(ResultExecutedContext context) {
      WriteMessage(context, $"<div>After Result:{message}</div>");
    }
    private void WriteMessage(FilterContext context, string msg) {
      byte[] bytes = Encoding.ASCII
        .GetBytes($"<div>{msg}</div>");
      context.HttpContext.Response
        .Body.Write(bytes, 0, bytes.Length);
    }
  }
}
```

В листинге 19.33 определен фильтр результатов, который записывает фрагменты HTML-разметки до и после обработки результата действия. Сообщение, записываемое фильтром, конфигурируется посредством аргумента конструктора, который может использоваться, когда применяется атрибут. В листинге 19.34 контроллер `Home` упрощен, а фильтры из предшествующих примеров заменены множеством экземпляров фильтра `Message`.

Листинг 19.34. Применение фильтра в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
  [Message("This is the Controller-Scoped Filter")]
  public class HomeController : Controller {
    [Message("This is the First Action-Scoped Filter")]
    [Message("This is the Second Action-Scoped Filter")]
    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");
  }
}
```

Набор глобальных фильтров изменен с целью использования также фильтра Message (листинг 19.35).

Листинг 19.35. Создание глобального фильтра в файле Startup.cs из папки Filters

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Filters.Infrastructure;

namespace Filters {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddScoped<IFilterDiagnostics, DefaultFilterDiagnostics>();
      services.AddScoped<TimeFilter>();
      services.AddScoped<ViewResultDiagnostics>();
      services.AddScoped<DiagnosticsFilter>();
      services.AddMvc().AddMvcOptions(options => {
        options.Filters.Add(new
          MessageAttribute("This is the Globally-Scoped Filter"));
      });
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

При реагировании метода действия Index() на запрос будут использоваться четыре экземпляра фильтра. Запустив приложение и запросив стандартный URL, вы увидите следующий вывод, отображаемый в браузере:

```
Before Result:This is the Globally-Scoped Filter
Before Result:This is the Controller-Scoped Filter
Before Result:This is the First Action-Scoped Filter
Before Result:This is the Second Action-Scoped Filter
After Result:This is the Second Action-Scoped Filter
After Result:This is the First Action-Scoped Filter
After Result:This is the Controller-Scoped Filter
After Result:This is the Globally-Scoped Filter
```

По умолчанию инфраструктура MVC запускает глобальные фильтры, затем фильтры, которые применены к классу контроллера, и в заключение фильтры, примененные к методам действий. После того, как метод действия вызван или результат действия обработан, стек фильтров раскручивается, из-за чего сообщения After Result в выводе расположены в обратном порядке.

Изменение порядка применения фильтров

Стандартный порядок может быть изменен за счет реализации интерфейса IOrderedFilter, который MVC ищет при выяснении, как помещать фильтры в последовательность. Вот определение этого интерфейса:

```
namespace Microsoft.AspNetCore.Mvc.Filters {
  public interface IOrderedFilter : IFilterMetadata {
    int Order { get; }
  }
}
```

Свойство Order возвращает значение int; низкое значение указывает инфраструктуре MVC на необходимость применения фильтра перед фильтрами с более высокими значениями Order. Удобные атрибуты уже реализуют интерфейс IOrderedFilter, и в листинге 19.36 свойство Order устанавливается для фильтров, примененных к контроллеру Home.

Совет. Атрибуты TypeFilter и ServiceFilter тоже реализуют интерфейс IOrderedFilter, т.е. в случае использования внедрения зависимостей порядок применения фильтров также можно изменять.

Листинг 19.36. Установка порядка применения фильтров в файле HomeController.cs из папки Controllers

```
using Filters.Infrastructure;
using Microsoft.AspNetCore.Mvc;
namespace Filters.Controllers {
  [Message("This is the Controller-Scoped Filter", Order = 10)]
  public class HomeController : Controller {
    [Message("This is the First Action-Scoped Filter", Order = 1)]
    [Message("This is the Second Action-Scoped Filter", Order = -1)]
    public ViewResult Index() => View("Message",
      "This is the Index action on the Home controller");
  }
}
```

Значения Order могут быть и отрицательными, что дает удобный способ гарантировать применение фильтра перед любыми глобальными фильтрами со стандартным порядком (хотя при создании глобальных фильтров также можно устанавливать порядок). Запустив приложение, вы увидите, что порядок следования сообщений в выводе изменился, чтобы отразить новые приоритеты:

```
Before Result:This is the Second Action-Scoped Filter
Before Result:This is the Globally-Scoped Filter
Before Result:This is the First Action-Scoped Filter
Before Result:This is the Controller-Scoped Filter
After Result:This is the Controller-Scoped Filter
After Result:This is the First Action-Scoped Filter
After Result:This is the Globally-Scoped Filter
After Result:This is the Second Action-Scoped Filter
```

Резюме

В главе было показано, как инкапсулировать логику сквозной ответственности в виде фильтров. Вы узнали о доступных типах фильтров и о том, как их реализовать. Были приведены объяснения, каким образом применять фильтры в форме атрибутов к контроллерам и методам действий, а также в качестве глобальных фильтров. В следующей главе будет показано, как использовать контроллеры для создания веб-служб.

ГЛАВА 20

Контроллеры API

Не все контроллеры используются для отправки HTML-документов клиентам. Существуют также контроллеры API, которые применяются для предоставления доступа к данным приложения. Это средство, которое ранее предлагалось через отдельную инфраструктуру Web API, но теперь интегрировано в ASP.NET Core MVC. В настоящей главе объясняется роль контроллеров API в веб-приложении, описываются задачи, которые они решают, и демонстрируются способы их создания, тестирования и использования. В табл. 20.1 приведена сводка, позволяющая поместить контроллеры API в контекст.

Таблица 20.1. Помещение контроллеров API в контекст

Вопрос	Ответ
Что это такое?	Контроллеры API подобны обычным контроллерам за исключением того, что ответы, порождаемые их методами действий, представляют собой объекты данных, которые отправляются клиенту без HTML-разметки
Чем они полезны?	Контроллеры API позволяют клиентам иметь доступ к данным в приложении, не получая также и HTML-разметку, которая требуется для представления содержимого пользователю. Не все клиенты являются браузерами, и не все клиенты отображают данные пользователям. Контроллер API делает приложение открытым для поддержки новых типов клиентов или клиентов, разработанных третьей стороной
Как они используются?	Контроллеры API применяются аналогично обычным контроллерам HTML
Существуют ли какие-то скрытые ловушки или ограничения?	Наиболее распространенные проблемы связаны со способом сериализации объектов данных, чтобы их можно было отправить клиенту. За подробными сведениями обращайтесь в раздел "Форматирование содержимого" далее в главе
Существуют ли альтернативы?	Вы не обязаны использовать контроллеры API в своих проектах, однако их применение может увеличить ценность вашей платформы для клиентов

В табл. 20.2 приведена сводка по главе.

Таблица 20.2. Сводка по главе

Задача	Решение	Листинг
Предоставление доступа к данным в приложении	Создайте контроллер API	20.10
Запрашивание данных из контроллера API	Используйте запрос Ajax, либо напрямую работая с API-интерфейсом браузера, либо через библиотеку вроде jQuery	20.11–20.13
Переопределение процесса согласования содержимого	Применяйте атрибут `Produces`	20.14–20.16
Разрешение клиентам переопределять заголовок `Accept` за счет указания формата данных в URL	Добавьте в класс `Startup` отображения форматеров, добавьте переменную сегмента, которая захватывает формат данных, и дополнительно примените атрибут `FormatFilter`	20.17, 20.18
Предоставление полной поддержки для процесса согласования содержимого	Включите форматер `HttpNotAcceptableOutputFormatter` и установите конфигурационное свойство `RespectBrowserAcceptHeader`	20.19, 20.20
Получение данных в разнообразных форматах с использованием разных методов действий	Примените атрибут `Consumes`	20.21

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `ApiControllers` с использованием шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core.

Создание модели и хранилища

Первым делом создадим папку `Models`, добавим в нее файл класса по имени `Reservation.cs` и определим класс модели, как показано в листинге 20.1.

Листинг 20.1. Содержимое файла `Reservation.cs` из папки `Models`

```
namespace ApiControllers.Models {

  public class Reservation {
    public int ReservationId { get; set; }
    public string ClientName { get; set; }
    public string Location { get; set; }
  }
}
```

Затем добавим в папку `Models` файл класса по имени `IRepository.cs` и определим в нем интерфейс для хранилища объектов модели (листинг 20.2).

Листинг 20.2. Содержимое файла IRepository.cs из папки Models

```
using System.Collections.Generic;

namespace ApiControllers.Models {

  public interface IRepository {

    IEnumerable<Reservation> Reservations { get; }
    Reservation this[int id] { get; }

    Reservation AddReservation(Reservation reservation);
    Reservation UpdateReservation(Reservation reservation);
    void DeleteReservation(int id);
  }
}
```

Добавим в папку Models файл класса по имени MemoryRepository.cs с определением непостоянной реализации интерфейса IRepository из листинга 20.3.

Листинг 20.3. Содержимое файла MemoryRepository.cs из папки Models

```
using System.Collections.Generic;

namespace ApiControllers.Models {

  public class MemoryRepository : IRepository {
    private Dictionary<int, Reservation> items;

    public MemoryRepository() {
      items = new Dictionary<int, Reservation>();
      new List<Reservation> {
        new Reservation { ClientName = "Alice", Location = "Board Room" },
        new Reservation { ClientName = "Bob", Location = "Lecture Hall" },
        new Reservation { ClientName = "Joe", Location = "Meeting Room 1"
}
      }.ForEach(r => AddReservation(r));
    }

    public Reservation this[int id] => items.ContainsKey(id) ? items[id] : null;

    public IEnumerable<Reservation> Reservations => items.Values;

    public Reservation AddReservation(Reservation reservation) {
      if (reservation.ReservationId == 0) {
        int key = items.Count;
        while (items.ContainsKey(key)) { key++; };
        reservation.ReservationId = key;
      }
      items[reservation.ReservationId] = reservation;
      return reservation;
    }

    public void DeleteReservation(int id) => items.Remove(id);

    public Reservation UpdateReservation(Reservation reservation)
      => AddReservation(reservation);
  }
}
```

При создании экземпляра хранилища строится простой набор объектов модели, а поскольку постоянство не поддерживается, то в случае останова или перезапуска приложения все внесенные изменения будут утеряны. (В главе 8 приводился пример создания постоянного хранилища как часть проекта приложения SportsStore.)

Создание контроллера и представлений

Позже в главе будут созданы контроллеры REST, но при подготовке нужно создать обычный контроллер, чтобы обеспечить основу для дальнейших примеров. Создадим папку `Controllers`, поместим в нее файл класса по имени `HomeController.cs` и определим в нем контроллер, код которого приведен в листинге 20.4.

Листинг 20.4. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;
namespace ApiControllers.Controllers {
  public class HomeController : Controller {
    private IRepository repository { get; set; }
    public HomeController(IRepository repo) => repository = repo;
    public ViewResult Index() => View(repository.Reservations);
    [HttpPost]
    public IActionResult AddReservation(Reservation reservation) {
      repository.AddReservation(reservation);
      return RedirectToAction("Index");
    }
  }
}
```

В контроллере определено действие `Index`, которое является стандартным для приложения и визуализирует модель данных. В нем также определено действие `AddReservation`, которое доступно только HTTP-запросам `POST` и применяется для получения данных формы от пользователя. Упомянутые действия следуют паттерну Post/Redirect/Get, описанному в главе 17, так что перезагрузка веб-страницы не будет инициировать повторную отправку формы.

Чтобы можно было отделить HTML-содержимое от заголовка документа, необходимо создать компоновку, которая упростит изменения, вносимые позже в главе. Создадим папку `Views/Shared` и добавим в нее файл компоновки по имени `_Layout.cshtml` с содержимым, показанным в листинге 20.5.

Листинг 20.5. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RESTful Controllers</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css"
rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @RenderBody()
</body>
</html>
```

Теперь создадим папку Views/Home, добавим в нее файл представления по имени Index.cshtml и поместим в него содержимое из листинга 20.6.

Листинг 20.6. Содержимое файла Index.cshtml из папки Views/Home

```
@model IEnumerable<Reservation>
@{ Layout = "_Layout"; }

<form id="addform" asp-action="AddReservation" method="post">
  <div class="form-group">
    <label for="ClientName">Name:</label>
    <input class="form-control" name="ClientName" />
  </div>
  <div class="form-group">
    <label for="Location">Location:</label>
    <input class="form-control" name="Location" />
  </div>
  <div class="text-center panel-body">
    <button type="submit" class="btn btn-sm btn-primary">Add</button>
  </div>
</form>

<table class="table table-sm table-striped table-bordered m-2">
  <thead><tr><th>ID</th><th>Client</th><th>Location</th></tr></thead>
  <tbody>
    @foreach (var r in Model) {
      <tr>
        <td>@r.ReservationId</td>
        <td>@r.ClientName</td>
        <td>@r.Location</td>
      </tr>
    }
  </tbody>
</table>
```

Это строго типизированное представление получает последовательность объектов Reservation в качестве своей модели и с помощью Razor-цикла foreach заполняет ими таблицу. Предусмотрен также элемент form, который сконфигурирован для отправки запросов POST к действию AddReservation.

Примеры в настоящей главе полагаются на CSS-пакет Bootstrap. Чтобы добавить Bootstrap в проект, создадим в корневой папке проекта файл bower.json с использованием шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел dependencies (листинг 20.7).

Листинг 20.7. Добавление пакета в файле bower.json

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Создадим в папке Views файл _ViewImports.cshtml и настроим в нем встроенные вспомогательные функции дескрипторов для применения в представлениях Razor, а также обеспечим импортирование пространства имен модели (листинг 20.8).

Листинг 20.8. Содержимое файла _ViewImports.cshtml из папки Views

```
@using ApiControllers.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Чтобы включить инфраструктуру MVC и компоненты промежуточного ПО, требующиеся для разработки, внесем в класс Startup изменения, приведенные в листинге 20.9. Здесь также с помощью метода AddSingleton() настраивается отображение службы для хранилища объектов модели.

Листинг 20.9. Содержимое файла Startup.cs из папки ApiControllers

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;

namespace ApiControllers {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IRepository, MemoryRepository>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Установка порта HTTP

Некоторые примеры в главе тестируются путем набора URL вручную. Чтобы облегчить их ввод, понадобится установить порт, который будет получать HTTP-запросы. Выберем пункт ApiControllers Properties (Свойства ApiControllers) в меню Project (Проект) среды Visual Studio, в открывшемся диалоговом окне перейдем на вкладку Debug (Отладка) и изменим значение в поле App URL (URL приложения) на http://localhost:7000/, как показано на рис. 20.1. После установки номера порта сохраним изменения.

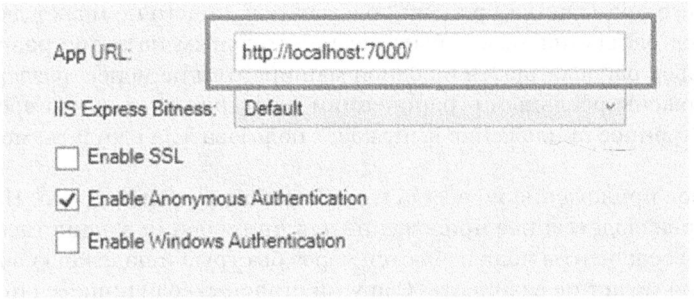


Рис. 20.1. Установка URL приложения

Запустим приложение, заполним форму и щелкнем на кнопке Add (Добавить); приложение добавит в хранилище новый объект `Reservation` (рис. 20.2). Вносимые в хранилище изменения непостоянны и будут утрачиваться при останове или перезапуске приложения.

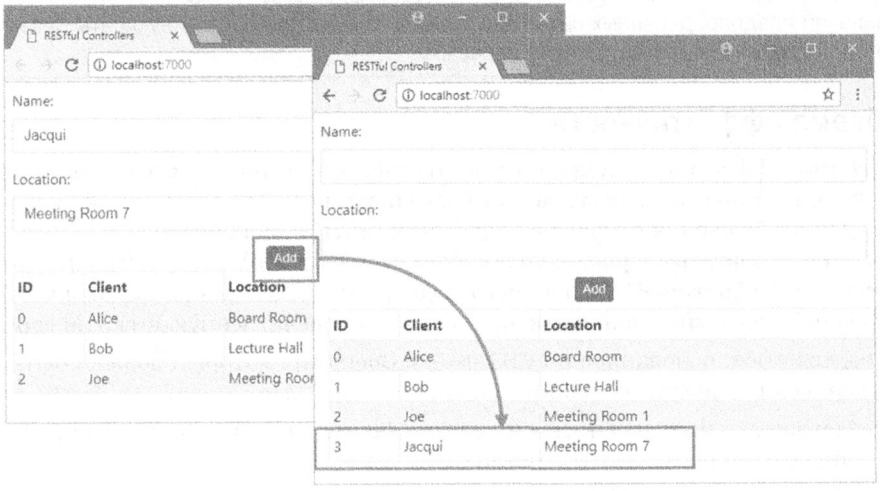


Рис. 20.2. Выполнение примера приложения

Роль контроллеров REST

Рассматриваемое приложение — пример классического веб-приложения. Вся логика приложения находится на сервере и содержится в классах C#, что упрощает ее управление, тестирование и сопровождение. Но приложению, спроектированному подобным образом, могут быть присущи серьезные недостатки в плане скорости, эффективности и открытости.

Проблема скорости

В настоящий момент пример приложения является *синхронным* веб-приложением. После того как пользователь щелкает на кнопке Add, браузер отправляет серверу запрос `POST`, ожидает ответа и затем визуализирует полученную HTML-разметку.

В течение этого промежутка времени пользователь ничего не может делать, а только ждать. Период ожидания может быть незначительным на этапе разработки, когда браузер и сервер располагаются на одной машине; тем не менее, развернутые приложения подвержены реальным ограничениям и задержкам, так что время, в течение которого синхронное приложение вынуждает пользователя ожидать, может оказаться ощутимым.

Синхронное приложение не всегда имеет проблему со скоростью. Например, если вы пишете производственное приложение для применения в единственном местоположении, где все клиенты подключаются через быструю и надежную локальную сеть, тогда проблема может не возникать. С другой стороны, если пишется приложение для мобильных клиентов на площадках со скудной инфраструктурой, то проблема скорости может быть существенной.

Совет. Некоторые браузеры позволяют эмулировать разные типы сетей, что представляет собой удобный инструмент для выяснения, склонны ли пользователи согласиться на работу с синхронным приложением в определенном диапазоне сценариев. Скажем, браузер Google Chrome предлагает средство под названием *сетевое регулирование*, которое доступно в разделе Network (Сеть) инструментов, вызываемых по нажатию <F12>. Доступен диапазон предопределенных сетей или же можно создать собственную сеть, указав скорости выгрузки и загрузки, а также задержку.

Проблема эффективности

Проблема эффективности проистекает из способа, которым синхронное веб-приложение трактует браузер как механизм визуализации HTML-разметки, используемый только для отображения отправляемых сервером HTML-документов.

Когда пользователь впервые запрашивает стандартный URL примера приложения, отправленный обратно HTML-документ содержит все, в чем браузер нуждается для отображения содержимого приложения, включая перечисленную ниже информацию:

- содержимое, основанное на CSS-файле Bootstrap, который должен быть загружен, если не доступна кешированная копия;
- содержимое, имеющее в своем составе форму, которая сконфигурирована для отправки запроса `POST` к действию `AddReservation`;
- содержимое, имеющее в своем составе таблицу, тело которой содержит три заполненные строки.

Пример приложения прост, и начальный запрос приводит к тому, что сервер посылает клиенту около 1,3 Кбайт данных. Однако когда пользователь отправляет форму, клиент перенаправляется снова на действие `Index`, в результате давая еще 1,3 Кбайт данных, чтобы отразить добавление одиночной строки таблицы. Браузер уже визуализировал форму и таблицу, но они были отброшены и заменены полностью новым представлением того, что является практически тем же самым содержимым.

Может показаться, что объем в 1,3 Кбайт данных не особенно велик и, несомненно, вы будете правы. Но если учесть соотношение между полезным и дублированным содержимым, тогда вы увидите, что подавляющее большинство данных, посылаемых браузеру, будет излишним. К тому же пример приложения намеренно упрощен; очень немногие реальные приложения требуют настолько мало HTML-разметки, и по мере возрастания сложности приложения объем дублированного содержимого значительно увеличивается.

Проблема открытости

Последняя проблема традиционного веб-приложения связана с тем, что его проектное решение закрыто, означая возможность доступа к данным в модели только через действия, которые предоставляет контроллер `Home`. Закрытые приложения становятся проблемой, если лежащие в основе данные необходимо использовать в другом приложении, особенно когда такое приложение разрабатывается другой командой или даже другой организацией.

Разработчики зачастую уверены, что ценность приложения выражается взаимодействиями, которые оно предлагает пользователям, в значительной степени из-за того, что они являются теми частями, на обдумывание и реализацию которых разработчики тратят немалое время. Но как только приложение установилось и накопило активную пользовательскую базу, содержащиеся в приложении данные часто становятся более важными.

Введение в REST и контроллеры API

Контроллер API — это контроллер MVC, который отвечает за предоставление доступа к данным в приложении, не инкапсулируя их в HTML-разметку. В результате появляется возможность извлекать или модифицировать данные в модели без необходимости в применении действий, предлагаемых обычными контроллерами, такими как контроллер `Home` в примере приложения.

Самый распространенный подход к доставке данных из приложения предусматривает использование паттерна REST (Representational State Transfer — передача состояния представления). Детальная спецификация REST отсутствует, что привело к появлению многочисленных подходов, позиционирующих себя как согласованные с REST. Тем не менее, существует несколько унифицированных идей, которые полезны в клиентской части разработки веб-приложений.

Основная предпосылка веб-службы REST — принятие характеристик HTTP, так что методы запросов (называемые *командами*) указывают серверу операцию, подлежащую выполнению, а URL запроса определяет один или большее число объектов данных, к которым операция будет применена.

В качестве примера взгляните на URL, который может ссылаться на специфический объект `Reservation` в рассматриваемом приложении:

```
/api/reservations/1
```

Первая часть URL, `api`, используется для отделения части данных приложения от стандартных контроллеров, которые генерируют HTML-разметку. Следующая часть, `reservations`, указывает коллекцию объектов, с которой будет производиться работа. Финальная часть, `1`, задает индивидуальный объект внутри коллекции `reservations`. В примере приложения это значение свойства `ReservationId`, которое уникальным образом идентифицирует объект, и будет применяться в URL.

Идентифицирующие объект URL объединяются с методами HTTP для указания операций. В табл. 20.3 перечислены наиболее часто используемые методы HTTP и описано, что они представляют, когда комбинируются с URL примера. В ней также приведены детали о том, какие данные (*полезная нагрузка*) включаются в запрос и ответ для каждой комбинации метода и URL. Контроллер API, который обрабатывает такие запросы, с помощью кода состояния ответа сообщает об исходе запроса.

Таблица 20.3. Объединение методов HTTP с URL для указания веб-службы REST

Команда	URL	Описание	Полезная нагрузка
GET	/api/reservations	Эта комбинация извлекает все объекты	Ответ содержит полную коллекцию объектов Reservation
GET	/api/reservations/1	Эта комбинация извлекает объект Reservation, свойство ReservationId которого имеет значение 1	Ответ содержит указанный объект Reservation
POST	/api/reservation	Эта комбинация создает новый объект Reservation	Запрос содержит значения для других свойств, которые требуются при создании объекта Reservation. Ответ содержит объект, который был сохранен, гарантируя получение клиентом сохраненных данных
PUT	/api/reservation	Эта комбинация заменяет существующий объект Reservation	Запрос содержит значения, требуемые для изменения свойств указанного объекта Reservation. Ответ содержит объект, который был сохранен, гарантируя получение клиентом сохраненных данных
PATCH	/api/reservation/1	Эта комбинация модифицирует существующий объект Reservation, свойство ReservationId которого имеет значение 1	Запрос содержит набор модификаций, которые должны быть применены к указанному объекту Reservation. Ответом является подтверждение о том, что изменения были применены
DELETE	/api/reservation/1	Эта комбинация удаляет объект Reservation, свойство ReservationId которого имеет значение 1	Полезная нагрузка в запросе или ответе отсутствует

Следование соглашению REST не является требованием, но содействует упрощению работы с приложением, потому что один и тот же подход широко применяется во многих уже установившихся веб-приложениях.

Создание контроллера API

Процесс создания контроллера API основан на подходе, используемом для стандартных контроллеров, с рядом дополнительных средств, которые помогают указывать API-интерфейс, представляемый клиентам. Добавим в папку `Controllers` файл класса по имени `ReservationController.cs` с определением, приведенным в листинге 20.10. Функциональность, предлагаемая контроллером `Reservation`, анализируется в последующих разделах.

Совет. Вспомните, что классы контроллеров могут определяться где угодно в проекте, а не только в папке `Controllers`. В крупных и сложных проектах зачастую удобно определять контроллеры API отдельно от обычных контроллеров HTML и помещать их в какую-нибудь подпапку или даже в полностью отдельную папку.

Листинг 20.10. Содержимое файла `ReservationController.cs` из папки `Controllers`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;
using Microsoft.AspNetCore.JsonPatch;

namespace ApiControllers.Controllers {

  [Route("api/[controller]")]
  public class ReservationController : Controller {
    private IRepository repository;

    public ReservationController(IRepository repo) => repository = repo;

    [HttpGet]
    public IEnumerable<Reservation> Get() => repository.Reservations;

    [HttpGet("{id}")]
    public Reservation Get(int id) => repository[id];

    [HttpPost]
    public Reservation Post([FromBody] Reservation res) =>
      repository.AddReservation(new Reservation {
        ClientName = res.ClientName,
        Location = res.Location
      });

    [HttpPut]
    public Reservation Put([FromBody] Reservation res) =>
      repository.UpdateReservation(res);

    [HttpPatch("{id}")]
    public StatusCodeResult Patch(int id,
        [FromBody]JsonPatchDocument<Reservation> patch) {
      Reservation res = Get(id);
      if (res != null) {
        patch.ApplyTo(res);
        return Ok();
      }
      return NotFound();
    }
```

```
    [HttpDelete("{id}")]
    public void Delete(int id) => repository.DeleteReservation(id);
  }
}
```

Контроллеры API работают в той же самой манере, что и обычные контроллеры, т.е. можно создать контроллер POCO или унаследовать класс от базового класса `Controller`, который предоставляет более удобный доступ к данным контекста запроса.

Адаптация паттерна REST

Паттерн REST поощряет определенную долю догматизма относительно того, каким образом API-интерфейсы веб-приложения должны быть представлены клиентам. Паттерн REST не является стандартным или хотя бы четко определенным, и существует несколько полезных подходов, которые облегчают адаптацию REST в приложениях ASP.NET Core MVC, но имеют тенденцию к крушению планов у тех программистов, кто обладает установившимися представлениями о том, что считается соответствующим REST.

В табл. 20.3 примеры URL, перечисленные для операций `POST` и `PUT`, не идентифицируют ресурс уникальным образом, что некоторые считают неотъемлемой характеристикой REST. В случае операции `POST` уникальный идентификатор объекта `Reservation` назначается моделью, т.е. клиент не может предоставить его как часть URL. В случае операции `PUT` средство привязки моделей MVC (описанное в главе 26 и являющееся причиной применения атрибута `FromBody` в листинге 20.10) упрощает получение из тела запроса деталей объекта `Reservation`, который подлежит модификации. Таким образом, именно здесь контроллер `Reservation` ожидает найти значение `ReservationId`, идентифицирующее объект модели, который должен быть модифицирован.

Подобно всем паттернам REST представляет собой отправную точку, предлагающую удобные и полезные идеи. Это не жесткий стандарт, который должен соблюдаться любой ценой; единственная важная цель в том, чтобы писать понятный, тестируемый и сопровождаемый код. Учет природы приложений MVC и проектного решения, положенного в основу хранилища, способствует получению более простого приложения и по-прежнему предоставляет практичный API-интерфейс для потребления клиентами. Я советую рассматривать паттерны как руководящие принципы, которые вы адаптируете под собственные нужды, что справедливо в отношении как REST, так и MVC в целом.

Определение маршрута

Маршрут, посредством которого достигаются контроллеры API, может определяться только с использованием атрибута `Route`, но не конфигурации приложения в классе `Startup`. По соглашению для контроллеров API применяется маршрут, предваренный префиксом `api`, за которым следует имя контроллера, так что контроллер `ReservationController` из листинга 20.10 достигается через URL вида `/api/reservation`:

```
...
[Route("api/[controller]")]
public class ReservationController : Controller {
...
```

Объявление зависимостей

Объекты контроллеров API создаются так же, как и объекты обычных контроллеров, а это значит, что они могут объявлять зависимости, которые будут распознаваться с использованием поставщика служб. Класс `ReservationController` объявляет в конструкторе зависимость от интерфейса `IRepository`, которая будет распознаваться для предоставления доступа к данным в модели:

```
...
public ReservationController(IRepository repo) => repository = repo;
...
```

Определение методов действий

Каждый метод действия декорируется атрибутом, указывающим метод HTTP, который он принимает, например:

```
...
[HttpGet]
public IEnumerable<Reservation> Get() => repository.Reservations;
...
```

Атрибут `HttpGet` входит в набор атрибутов, применяемых для ограничения доступа к методам действий запросами, которые имеют специфический метод или команду HTTP. Полный набор атрибутов представлен в табл. 20.4.

Таблица 20.4. Атрибуты методов HTTP

Имя	Описание
`HttpGet`	Этот атрибут указывает, что действие может быть вызвано только HTTP-запросами, которые используют команду `GET`
`HttpPost`	Этот атрибут указывает, что действие может быть вызвано только HTTP-запросами, которые применяют команду `POST`
`HttpDelete`	Этот атрибут указывает, что действие может быть вызвано только HTTP-запросами, которые используют команду `DELETE`
`HttpPut`	Этот атрибут указывает, что действие может быть вызвано только HTTP-запросами, которые применяют команду `PUT`
`HttpPatch`	Этот атрибут указывает, что действие может быть вызвано только HTTP-запросами, которые используют команду `PATCH`
`HttpHead`	Этот атрибут указывает, что действие может быть вызвано только HTTP-запросами, которые применяют команду `HEAD`
`AcceptVerbs`	Этот атрибут используется для указания множества команд HTTP

Маршруты могут дополнительно конкретизироваться за счет включения фрагмента маршрутизации в качестве аргумента для атрибута метода HTTP:

```
...
[HttpGet("{id}")]
public Reservation Get(int id) => repository[id];
...
```

Фрагмент маршрутизации, `{id}`, объединяется с маршрутом, определенным в атрибуте `Route`, который применен к контроллеру, и ограничением, основанным на методе HTTP. В данном случае действие может быть достигнуто путем отправки запроса

GET с URL, соответствующим шаблону маршрутизации /api/reservations/{id}, где сегмент id затем используется для идентификации объекта Reservation, подлежащего извлечению.

Обратите внимание, что маршруты, генерируемые для контроллера API, не включают переменную сегмента {action}, т.е. имя метода действия не является частью URL, требующейся для нацеливания на специфический метод. Все действия в контроллере API достигаются через один и тот же базовый URL (/api/reservation в рассматриваемом примере), а метод HTTP и дополнительные сегменты применяются для их различения.

Определение результатов действий

При предоставлении результатов методы действий для контроллеров API не полагаются на объекты ViewResult, поскольку доставка данных не требует каких-либо представлений. Взамен методы действий контроллера API возвращают объекты данных:

```
...
[HttpGet]
public IEnumerable<Reservation> Get() => repository.Reservations;
...
```

Приведенное действие возвращает последовательность объектов Reservation и возлагает на инфраструктуру MVC ответственность за их сериализацию в формат, который может быть обработан клиентом. В разделе "Форматирование содержимого" далее в главе данный процесс объясняется более подробно.

Настройка результатов, возвращаемых действиями контроллеров API

Один из наиболее привлекательных аспектов контроллеров API заключается в том, что вы можете просто возвращать из методов действий объекты C# и позволить инфраструктуре MVC самостоятельно выяснить, что с ними делать. Инфраструктура MVC справляется с такой задачей довольно хорошо. Например, если вы возвратите из метода действия контроллера API значение null, то клиенту будет отправлен ответ 204 - No Content (204 — содержимое отсутствует).

Но контроллеры API способны использовать также и средства, доступные обычным контроллерам. Это означает, что стандартное поведение можно переопределить, возвращая из методов действий тип IActionResult, который указывает вид результата для отправки. Ниже показана реализация метода действия из примера контроллера, отправляющая ответ 404 - Not Found (404 — не найдено) для запросов, которые не соответствуют каким-либо объектам в модели:

```
...
[HttpGet("{id}")]
public IActionResult Get(int id) {
  Reservation result = repository[id];
  if (result == null) {
    return NotFound();
  } else {
    return Ok(result);
  }
}
...
```

Если в хранилище не обнаруживается объект с указанным идентификатором, тогда вызывается метод `NotFound()`, создающий объект `NotFoundResult`, который приводит к отправке клиенту ответа `404 - Not Found`. Если объект в хранилище найден, тогда вызывается метод `Ok()`, чтобы создать объект `ObjectResult`. Метод `Ok()` позволяет отправлять объект клиенту внутри действия, которое возвращает `IActionResult`, как было описано в главе 17. Потребность в переопределении стандартных ответов, возвращаемых действиями контроллеров API, возникает нечасто, но если она все-таки появляется, то для таких ситуаций доступен полный диапазон результатов действий.

Тестирование контроллера API

Существует множество инструментов, которые помогают тестировать API-интерфейсы веб-приложения. В числе хороших примеров можно назвать Fiddler (`https://www.telerik.com/fiddler`), представляющий собой автономный инструмент для отладки HTTP-запросов, и Swashbuckle (`https://github.com/domaindrivendev/Swashbuckle`), являющийся пакетом NuGet, который добавляет в приложение страницу сводки с описаниями его операций API и позволяет их тестировать.

Но самый простой способ удостовериться в работоспособности контроллера API предусматривает применение инструмента PowerShell, который облегчает создание HTTP-запросов в командной строке и позволяет сосредоточиться на результатах операций API, не погружаясь в детали. Инструмент PowerShell происходит из Windows, но теперь доступен также для Linux и macOS.

В последующих разделах будет показано, как использовать PowerShell для тестирования операций, предоставляемых контроллером `Reservation`. Для выполнения тестовых команд можно открыть новое окно PowerShell или работать в окне консоли диспетчера пакетов Visual Studio, которое само взаимодействует с PowerShell.

Тестирование операций GET

Чтобы протестировать операцию `GET`, предлагаемую контроллером `Reservation`, запустим приложение, выбрав пункт Start Without Debugging (Запустить без отладки) в меню Debug (Отладка) среды Visual Studio, и подождем появления синхронного ответа, предоставляемого контроллером `Home`. Пока приложение выполняется, откроем окно PowerShell и наберем такую команду:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

Эта команда применяет командлет PowerShell по имени `Invoke-RestMethod` для отправки запроса `GET` на URL вида `/api/reservation`. Результат разбирается и форматируется для удобства восприятия данных:

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Lecture Hall
            2 Joe        Meeting Room 1
```

Сервер отвечает на запрос `GET` представлением JSON объектов `Reservation`, содержащихся в модели, которое командлет `Invoke-RestMethod` выводит в табличном формате.

Формат JSON

Формат *JSON* (JavaScript Object Notation — система обозначений для объектов JavaScript) стал стандартным форматом данных для веб-приложений. Формат JSON популярен благодаря простоте, лаконичности и легкости в работе с ним. Данные JSON особенно легко обрабатывать в коде JavaScript, потому что формат JSON подобен способу выражения литеральных объектов на языке JavaScript. Современные браузеры включают встроенную поддержку для генерации и разбора данных JSON, а популярные библиотеки JavaScript, такие как jQuery, будут автоматически преобразовывать в и из формата JSON.

Хотя формат JSON эволюционировал из JavaScript, его структура легка в чтении и понимании для разработчиков на языке C#. Вот как выглядит ответ из контроллера API в примере приложения:

```
...
[{"reservationId":0,"clientName":"Alice","location":"Board Room"},
 {"reservationId":1,"clientName":"Bob","location":"Lecture Hall"},
 {"reservationId":2,"clientName":"Joe","location":"Meeting Room 1"}]
...
```

Здесь описан массив объектов. Массив обозначается символами `[` и `]`, а каждый объект — символами `{` и `}`. Объекты являются коллекциями пар "ключ-значение", где каждый ключ отделяется от своего значения с помощью двоеточия (`:`), а пары разделяются запятыми (`,`). Это отдаленно напоминает синтаксис литералов C#, который использовался в классе `MemoryRepository` для определения данных в листинге 20.3:

```
...
new List<Reservation> {
  new Reservation { ClientName = "Alice", Location = "Board Room" },
  new Reservation { ClientName = "Bob", Location = "Lecture Hall" },
  new Reservation { ClientName = "Joe", Location = "Meeting Room 1" }
...
```

Однако обратите внимание, что MVC заменяет соглашение C# по записи имен свойств, начиная с заглавной буквы (например, `ClientName`), соглашением JavaScript (`clientName` начинается со строчной буквы).

Несмотря на то что форматы не идентичны, имеющегося подобия вполне достаточно для того, чтобы разработчик C# мог читать и понимать данные JSON, не прикладывая больших усилий. В большинстве веб-приложений вам не придется погружаться в детали JSON, т.к. всю рутинную работу берет на себя инфраструктура MVC, но вы можете почитать о формате JSON на веб-сайте `www.json.org`.

Контроллер `Reservation` предоставляет две операции `GET`. Когда запрос `GET` отправляется на URL вида `/api/reservation`, возвращается ответ, содержащий все объекты. Чтобы извлечь одиночный объект, его значение `ReservationId` указывается как финальный сегмент в URL, например:

```
Invoke-RestMethod http://localhost:7000/api/reservation/1 -Method GET
```

Команда запрашивает объект `Reservation` со значением `1` в свойстве `ReservationId` и выводит следующий результат:

```
reservationId clientName location
------------- ---------- --------
            1 Bob        Lecture Hall
```

Тестирование операции POST

Все операции, предоставляемые контроллером API, могут быть протестированы с применением PowerShell, хотя формат команд может оказаться слегка неудобным. Вот как выглядит команда, которая отправляет контроллеру API запрос POST для создания нового объекта Reservation в хранилище и записывает полученные данные в ответ:

```
Invoke-RestMethod http://localhost:7000/api/reservation
      -Method POST -Body
(@{clientName="Anne"; location="Meeting Room 4"} |
  ConvertTo-Json) -ContentType "application/json"
```

В приведенной команде с помощью аргумента -Body указывается тело запроса, кодируемое как JSON. Аргумент -ContentType используется для установки заголовка Content-Type в запросе. Команда даст следующий результат:

```
reservationId clientName location
------------- ---------- --------
            3 Anne       Meeting Room 4
```

Операция POST применяет значения clientName и location, чтобы создать объект Reservation и возвратить клиенту представление JSON нового объекта, которое включает присвоенное ему значение ReservationId. Может показаться, что клиент просто получает значения данных, которые он отправил серверу в запросе. Тем не менее, такой подход гарантирует, что клиент работает с теми же самыми данными, которые использует сервер, и учитывает любое форматирование или трансляции, выполняемые сервером в отношении полученных от клиента данных. Чтобы взглянуть на эффект от запроса POST, отправим еще один запрос GET на URL вида /api/reservation:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

Возвращенные данные отражают добавление нового объекта Reservation:

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Lecture Hall
            2 Joe        Meeting Room 1
            3 Anne       Meeting Room 4
```

Тестирование операции PUT

Метод PUT применяется для замены существующих объектов в модели. Значение ReservationId объекта указывается как часть URL запроса, а значения clientName и location предоставляются в теле запроса. Вот команда PowerShell, которая отправляет запрос PUT для замены объекта Reservation:

```
Invoke-RestMethod http://localhost:7000/api/reservation
   -Method PUT -Body
(@{reservationId="1"; clientName="Bob"; location="Media Room"} |
  ConvertTo-Json) -ContentType "application/json"
```

Запрос изменяет объект Reservation со значением 1 в свойстве ReservationId и указывает новое значение для свойства Location. Выполнив команду, вы увидите следующий ответ, который отражает внесенное изменение:

```
reservationId clientName location
------------- ---------- --------
            1 Bob        Media Room
```

Чтобы взглянуть на эффект от запроса POST, отправим запрос GET на URL вида /api/reservation:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

Возвращенные данные содержат замененный объект Reservation:

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            2 Joe        Meeting Room 1
            3 Anne       Meeting Room 4
```

Тестирование операции Patch

Метод PATCH используется для модификации существующего объекта в модели. Многие приложения применяют запросы PUT и вообще игнорируют запросы PATCH, что является разумным подходом, если клиенты имеют доступ ко всем свойствам, определенным объектами в модели. Но в сложных приложениях по причинам, связанным с безопасностью, клиенты могут получать специфический набор значений свойств, что препятствует отправке ими полного объекта как части запроса PUT. Запросы PATCH обладают большей избирательностью, позволяя клиентам указывать набор детализированных изменений, которые необходимо внести в объект.

В ASP.NET Core MVC имеется поддержка работы со стандартом JSON Patch, который делает возможным указание изменений в унифицированной манере. Я не собираюсь погружаться в детали стандарта JSON Patch, о котором можно почитать по ссылке https://tools.ietf.org/html/rfc6902, но в примере приложения клиент будет отправлять контроллеру API в HTTP-запросе PATCH данные JSON следующего вида:

```
[
 { "op": "replace", "path": "clientName", "value": "Bob"},
 { "op": "replace", "path": "location", "value": "Lecture Hall"}
]
```

Документ JSON Patch выражается как массив операций. Каждая операция имеет свойство op, указывающее тип операции, и свойство path, которое определяет, где операция будет применена.

В примере приложения — и фактически в большинстве приложений — требуется только операция replace, которая используется для изменения значения свойства. Показанные выше данные JSON Patch устанавливают новые значения для свойств clientName и location, в то время как подлежащий модификации объект будет идентифицироваться URL запроса. Инфраструктура ASP.NET Core MVC автоматически обработает данные JSON и предоставит их методу действия в виде объекта JsonPatchDocument<T>, где T — тип модифицируемого объекта модели. Объект JsonPatchDocument<T> затем может применяться для модификации объекта их хранилища с использованием метода ApplyTo(). Вот команда PowerShell, которая посылает запрос PATCH:

```
Invoke-RestMethod http://localhost:7000/api/reservation/2
      -Method PATCH -Body
(@{ op="replace"; path="clientName";
    value="Bob"},@{ op="replace"; path="location";
    value="Lecture Hall"} | ConvertTo-Json)
    -ContentType "application/json"
```

Показанный запрос предлагает серверу модифицировать свойства `clientName` и `location` объекта `Reservation`, у которого значение `ReservationId` равно 2. Чтобы увидеть эффект от запроса `PATCH`, отправим запрос `GET` на `/api/reservation`:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

Данные, возвращаемые клиенту, отражают обновленный объект `Reservation`:

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            2 Bob        Lecture Hall
            3 Anne       Meeting Room 4
```

Тестирование операции `Delete`

Последний тест предусматривает отправку запроса `DELETE`, который удалит объект `Reservation` из хранилища:

```
Invoke-RestMethod http://localhost:7000/api/reservation/2
      -Method DELETE
```

Действие, которое принимает запросы `DELETE` в контроллере `Reservation`, результат не возвращает, поэтому после завершения команды какие-либо данные не отображаются. Чтобы просмотреть эффект от удаления, запросим содержимое хранилища с применением такой команды:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

Объект `Reservation` со значением 2 в свойстве `ReservationId` был удален из хранилища:

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            3 Anne       Meeting Room 4
```

Использование контроллера API в браузере

Определение контроллера API решает проблему открытости в приложении, но ничего не делает в плане решения проблем скорости и эффективности. Для этого понадобится обновить часть HTML приложения, чтобы при отправке HTTP-запросов к контроллеру API с целью выполнения операций над данными она опиралась на JavaScript.

В браузере асинхронные HTTP-запросы обычно известны как *запросы Ajax*, где Ajax представляет собой аббревиатуру для *Asynchronous JavaScript and XML* (асинхронный JavaScript и XML). В последние годы формат данных XML подрастерял свою популярность, но название Ajax по-прежнему применяется для ссылки на асинхронные HTTP-

запросы, даже когда они возвращают данные JSON. Выражаясь более широко, описанный в настоящем разделе прием является основой одностраничных приложений, где код JavaScript внутри единственной HTML-страницы используется для извлечения данных, которые предназначены множеству разделов приложения, с динамической генерацией содержимого, подлежащего отображению.

На заметку! Разработка клиентской стороны является отдельной темой и выходит за рамки настоящей книги. Здесь мы создадим только базовый асинхронный HTTP-запрос без подробных объяснений, просто чтобы дать вам представление о том, как это делается. В моих книгах *Pro Angular* и *Essential Angular for ASP.NET Core MVC*, опубликованных издательством Apress, подробно раскрывается разработка клиентской стороны с помощью инфраструктуры Angular и поддержка клиентов Angular с применением ASP.NET Core MVC.

Браузеры предоставляют API-интерфейс JavaScript для выполнения запросов Ajax, но работать с ним несколько неудобно, и существуют отличия в том, как браузеры реализуют некоторые дополнительные средства. Делать запросы Ajax проще всего с использованием библиотеки jQuery, которая является бесконечно полезным инструментом для разработки клиентской стороны. В листинге 20.11 демонстрируется добавление пакета jQuery в файле `bower.json`.

Листинг 20.11. Добавление пакета jQuery в файле `bower.json` из папки `ApiControllers`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

На самом деле, поскольку средства Bootstrap зависят от jQuery, инструмент Bower уже установил пакет в папку `wwwroot/lib`. Добавление в листинге 20.11 просто делает такую зависимость явной. Чтобы задействовать средства, предлагаемые jQuery, создадим папку `wwwroot/js` и добавим в нее файл по имени `client.js` с содержимым из листинга 20.12.

Листинг 20.12. Содержимое файла `client.js` из папки `wwwroot/js`

```
$(document).ready(function () {
  $("form").submit(function (e) {
    e.preventDefault();
    $.ajax({
      url: "api/reservation",
      contentType: "application/json",
      method: "POST",
      data: JSON.stringify({
        clientName: this.elements["ClientName"].value,
        location: this.elements["Location"].value
      }),
```

```
      success: function(data) {
        addTableRow(data);
      }
    })
  });
});

var addTableRow = function (reservation) {
    $("table tbody").append("<tr><td>" + reservation.reservationId
      + "</td><td>"
      + reservation.clientName + "</td><td>"
      + reservation.location + "</td></tr>");
}
```

Когда пользователь отправляет форму в браузере, файл JavaScript создает ответ, кодирует данные формы как JSON и отправляет их серверу с применением HTTP-запроса `POST`. Данные JSON, возвращаемые сервером, автоматически разбираются jQuery и затем используются для добавления строки в HTML-таблицу.

В листинге 20.13 обновлена компоновка с целью включения элементов `script` для библиотеки jQuery и файла `client.js`.

Листинг 20.13. Добавление ссылок на файлы JavaScript в файле `_Layout.cshtml` из папки `Views`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RESTful Controllers</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css"
        rel="stylesheet" />
  <script src="lib/jquery/dist/jquery.js"></script>
  <script src="js/client.js"></script>
</head>
<body class="m-1 p-1">
  @RenderBody()
</body>
</html>
```

Первый элемент `script` сообщает браузеру о необходимости загрузки библиотеки jQuery, а второй элемент `script` указывает файл, который будет содержать специальный код. Запустив приложение и создав с помощью HTML-формы объект `Reservation` в хранилище приложения, вы не заметите каких-либо визуальных отличий, но если вы просмотрите HTTP-запрос, отправленный браузером, то увидите, что он требует намного меньше данных, чем было в синхронной версии приложения. При простом тестировании асинхронный запрос потребовал 440 байтов данных, что составляет около 40% объема, необходимого в случае синхронного запроса. В реальных приложениях улучшения будут более существенными, т.к. размер данных имеет тенденцию быть гораздо меньше размера HTML-документа, который применяется для их отображения.

Форматирование содержимого

Когда метод действия возвращает в качестве своего результата объект C#, инфраструктура MVC обязана выяснить, какой формат данных должен использоваться для кодирования объекта и его отправки клиенту. В настоящем разделе объясняется стандартный процесс, а также влияние на него запроса, отправляемого клиентом, и конфигурации приложения. Для содействия прояснению работы процесса добавим в папку `Controllers` файл класса по имени `ContentController.cs` и определим в нем контроллер, приведенный в листинге 20.14.

Листинг 20.14. Содержимое файла `ContentController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

  [Route("api/[controller]")]
  public class ContentController : Controller {

    [HttpGet("string")]
    public string GetString() => "This is a string response";

    [HttpGet("object")]
    public Reservation GetObject() => new Reservation {
      ReservationId = 100,
      ClientName = "Joe",
      Location = "Board Room"
    };
  }
}
```

Для двух действий в контроллере `Content` указаны статические переменные сегментов как аргументы атрибута `HttpGet`, что означает возможность достичь их посредством URL вида `/api/controller/string` и `/api/controller/object`. Контроллер `Content` даже близко не следует паттерну REST, но он облегчит понимание, каким образом работает согласование содержимого.

Формат содержимого, выбираемый MVC, зависит от четырех факторов: форматов, которые клиент будет принимать, форматов, которые MVC может выпускать, политики содержимого, указанной действием, и типа, возвращаемого методом действия. Процесс выяснения, каким образом все сочетается друг с другом, может выглядеть пугающим, но хорошая новость в том, что стандартная политика прекрасно подходит для большинства приложений. Вы должны понимать, что происходит "за кулисами", только при необходимости внесения изменений или в случае, если результаты получаются не в том формате, который ожидался.

Стандартная политика содержимого

Отправной точкой является стандартная конфигурация приложения, которая применяется, когда ни клиент, ни метод действия не накладывает ограничения на форматы, которые можно использовать. В такой ситуации исход прост и предсказуем.

- Если метод действия возвращает объект `string`, то он отправляется клиенту неизмененным, а заголовок `Content-Type` ответа устанавливается в `text/plain`.

- Для всех остальных типов данных, включая прочие простые типы вроде `int`, данные форматируются как JSON, а заголовок `Content-Type` ответа устанавливается в `application/json`.

Строки получают специальную интерпретацию из-за того, что могут вызвать проблемы при кодировании в JSON. В результате кодирования других простых типов, таких как значение `2` типа `int` языка C#, получается строка в кавычках, подобная `"2"`. Когда кодируется строка, итогом будет два набора кавычек, так что `"Hello"` становится `""Hello""`. Не все клиенты хорошо стравляются с таким кодированием, поэтому надежнее применять формат `text/plain` и полностью обойти проблему. Проблема возникает редко, потому что лишь немногие приложения отправляют значения `string`; гораздо чаще посылаются объекты в формате JSON. Оба исхода можно просмотреть с использованием PowerShell. Вот команда, которая вызывает метод `GetString()`, возвращающий значение `string`:

```
Invoke-WebRequest http://localhost:7000/api/content/string | select
@{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

Данная команда отправляет запрос `GET` на URL вида `/api/content/string` и обрабатывает ответ для отображения заголовка `Content-Type` и содержимого ответа.

Совет. Во время применения командлета `Invoke-WebRequest` может быть получена ошибка, если не была выполнена начальная настройка для Internet Explorer. Это особенно вероятно на машине Windows 10, где Internet Explorer заменен браузером Edge. Проблему легко устранить, запустив IE и выбрав требуемую начальную конфигурацию.

Команда выдает следующий вывод:

```
Content-Type                Content
------------                -------
text/plain; charset=utf-8   This is a string response
```

Ту же самую команду можно также использовать для отображения результата в формате JSON, изменив лишь запрашиваемый URL:

```
Invoke-WebRequest http://localhost:7000/api/content/object | select
@{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

Команда генерирует вывод (отформатированный для ясности), который показывает, что ответ был закодирован в JSON:

```
Content-Type                     Content
------------                     -------
application/json; charset=utf-8  {"reservationId":100,
                                  "clientName":"Joe",
                                  "location":"Board Room"}
```

Согласование содержимого

Большинство клиентов будут включать в запрос заголовок `Accept`, указывающий набор форматов, которые они готовы получать в ответе, выраженный в виде множества типов MIME. Ниже приведен заголовок `Accept`, который посылает в запросах браузер Google Chrome:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
  image/webp,*/*;q=0.8
```

Заголовок указывает, что Chrome может обрабатывать форматы HTML и XHTML (XHTML — это совместимый с XML диалект HTML), XML, а также формат изображений WEBP. Значения q в заголовке задают относительные предпочтения, где значение 1.0 является стандартным. Указание значения q, равного 0.9, для application/xml сообщает серверу о том, что Chrome будет принимать данные XML, но предпочитает иметь дело с HTML или XHTML. Последний элемент, */*, уведомляет сервер о том, что Chrome будет принимать любой формат, но его значение q указывает наименьшее предпочтение среди перечисленных типов. В итоге заголовок Accept, отправляемый браузером Chrome, снабжает сервер следующей информацией.

1. Браузер Chrome предпочитает получать данные HTML или XHTML либо изображения WEBP.
2. Если упомянутые форматы не доступны, тогда следующим наиболее предпочтительным форматом является XML.
3. Если ни один из предпочтительных форматов не доступен, то Chrome будет принимать любой формат.

Вы можете предположить, что для изменения формата запроса, полученного от приложения MVC, достаточно установить заголовок Accept, но он не работает подобным образом — точнее он не работает подобным образом из-за того, что требуется определенная подготовка. Для начала вот команда PowerShell, посылающая GET методу GetObject() запрос с заголовком Accept, в котором указано, что клиент будет принимать только данные XML:

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers
@{Accept="application/xml"} | select @{n='Content-Type';
e={ $_.Headers."Content-Type" }}, Content
```

Ниже показаны результаты, где видно, что сервер отправил ответ application/json:

```
Content-Type                          Content
------------                          -------
application/json; charset=utf-8       {"reservationId":100,
                                       "clientName":"Joe",
                                       "location":"Board Room"}
```

Включение заголовка Accept никак не влияет на формат, хотя сервер отправил клиенту ответ в формате, который не был указан. Проблема в том, что по умолчанию инфраструктура MVC сконфигурирована для поддержки только формата JSON, поэтому нет других форматов, которые можно было бы применять. Вместо возвращения ошибки инфраструктура MVC посылает данные JSON в надежде на то, что клиент сумеет их обработать, даже при отсутствии такого формата в числе указанных в заголовке Accept запроса.

Конфигурирование сериализатора JSON

Для сериализации объектов в формат JSON инфраструктура ASP.NET Core MVC использует популярный сторонний пакет JSON под названием Json.Net. Стандартная конфигурация подходит в большинстве проектов, но может быть изменена, если данные JSON необходимо создавать специфическим образом. Расширяющий метод AddMvc().AddJsonOptions(), вызываемый внутри класса Startup, позволяет получить доступ к объекту MvcJsonOptions, посредством которого конфигурируется пакет Json.Net. Описание доступных конфигурационных параметров ищите по ссылке https://www.newtonsoft.com/json.

Включение форматирования XML

Чтобы увидеть согласование содержимого в работе, понадобится предоставить инфраструктуре MVC определенный выбор в форматах, которые она применяет для кодирования данных ответа. Несмотря на то что JSON стал стандартным форматом для веб-приложений, инфраструктура MVC способна также поддерживать кодирование данных как XML (листинг 20.15).

Совет. Можно создать собственный формат содержимого, унаследовав его от класса `Microsoft.AspNetCore.Mvc.Formatters.OutputFormatter`. Это редко приносит пользу, потому что создание специального формата данных не является удобным способом открытия доступа к данным в приложении, а самые распространенные форматы — JSON и XML — уже реализованы.

Листинг 20.15. Включение форматирования XML в файле `Startup.cs` из папки `ApiControllers`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;

namespace ApiControllers {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IRepository, MemoryRepository>();
      services.AddMvc().AddXmlDataContractSerializerFormatters();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Когда инфраструктуре MVC доступен только формат JSON, у нее нет никакого выбора кроме кодирования ответов как JSON. Теперь, когда выбор есть, можно понаблюдать за более полной работой процесса согласования содержимого.

Совет. В листинге 20.15 применялся расширяющий метод `AddXmlDataContractSerializerFormatters()`, но можно также использовать расширяющий метод `AddXmlSerializerFormatters()`, который предоставляет доступ к более старому классу сериализации. Он может быть полезен при необходимости генерации XML-содержимого для более старых клиентов .NET.

Вот команда PowerShell, которая снова запрашивает данные XML:

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers
@{Accept="application/xml"} | select @{n='Content-Type';
e={ $_.Headers."Content-Type" }}, Content
```

Запуск команды позволит увидеть, что теперь сервер возвращает данные XML, а не JSON (для краткости атрибуты пространств имен XML опущены):

```
Content-Type                    Content
------------                    -------
application/xml; charset=utf-8 <Reservation>
                                  <ClientName>Joe</ClientName>
                                  <Location>Board Room</Location>
                                  <ReservationId>100</ReservationId>
                               </Reservation>
```

Указание формата данных для действия

Систему согласования содержимого можно переопределить и указать формат данных прямо на методе действия, применяя атрибут `Produces` (листинг 20.16).

Листинг 20.16. Указание формата данных в файле `ContentController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

  [Route("api/[controller]")]
  public class ContentController : Controller {

    [HttpGet("string")]
    public string GetString() => "This is a string response";

    [HttpGet("object")]
    [Produces("application/json")]
    public Reservation GetObject() => new Reservation {
      ReservationId = 100,
      ClientName = "Joe",
      Location = "Board Room"
    };
  }
}
```

Атрибут `Produces` — это фильтр, изменяющий тип содержимого объектов `ObjectResult`, которые используются инфраструктурой MVC "за кулисами" для представления результатов действий в контроллерах API. В аргументе атрибута указывается формат, который будет применяться для результата, возвращаемого действием, и допускается также указание дополнительных разрешенных типов. Атрибут `Produces` принудительно устанавливает формат, используемый ответом, который можно просмотреть с помощью следующей команды PowerShell:

```
(Invoke-WebRequest http://localhost:7000/api/content/object -Headers
@{Accept="application/xml"}).Headers."Content-Type"
```

Команда отображает значение заголовка Content-Type из ответа на запрос GET к URL вида /api/content/object. Выполнение команды показывает, что применяется JSON, как указано в атрибуте Produces, хотя заголовок Accept запроса определяет, что должен использоваться XML.

Получение формата данных из маршрута или строки запроса

Заголовок Accept не всегда находится под контролем программиста, пишущего код клиента, особенно если разработка проводится с применением старого браузера либо инструментального набора. В таких ситуациях может быть удобно разрешить указание формата данных ответа через маршрут, используемый для нацеливания на метод действия, или в разделе строки запроса в URL. Первым делом в классе Startup нужно определить сокращенные значения, которые могут применяться для ссылки на форматы в маршруте или в строке запроса. По умолчанию имеется одно отображение, в котором json используется в качестве сокращения для application/json. В листинге 20.17 добавляется дополнительное отображение для XML.

Листинг 20.17. Добавление сокращения формата в файле Startup.cs из папки ApiControllers

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;
using Microsoft.Net.Http.Headers;

namespace ApiControllers {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IRepository, MemoryRepository>();
      services.AddMvc()
        .AddXmlDataContractSerializerFormatters()
        .AddMvcOptions(opts => {
          opts.FormatterMappings.SetMediaTypeMappingForFormat("xml",
            new MediaTypeHeaderValue("application/xml"));
        });
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Свойство `MvcOptions.FormatterMappings` служит для установки и управления отображениями.

В листинге 20.17 с использованием метода `SetMediaTypeMappingForFormat()` создается новое отображение, так что сокращение `xml` будет ссылаться на формат `application/xml`. Далее понадобится применить атрибут `FormatFilter` к методу действия и дополнительно подкорректировать маршрут для действия, чтобы он включал переменную сегмента `format` (листинг 20.18).

Листинг 20.18. Использование атрибута `FormatFilter` в файле `ContentController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

  [Route("api/[controller]")]
  public class ContentController : Controller {

    [HttpGet("string")]
    public string GetString() => "This is a string response";

    [HttpGet("object/{format?}")]
    [FormatFilter]
    [Produces("application/json", "application/xml")]
    public Reservation GetObject() => new Reservation {
      ReservationId = 100,
      ClientName = "Joe",
      Location = "Board Room"
    };
  }
}
```

К методу `GetObject()` был применен атрибут `FormatFilter`, а маршрут для данного действия модифицирован так, что он включает необязательный сегмент `format`. Вы не обязаны использовать атрибут `Produces` в сочетании с атрибутом `FormatFilter`, но если сделаете это, тогда будут работать только запросы, указывающие форматы, для которых был сконфигурирован атрибут `Produces`. Запросы, указывающие формат, для которого атрибут `Produces` не был сконфигурирован, получат ответ `404 - Not Found`. Если вы не применяли атрибут `Produces`, то запрос может задавать любой формат, который инфраструктура MVC сконфигурировала для использования.

Кроме того, к атрибуту `Produces` добавлен формат `application/xml`, чтобы метод действия поддерживал запросы JSON и XML.

В следующей команде PowerShell формат `xml` указывается как часть URL запроса:

```
(Invoke-WebRequest
  http://localhost:7000/api/content/object/xml).Headers."Content-Type"
```

Запуск команды приводит к отображению типа содержимого ответа:

```
application/xml; charset=utf-8
```

Атрибут `FormatFilter` ищет переменную сегмента маршрутизации по имени `format`, получает содержащееся в ней сокращенное значение и извлекает ассоции-

рованный формат данных из конфигурации приложения. Затем этот формат применяется для ответа. Если данные маршрутизации отсутствуют, тогда инспектируется строка запроса. Вот команда PowerShell, которая требует использования формата XML посредством строки запроса:

```
(Invoke-WebRequest http://localhost:7000/api/content/object?format=xml).
Headers."Content-Type"
```

Формат, найденный атрибутом `FormatFilter`, переопределяет любые форматы, указанные в заголовке `Accept`, что передает выбор формата в руки разработчика клиентов, даже когда работа производится с инструментальными наборами и браузерами, которые не позволяют устанавливать заголовок `Accept`.

Включение полного согласования содержимого

Для большинства приложений отправка данных JSON, когда нет других доступных форматов, является практичной политикой, поскольку клиент веб-приложения скорее некорректно установит свой заголовок `Accept`, чем не сумеет обработать JSON. Тем не менее, определенным приложениям придется иметь дело с клиентами, которые вызывают проблемы в случае возвращения данных JSON безотносительно к тому, что указано в заголовке `Accept`. Обеспечение работоспособности согласования содержимого требует внесения в конфигурацию внутри класса `Startup` двух изменений, как иллюстрируется в листинге 20.19.

Листинг 20.19. Включение полного согласования содержимого в файле `Startup.cs` из папки `ApiControllers`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;
using Microsoft.Net.Http.Headers;

namespace ApiControllers {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IRepository, MemoryRepository>();
      services.AddMvc()
        .AddXmlDataContractSerializerFormatters()
        .AddMvcOptions(opts => {
          opts.FormatterMappings.SetMediaTypeMappingForFormat("xml",
            new MediaTypeHeaderValue("application/xml"));
          opts.RespectBrowserAcceptHeader = true;
          opts.ReturnHttpNotAcceptable = true;
        });
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
```

```
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Параметр `RespectBrowserAcceptHeader` применяется для управления тем, полностью ли соблюдается заголовок `Accept`. Параметр `ReturnHttpNotAcceptable` используется для управления тем, будет ли клиенту отправляться ответ `406 - Not Acceptable` (406 — неприемлемо), если подходящий формат отсутствует.

Необходимо также удалить атрибут `Produces` из метода действия, чтобы процесс согласования содержимого не переопределялся (листинг 20.20).

Листинг 20.20. Удаление атрибута `Produces` в файле `ContentController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

  [Route("api/[controller]")]
  public class ContentController : Controller {

    [HttpGet("string")]
    public string GetString() => "This is a string response";

    [HttpGet("object/{format?}")]
    [FormatFilter]
    // [Produces("application/json", "application/xml")]
    public Reservation GetObject() => new Reservation {
      ReservationId = 100,
      ClientName = "Joe",
      Location = "Board Room"
    };
  }
}
```

Ниже показана команда PowerShell, которая отправляет запрос `GET` на URL вида `/api/content/object` с заголовком `Accept`, указывающий тип содержимого, который приложение не может предоставить:

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers
@{Accept="application/custom"}
```

В результате запуска команды отображается сообщение об ошибке 406, указывающее клиенту на то, что сервер не смог предоставить затребованный формат.

Получение разных форматов данных

Когда клиент посылает данные контроллеру, скажем, в запросе `POST`, с помощью атрибута `Consumes` можно указывать разные методы действий для обработки специфических форматов данных (листинг 20.21).

Листинг 20.21. Обработка разных форматов данных в файле ContentController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

  [Route("api/[controller]")]
  public class ContentController : Controller {

    [HttpGet("string")]
    public string GetString() => "This is a string response";

    [HttpGet("object/{format?}")]
    [FormatFilter]
    //[Produces("application/json", "application/xml")]
    public Reservation GetObject() => new Reservation {
      ReservationId = 100,
      ClientName = "Joe",
      Location = "Board Room"
    };

    [HttpPost]
    [Consumes("application/json")]
    public Reservation ReceiveJson([FromBody] Reservation reservation) {
      reservation.ClientName = "Json";
      return reservation;
    }

    [HttpPost]
    [Consumes("application/xml")]
    public Reservation ReceiveXml([FromBody] Reservation reservation) {
      reservation.ClientName = "Xml";
      return reservation;
    }
  }
}
```

Методы действий `ReceiveJson()` и `ReceiveXml()` принимают запросы `POST`. Отличие между ними связано с форматом данных, указанным в атрибуте `Consumes`, который исследует заголовок `Content-Type`, чтобы выяснить, способен ли метод действия обработать запрос. В результате, когда поступает запрос с заголовком `Content-Type`, установленным в `application/json`, будет применяться метод `ReceiveJson()`, но если заголовок `Content-Type` установлен в `application/xml`, тогда будет использоваться метод `ReceiveXml()`.

Резюме

В главе объяснялась роль, которую контроллеры API играют в приложениях MVC. Было продемонстрировано, каким образом создавать и тестировать контроллер API и как выполнять HTTP-запросы с применением jQuery. Вдобавок был описан процесс форматирования содержимого. В следующей главе более подробно рассматривается работа представлений и механизмов визуализации.

ГЛАВА 21

Представления

В главе 17 было показано, что методы действий могут возвращать объекты `ViewResult`, которые сообщают инфраструктуре MVC о необходимости визуализации представления и возвращения HTML-ответа клиенту.

Вы уже видели использование представлений во многих примерах книги, поэтому должны приблизительно понимать, что они делают, но в настоящей главе мы погрузимся в детали.

Мы начнем с того, что покажем, как инфраструктура MVC обрабатывает объекты `ViewResult` с применением *механизмов визуализации*, и продемонстрируем создание специального механизма визуализации. Кроме того, мы опишем приемы эффективной работы со встроенным механизмом визуализации Razor, в том числе использование частичных представлений и разделов компоновки, которые являются важными темами для построения эффективных приложений MVC. В табл. 21.1 приведена сводка, позволяющая поместить представления в контекст.

Таблица 21.1. Помещение представлений в контекст

Вопрос	Ответ
Что это такое?	Представления — часть паттерна MVC, применяемая для отображения содержимого пользователю. В приложении ASP.NET Core MVC представление является файлом, содержащим HTML-элементы и код C#, который обрабатывается для генерации ответа
Чем они полезны?	Представления позволяют отделять презентацию данных от логики, которая обрабатывает запросы. Представления также дают возможность применять ту же самую презентацию повсюду в приложении, т.к. многие контроллеры могут использовать одно и то же представление
Как они используются?	В большинстве приложений MVC применяется механизм визуализации Razor, который облегчает смешивание содержимого HTML и C#. Представления выбираются за счет возвращения объекта `ViewResult` в качестве результата метода действия, как было описано в главе 17
Существуют ли какие-то скрытые ловушки или ограничения?	Привыкание к использованию Razor и его смеси HTML и C# может занять какое-то время. В главе объясняется работа механизма Razor, что поможет в прояснении ряда его операций
Существуют ли альтернативы?	Для инфраструктуры MVC доступно несколько сторонних механизмов визуализации, но их применение ограничено

В табл. 21.2 приведена сводка по главе.

Таблица 21.2. Сводка по главе

Задача	Решение	Листинг
Создание специального механизма визуализации	Реализуйте интерфейсы `IViewEngine` и `IView`	21.3–21.6
Облегченное создание ответов со смесью разметки HTML и кода C#	Применяйте механизм визуализации Razor	21.7–21.11
Определение областей содержимого для использования в компоновке	Применяйте разделы Razor	21.12–21.18
Создание многократно используемых фрагментов разметки	Применяйте частичные представления	21.19–21.22
Добавление содержимого JSON в представления	Используйте выражение `@Json.Serialze`	21.23–21.25
Изменение местоположений, в которых Razor ищет представления	Создайте расширитель местоположений представлений	21.26–21.30

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `Views` с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core. Чтобы включить инфраструктуру MVC и другие компоненты промежуточного ПО, внесем в класс `Startup` изменения, показанные в листинге 21.1.

Листинг 21.1. Содержимое файла `Startup.cs` из папки `Views`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Views {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создадим папку Controllers, добавим в нее файл класса по имени HomeController.cs и определим контроллер, приведенный в листинге 21.2.

Листинг 21.2. Содержимое файла HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace Views.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() {
      ViewBag.Message = "Hello, World";
      ViewBag.Time = DateTime.Now.ToString("HH:mm:ss");
      return View("DebugData");
    }

    public ViewResult List() => View();
  }
}
```

Создание специального механизма визуализации

Мы собираемся впасть в крайность и создать специальный механизм визуализации. В большинстве проектов поступать так не придется, потому что инфраструктура MVC содержит встроенный механизм визуализации Razor, синтаксис которого был описан в главе 5 и который применялся во всех примерах, рассмотренных до сих пор (и вскоре будет использоваться снова).

Ценность создания специального механизма визуализации в том, что вы увидите происходящее "за кулисами" и расширите свои знания работы MVC. Вдобавок вы поймете, насколько большую свободу имеют механизмы визуализации при трансляции объектов ViewResult в ответы, предназначенные клиентам.

Механизмы визуализации — это классы, реализующие интерфейс IViewEngine, который определен в пространстве имен AspNetCore.Mvc.ViewEngines. Вот определение интерфейса IViewEngine:

```
namespace Microsoft.AspNetCore.Mvc.ViewEngines {
  public interface IViewEngine {
    ViewEngineResult GetView(string executingFilePath,
      string viewPath, bool isMainPage);
    ViewEngineResult FindView(ActionContext context, string viewName,
      bool isMainPage);
  }
}
```

Роль механизма визуализации заключается в трансляции запросов к представлениям в объекты ViewEngineResult. Когда инфраструктура MVC нуждается в представлении, она начинает с вызова метода GetView(), который дает механизму визуализации возможность предоставить представление, просто используя его имя.

Если методу GetView() не удалось предоставить представление, тогда вызывается метод FindView(), так что механизм визуализации получает шанс поискать представление с применением объекта ActionContext, который содержит информацию о методе действия, создавшем объект ViewResult.

Работа механизма визуализации связана со снабжением инфраструктуры MVC объектами ViewEngineResult, которые могут использоваться для генерации ответов. Создавать экземпляры класса ViewEngineResult напрямую нельзя, но для создания экземпляров он предлагает статические методы, которые перечислены в табл. 21.3.

Таблица 21.3. Статические методы класса ViewEngineResult

Имя	Описание
Found(name, view)	Вызов этого метода предоставляет инфраструктуре MVC запрошенное представление, которое указывается в параметре view. Представления реализуют интерфейс IView
NotFound(name, locations)	Вызов этого метода создает объект ViewEngineResult, который сообщает инфраструктуре MVC о том, что запрошенное представление найти не удалось. Параметр locations — это перечисление значений string, описывающих местоположения, в которых механизм визуализации искал представление

При написании механизма визуализации выбирается один из методов, описанных в табл. 21.3, для указания исхода запроса к представлению. Метод Found() создает объект ViewEngineResult, который обозначает успешный запрос и передает представление инфраструктуре MVC на обработку. Метод NotFound() создает объект ViewEngineResult, который отражает неудачный запрос и снабжает инфраструктуру MVC списком местоположений, просмотренных механизмом визуализации во время поиска представления (они будут отображаться разработчику как часть сообщения об ошибке).

Еще одним строительным блоком системы механизма визуализации является интерфейс IView, который применяется для описания функциональности, обеспечиваемой представлениями вне зависимости от того, какой механизм визуализации их создал. Интерфейс IView определен следующим образом:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc.ViewEngines {

  public interface IView {

    string Path { get; }
    Task RenderAsync(ViewContext context);
  }
}
```

Свойство Path возвращает путь к представлению, что предполагает определение представлений как файлов на диске. Метод RenderAsync() вызывается инфраструктурой MVC для генерации ответа клиенту. Данные контекста передаются представлению через экземпляр класса ViewContext, который является производным от класса ActionContext. В дополнение к свойствам контекста, унаследованным от родительского класса (которые обеспечивают доступ к запросу, данным маршрутизации, контроллеру и т.д.), класс ViewContext предлагает свойства, удобные для использования при визуализации ответов; наиболее полезные свойства такого рода приведены в табл. 21.4.

Таблица 21.4. Полезные свойства класса `ViewContext`

Имя	Описание
`ViewData`	Это свойство возвращает объект `ViewDataDictionary`, который содержит данные представления, предоставленные контроллером
`TempData`	Это свойство возвращает словарь, содержащий временные данные (как описано в главе 17)
`Writer`	Это свойство возвращает объект `TextWriter`, который должен применяться для записи вывода из представления

Самым интересным из всех перечисленных является свойство `ViewData`, возвращающее объект `ViewDataDictionary`. В классе `ViewDataDictionary` определено несколько полезных свойств, которые предоставляют доступ к модели представления, данным `ViewBag` и метаданным модели представления. Наиболее полезные свойства описаны в табл. 21.5.

Таблица 21.5. Полезные свойства класса `ViewDataDictionary`

Имя	Описание
`Model`	Это свойство типа `object` возвращает данные модели, предоставленные контроллером
`ModelMetadata`	Это свойство возвращает объект `ModelMetadata`, который может использоваться для рефлексии типа данных модели
`ModelState`	Это свойство возвращает состояние модели, которое рассматривается в главе 27
`Keys`	Это свойство возвращает перечисление значений ключей, которые могут применяться для доступа к данным `ViewBag`

Простейший способ выяснить, как работают типы `IViewEngine`, `ViewEngine Result` и `IView`, предусматривает создание механизма визуализации. Мы построим простой механизм визуализации, который возвращает один вид представления. Представление будет визуализировать результат, содержащий информацию о запросе и данные представления, которые сгенерированы методом действия. Такой подход позволяет продемонстрировать способ оперирования механизмов визуализации, не увязая в разборе шаблонов представлений и не занимаясь воссозданием других средств, которые поддерживает Razor.

Создание специальной реализации интерфейса `IView`

Начнем с создания реализации интерфейса `IView`. Добавим в проект папку `Infrastructure` и создадим в ней файл класса по имени `DebugDataView.cs`, содержимое которого показано в листинге 21.3.

Листинг 21.3. Содержимое файла `DebugDataView.cs` из папки `Infrastructure`

```
using System;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Rendering;
```

```
using Microsoft.AspNetCore.Mvc.ViewEngines;

namespace Views.Infrastructure {

  public class DebugDataView : IView {
    public string Path => String.Empty;

    public async Task RenderAsync(ViewContext context) {
      context.HttpContext.Response.ContentType = "text/plain";

      StringBuilder sb = new StringBuilder();

      sb.AppendLine("---Routing Data---");  // Данные маршрутизации
      foreach (var kvp in context.RouteData.Values) {
        sb.AppendLine($"Key: {kvp.Key}, Value: {kvp.Value}");
      }

      sb.AppendLine("---View Data---");     // Данные представления
      foreach (var kvp in context.ViewData) {
        sb.AppendLine($"Key: {kvp.Key}, Value: {kvp.Value}");
      }

      await context.Writer.WriteAsync(sb.ToString());
    }
  }
}
```

Когда представление `DebugDataView` визуализируется, оно записывает детали данных маршрутизации и данных представления, полученные с использованием аргумента `ViewContext` метода `RenderAsync()`. Ответом является простой текст, поэтому с помощью объекта контекста заголовок `Content-Type` запроса устанавливается в `text/plain`. Без такого присваивания ASP.NET по умолчанию применит `text/html`, что заставит браузер отображать данные в виде единственной неразбитой строки символов.

Создание реализации интерфейса `IViewEngine`

Цель механизма визуализации — выпуск объекта `ViewEngineResult`, который содержит либо экземпляр реализации `IView`, либо список мест, где производился поиск подходящего представления. Теперь, имея рабочую реализацию интерфейса `IView`, можно создать механизм визуализации. Добавим в папку `Infrastructure` файл класса по имени `DebugDataViewEngine.cs` с содержимым из листинга 21.4.

Листинг 21.4. Содержимое файла `DebugDataViewEngine.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewEngines;

namespace Views.Infrastructure {

  public class DebugDataViewEngine : IViewEngine {

    public ViewEngineResult GetView(string executingFilePath, string viewPath,
        bool isMainPage) {
      return ViewEngineResult.NotFound(viewPath,
        new string[] { "(Debug View Engine - GetView)" });
    }
```

```
    public ViewEngineResult FindView(ActionContext context, string viewName,
        bool isMainPage) {
      if (viewName == "DebugData") {
        return ViewEngineResult.Found(viewName, new DebugDataView());
      } else {
        return ViewEngineResult.NotFound(viewName,
          new string[] { "(Debug View Engine - FindView)" });
      }
    }
  }
}
```

Метод `GetView()` в данном механизме визуализации всегда возвращает ответ `NotFound`. Метод `FindView()` поддерживает только одно представление, которое называется `DebugData`. Когда механизм визуализации получает запрос представления с таким именем, он возвращает новый экземпляр класса `DebugDataView`:

```
...
if (viewName == "DebugData") {
  return ViewEngineResult.Found(viewName, new DebugDataView());
}
...
```

При реализации более совершенного механизма визуализации эта возможность использовалась бы для поиска шаблонов. В рассматриваемом простом примере требуется только создание нового экземпляра класса `DebugDataView`. В случае получения запроса представления, отличающегося от `DebugData`, создается ответ `NotFound`:

```
...
return ViewEngineResult.NotFound(viewName,
  new string[] { "(Debug View Engine - FindView)" });
...
```

Метод `ViewEngineResult.NotFound()` предполагает, что механизм визуализации располагает информацией о местах, в которых необходимо искать представления. Предположение вполне обосновано, поскольку представления обычно являются шаблонами, хранящимися как файлы проекта. В текущем случае искать негде, потому просто возвращается фиктивное местоположение, которое укажет, какой метод был вызван для поиска представления.

Регистрация специального механизма визуализации

Механизмы визуализации регистрируются в классе `Startup` путем конфигурирования объекта `MvcViewOptions`, как показано в листинге 21.5.

Листинг 21.5. Регистрация специального механизма визуализации в файле `Startup.cs` из папки `Views`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;

namespace Views {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.Configure<MvcViewOptions>(options => {
        options.ViewEngines.Insert(0, new DebugDataViewEngine());
      });
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

В классе `MvcViewOptions` определено свойство `ViewEngines`, которое представляет собой коллекцию объектов реализации `IViewEngine`. Механизм Razor добавляется в коллекцию `ViewEngine` с помощью метода `AddMvc()`, и стандартный механизм визуализации дополняется нашим специальным классом.

Когда инфраструктура MVC получает объект `ViewResult` от метода действия, она вызывает метод `FindView()` каждого механизма визуализации, содержащегося в коллекции `MvcViewOptions.ViewEngines`, до тех пор пока не получит объект `ViewEngineResult`, который был создан с применением метода `Found()`.

Порядок, в котором механизмы добавляются в коллекцию `MvcViewOptions.ViewEngines`, важен в случае, если два или большее число механизмов способны обслуживать запрос к представлению с одним и тем же именем. Чтобы специальный механизм визуализации имел приоритет, он должен быть вставлен в начало коллекции `ViewEngines`, как делалось в листинге 21.5.

Тестирование механизма визуализации

Когда приложение запускается, браузер автоматически переходит на корневой URL проекта, который будет отображен на действие `Index` контроллера `Home`. Этот метод действия использует метод `View()` для возвращения объекта `ViewResult`, который указывает представление `DebugData`.

Инфраструктура MVC обратится к коллекции механизмов визуализации и начнет вызывать их методы `FindView()`. Поскольку запрошенное представление является именно тем, на обработку которого настроен специальный механизм визуализации, он снабжает MVC представлением, отображающим результаты, как показано на рис. 21.1.

Чтобы посмотреть, что произойдет, когда ни один из механизмов визуализации не может предоставить представление, нужно запросить URL вида `/Home/List`. В таком случае будет создан объект `ViewResult`, указывающий представление по имени `List`, которое не способен предоставить ни Razor, ни специальный механизм визуализации. Отобразится сообщение об ошибке, продемонстрированное на рис. 21.2.

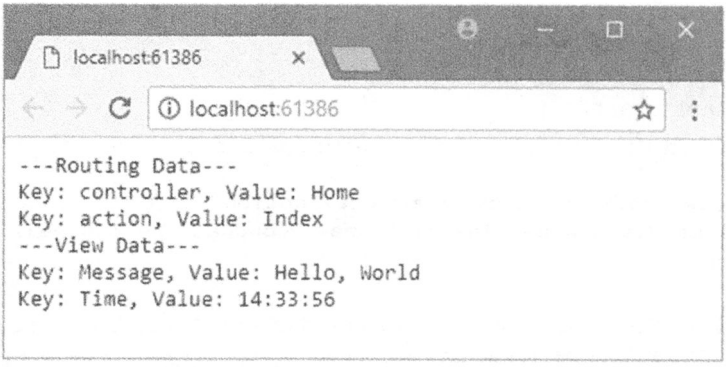


Рис. 21.1. Использование специального механизма визуализации

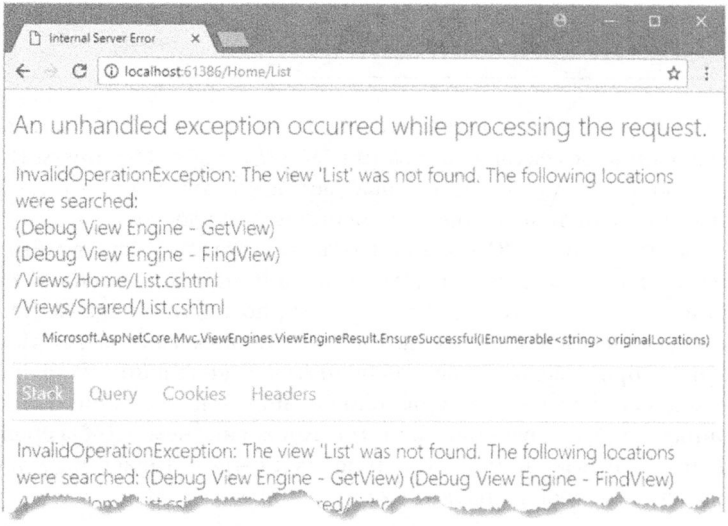


Рис. 21.2. Запрашивание представления, которое не может быть предоставлено

Здесь видно, что сообщения, выдаваемые специальным механизмом визуализации, присутствуют в списке местоположений, в которых выполнялся поиск представления `List`, наряду с местоположениями, проверенными механизмом Razor.

Когда необходимо гарантировать, что будет применяться только специальный механизм визуализации, понадобится вызвать метод `Clear()` на коллекции механизмов визуализации, чтобы удалить из нее Razor (листинг 21.6).

Листинг 21.6. Удаление других механизмов визуализации в файле `Startup.cs` из папки `Views`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
```

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;

namespace Views {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.Configure<MvcViewOptions>(options => {
        options.ViewEngines.Clear();
        options.ViewEngines.Insert(0, new DebugDataViewEngine());
      });
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Если запустить приложение и снова перейти на URL вида `/Home/List`, то легко заметить, что используется только специальный механизм визуализации (рис. 21.3).

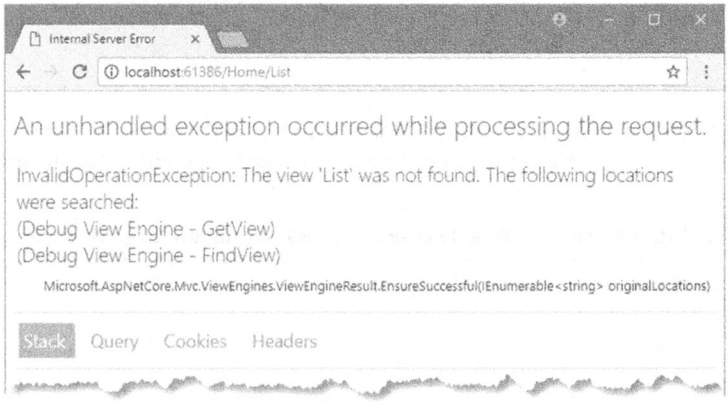


Рис. 21.3. Применение только специального механизма визуализации в примере приложения

Работа с механизмом Razor

В предыдущем разделе у нас была возможность создать специальный механизм визуализации за счет реализации всего лишь двух интерфейсов. Правда, итогом стало простое подобие механизма, который генерировал неуклюжее содержимое, но зато вы увидели, насколько легко инфраструктура MVC позволяет добавлять или заменять основную функциональность.

Сложность в механизм визуализации привносится системой шаблонов представлений, которая включает фрагменты кода, поддерживает компоновки и обеспечивает оптимизацию производительности. В своем простом механизме визуализации мы ничего подобного не делали (и по большому счету не должны), потому что встроенный механизм Razor предоставляет все эти средства и многое другое. На самом деле в Razor доступна функциональность, которая требуется практически во всех приложениях MVC. Лишь в ничтожно малом количестве проектов разработчики берут на себя труд создавать специальный механизм визуализации.

Основы синтаксиса Razor приводились в главе 5, а в настоящем разделе будет показано, как использовать другие средства для создания и визуализации представлений Razor. Вы также научитесь настраивать механизм Razor.

Подготовка проекта для примера

Подготовка проекта к работе с Razor требует внесения нескольких изменений. Модифицируем действие `Index` контроллера `Home`, чтобы оно выбирало стандартное представление и предоставляло некоторые данные модели (листинг 21.7).

Листинг 21.7. Изменение действия `Index` в файле `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace Views.Controllers {
  public class HomeController : Controller {
    public ViewResult Index() =>
      View(new string[] { "Apple", "Orange", "Pear" });
    public ViewResult List() => View();
  }
}
```

Для снабжения метода действия `Index()` представлением создадим папку `Views/Home` и добавим в нее файл по имени `Index.cshtml` с содержимым, приведенным в листинге 21.8.

Листинг 21.8. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model string[]
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Razor</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  This is a list of fruit names:
  @foreach (string name in Model) {
    <span><b>@name</b></span>
  }
</body>
</html>
```

Представление полагается на CSS-библиотеку Bootstrap. Чтобы добавить Bootstrap в проект, создадим в корневой папке проекта файл `bower.json` с применением шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в его раздел `dependencies` (листинг 20.9).

Листинг 21.9. Содержимое файла `bower.json` из папки `Views`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Создадим в папке `Views` файл `_ViewImports.cshtml` с выражением, показанным в листинге 21.10, чтобы включить встроенные вспомогательные функции дескрипторов.

Листинг 21.10. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Финальный подготовительный шаг связан с переустановкой механизмов визуализации в классе `Startup` с целью удаления специального механизма и удаления вызова метода `Clear()`, который отключает Razor (листинг 21.11).

Листинг 21.11. Переустановка механизмов визуализации в файле `Startup.cs` из папки `Views`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;

namespace Views {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      // services.Configure<MvcViewOptions>(options => {
      //   options.ViewEngines.Clear();
      //   options.ViewEngines.Insert(0, new DebugDataViewEngine());
      // });
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
```

```
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Результат запуска приложения представлен на рис. 21.4.

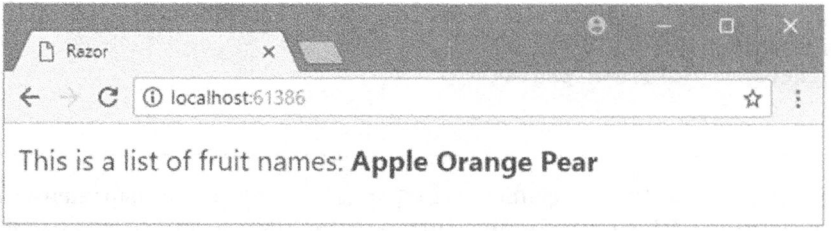


Рис. 21.4. Выполнение примера приложения

Прояснение представлений Razor

Даже начальное понимание работы механизма Razor может помочь поместить большой объем функциональности в контекст и приоткрыть тайну обработки файлов CSHTML.

Так каким же образом Razor берет смесь элементов HTML с операторами C# и выпускает содержимое для HTTP-ответа? Ответ довольно прост и основан на функциональности MVC, о которой вы уже узнали в предшествующих главах. Механизм Razor преобразует файлы CSHTML в классы C#, компилирует их и затем создает новые экземпляры каждый раз, когда представлению требуется сгенерировать результат. Вот класс C#, который Razor создает для представления `Index.cshtml` из листинга 21.8:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Razor;
using Microsoft.AspNetCore.Mvc.Razor.Internal;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace Asp {

  public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]> {
    public IUrlHelper Url { get; private set; }
    public IViewComponentHelper Component { get; private set; }
    public IJsonHelper Json { get; private set; }
    public IHtmlHelper<string[]> Html { get; private set; }
    public override async Task ExecuteAsync() {
      Layout = null;

      WriteLiteral(@"<!DOCTYPE html><html><head>
        <meta name=""viewport"" content=""width=device-width"" />
        <title>Razor</title>
        <link asp-href-include=""lib/bootstrap/dist/css/*.min.css""
          rel=""stylesheet"" />
        </head><body class=""m-1 p-1"">This is a list of fruit names:");
```

```
        foreach (string name in Model) {
          WriteLiteral("<span><b>");
          Write(name);
          WriteLiteral("</b></span>");
        }
        WriteLiteral("</body></html>");
      }
    }
}
```

Код класса был приведен в порядок, чтобы облегчить его чтение. Кроме того, были удалены операторы C#, которые Razor добавляет с целью оснащения инструментами при генерации класса. В последующих разделах данный класс разбивается на части, и приводятся объяснения работы скомпилированных представлений.

На заметку! Прежде было легко просматривать классы, созданные более ранними версиями Razor, т.к. для каждого представления генерировался дисковый файл C#, который затем компилировался и использовался в приложении. Изучение класса сводилось просто к нахождению правильного файла. Текущая версия Razor опирается на усовершенствования компилятора C#, которые позволяют коду генерироваться и компилироваться в памяти, что обеспечивает улучшение производительности, но затрудняет выяснение происходящего. Чтобы получить показанный выше класс, пришлось изменить предназначение ряда модульных тестов, входящих в состав исходного кода ASP.NET Core MVC, которые предоставляли фиктивные реализации классов, применяемые Razor для нахождения и обработки файлов представлений. Это не то, что вам придется предпринимать при повседневной разработке, но такой процесс позволяет выявить очень многие детали о том, как работают представления.

Имя класса

Лучше всего начать с имени класса, который создает механизм Razor:

```
...
public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]> {
...
```

Механизму Razor нужен какой-нибудь способ для трансляции имени и пути к файлу CSHTML в класс, который он создает, когда производит разбор файла, и Razor делает это за счет кодирования информации в имени класса. Механизм Razor снабжает имя класса префиксом `ASPV`, за которым следует имя проекта, имя контроллера и, наконец, имя файла представления; такая комбинация облегчает проверку доступности класса при запрашивании инфраструктурой MVC представления через интерфейс `IViewEngine`, описанный ранее в главе.

Базовый класс

Многие основные средства Razor, такие как ссылка на модель представления в виде `@Model`, возможны благодаря базовому классу, от которого унаследованы сгенерированные классы:

```
...
public class ASPV_Views_Home_Index_cshtml : RazorPage<string[]> {
...
```

Классы представлений наследуются от класса `RazorPage` или класса `RazorPage<T>`, если с помощью директивы `@model` был указан тип модели. Класс `RazorPage` предлагает методы и свойства, которые могут использоваться в файлах CSHTML для доступа к средствам MVC; наиболее полезные методы и свойства класса `RazorPage<T>` описаны в табл. 21.6.

Таблица 21.6. Методы и свойства класса `RazorPage<T>`, полезные при разработке представлений

Имя	Описание
`Model`	Это свойство возвращает данные модели, предоставляемые методом действия
`ViewData`	Это свойство возвращает объект `ViewDataDictionary`, который обеспечивает доступ к другим средствам данных представления
`ViewContext`	Это свойство возвращает объект `ViewContext`, описанный в табл. 21.4
`Layout`	Это свойство применяется для указания компоновки, как было показано в главе 5 и снова упоминается в разделе "Использование разделов компоновки" далее в настоящей главе
`ViewBag`	Это свойство предоставляет доступ к объекту `ViewBag`, который был описан в главе 17
`TempData`	Это свойство предоставляет доступ к объекту `TempData`, который рассматривался в главе 17
`Context`	Это свойство возвращает объект `HttpContext`, который описывает текущий запрос и подготавливаемый ответ
`User`	Это свойство возвращает профиль пользователя, ассоциированного с текущим запросом. Аутентификация и авторизация пользователей подробно рассматриваются в главе 28
`RenderSection()`	Этот метод применяется для вставки раздела содержимого из представления в компоновку, как описано в разделе "Использование разделов компоновки" далее в главе
`RenderBody()`	Этот метод вставляет в компоновку все содержимое представления, которое не находится внутри раздела. Подробности ищите в разделе "Использование разделов компоновки" далее в главе
`IsSectionDefined()`	Этот метод используется для выяснения, определен ли раздел в представлении

Средство Razor Pages

В версию ASP.NET Core 2 было добавлено средство Razor Pages, нарушающее модель MVC. Оно помещает код, который требуется для поддержки представления, в файл, ассоциированный с представлением Razor. Структура похожа на ту, что применялась в ASP.NET Web Forms. Компания Microsoft периодически обращается к такому проектному подходу в попытках возвратить простоту старых платформ для создания веб-страниц, но без присущих им недостатков, которые были описаны в главе 1.

Не следует путать базовый класс RazorPage, рассматриваемый в этом разделе, со средством Razor Pages. Несмотря на похожие имена, базовый класс RazorPage предоставляет основу для механизма визуализации Razor, используемого MVC Framework. Средство Razor Pages в книге не обсуждается, т.к. оно не соответствует модели MVC и не является частью платформы MVC.

Механизм Razor также предлагает несколько вспомогательных свойств, которые можно применять в представлениях для генерации содержимого (табл. 21.7).

Таблица 21.7. Вспомогательные свойства Razor

Имя	Описание
HtmlEncoder	Это свойство возвращает объект HtmlEncoder, который может использоваться для безопасного кодирования HTML-содержимого в представлении
Component	Это свойство возвращает вспомогательный объект для компонента представления (глава 22)
Json	Это свойство возвращает вспомогательный объект JSON, как описано в разделе "Добавление содержимого JSON в представления" далее в главе
Url	Это свойство возвращает вспомогательный объект URL, который может применяться для генерации URL, используя конфигурацию маршрутизации, как было описано в главе 16
Html	Это свойство возвращает вспомогательный объект HTML, который может применяться для генерации динамического содержимого. Данное средство было почти полностью замещено вспомогательными функциями дескрипторов, но все еще используется для частичных представлений, как объясняется в разделе "Использование частичных представлений" далее в главе

Методы и свойства, перечисленные в табл. 21.6 и 21.7, вы будете применять при ежедневной разработке приложений MVC для доступа к данным модели, конфигурирования представлений и решения других важных задач. Они приподнимают занавес над использованием механизма Razor, возвращая его в хорошо понимаемый мир C#. Например, когда вы получаете доступ к объекту модели представления либо извлекаете значение из TempData посредством @TempData, то ссылаетесь на свойства, которые определены в классе RazorPage.

Визуализация представлений

В дополнение к свойствам и методам, предоставляющим функциональные средства для разработчиков, в обязанности класса RazorPage также входит генерирование содержимого ответов через его метод ExecuteAsync(). Этот метод показывает, каким образом Razor обрабатывает файл Index.cshtml с помощью набора операторов C#:

```
...
public override async Task ExecuteAsync() {
  Layout = null;
  WriteLiteral(@"<!DOCTYPE html><html><head>
    <meta name=""viewport"" content=""width=device-width"" />
    <title>Razor</title>
    <link asp-href-include=""lib/bootstrap/dist/css/*.min.css""
      rel=""stylesheet"" />
```

```
    </head><body class=""m-1 p-1"">This is a list of fruit names:");
    foreach (string name in Model) {
      WriteLiteral("<span><b>");
      Write(name);
      WriteLiteral("</b></span>");
    }
    WriteLiteral("</body></html>");
  }
  ...
```

Значения данных, такие как значения свойства `Model`, отправляются клиенту с применением метода `Write()`, который защищает строки так, что они не будут интерпретироваться браузером как HTML-элементы. Упомянутый аспект важен, поскольку предотвращает попадание в вывод приложения злонамеренных данных из добавленного содержимого.

Метод `WriteLiteral()` не защищает строки и используется для статического содержимого в файле `Index.cshtml`, которое браузер, конечно же, должен интерпретировать как HTML-элементы. В результате статическое и динамическое содержимое файла CSHTML находится в обычном классе C# и выпускается посредством простого вызова метода.

Добавление динамического содержимого к представлению Razor

Основная цель представлений — сделать возможной визуализацию частей модели предметной области пользователям. Для этого необходимо уметь добавлять к представлениям *динамическое содержимое*. Динамическое содержимое генерируется во время выполнения и может быть разным для каждого запроса. Оно является противоположностью *статического содержимого*, такого как HTML-разметка, которое создается при написании кода приложения и одинаково для всех запросов. Добавлять динамическое содержимое к представлениям можно различными способами, которые описаны в табл. 21.8.

Таблица 21.8. Добавление динамического содержимого к представлению

Прием	Когда применяется
Встраиваемый код	Используется для небольших и самодостаточных порций логики представления, таких как операторы `if` и `foreach`. Это фундаментальный инструмент для создания динамического содержимого в представлениях, на основе которого построены другие подходы. Данный прием был описан в главе 5 и затем применялся в многочисленных примерах
Вспомогательные функции дескрипторов	Используются для генерации атрибутов в HTML-элементах. Вспомогательные функции дескрипторов рассматриваются в главах 23–25
Разделы	Применяются для создания разделов содержимого, которые будут вставляться в специфические места внутри компоновки, как объясняется далее в главе

Окончание табл. 21.8

Прием	Когда применяется
Частичные представления	Применяются для совместного использования подразделов компоновки представления несколькими представлениями. Частичные представления могут содержать встраиваемый код, вспомогательные методы объектов HTML и ссылки на другие частичные представления. Частичные представления не вызывают какой-либо метод действия, поэтому не могут применяться для выполнения бизнес-логики. Частичные представления описаны далее в разделе
Компоненты представлений	Применяются для создания многократно используемых элементов управления или виджетов пользовательского интерфейса, которые должны содержать бизнес-логику. Компоненты представлений рассматриваются в главе 22

Использование разделов компоновки

Механизм визуализации Razor поддерживает концепцию *разделов*, которые позволяют организовывать области содержимого внутри компоновки. Разделы Razor обеспечивают больший контроль над тем, какие части представления вставляются в компоновку и куда они помещаются. Чтобы взглянуть на работу разделов, приведем содержимое файла `/Views/Home/Index.cshtml` в соответствие с листингом 21.12.

Листинг 21.12. Определение разделов в файле `Index.cshtml` из папки `Views/Home`

```
@model string[]
@{ Layout = "_Layout"; }

@section Header {
  <div class="bg-success">
    @foreach (string str in new [] {"Home", "List", "Edit"}) {
      <a class="btn btn-sm btn-primary" asp-action="str">@str</a>
    }
  </div>
}

This is a list of fruit names:
@foreach (string name in Model) {
  <span><b>@name</b></span>
}

@section Footer {
  <div class="bg-success">
    This is the footer
  </div>
}
```

Из представления были удалены некоторые HTML-элементы и установлено свойство `Layout` для указания на то, что при визуализации содержимого должен применяться файл компоновки по имени `_Layout.cshtml`.

Кроме того, в представление были добавлены разделы. Разделы определяются с использованием Razor-выражения `@section`, за которым следует имя раздела.

В листинге 21.12 созданы разделы с именами `Header` и `Footer`. Раздел содержит обычную смесь разметки HTML и выражений Razor, которую можно встретить за пределами разделов в других примерах.

Разделы определяются в представлении, но применяются в компоновке с помощью выражения `@RenderSection`. Чтобы посмотреть, как работают разделы, создадим папку `Views/Shared` и добавим в нее файл компоновки по имени `_Layout.cshtml`, содержимое которого показано в листинге 21.13.

Листинг 21.13. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @RenderSection("Header")

  <div class="bg-info">
    This is part of the layout
  </div>

  @RenderBody()

  <div class="bg-info">
    This is part of the layout
  </div>

  @RenderSection("Footer")

  <div class="bg-info">
    This is part of the layout
  </div>
</body>
</html>
```

Когда механизм Razor производит разбор компоновки, выражение `@RenderSection` заменяется содержимым раздела с указанным именем из представления. Части представления, которые не содержатся в каком-либо разделе, вставляются в компоновку с использованием выражения `@RenderBody`.

Запустив приложение, можно увидеть эффект от применения разделов (рис. 21.5). Чтобы было совершенно ясно, какие разделы вывода поступают из представления, а какие — из компоновки, используются стили Bootstrap. Результат далек от идеала, но четко демонстрирует, как можно помещать области содержимого из представления в специфические местоположения внутри компоновки.

На заметку! В представлении можно определять только разделы, на которые имеются ссылки внутри компоновки. При попытке определить в представлении разделы, для которых соответствующие выражения `@RenderSection` в компоновке отсутствуют, инфраструктура MVC сгенерирует исключение.

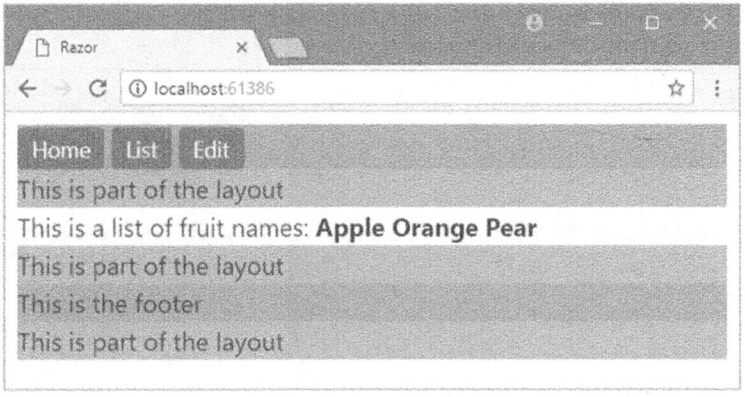


Рис. 21.5. Использование разделов представления для размещения содержимого в компоновке

Обычно разделы не смешиваются с остальной частью представления. По соглашению разделы определяются либо в начале, либо в конце представления, чтобы легче было видеть, какие области содержимого будут трактоваться как разделы, а какие будут обслуживаться выражением `@RenderBody`. Еще один подход предусматривает определение представления полностью в терминах разделов, включая раздел для тела (листинг 21.14).

Листинг 21.14. Определение представления с использованием разделов Razor в файле `Index.cshtml` из папки `Views/Home`

```
@model string[]
@{ Layout = "_Layout"; }

@section Header {
  <div class="bg-success">
    @foreach (string str in new [] {"Home", "List", "Edit"}) {
      <a class="btn btn-sm btn-primary" asp-action="str">@str</a>
    }
  </div>
}
@section Body {
  This is a list of fruit names:
  @foreach (string name in Model) {
    <span><b>@name</b></span>
  }
}
@section Footer {
  <div class="bg-success">
    This is the footer
  </div>
}
```

При таком подходе получаются более чистые представления и уменьшаются шансы захвата излишнего содержимого выражением `@RenderBody`. Чтобы задействовать подход, выражение `@RenderBody` понадобится заменить выражением `@RenderSection("Body")`, как показано в листинге 21.15.

Листинг 21.15. Визуализация тела в виде раздела в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @RenderSection("Header")

  <div class="bg-info">
    This is part of the layout
  </div>

  @RenderSection("Body")

  <div class="bg-info">
    This is part of the layout
  </div>

  @RenderSection("Footer")

  <div class="bg-info">
    This is part of the layout
  </div>
</body>
</html>
```

Проверка существования разделов

При необходимости можно выполнять проверку, определен ли в представлении специфический раздел из компоновки. Это удобный способ предоставления стандартного содержимого для раздела, если представление не нуждается или не желает предоставлять специальное содержимое. В листинге 21.16 приведено модифицированное содержимое файла `_Layout.cshtml`, в котором предпринимается проверка, определен ли раздел `Footer`.

Листинг 21.16. Проверка, определен ли раздел в представлении, в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @RenderSection("Header")

  <div class="bg-info">
    This is part of the layout
  </div>
```

```
  @RenderSection("Body")
  <div class="bg-info">
    This is part of the layout
  </div>
  @if (IsSectionDefined("Footer")) {
    @RenderSection("Footer")
  } else {
    <h4>This is the default footer</h4>
  }
  <div class="bg-info">
    This is part of the layout
  </div>
</body>
</html>
```

Вспомогательный метод `IsSectionDefined()` принимает имя проверяемого раздела и возвращает `true`, если он определен в визуализируемом представлении. В примере метод `IsSectionDefined()` применяется для выяснения, должно ли визуализироваться стандартное содержимое, когда в представлении не определен раздел `Footer`.

Визуализация необязательных разделов

По умолчанию представление должно содержать все разделы, для которых в компоновке имеются выражения `@RenderSection`. Если разделы отсутствуют, тогда инфраструктура MVC сгенерирует исключение. Чтобы удостовериться в сказанном, добавим в файл `_Layout.cshtml` новое выражение `@RenderSection` для раздела по имени `scripts` (листинг 21.17).

Листинг 21.17. Визуализация несуществующего раздела в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @RenderSection("Header")
  <div class="bg-info">
    This is part of the layout
  </div>
  @RenderSection("Body")
  <div class="bg-info">
    This is part of the layout
  </div>
  @if (IsSectionDefined("Footer")) {
    @RenderSection("Footer")
  } else {
    <h4>This is the default footer</h4>
  }
```

```
    @RenderSection("scripts")
    <div class="bg-info">
      This is part of the layout
    </div>
</body>
</html>
```

После запуска приложения механизм Razor пытается визуализировать компоновку и представление, в результате чего возникает ошибка (рис. 21.6).

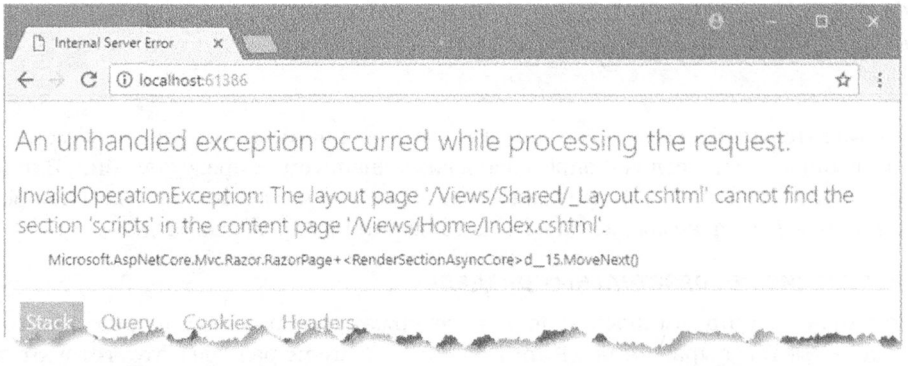


Рис. 21.6. Сообщение об ошибке, отображаемое из-за отсутствия раздела

Можно задействовать метод `IsSectionDefined()`, чтобы избежать выполнения выражений `@RenderSection` для разделов, которые не определены в представлении, но более элегантный подход предусматривает использование необязательных разделов, для которых в `@RenderSection` указывается дополнительный аргумент `false` (листинг 21.18).

Листинг 21.18. Создание необязательного раздела

```
...
@RenderSection("scripts", false)
...
```

В итоге создается необязательный раздел, содержимое которого будет вставлено в результат, если представление определено, а его отсутствие не приведет к генерации исключения.

Использование частичных представлений

Часто возникает необходимость в применении одних и тех же фрагментов дескрипторов Razor и разметки HTML в нескольких местах приложения. Вместо дублирования содержимого можно использовать *частичные представления* — отдельные файлы представлений, содержащие фрагменты дескрипторов и разметку, которые могут быть включены в другие представления. В настоящем разделе будет показано, как создавать и применять частичные представления, объяснена их работа и продемонстрированы приемы, доступные для передачи данных представления в частичное представление.

Создание частичного представления

Частичные представления являются всего лишь обычными файлами CSHTML, отличаясь от нормальных представлений только способом их использования. Среда Visual Studio предлагает инструментальную поддержку для создания заранее наполненных частичных представлений, но создать частичное представление проще всего, создав обычное представление с применением шаблона элемента MVC View Page (Страница представления MVC). Добавим в папку `Views/Home` файл по имени `MyPartial.cshtml` и поместим в него содержимое из листинга 21.19.

Листинг 21.19. Содержимое файла `MyPartial.cshtml` из папки `Views/Home`

```
<div class="bg-info">
  <div>This is the message from the partial view.</div>
  <a asp-action="Index">This is a link to the Index action</a>
</div>
```

Здесь планируется продемонстрировать возможность смешивания в частичном представлении статического и динамического содержимого, поэтому было определено простое сообщение и якорный элемент, в котором используется вспомогательная функция дескриптора.

Применение частичного представления

Частичное представление потребляется посредством выражения `@Html.Partial` внутри другого представления. Создадим в папке `Views/Home` файл по имени `List.cshtml` и добавим в него содержимое, показанное в листинге 21.20.

Листинг 21.20. Содержимое файла `List.cshtml` из папки `Views/Home`

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Razor</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  This is the List View
  @Html.Partial("MyPartial")
</body>
</html>
```

Метод `Partial()` — это расширяющий метод, применяемый к свойству `Html`, которое добавлено в класс, сгенерированный механизмом Razor из файла представления. Он является примером вспомогательного метода HTML, который использовался в качестве способа генерирования динамического содержимого внутри представлений в предшествующих версиях MVC, но теперь в значительной степени замещен вспомогательными функциями дескрипторов. Методу `Partial()` передается аргумент, указывающий имя частичного представления, содержимое которого вставляется в вывод, отправляемый клиенту.

Совет. Механизм Razor ищет частичные представления там же, где и обычные представления (в папках `~/Views/<контроллер>` и `~/Views/Shared`). Другими словами, можно создавать специализированные версии частичных представлений, специфичные для контроллера, и переопределять частичные представления с таким же именем из папки `Shared`.

Запустив приложение и перейдя на URL вида `/Home/List`, можно оценить эффект от потребления частичного представления (рис. 21.7).

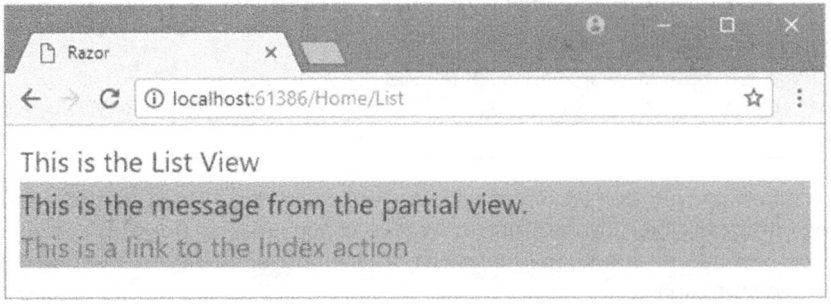


Рис. 21.7. Применение частичного представления

Использование строго типизированных частичных представлений

Допускается создавать строго типизированные частичные представления и снабжать их объектами модели представления, подлежащими применению при визуализации частичных представлений. Создадим в папке `Views/Home` файл представления по имени `MyStronglyTypedPartial.cshtml` и поместим в него содержимое из листинга 21.21.

Листинг 21.21. Содержимое файла `MyStronglyTypedPartial.cshtml` из папки `Views/Home`

```
@model IEnumerable<string>

<div class="bg-info">
  This is the message from the partial view.
  <ul>
    @foreach (string str in Model) {
      <li>@str</li>
    }
  </ul>
</div>
```

С использованием стандартного выражения `@model` определяется тип модели представления, а с помощью цикла `@foreach` отображается содержимое объекта модели представления в виде элементов списка HTML. Чтобы задействовать новое частичное представление, модифицируем содержимое файла `/Views/Common/List.cshtml`, как показано в листинге 21.22.

Листинг 21.22. Применение строго типизированного частичного представления в файле `List.cshtml` из папки `Views/Common`

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Razor</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  This is the List View
  @Html.Partial("MyStronglyTypedPartial",
    new string[] { "Apple", "Orange", "Pear" })
</body>
</html>
```

Отличие от предыдущего примера заключается в том, что здесь вспомогательному методу `Partial()` передается дополнительный аргумент, который поставляет модель представления. Запустив приложение и перейдя на URL вида `/Home/List`, можно посмотреть на работу строго типизированного частичного представления (рис. 21.8).



Рис. 21.8. Использование строго типизированного частичного представления

Добавление содержимого JSON в представления

Содержимое JSON часто включается в представления с целью снабжения кода JavaScript клиентской стороны данными, которые могут применяться при динамической генерации содержимого. Для подготовки к такому примеру добавим в приложение пакет jQuery, отредактировав файл `bower.json` согласно листингу 21.23. Пакет jQuery облегчает обработку данных JSON, когда они получаются браузером как часть HTML-документа.

Листинг 21.23. Добавление пакета jQuery в файле `bower.json` из папки `Views`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

В листинге 21.24 приведены добавления внутри представления `List.cshtml`, которые используют Razor для включения данных JSON в ответ, отправляемый браузеру.

Листинг 21.24. Работа с данными JSON в файле `List.cshtml` из папки `Views/Common`

```
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Razor</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
  <script id="jsonData" type="application/json">
    @Json.Serialize(new string[] { "Apple", "Orange", "Pear" })
  </script>
</head>
<body class="m-1 p-1">
  This is the List View
  <ul id="list"></ul>
</body>
</html>
```

Выражение `@Json.Serialize` принимает объект и сериализирует его в формат JSON. В листинге 21.24 к представлению добавлен элемент `script`, содержащий данные JSON. Когда представление визуализируется и отправляется браузеру, оно включает элемент, подобный показанному ниже:

```
...
<script id="jsonData" type="application/json">["Apple","Orange","Pear"]
</script>
...
```

Чтобы можно было работать с данными JSON, к представлению `List.cshtml` добавляется библиотека jQuery и встраиваемый код JavaScript, который применяет ее для разбора данных JSON и динамического создания HTML-элементов (листинг 21.25).

Листинг 21.25. Использование данных JSON в файле `List.cshtml` 0 из папки `Views/Common`

```
@{ Layout = null; }

<!DOCTYPE html>
```

```
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Razor</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
  <script id="jsonData" type="application/json">
    @Json.Serialize(new string[] { "Apple", "Orange", "Pear" })
  </script>
  <script asp-src-include="lib/jquery/dist/*.min.js"></script>
  <script type="text/javascript">
    $(document).ready(function () {
      var list = $("#list")
      JSON.parse($("#jsonData").text()).forEach(function (val) {
        console.log("Val: " + val);
        list.append($("<li>").text(val));
      });
    });
  </script>
</head>
<body class="m-1 p-1">
  This is the List View
  <ul id="list"></ul>
</body>
</html>
```

После запуска приложения и запроса URL вида `/Home/List` отобразится содержимое, показанное на рис. 21.9. Не самый впечатляющий результат работы с данными JSON, но он демонстрирует, как их можно включать в представления.

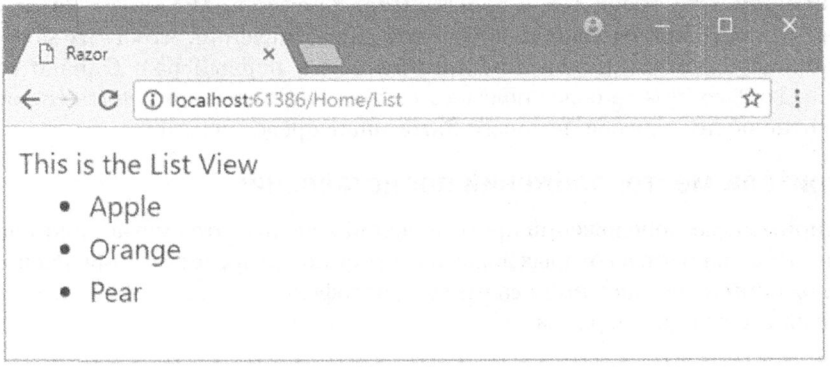


Рис. 21.9. Использование данных JSON в представлении

Конфигурирование механизма Razor

Механизм Razor можно конфигурировать с применением класса `RazorViewEngine Options`, который находится в пространстве имен `Microsoft.AspNetCore.Mvc. Razor`. Класс определяет два конфигурационных свойства, описанные в табл. 21.9.

Таблица 21.9. Свойства класса `RazorViewEngineOptions`

Имя	Описание
`FileProvider`	Это свойство используется для установки объекта, который предоставляет механизму Razor содержимое файлов и каталогов. Функциональность определяется интерфейсом `Microsoft.AspNetCore.FileProviders.IFileProvider`, а стандартной реализацией является класс `PhysicalFileProvider`, который обеспечивает чтение файлов из диска
`ViewLocationExpanders`	Это свойство применяется при конфигурировании расширителей местоположений представлений, которые используются для изменения способа поиска представления механизмом Razor

Совет. Если вы действительно заинтересованы в тщательном исследовании, тогда можете заменить внутренние компоненты Razor, создав классы, которые реализуют интерфейсы из пространства имен `Microsoft.AspNetCore.Mvc.Razor`, и зарегистрировав их с помощью поставщика служб в классе `Startup`. У большинства разработчиков никогда не возникнет необходимость выполнять такую работу, и к ней нельзя относиться легкомысленно, но это удобный вариант, когда желательно иметь полный контроль над обработкой содержимого внутри приложения. Для начала потребуется загрузить исходный код Razor из `https://github.com/aspnet/`.

Свойство `FileProvider` не относится к тем свойствам, которые будут изменять многие приложения, поскольку чтение файлов представлений из диска является в точности тем, что требуется в большинстве проектов. Механизм Razor использует данный поставщик только для загрузки представлений, чтобы их можно было скомпилировать, когда приложение запускается в первый раз. Однако свойство `ViewLocationExpanders` более полезно, т.к. оно позволяет применять специальную логику в отношении способа, которым Razor ищет представления.

Расширители местоположений представлений

Расширители местоположений представлений используются механизмом Razor для построения списка мест, в которых должен вестись поиск представления. Расширители местоположений представлений реализуют интерфейс `IViewLocationExpander`, определенный следующим образом:

```
using System.Collections.Generic;

namespace Microsoft.AspNetCore.Mvc.Razor {

  public interface IViewLocationExpander {

    void PopulateValues(ViewLocationExpanderContext context);

    IEnumerable<string> ExpandViewLocations(
      ViewLocationExpanderContext context,
      IEnumerable<string> viewLocations);
  }
}
```

В приведенных далее разделах объясняется, как работают расширители местоположений представлений, и создается специальная реализация интерфейса IViewLocationExpander. Для подготовки к созданию расширителей местоположений представлений потребуется изменить метод действия Index() контроллера Home, чтобы он запрашивал несуществующее представление (листинг 21.26). В результирующем сообщении об ошибке будут присутствовать местоположения, в которых механизм Razor искал представление, и отражен эффект, оказываемый на них расширителями местоположений представлений.

Листинг 21.26. Запрашивание несуществующего представления в файле HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
namespace Views.Controllers {
  public class HomeController : Controller {
    public ViewResult Index() =>
      View("MyView", new string[] { "Apple", "Orange", "Pear" });
    public ViewResult List() => View();
  }
}
```

Запуск приложения и запрос стандартного URL позволяет увидеть, что в сообщении об ошибке отображаются стандартные местоположения поиска представления:

```
/Views/Home/MyView.cshtml
/Views/Shared/MyView.cshtml
```

Создание простого расширителя местоположений представлений

Простейший расширитель местоположений представлений всего лишь изменяет набор мест, где Razor проводит поиск всех представлений. Для этого необходимо реализовать метод ExpandViewLocations() и возвратить список местоположений, которые должны поддерживаться. Добавим в папку Infrastructure файл класса по имени SimpleExpander.cs и поместим в него определение, показанное в листинге 21.27.

Листинг 21.27. Содержимое файла SimpleExpander.cs из папки Infrastructure

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc.Razor;
namespace Views.Infrastructure {
  public class SimpleExpander : IViewLocationExpander {
    public void PopulateValues(ViewLocationExpanderContext context) {
      // Ничего не делать - метод не требуется.
    }
    public IEnumerable<string> ExpandViewLocations(
          ViewLocationExpanderContext context,
          IEnumerable<string> viewLocations) {
      foreach (string location in viewLocations) {
        yield return location.Replace("Shared", "Common");
      }
      yield return "/Views/Legacy/{1}/{0}/View.cshtml";
    }
  }
}
```

Механизм Razor вызывает метод ExpandViewLocations(), когда ему требуется список местоположений поиска, и предоставляет стандартные местоположения как последовательность строк в параметре viewLocations. Местоположения выражаются в виде шаблонов с заполнителями для имени действия и имени контроллера. Вот как выглядят шаблоны местоположений, которые применяются по умолчанию в приложении, не использующем области маршрутизации:

```
"/Views/{1}/{0}.cshtml"
"/Views/Shared/{0}.cshtml"
```

Заполнитель {0} применяется для ссылки на имя метода действия, а заполнитель {1} — для ссылки на имя контроллера. Работа расширителя местоположений представлений заключается в возвращении набора мест, где должен производиться поиск, и в листинге 21.27 используется метод string.Replace() для изменения Shared на Common в стандартных местоположениях, а также добавляется собственное местоположение с другой структурой файлов и папок.

Применение расширителя местоположений представлений

В листинге 21.28 расширитель местоположений представлений настраивается путем конфигурирования механизма Razor в классе Startup. Свойство ViewLocationExpanders возвращает объект List<IViewLocationExpander>, на котором вызывается метод Add().

Листинг 21.28. Конфигурирование механизма Razor в файле Startup.cs из папки Views

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;
using Microsoft.AspNetCore.Mvc.Razor;

namespace Views {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.Configure<RazorViewEngineOptions>(options => {
        options.ViewLocationExpanders.Add(new SimpleExpander());
      });
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

После запуска приложения сообщение об ошибке отобразит набор местоположений, который был предоставлен механизму Razor специальным расширителем местоположений представлений:

```
/Views/Home/MyView.cshtml
/Views/Common/MyView.cshtml
/Views/Legacy/Home/MyView/View.cshtml
```

Выбор специфических представлений для запросов

Расширители местоположений представлений облегчают изменение мест поиска для всех запросов, но могут также делать это для индивидуальных запросов. В предыдущем примере был реализован только метод `ExpandViewLocations()`, но реальную мощь демонстрирует метод `PopulateValues()`, являющийся еще одним методом в интерфейсе `IViewLocationExpander`.

Всякий раз, когда механизму Razor требуется представление, он вызывает метод `PopulateValues()` своих расширителей местоположений представлений, передавая ему объект `ViewLocationExpanderContext` для данных контекста.

В классе `ViewLocationExpanderContext` определены свойства, перечисленные в табл. 21.10.

Таблица 21.10. Свойства класса `ViewLocationExpanderContext`

Имя	Описание
`ActionContext`	Это свойство возвращает объект `ActionContext`, который описывает метод действия, запросивший представление, а также включает детали запроса и ответа
`ViewName`	Это свойство возвращает имя представления, которое запросил метод действия
`ControllerName`	Это свойство возвращает имя контроллера, который содержит метод действия
`AreaName`	Это свойство возвращает имя области, содержащей контроллер, если области были определены
`IsMainPage`	Это свойство возвращает `false`, если механизм Razor ищет частичное представление, и `true` в противном случае
`Values`	Это свойство возвращает объект `IDictionary<string, string>`, к которому расширитель местоположений представлений добавляет пары "ключ-значение", уникально идентифицирующие категорию запроса, как объясняется ниже

Метод `PopulateValues()` предназначен для категоризации запроса за счет добавления пар "ключ-значение" в словарь, возвращаемый свойством `Values` объекта контекста. Механизм Razor не беспокоит то, каким образом категоризируется запрос, и реализация метода для заполнения словаря возложена полностью на расширитель местоположений представлений. Легче всего это объяснить с помощью примера, так что добавим в папку `Infrastructure` файл класса по имени `ColorExpander.cs` с определением из листинга 21.29.

Листинг 21.29. Содержимое файла `ColorExpander.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc.Razor;

namespace Views.Infrastructure {

  public class ColorExpander : IViewLocationExpander {
    private static Dictionary<string, string> Colors
      = new Dictionary<string, string> {
       ["red"] = "Red", ["green"] = "Green", ["blue"] = "Blue"
      };

    public void PopulateValues(ViewLocationExpanderContext context) {

      var routeValues = context.ActionContext.RouteData.Values;
      string color;

      if (routeValues.ContainsKey("id")
          && Colors.TryGetValue(routeValues["id"] as string, out color)
          && !string.IsNullOrEmpty(color)) {
        context.Values["color"] = color;
      }
    }

    public IEnumerable<string> ExpandViewLocations(
           ViewLocationExpanderContext context,
           IEnumerable<string> viewLocations) {

      string color;
      context.Values.TryGetValue("color", out color);
      foreach (string location in viewLocations) {
        if (!string.IsNullOrEmpty(color)) {
          yield return location.Replace("{0}", color);
        } else {
          yield return location;
        }
      }
    }
  }
}
```

Метод `PopulateValues()` использует объект `ActionContext` для получения данных маршрутизации и ищет значение сегмента `id` в URL. Если есть сегмент `id` и его значением является `red`, `green` или `blue`, тогда расширитель местоположений представлений добавляет в словарь `Values` свойство `color`. Мы имеем дело с процессом категоризации: запрос, сегмент `id` которого соответствует цвету, категоризируется с ключом `color`, чье значение выводится из значения сегмента.

Далее механизм Razor вызывает метод `ExpandViewLocations()` и предоставляет тот же объект контекста, который применялся для метода `PopulateValues()`. Это позволяет расширителю местоположений представлений просматривать проведенную ранее категоризацию и генерировать набор мест, в которых механизм Razor должен искать представления. В рассматриваемом примере с помощью метода `string.Replace()` заполнитель `{0}` заменялся именем цвета.

Совет. Механизм Razor вызывает метод `PopulateValues()` для каждого запроса представления, но кеширует набор местоположений поиска, возвращаемый методом `ExpandViewLocations()`. Таким образом, последующие запросы, для которых метод `PopulateValues()` генерирует тот же самый набор ключей и значений категоризации, не требует вызова метода `ExpandViewLocations()`.

В листинге 21.30 механизм Razor конфигурируется для использования класса `ColorExpander`.

Листинг 21.30. Добавление расширителя местоположений представлений в файле `Startup.cs` из папки `Views`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc;
using Views.Infrastructure;
using Microsoft.AspNetCore.Mvc.Razor;

namespace Views {

  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.Configure<RazorViewEngineOptions>(options => {
        options.ViewLocationExpanders.Add(new SimpleExpander());
        options.ViewLocationExpanders.Add(new ColorExpander());
      });
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Запустив приложение и запросив URL вида `/Home/Index/red`, можно заметить эффект от нового расширителя местоположений представлений, который приведет к тому, что механизм Razor будет выполнять поиск в следующих местах:

```
/Views/Home/Red.cshtml
/Views/Common/Red.cshtml
/Views/Legacy/Home/Red/View.cshtml
```

Аналогично запрос URL вида `/Home/Index/green` заставит Razor искать в таких местоположениях:

```
/Views/Home/Green.cshtml
/Views/Common/Green.cshtml
/Views/Legacy/Home/Green/View.cshtml
```

Порядок, в котором регистрируются расширители местоположений представлений, важен, потому что набор местоположений, генерируемый методом `ExpandViewLocations()` одного расширителя, применяется в качестве аргумента `viewLocations` для следующего расширителя в списке. Это можно заметить в показанных ранее местоположениях, где `Views/Common` и `Views/Legacy` генерировались классом `SimpleExpander`, который регистрировался перед `ColorExpander` в классе `Startup`.

Резюме

В главе было продемонстрировано, как создавать специальный механизм визуализации, и объяснялось, каким образом работает механизм Razor при трансляции файлов CSHTML в классы C#. Вы узнали, как использовать разделы компоновки и частичные представления, и научились изменять местоположения, в которых Razor ищет файлы представлений. В следующей главе будут описаны компоненты представлений, которые применяются для обеспечения логики поддержки частичных представлений.

ГЛАВА 22

Компоненты представлений

В настоящей главе рассматриваются *компоненты представлений*, которые являются новым добавлением к инфраструктуре ASP.NET Core MVC и заменяют собой средство дочерних действий из предшествующих версий. Компоненты представлений — это классы, которые для поддержки частичных представлений предлагают логику в стиле действий, что позволяет возможность встраивать в представления сложное содержимое и одновременно сохранять поддерживающий его код C# простым в плане сопровождения и модульного тестирования. В табл. 22.1 приведена сводка, позволяющая поместить компоненты представлений в контекст.

Таблица 22.1. Помещение компонентов представлений в контекст

Вопрос	Ответ
Что это такое?	Компоненты представлений являются классами, которые обеспечивают прикладную логику для поддержки частичных представлений либо для внедрения небольших фрагментов HTML-разметки или данных JSON в родительское представление
Чем они полезны?	Без компонентов представлений трудно создавать встроенную функциональность наподобие корзины для покупок или панели входа таким способом, который допускает легкое сопровождение и модульное тестирование
Как они используются?	Компоненты представлений обычно являются производными от класса `ViewComponent` и применяются в дочернем представлении с использованием выражения `@await Component.InvokeAsync()`
Существуют ли какие-то скрытые ловушки или ограничения?	Нет, компоненты представлений — простое и предсказуемое средство. Основное заблуждение связано с отказом от их применения и попыткой помещать прикладную логику внутрь представлений, где ее трудно тестировать и сопровождать
Существуют ли альтернативы?	Логику доступа и обработки данных можно было бы поместить прямо в частичное представление, но в результате работа с ней и эффективное тестирование станут значительно труднее

В табл. 22.2 приведена сводка по главе.

Таблица 22.2. Сводка по главе

Задача	Решение	Листинг
Создание частичного представления с собственной логикой и данными	Используйте компонент представления	22.12
Вызов компонента представления	Применяйте выражение `@await Component.InvokeAsync()` в представлении	22.13
Упрощение доступа к данным контекста и результатам	Унаследуйте класс от класса `ViewComponent`	22.14–22.16
Выбор частичного представления	Используйте метод `View()` для создания и возвращения объекта `ViewViewComponentResult`	22.17–22.19
Создание фрагмента HTML-разметки	Вызовите метод `Content()` для создания объекта `ContentViewComponentResult` или явно создайте объект `HtmlContentViewComponentResult`, если не хотите, чтобы фрагмент был закодированным	22.20, 22.21
Использование деталей запроса для генерирования результата	Применяйте данные контекста компонента представления	22.22
Предоставление данных контекста при вызове компонента представления	Передайте аргументы методу `InvokeAsync()`	22.23–22.25
Создание асинхронного компонента представления	Реализуйте метод `InvokeAsync()` и возвратите объект `Task`, который выдает требуемый результат	22.26–22.29
Создание гибрида контроллера и компонента представления	Применяйте атрибут `ViewComponent` к классу контроллера	22.30–22.33

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `UsingViewComponents` с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core.

Создание моделей и хранилищ

Для демонстрации работы компонентов представлений понадобятся два источника данных. Часть приложения будет оперировать с набором описаний товаров, а потому создадим папку `Models` и добавим в нее файл класса по имени `Product.cs` с определением из листинга 22.1.

Листинг 22.1. Содержимое файла `Product.cs` из папки `Models`

```
namespace UsingViewComponents.Models {
  public class Product {
```

```
    public string Name { get; set; }
    public decimal Price { get; set; }
  }
}
```

Чтобы создать хранилище для объектов `Product`, добавим в папку `Models` файл по имени `ProductRepository.cs` и определим в нем интерфейс и класс реализации, как показано в листинге 22.2.

Листинг 22.2. Содержимое файла `ProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace UsingViewComponents.Models {

  public interface IProductRepository {
    IEnumerable<Product> Products { get; }
    void AddProduct(Product newProduct);
  }

  public class MemoryProductRepository : IProductRepository {
    private List<Product> products = new List<Product> {
      new Product { Name = "Kayak", Price = 275M },
      new Product { Name = "Lifejacket", Price = 48.95M },
      new Product { Name = "Soccer ball", Price = 19.50M }
    };

    public IEnumerable<Product> Products => products;

    public void AddProduct(Product newProduct) {
      products.Add(newProduct);
    }
  }
}
```

В интерфейсе `IProductRepository` определен ограниченный набор функциональных средств хранилища, а класс `MemoryProductRepository` реализует данный интерфейс с применением объекта `List`, находящегося в памяти.

Другая часть приложения будет оперировать с описаниями городов. С этой целью добавим в папку `Models` файл класса по имени `City.cs`, содержимое которого приведено в листинге 22.3.

Листинг 22.3. Содержимое файла `City.cs` из папки `Models`

```
namespace UsingViewComponents.Models {

  public class City {
    public string Name { get; set; }
    public string Country { get; set; }
    public int Population { get; set; }
  }
}
```

Для хранилища объектов `City` создадим в папке `Models` файл класса по имени `CityRepository.cs` и определим в нем интерфейс и класс реализации, как показано в листинге 22.4.

Листинг 22.4. Содержимое файла `CityRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace UsingViewComponents.Models {

  public interface ICityRepository {
    IEnumerable<City> Cities { get; }

    void AddCity(City newCity);
  }

  public class MemoryCityRepository : ICityRepository {
    private List<City> cities = new List<City> {
      new City { Name = "London", Country = "UK", Population = 8539000},
      new City { Name = "New York", Country = "USA", Population = 8406000 },
      new City { Name = "San Jose", Country = "USA", Population = 998537 },
      new City { Name = "Paris", Country = "France", Population = 2244000 }
    };

    public IEnumerable<City> Cities => cities;
    public void AddCity(City newCity) {
      cities.Add(newCity);
    }
  }
}
```

Интерфейс `ICityRepository` предлагает ограниченный набор функциональных средств хранилища, а класс `MemoryCityRepository` реализует данный интерфейс с использованием объекта `List`, находящегося в памяти.

Создание контроллера и представлений

Для начала необходим только один контроллер, поэтому создадим папку `Controllers`, добавим в нее файл по имени `HomeController.cs` и определим в нем класс, приведенный в листинге 22.5.

Листинг 22.5. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;

namespace UsingViewComponents.Controllers {

  public class HomeController : Controller {
    private IProductRepository repository;
    public HomeController(IProductRepository repo) {
      repository = repo;
    }
    public ViewResult Index() => View(repository.Products);
    public ViewResult Create() => View();
    [HttpPost]
    public IActionResult Create(Product newProduct) {
      repository.AddProduct(newProduct);
      return RedirectToAction("Index");
    }
  }
}
```

Контроллер Home применяет свой конструктор для объявления зависимости от интерфейса IProductRepository, которая будет распознаваться поставщиком служб, когда контроллер станет использоваться при обработке запросов. Действие Index извлекает все объекты Product из хранилища и визуализирует их с применением стандартного представления. Два метода Create() задействуют паттерн Post/Redirect/Get при добавлении новых объектов в хранилище с использованием данных формы, передаваемых клиентом.

Представления в рассматриваемом примере будут разделять общую компоновку. Создадим папку Views/Shared и добавим в нее файл по имени _Layout.cshtml с разметкой из листинга 22.6.

Листинг 22.6. Содержимое файла _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div class="bg-primary m-1 p-1">
    <div class="row text-white">
      <div class="col-7"><h1>Products</h1></div>
      <div class="col-5">
        <div class="bg-info text-center m-1 p-1">City Placeholder</div>
      </div>
    </div>
  </div>
  <div class="m-1 p-1">@RenderBody()</div>
</body>
</html>
```

В компоновке определен заголовок, включающий заполнитель для содержимого, которое будет создано позже в главе с применением хранилища городов. Далее создадим папку Views/Home и поместим в нее файл по имени Index.cshtml с разметкой, которая выводит детали объектов Product в таблице (листинг 22.7).

Листинг 22.7. Содержимое файла Index.cshtml из папки Views/Home

```
@model IEnumerable<Product>
@{
  ViewData["Title"] = "Products";
  Layout = "_Layout";
}
<table class="table table-sm table-striped table-bordered">
  <thead>
    <tr><th>Name</th><th>Price</th></tr>
  </thead>
  <tbody>
    @foreach (var product in Model) {
```

```
      <tr>
        <td>@product.Name</td>
        <td>@product.Price</td>
      </tr>
    }
  </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create</a>
```

Финальным элементом в представлении `Index` является элемент `a`, который стилизован как кнопка и нацелен на действие `Create`, так что пользователь может создавать новый объект `Product` в хранилище. Чтобы предоставить форму, которую будет заполнять пользователь, добавим в папку `Views/Home` файл `Create.cshtml` с разметкой из листинга 22.8.

Листинг 22.8. Содержимое файла `Create.cshtml` из папки `Views/Home`

```
@model Product
@{
  ViewData["Title"] = "Create Product";
  Layout = "_Layout";
}
<form method="post" asp-action="Create">
  <div class="form-group">
    <label asp-for="Name">Name:</label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label asp-for="Price">Price:</label>
    <input class="form-control" asp-for="Price" />
  </div>
  <button type="submit" class="btn btn-primary">Create</button>
  <a class="btn btn-secondary" asp-action="Index">Cancel</a>
</form>
```

В представлениях используются встроенные вспомогательные функции дескрипторов. Чтобы сделать их доступными, создадим в папке `Views` файл `_ViewImports.cshtml` и добавим в него выражения, показанные в листинге 22.9, которые также обеспечивают доступ к классам из папки `Models` без указания пространства имен.

Листинг 22.9. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@using UsingViewComponents.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Представления также полагаются на CSS-пакет Bootstrap для стилизации своего содержимого. Создадим в корневой папке проекта файл `bower.json` с применением шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел `dependencies` (листинг 22.10).

Листинг 22.10. Добавление пакета Bootstrap в файле `bower.json` из папки `UsingViewComponents`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Конфигурирование приложения

Последний подготовительный шаг связан с конфигурированием приложения, как демонстрируется в листинге 22.11. В дополнение к настройке служб MVC и промежуточного ПО создаются службы-одиночки для двух хранилищ данных.

Листинг 22.11. Содержимое файла `Startup.cs` из папки `UsingViewComponents`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using UsingViewComponents.Models;

namespace UsingViewComponents {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IProductRepository, MemoryProductRepository>();
      services.AddSingleton<ICityRepository, MemoryCityRepository>();
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Запуск приложения приводит к отображению списка объектов `Product` из хранилища товаров. Приложение позволяет добавлять новые товары, щелкая на кнопке Create (Создать), заполняя форму и отправляя ее серверу, который затем перенаправит браузер обратно на список (рис. 22.1). Поскольку представления в приложении разделяют общую компоновку, в течение всего процесса виден заполнитель для данных по городам.

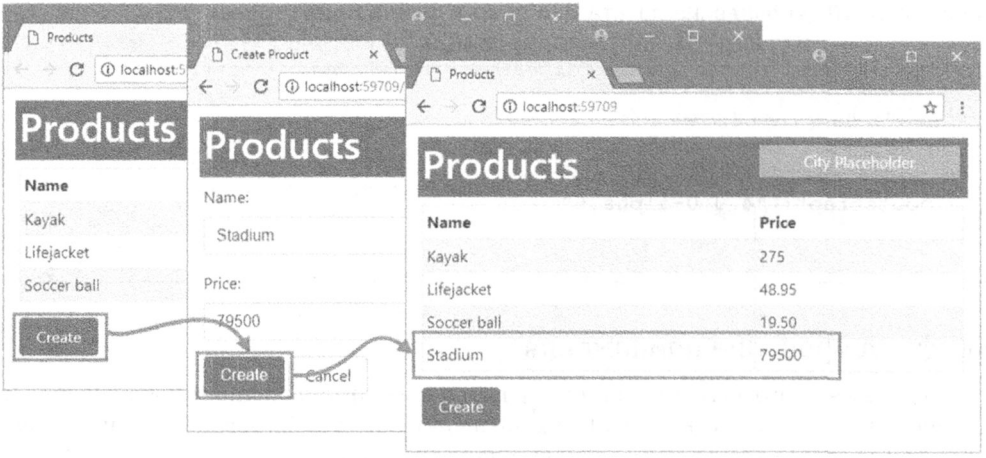


Рис. 22.1. Выполнение примера приложения

Понятие компонентов представлений

Приложениям обычно необходимо встраивать в представления содержимое, которое не связано с главным назначением приложений. Распространенные примеры включают инструменты навигации по сайту и панели аутентификации, которые позволяют пользователю входить, не посещая отдельную страницу.

Общая идея всех упомянутых примеров состоит в том, что данные, требуемые для отображения встроенного содержимого, не являются частью данных модели, которые передаются из действия в представление. Именно по этой причине в примере приложения были созданы два хранилища: мы собираемся отображать содержимое, генерируемое с использованием хранилища объектов `City`, что нелегко делать в представлении, которое получает от своих действий данные из хранилища объектов `Product`.

В главе 21 объяснялось, как применять частичные представления для создания многократно используемой разметки, которая требуется в приложениях, избегая дублирования одного и того же содержимого во множестве мест приложения. Частичные представления — удобное средство, но они просто содержат фрагменты HTML-разметки и директивы Razor, а данные, с которыми они имеют дело, получаются от родительского представления. Если нужно отображать другие данные, тогда возникает проблема. Можно было бы получать доступ к необходимым данным прямо из частичного представления, но такой подход нарушит принцип разделения обязанностей, являющийся фундаментом паттерна MVC, и его результатом окажется нахождение логики извлечения и обработки данных в файле представления, где она не может быть подвергнута модульному тестированию. Альтернативно можно было бы расширить применяемые в приложении модели представлений, чтобы включить требующиеся данные, но тогда пришлось бы изменить каждый метод действия и тем самым затруднить изолирование функциональности методов действий в целях эффективного тестирования.

Здесь на помощь приходят компоненты представлений. Компонент представления — это класс C#, который предлагает частичное представление с необходимыми ему данными, независимое от родительского представления и от визуализирующего действия. В таком отношении компонент представления можно трактовать как спе-

циализированное действие, которое используется только для доставки частичного представления с данными; оно не может получать HTTP-запросы, а выдаваемое им содержимое будет всегда включаться в родительское представление.

Создание компонента представления

Компоненты представлений можно создавать тремя способами: определяя компонент представления POCO, наследуя от базового класса `ViewComponent` и применяя атрибут `ViewComponent`. Приемы с классом POCO и базовым классом описаны в последующих разделах, а использование атрибута `ViewComponent` объясняется в разделе "Создание гибридных классов контроллеров и компонентов представлений" позже в главе.

Создание компонентов представлений POCO

Компонент представления POCO является классом, который поддерживает функциональность компонента представления, не полагаясь на какие-то API-интерфейсы MVC. Как и в случае контроллеров POCO, работать с таким видом компонента представления неудобно, но полезно знать особенности его функционирования. Компонентом представления POCO будет любой класс с именем, заканчивающимся на `ViewComponent`, в котором определен метод `Invoke()`. Классы компонентов представлений могут определяться где угодно в приложении, но по соглашению они собираются вместе в папке по имени `Components`, расположенной на корневом уровне проекта. Создадим упомянутую папку и добавим в нее файл класса `PocoViewComponent.cs` с определением, приведенным в листинге 22.12.

Листинг 22.12. Содержимое файла `PocoViewComponent.cs` из папки `Components`

```
using System.Linq;
using UsingViewComponents.Models;

namespace UsingViewComponents.ViewComponents {

  public class PocoViewComponent {
    private ICityRepository repository;

    public PocoViewComponent(ICityRepository repo) {
      repository = repo;
    }

    public string Invoke() {
      return $"{repository.Cities.Count()} cities, "
          + $"{repository.Cities.Sum(c => c.Population)} people";
    }
  }
}
```

Для получения требуемых служб компоненты представлений могут задействовать средство внедрения зависимостей. В рассматриваемом примере компонент представления POCO объявляет зависимость от интерфейса `ICityRepository`, реализация которого затем применяется в методе `Invoke()` при создании объекта `string`, описывающего количество городов и суммарное число жителей.

Для использования компонента представления требуется Razor-выражение `@await Component.InvokeAsync()`. Компонент представления выбирается за счет

указания в аргументе имени класса без окончания `ViewComponent`. В листинге 22.13 из разделяемой компоновки удаляется заполнитель, а взамен применяется компонент представления POCO.

Листинг 22.13. Применение компонента представления в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div class="bg-primary m-1 p-1">
    <div class="row text-white">
      <div class="col-7"><h1>Products</h1></div>
      <div class="col-5">
        @await Component.InvokeAsync("Poco")
      </div>
    </div>
  </div>
  <div class="m-1 p-1">@RenderBody()</div>
</body>
</html>
```

Чтобы применить компонент представления, методу `Invoke()` в качестве аргумента передается имя `Poco`. Когда представление использует компоновку, оно находит класс `PocoViewController`, вызывает его метод `Invoke()` и вставляет результат в вывод родительского представления (рис. 22.2).

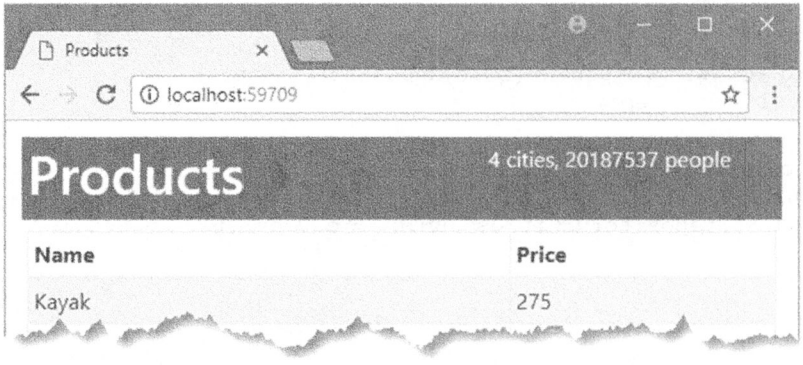


Рис. 22.2. Применение простого компонента представления

Хотя пример довольно прост, он иллюстрирует ряд важных характеристик компонентов представлений. Во-первых, класс `PocoViewComponent` смог получить доступ к требующимся ему данным без какой-либо зависимости от обработки HTTP-запроса или от его родительского представления. Во-вторых, определение логики, необходимой для получения и обработки сводки по городам, в классе C# означает, что она

может быть легко подвергнута модульному тестированию (за примером обращайтесь к врезке "Модульное тестирование компонентов представлений" далее в главе). В-третьих, форма приложения не нарушается из-за попытки включить объекты `City` в модели представлений, которые ориентированы на объекты `Product`. Словом, компонент представления является самодостаточной порцией многократно используемой функциональности, которая может применяться повсюду в приложении, к тому же разрабатываться и тестироваться в изоляции.

Внимание! Когда компонент представления применяется в представлении, должно быть указано ключевое слово `await`. В случае простого вызова `@Component.Invoke()` ошибка не возникнет, но отобразится строковое представление объекта `Task` такого вида: `System.Threading.Tasks.Task`1[Microsoft.AspNetCore.Html.IHtmlContent]`.

Наследование от базового класса `ViewComponent`

Компоненты представлений POCO ограничены в функциональности, если только не задействован API-интерфейс MVC, что вполне возможно, но требует гораздо больших усилий, чем другой распространенный подход, предусматривающий наследование от класса `ViewComponent`. Класс `ViewComponent`, определенный в пространстве имен `Microsoft.AspNetCore.Mvc`, обеспечивает удобный доступ к данным контекста и облегчает генерацию результатов. В листинге 22.14 приведено содержимое файла `CitySummary.cs`, добавленного в папку `Components`.

Листинг 22.14. Содержимое файла `CitySummary.cs` из папки `Components`

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;

namespace UsingViewComponents.Components {

  public class CitySummary : ViewComponent {
    private ICityRepository repository;

    public CitySummary(ICityRepository repo) {
      repository = repo;
    }

    public string Invoke() {
      return $"{repository.Cities.Count()} cities, "
          + $"{repository.Cities.Sum(c => c.Population)} people";
    }
  }
}
```

При наследовании от базового класса `ViewComponent` указывать суффикс `ViewComponent` в имени класса не обязательно. За исключением использования базового класса этот компонент представления функционально идентичен версии POCO. В последующих разделах будет показано, как применять удобные средства, предлагаемые базовым классом для работы с различными функциями компонентов представлений.

Совет. Обратите внимание, что в листинге 22.14 метод `Invoke()` не переопределялся. Класс `ViewComponent` не предоставляет стандартной реализации метода `Invoke()`, который должен быть определен явным образом.

В рамках подготовки к демонстрации функциональных средств компонентов представлений изменим компонент, используемый в разделяемой компоновке, как показано в листинге 22.15. Вместо литеральной строки с именем компонента представления можно применять выражение `nameof`, описанное в главе 4, что сократит шансы неправильно набора имени класса.

Листинг 22.15. Изменение компонента представления в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div class="bg-primary m-1 p-1">
    <div class="row text-white">
      <div class="col-7"><h1>Products</h1></div>
      <div class="col-5">
        @await Component.InvokeAsync(nameof(CitySummary))
      </div>
    </div>
  </div>
  <div class="m-1 p-1">@RenderBody()</div>
</body>
</html>
```

Чтобы в выражении `nameof` можно было ссылаться на класс `CitySummary` без указания его пространства имен, внесем в файл импортирования представлений изменение, приведенное в листинге 22.16.

Листинг 22.16. Добавление пространства имен в файле `_ViewImports.cshtml` из папки `Views`

```
@using UsingViewComponents.Models
@using UsingViewComponents.Components
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Понятие результатов компонентов представлений

Возможность вставки в родительское представление простых строковых значений не особенно полезна, но к счастью компоненты представлений способны на гораздо большее. Более сложных эффектов можно достичь, заставив метод `Invoke()` возвращать объект, который реализует интерфейс `IViewComponentResult`. В табл. 22.3 описаны три встроенных класса, которые реализуют интерфейс `IViewComponentResult`,

и удобные методы для создания их экземпляров, предоставляемые базовым классом ViewComponent. Использование каждого типа результата рассматривается в последующих разделах.

На заметку! В случае применения компонентов представлений POCO создавать экземпляры классов результатов допускается напрямую, хотя работать с ними может быть неудобно, т.к. они имеют сложные аргументы конструкторов, которые удобные методы из класса ViewComponent предоставляют самостоятельно.

Таблица 22.3. Встроенные классы реализации IViewComponentResult

Имя	Описание
ViewViewComponentResult	Этот класс используется для указания представления Razor с дополнительными данными модели представления. Экземпляры данного класса создаются с применением метода View()
ContentViewComponentResult	Этот класс используется для указания текстового результата, который будет безопасно закодирован с целью включения в HTML-документ. Экземпляры данного класса создаются с применением метода Content()
HtmlContentViewComponentResult	Этот класс используется для указания фрагмента HTML-разметки, которая будет включена в HTML-документ без добавочного кодирования. Для создания такого типа результата методы в классе ViewComponent не предусмотрены

Два типа результатов обрабатываются специальным образом. Если компонент представления возвращает объект string, тогда он используется для создания объекта ContentViewComponentResult, на который полагались предшествующие примеры. Если компонент представления возвращает объект реализации IHtmlContent, то он применяется для создания объекта HtmlContentViewComponentResult.

Возвращение частичного представления

Самым полезным ответом является неуклюже именованный объект ViewViewComponentResult, который сообщает механизму Razor о необходимости визуализации частичного представления и включения результата в родительское представление. Для создания объектов ViewViewComponentResult базовый класс ViewComponent предлагает метод View(), который имеет четыре версии (табл. 22.4).

Описанные в табл. 22.4 методы соответствуют методам, которые предоставляет базовый класс Controller, и используются аналогичным образом. Добавим в папку Models файл класса по имени CityViewModel.cs и определим в нем модель представления, как показано в листинге 22.17.

В листинге 22.18 приведен модифицированный метод Invoke() компонента представления CitySummary, который теперь применяет метод View() для выбора частичного представления и передачи ему данных представления, используя объект CityViewModel.

Таблица 22.4. Методы `ViewComponent.View()`

Имя	Описание
`View()`	В случае использования этого метода выбирается стандартное представление для компонента представления, а модель представления не указывается
`View(model)`	В случае применения этого метода выбирается стандартное представление, а указанный объект используется в качестве модели представления
`View(viewName)`	В случае применения этого метода выбирается указанное представление, а модель представления не задается
`View(viewName, model)`	В случае использования этого метода выбирается указанное представление, а заданный объект применяется в качестве модели представления

Листинг 22.17. Содержимое файла `CityViewModel.cs` из папки `Models`

```
namespace UsingViewComponents.Models {
  public class CityViewModel {
    public int Cities { get; set; }
    public int Population { get; set; }
  }
}
```

Листинг 22.18. Выбор частичного представления в файле `CitySummary.cs` из папки `Components`

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;

namespace UsingViewComponents.Components {

  public class CitySummary : ViewComponent {
    private ICityRepository repository;

    public CitySummary(ICityRepository repo) {
      repository = repo;
    }
    public IViewComponentResult Invoke() {
      return View(new CityViewModel{
        Cities = repository.Cities.Count(),
        Population = repository.Cities.Sum(c => c.Population)
      });
    }
  }
}
```

Выбор частичного представления в компоненте представления похож на выбор представления в контроллере, но с двумя важными отличиями: механизм Razor ищет представления в других местоположениях и применяет другое стандартное имя представления, если оно не указано.

Поскольку для компонента представления частичное представление не создавалось, после запуска приложения отобразится сообщение об ошибке, которое показывает, какие файлы искал механизм Razor:

- `/Views/Home/Components/CitySummary/Default.cshtml`
- `/Views/Shared/Components/CitySummary/Default.cshtml`

Если имя не указано, тогда Razor ищет файл `Default.cshtml`. В поисках частичного представления механизм Razor просматривает два местоположения. Первое местоположение учитывает имя контроллера, обрабатывающего HTTP-запрос, что позволяет каждому контроллеру иметь собственное представление. Второе местоположение разделяется между всеми контроллерами.

Совет. Обратите внимание, что компонент представления все же различает разделяемые частичные представления, а это значит, что компоненты представления не разделяют частичные представления. Такое поведение можно переопределить, включая путь в имя представления при вызове метода `View()`; таким образом, вызов `View("Views/Shared/Components/Common/Default.html")` переопределяет нормальные местоположения для поиска.

Чтобы завершить пример, создадим папку `Views/Home/Components/CitySummary` и добавим в нее новый файл по имени `Default.cshtml`, поместив в него разметку из листинга 22.19.

Листинг 22.19. Содержимое файла `Default.cshtml` из папки `Views/Home/Components/CitySummary`

```
@model CityViewModel

<table class="table table-sm table-bordered">
  <tr>
    <td>Cities:</td>
    <td class="text-right">
      @Model.Cities
    </td>
  </tr>
  <tr>
    <td>Population:</td>
    <td class="text-right">
      @Model.Population.ToString("#,###")
    </td>
  </tr>
</table>
```

Частичные представления для компонентов представлений работают таким же способом, как и для контроллеров. В данном случае было создано строго типизированное представление, которое ожидает объекта `CityViewModel` и отображает значения его свойств `Cities` и `Population` в таблице (рис. 22.3).

Возвращение фрагментов HTML-разметки

Класс `ContentViewComponentResult` применяется для включения фрагментов HTML-разметки в родительское представление, не используя представление.

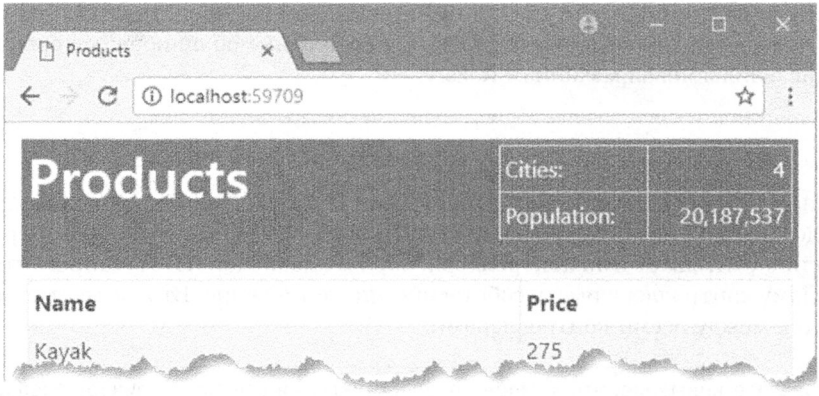


Рис. 22.3. Визуализация представления с использованием компонента представления

Экземпляры класса `ContentViewComponentResult` создаются с применением метода `Content()`, унаследованного от базового класса `ViewComponent`, который принимает значение `string`. Использование метода `Content()` демонстрируется в листинге 22.20. В дополнение к методу `Content()` возвращать `string` способен также метод `Invoke()`, и MVC будет автоматически преобразовывать это значение `string` в объект `ContentViewComponentResult`.

Листинг 22.20. Применение метода `Content()` в файле `CitySummary.cs` из папки `Components`

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;

namespace UsingViewComponents.Components {

  public class CitySummary : ViewComponent {
    private ICityRepository repository;

    public CitySummary(ICityRepository repo) {
      repository = repo;
    }

    public IViewComponentResult Invoke() {
      return Content("This is a <h3><i>string</i></h3>");
    }
  }
}
```

Полученная методом `Content()` строка кодируется, чтобы стать безопасной для включения в HTML-документ. Данный факт особенно важен при работе с содержимым, которое было предоставлено пользователями или внешними системами, т.к. предотвращает внедрение JavaScript-содержимого в HTML-разметку, генерируемую приложением. В текущем примере строка, которая передается методу `Content()`, содержит базовые HTML-дескрипторы, и после запуска приложения легко заметить, что они были безопасным образом закодированы (рис. 22.4).

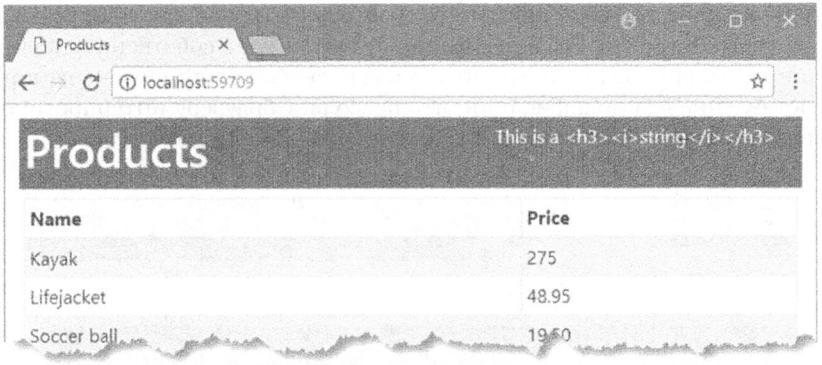


Рис. 22.4. Возвращение закодированного фрагмента HTML-разметки с использованием компонента представления

Взглянув на HTML-разметку, выпускаемую компонентом представления, можно увидеть, что символы угловых скобок были заменены так, что браузер не интерпретирует содержимое как HTML-элементы:

```
...
<div class="col-5">This is a &lt;h3&gt;&lt;i&gt;string&lt;/i&gt;&lt;/h3&gt;
</div>
...
```

Кодировать содержимое не понадобится, если вы доверяете источнику и хотите его интерпретировать как HTML-разметку. Метод `Content()` всегда кодирует свой аргумент. По указанной причине потребуется создать объект `HtmlContentViewComponentResult` напрямую и передать его конструктору объект `HtmlString`, представляющий строку, о которой известно, что отображать ее безопасно — либо потому, что она поступила из надежного источника, либо из-за того, что есть уверенность в том, что строка уже закодирована (листинг 22.21).

Листинг 22.21. Возвращение фрагмента надежной HTML-разметки в файле CitySummary.cs из папки Components

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Html;

namespace UsingViewComponents.Components {
  public class CitySummary : ViewComponent {
    private ICityRepository repository;

    public CitySummary(ICityRepository repo) {
      repository = repo;
    }

    public IViewComponentResult Invoke() {
      return new HtmlContentViewComponentResult(
        new HtmlString("This is a <h3><i>string</i></h3>"));
    }
  }
}
```

Показанный прием должен применяться осторожно и только с источниками содержимого, которые не могут быть подделаны и выполняют собственное кодирование. Запустив приложение, вы увидите, что угловые скобки были включены в родительское представление без модификации, что позволяет браузеру интерпретировать вывод компонента представления как HTML-элементы (рис. 22.5).

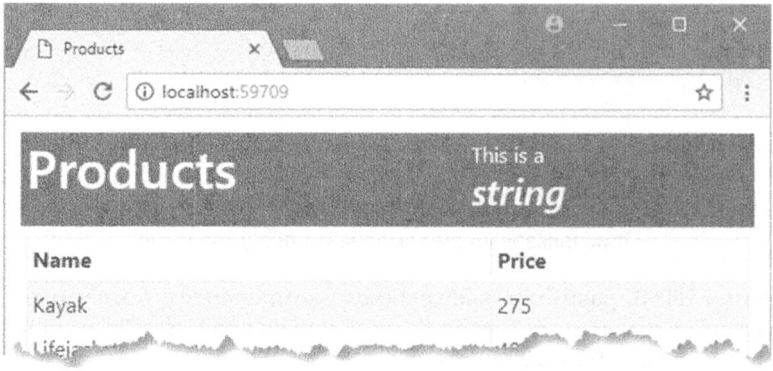


Рис. 22.5. Возвращение фрагмента незакодированной HTML-разметки с использованием компонента представления

Получение данных контекста

Детали о текущем запросе и родительском представлении компонент представления получает через свойства класса `ViewComponentContext`, наиболее полезные из которых описаны в табл. 22.5.

Таблица 22.5. Полезные свойства класса `ViewComponentContext`

Имя	Описание
`Arguments`	Это свойство возвращает словарь аргументов, предоставляемых представлением, который можно также получить через метод `Invoke()`
`HtmlEncoder`	Это свойство возвращает объект `HtmlEncoder`, который можно применять для безопасного кодирования фрагментов HTML-разметки
`ViewComponentDescriptor`	Это свойство возвращает объект `ViewComponentDescriptor`, который предоставляет описание компонента представления
`ViewContext`	Это свойство возвращает объект `ViewContext` из родительского представления. Функциональные средства класса `ViewContext` обсуждались в главе 21
`ViewData`	Это свойство возвращает объект `ViewDataDictionary`, который открывает доступ к данным представления, предназначенным для компонента представления

Базовый класс `ViewComponent` предлагает набор удобных свойств, которые облегчают доступ к специфической информации контекста (табл. 22.6).

Таблица 22.6. Удобные свойства класса `ViewComponent`

Имя	Описание
`ViewComponentContext`	Это свойство возвращает объект `ViewComponentContext`
`HttpContext`	Это свойство возвращает объект `HttpContext`, который описывает текущий запрос и подготавливаемый ответ
`Request`	Это свойство возвращает объект `HttpRequest`, который описывает текущий HTTP-запрос
`User`	Это свойство возвращает объект реализации `IPrincipal`, который описывает текущего пользователя (глава 28)
`RouteData`	Это свойство возвращает объект `RouteData`, который описывает данные маршрутизации для текущего запроса (глава 15)
`ViewBag`	Это свойство возвращает динамический объект `ViewBag`, который можно использовать для передачи данных между компонентом представления и представлением
`ModelState`	Это свойство возвращает объект `ModelStateDictionary`, который предоставляет детали процесса привязки моделей (глава 26)
`ViewContext`	Это свойство возвращает объект `ViewContext`, который был передан родительскому представлению (глава 21)
`ViewData`	Это свойство возвращает объект `ViewDataDictionary`, который обеспечивает доступ к данным представления для компонента представления
`Url`	Это свойство возвращает объект реализации `IUrlHelper`, который можно применять для генерации URL, как объяснялось в главе 15

Данные контекста могут использоваться любым способом, который помогает компоненту представления выполнять свою работу, включая варьирование метода выбора данных или визуализацию другого содержимого либо представлений.

В листинге 22.22 данные маршрутизации применяются для сужения выборки объектов `City`.

Листинг 22.22. Использование данных контекста в файле `CitySummary.cs` из папки `Components`

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace UsingViewComponents.Components {

  public class CitySummary : ViewComponent {
    private ICityRepository repository;

    public CitySummary(ICityRepository repo) {
      repository = repo;
    }
```

```
    public IViewComponentResult Invoke() {
      string target = RouteData.Values["id"] as string;
      var cities = repository.Cities
        .Where(city => target == null ||
          string.Compare(city.Country, target, true) == 0);
      return View(new CityViewModel{
        Cities = cities.Count(),
        Population = cities.Sum(c => c.Population)
      });
    }
  }
}
```

Браузер применяет сегмент `id` из маршрута для указания страны, которая используется LINQ при фильтрации объектов в хранилище. В результате запуска приложения и запрашивания стандартного URL отобразится информация по всем городам. Сузить выборку можно путем запрашивания URL вроде `/Home/Index/USA`, который ограничит выбор городами в США (рис. 22.6).

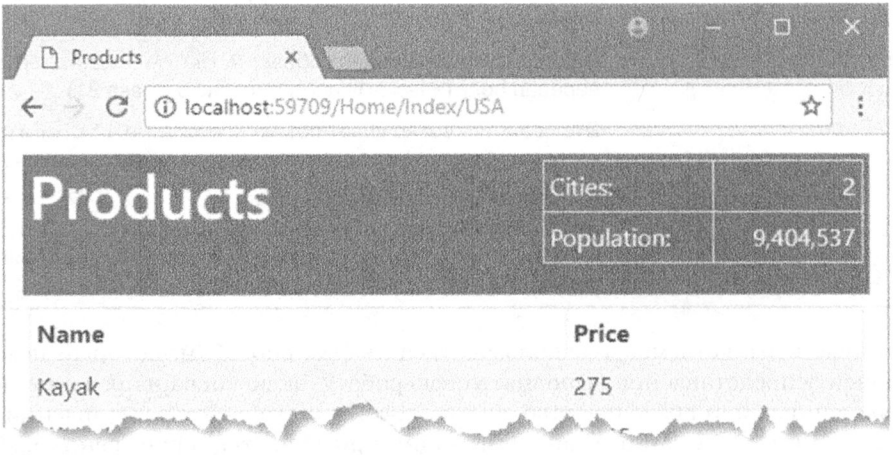


Рис. 22.6. Применение данных контекста в компоненте представления

Получение данных контекста из родительского представления с использованием аргументов

Родительские представления могут предоставлять дополнительные данные контекста в виде аргументов для выражения `@await Component.InvokeAsync()`. Такую функциональную возможность можно применять для получения данных из модели родительского представления или для сообщения инструкции о типе содержимого, которое должен выдавать компонент представления. Чтобы посмотреть на нее в работе, создадим в папке `Views/Home/Component/CitySummary` файл представления по имени `CityList.cshtml` и добавим в него разметку из листинга 22.23.

Листинг 22.23. Содержимое файла `CityList.cshtml` из папки `Views/Home/Component/CitySummary`

```
@model IEnumerable<City>

<table class="table table-sm table-bordered">
  @foreach (var city in Model) {
    <tr>
      <td>@city.Name</td>
      <td class="text-right">
        @city.Population.ToString("#,###")
      </td>
    </tr>
  }
  <tr>
    <th>Total:</th>
    <td class="text-right">
      @Model.Sum(p => p.Population).ToString("#,###")
    </td>
  </tr>
</table>
```

Определение второго представления позволит компоненту представления производить выбор между ними на основе аргумента, добавленного к методу `Invoke()`, как показано в листинге 22.24.

Листинг 22.24. Выбор представления в файле `CitySummary.cs` из папки `Components`

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using UsingViewComponents.Models;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace UsingViewComponents.Components {

  public class CitySummary : ViewComponent {
    private ICityRepository repository;

    public CitySummary(ICityRepository repo) {
      repository = repo;
    }

    public IViewComponentResult Invoke(bool showList) {
      if (showList) {
        return View("CityList", repository.Cities);
      } else {
        return View(new CityViewModel {
          Cities = repository.Cities.Count(),
          Population = repository.Cities.Sum(c => c.Population)
        });
      }
    }
  }
}
```

Если аргумент `showList` метода `Invoke()` равен `true`, то компонент представления выбирает `CityList` и передает все объекты `City` из хранилища как модель представления. Если аргумент `showList` равен `false`, тогда выбирается стандартное представление с указанием объекта `CityViewModel` в качестве модели представления.

Последний шаг связан с предоставлением данных контекста, когда компонент представления применяется в родительском представлении, для чего методу `Invoke()` передается анонимный объект (листинг 22.25).

Листинг 22.25. Предоставление данных контекста во время применения компонента представления в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div class="bg-primary m-1 p-1">
    <div class="row text-white">
      <div class="col-7"><h1>Products</h1></div>
      <div class="col-5">
        @await Component.InvokeAsync("CitySummary", new { showList = true })
      </div>
    </div>
  </div>
  <div class="m-1 p-1">@RenderBody()</div>
</body>
</html>
```

После запуска приложения компонент представления получит значение, указываемое родительским представлением, и отреагирует соответствующим образом (рис. 22.7).

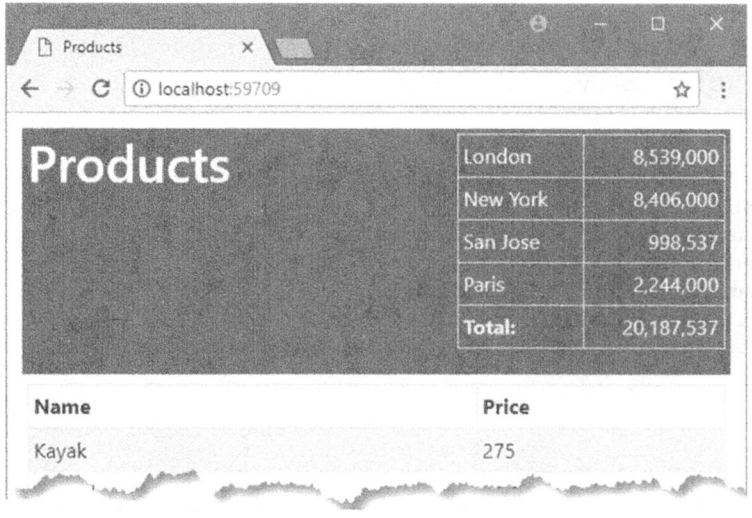


Рис. 22.7. Предоставление данных контекста компоненту представления

Модульное тестирование компонентов представлений

Компоненты представлений следуют общему подходу MVC, предусматривающему отделение логики выбора и обработки данных модели от разметки представления, которая их форматирует и отображает, что облегчает проведение модульного тестирования. Вот пример модульного теста для класса `CitySummary` из рассмотренного приложения:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Moq;
using UsingViewComponents.Models;
using UsingViewComponents.Components;
using Xunit;

namespace UsingViewComponents.Tests {

  public class SummaryViewComponentTests {

    [Fact]
    public void TestSummary() {

      // Организация
      var mockRepository = new Mock<ICityRepository>();
      mockRepository.SetupGet(m => m.Cities).Returns(new List<City> {
        new City { Population = 100 },
        new City { Population = 20000 },
        new City { Population = 1000000 },
        new City { Population = 500000 }
      });
      var viewComponent
        = new CitySummary(mockRepository.Object);

      // Действие
      ViewViewComponentResult result
        = viewComponent.Invoke(false) as ViewViewComponentResult;

      // Утверждение
      Assert.IsType(typeof(CityViewModel), result.ViewData.Model);
      Assert.Equal(4, ((CityViewModel)result.ViewData.Model).Cities);
      Assert.Equal(1520100,
        ((CityViewModel)result.ViewData.Model).Population);
    }
  }
}
```

Для организации теста создается имитированное хранилище, которое передается конструктору класса `CitySummary`, чтобы создать экземпляр компонента представления. В разделе действия теста вызывается метод `Invoke()`, который предоставляет объект результата. Компонент представления выбирает представление Razor, поэтому результат приводится к типу `ViewViewComponentResult` и затем через предлагаемое им свойство `ViewData.Model` осуществляется доступ к объекту модели представления. В разделе утверждения теста проверяется тип данных модели представления и содержащиеся в них значения.

Создание асинхронных компонентов представлений

Все рассмотренные до сих пор примеры были синхронными компонентами представлений, которые легко опознать по тому, что они определяли метод `Invoke()`.

Если компонент представления полагается на асинхронные API-интерфейсы, то можно создать асинхронный компонент представления, определив метод `InvokeAsync()`, который возвращает объект `Task`. Когда механизм Razor получает объект `Task` из метода `InvokeAsync()`, он будет ожидать его завершения и затем вставит результат в главное представление. Для подготовки настоящего примера щелкнем правой кнопкой мыши на элементе проекта `UsingViewComponents` в окне Solution Explorer, выберем в контекстном меню пункт Edit UsingViewComponents.csproj (Редактировать `UsingViewComponents.csproj`) и внесем в файл `UsingViewComponents.csproj` изменения, приведенные в листинге 22.26, чтобы добавить в проект новый пакет.

Листинг 22.26. Добавление пакета в файле `UsingViewComponents.csproj` из папки `UsingViewComponents`

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0 " />
    <PackageReference Include="System.Net.Http" Version="4.3.2" />
  </ItemGroup>

</Project>
```

Пакет `System.Net.Http` предоставляет API-интерфейс для выполнения асинхронных HTTP-запросов, который будет использоваться для отправки запросов к веб-сайту Apress.com. Добавим в папку `Components` файл класса по имени `PageSize.cs`, содержимое которого показано в листинге 22.27.

Листинг 22.27. Содержимое файла `PageSize.cs` из папки `Components`

```
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace UsingViewComponents.Components {

  public class PageSize : ViewComponent {

    public async Task<IViewComponentResult> InvokeAsync() {
      HttpClient client = new HttpClient();
      HttpResponseMessage response
        = await client.GetAsync("http://apress.com");
      return View(response.Content.Headers.ContentLength);
    }
  }
}
```

В методе InvokeAsync() применяются ключевые слова async и await, описанные в главе 4, для потребления асинхронного API-интерфейса, предлагаемого классом HttpClient, и получения длины содержимого, возвращаемого в результате отправки запроса GET веб-сайту Apress.com. Длина передается методу View(), который выбирает стандартное частичное представление, ассоциированное с компонентом представления.

Чтобы создать это представление, добавим в проект папку Views/Shared/Components/PageSize и поместим в нее файл представления по имени Default.cshtml с содержимым, приведенным в листинге 22.28.

Листинг 22.28. Содержимое файла Default.cshtml из папки Views/Shared/Components/PageSize

```
@model long
<div class="m-1 p-1 bg-info text-white">Page size: @Model</div>
```

Осталось лишь воспользоваться компонентом, что и делается в файле _Layout.cshtml (листинг 22.29).

Листинг 22.29. Применение асинхронного компонента представления в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div class="bg-primary m-1 p-1">
    <div class="row text-white">
      <div class="col-7"><h1>Products</h1></div>
      <div class="col-5">
        @await Component.InvokeAsync("CitySummary",
          new { showList = true })
      </div>
    </div>
  </div>
  <div class="m-1 p-1">@RenderBody()</div>
  @await Component.InvokeAsync("PageSize")
</body>
</html>
```

В результате запуска приложения выяснится, что браузер отобразит новое добавление к содержимому (рис. 22.8). Когда вы запустите пример приложения, выводимое число может быть другим, т.к. веб-сайт Apress часто обновляется.

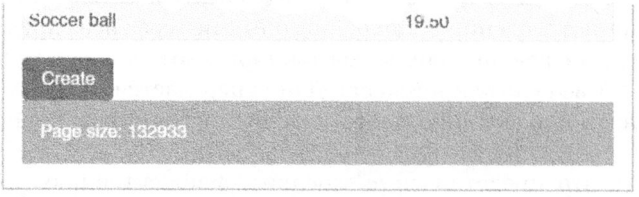


Рис. 22.8. Создание асинхронного компонента представления

Создание гибридных классов контроллеров и компонентов представлений

Компоненты представлений часто обеспечивают сводку или краткую характеристику функциональности, которая поддерживается контроллером. Например, компоненты представлений, отображающие сводку по корзине для покупок, нередко содержат ссылку, нацеленную на контроллер, который выводит подробный список товаров в корзине и может использоваться для завершения покупок и перехода к оплате.

В такой ситуации можно создать класс, являющийся одновременно контроллером и компонентом представления, который позволяет сгруппировать вместе связанную функциональность и сократить дублирование кода. Добавим в папку `Controllers` файл класса по имени `CityController.cs` и определим в нем контроллер, как показано в листинге 22.30.

Листинг 22.30. Содержимое файла `CityController.cs` из папки `Controllers`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using UsingViewComponents.Models;

namespace UsingViewComponents.Controllers {

  [ViewComponent(Name = "ComboComponent")]
  public class CityController : Controller {
    private ICityRepository repository;

    public CityController(ICityRepository repo) {
      repository = repo;
    }
    public ViewResult Create() => View();
    [HttpPost]
    public IActionResult Create(City newCity) {
      repository.AddCity(newCity);
      return RedirectToAction("Index", "Home");
    }

    public IViewComponentResult Invoke() => new ViewViewComponentResult() {
      ViewData = new ViewDataDictionary<IEnumerable<City>>(ViewData,
        repository.Cities)
    };
  }
}
```

Атрибут ViewComponent применяется к классам, которые не унаследованы от базового класса ViewComponent и не имеют в своих именах суффикс ViewComponent, т.е. нормальный процесс обнаружения не сможет отнести их к категории компонентов представлений. Свойство Name устанавливает имя, по которому на класс можно ссылаться во время его использования в выражении @Component.Invoke() внутри родительского представления. В данном примере свойство Name применяется для установки имени части класса, относящейся к компоненту представления, в ComboComponent. Это имя будет использоваться для обращения к компоненту представления и поиска его представлений.

Поскольку гибридные классы не наследуются от базового класса ViewComponent, они не имеют доступа к удобным методам для создания объектов реализации IViewComponentResult, что означает необходимость в создании объекта ViewViewComponentResult напрямую, как требовалось бы в компоненте представления POCO.

Создание гибридных представлений

Гибридный класс требует двух наборов представлений: того, который визуализируется в случае применения класса как контроллера, и того, который визуализируется при использовании класса как компонента представления.

Создадим папку Views/City и добавим в нее файл представления по имени Create.cshtml с содержимым из листинга 22.31.

Листинг 22.31. Содержимое файла Create.cshtml из папки Views/City

```
@model City
@{
  ViewData["Title"] = "Create City";
  Layout = "_Layout";
}
<form method="post" asp-action="Create">
  <div class="form-group">
    <label asp-for="Name">Name:</label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label asp-for="Country">Country:</label>
    <input class="form-control" asp-for="Country" />
  </div>
  <div class="form-group">
    <label asp-for="Population">Population:</label>
    <input class="form-control" asp-for="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Create</button>
  <a class="btn btn-secondary" asp-controller="Home" asp-action="Index">
    Cancel
  </a>
</form>
```

Представление Create выводит простую форму для создания новых объектов City. Щелчок на кнопке Create (Создать) приводит к отправке запроса POST к действию Create контроллера City, а на кнопке Cancel (Отмена) — к отправке запроса GET к действию Index контроллера Home.

Создадим папку `Views/Shared/Components/ComboComponent` и добавим в нее файл представления по имени `Default.cshtml`, содержимое которого показано в листинге 22.32. Частичное представление помещено в папку `Views/Shared`, потому что она будет применяться при поиске представлений контроллером, в чьем представлении используется компонент представления, имя которого включается в путь.

Листинг 22.32. Содержимое файла `Default.cshtml` из папки `Views/Shared/Components/ComboComponent`

```
@model IEnumerable<City>

<table class="table table-sm table-bordered">
  <tr>
    <td>Biggest City:</td>
    <td>
      @Model.OrderByDescending(c => c.Population).First().Name
    </td>
  </tr>
</table>
<a class="btn btn-sm btn-info" asp-controller="City" asp-action="Create">
  Create City
</a>
```

Частичное представление `Default.cshtml` получает последовательность объектов `City` и сортирует ее с помощью LINQ, чтобы выбрать объект с наибольшим значением `Population`. Кроме того, определен элемент `a`, стилизованный под кнопку, который нацелен на действие `Create` контроллера `City`.

Совет. В листинге 22.32 обратите внимание на явное указание контроллера `City` для элемента `a`. Дело в том, что URL генерируются с применением данных контекста, предоставляемых родительским представлением; это значит, что запрос обрабатывается стандартным контроллером, а не контроллером, который является также компонентом представления. Если опустить атрибут `asp-controller`, то ссылка будет нацелена на действие `Create` контроллера `Home`.

Применение гибридного класса

Финальный шаг предусматривает применение гибридного класса как компонента представления в разделяемой компоновке с использованием имени, которое было указано посредством атрибута `ViewComponent` (листинг 22.33).

Листинг 22.33. Применение гибридного класса в файле `_Layout.cshtml`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-primary m-1 p-1">
    <div class="row text-white">
      <div class="col-7"><h1>Products</h1></div>
      <div class="col-5">
        @await Component.InvokeAsync("ComboComponent")
      </div>
    </div>
  </div>
  <div class="m-1 p-1">@RenderBody()</div>
  @await Component.InvokeAsync("PageSize")
</body>
</html>
```

Результатом будет компонент представления, оснащенный собственным интегрированным контроллером (или, если хотите, контроллер, который имеет собственный интегрированный компонент представления). Запустив приложение, можно заметить, что в качестве самого густонаселенного города указан Лондон (London). Щелкнем на кнопке Create City (Создать город), чтобы отобразить форму, которая позволит добавить новый объект `City` в приложение. Заполним и отправим форму, после чего контроллер получит данные, обновит хранилище и перенаправит браузер на стандартный URL приложения. В случае добавления объекта `City`, население которого превышает показатели населения других городов, имеющихся в хранилище, вывод из компонента представления изменится, как показано на рис. 22.9.

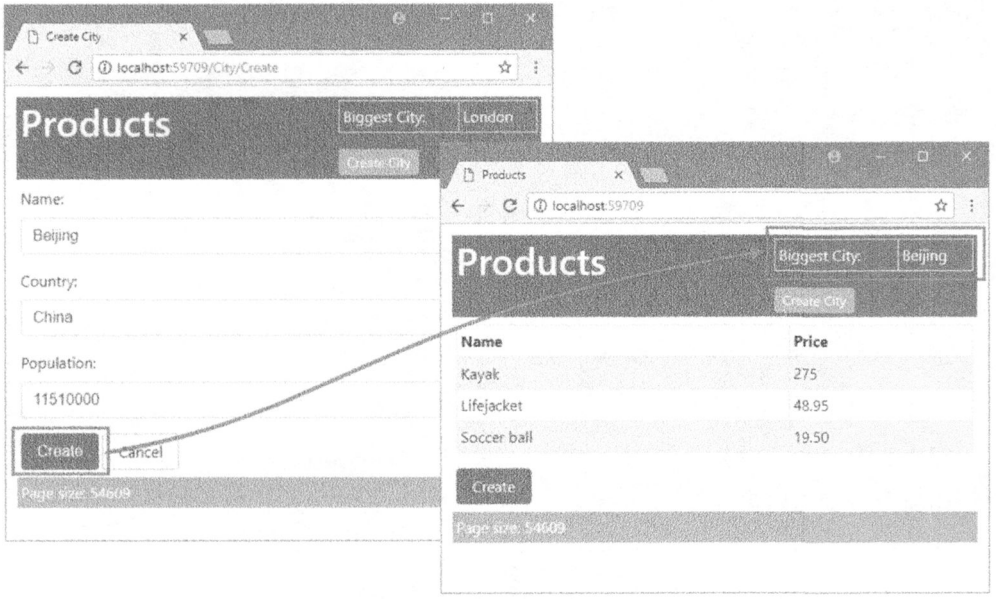


Рис. 22.9. Использование гибридного класса контроллера и компонента представления

Резюме

В главе было дано введение в компоненты представлений, которые являются функциональной возможностью ASP.NET Core MVC, заменяющей средство дочерних действий из предыдущих версий MVC. Вы узнали, каким образом создавать компоненты представлений POCO и как применять базовый класс `ViewComponent`. Были продемонстрированы три разных типа результатов, которые могут выпускать компоненты представлений, в том числе выбор частичных представлений для их включения в родительское представление. В завершение главы было показано, как добавить функциональность компонентов представлений в класс контроллера, чтобы сократить дублирование кода и упростить приложение. В следующей главе рассматриваются вспомогательные функции дескрипторов, которые используются для трансформации HTML-элементов в представлениях.

ГЛАВА 23

Вспомогательные функции дескрипторов

Вспомогательные функции дескрипторов — это средство, появившееся в ASP.NET Core MVC. Они являются классами, которые трансформируют HTML-элементы в представлении. Распространенные случаи использования вспомогательных функций дескрипторов включают генерирование URL для форм с применением конфигурации маршрутизации приложения, обеспечение согласованной стилизации элементов специфического типа и замена специальных сокращающих элементов ходовыми фрагментами содержимого. В настоящей главе будет показано, как работают вспомогательные функции дескрипторов и каким образом создавать и использовать специальные вспомогательные функции дескрипторов. В главе 24 будут описаны встроенные вспомогательные функции дескрипторов, которые поддерживают HTML-формы, а в главе 25 — другие встроенные вспомогательные функции дескрипторов, предоставляемые инфраструктурой MVC. В табл. 23.1 приведена сводка, позволяющая поместить вспомогательные функции дескрипторов в контекст.

Компоненты вспомогательных функций дескрипторов

В платформе ASP.NET Core 2 вводятся *компоненты вспомогательных функций дескрипторов*, представляющие собой классы, которые модифицируют специфическую порцию HTML-разметки в ответах, отправляемых клиентам, и могут применяться вспомогательными функциями дескрипторов. В книге данное средство не рассматривается по причине его неудобной реализации и малой пригодности для большинства разработчиков приложений MVC. Если необходимо трансформировать содержимое в ответах, которые отправляются клиентам, тогда следует использовать средства вспомогательных функций дескрипторов, описанные в текущей главе.

Таблица 23.1. Помещение вспомогательных функций дескрипторов в контекст

Вопрос	Ответ
Что это такое?	Вспомогательные функции дескрипторов — это классы, которые манипулируют HTML-элементами с целью их изменения каким-либо образом, дополнения добавочным содержимым или полной замены новым содержимым
Чем они полезны?	Вспомогательные функции дескрипторов позволяют генерировать либо трансформировать содержимое представления с использованием логики C#, гарантируя, что отправляемая браузеру HTML-разметка отражает состояние приложения

Окончание табл. 23.1

Вопрос	Ответ
Как они используются?	Элементы HTML, к которым применяются вспомогательные функции дескрипторов, выбираются на основе имени класса или за счет использования атрибута `HTMLTargetElement`. Когда представление визуализируется, элементы трансформируются вспомогательными функциями дескрипторов и включаются в HTML-разметку, отправляемую клиенту
Существуют ли какие-то скрытые ловушки или ограничения?	Довольно легко увлечься и генерировать с применением вспомогательных функций дескрипторов сложные разделы HTML-разметки, что намного проще достигать с помощью компонентов представлений, как объяснялось в главе 22
Существуют ли альтернативы?	Вы не обязаны использовать вспомогательные функции дескрипторов, но они облегчают генерацию сложной HTML-разметки в приложениях MVC

В табл. 23.2 приведена сводка по главе.

Таблица 23.2. Сводка по главе

Задача	Решение	Листинг
Трансформация HTML-элемента	Создайте вспомогательную функцию дескриптора и зарегистрируйте ее с применением выражения `@addTagHelper` в представлении или в файле импортирования представлений	23.10–23.12
Управление областью действия вспомогательной функции дескриптора	Используйте атрибут `HtmlTargetElement`	23.13–23.17
Поддержка сокращающего элемента	Применяйте объект `TagHelperOutput` для генерирования заменяющих элементов	23.18, 23.19
Вставка содержимого вокруг или внутрь целевого элемента	Используйте предоставляемые классом `TagHelperOutput` свойства, имена которых начинаются на `Pre` и `Post`	23.22, 23.23
Получение данных контекста во вспомогательной функции дескриптора	Декорируйте свойство атрибутами `ViewContext` и `HtmlAttributeNotBound`	23.24, 23.25
Получение доступа к модели представления	Применяйте свойство `ModelExpression`	23.26, 23.27
Согласование вспомогательных функций дескрипторов	Используйте свойство `TagHelperContext.Items`	23.28, 23.29
Подавление элемента	Применяйте метод `SuppressOutput()`	23.30, 23.31

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `Cities` с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core.

Создание модели и хранилища

Создадим папку Models и добавим в нее файл класса по имени City.cs с определением из листинга 23.1.

Листинг 23.1. Содержимое файла City.cs из папки Models

```
namespace Cities.Models {
  public class City {
    public string Name { get; set; }
    public string Country { get; set; }
    public int? Population { get; set; }
  }
}
```

Чтобы создать хранилище для объектов City, добавим в папку Models файл по имени Repository.cs и определим в нем интерфейс и класс реализации, как показано в листинге 23.2.

Листинг 23.2. Содержимое файла Repository.cs из папки Models

```
using System.Collections.Generic;

namespace Cities.Models {
  public interface IRepository {
    IEnumerable<City> Cities { get; }
    void AddCity(City newCity);
  }
  public class MemoryRepository : IRepository {
    private List<City> cities = new List<City> {
      new City { Name = "London", Country = "UK", Population = 8539000},
      new City { Name = "New York", Country = "USA", Population = 8406000 },
      new City { Name = "San Jose", Country = "USA", Population = 998537 },
      new City { Name = "Paris", Country = "France", Population = 2244000 }
    };
    public IEnumerable<City> Cities => cities;
    public void AddCity(City newCity) {
      cities.Add(newCity);
    }
  }
}
```

Создание контроллера, компоновки и представлений

В примерах, рассматриваемых в главе, требуется только один контроллер. Создадим папку Controllers, добавим в нее файл класса по имени HomeController.cs и определим контроллер, код которого приведен в листинге 23.3.

Листинг 23.3. Содержимое файла HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using Cities.Models;

namespace Cities.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public ViewResult Index() => View(repository.Cities);

    public ViewResult Create() => View();

    [HttpPost]
    public IActionResult Create(City city) {
      repository.AddCity(city);
      return RedirectToAction("Index");
    }
  }
}
```

Контроллер поддерживает метод действия `Index()`, выводящий список объектов в хранилище, и пару методов `Create()`, которые позволят пользователю применять форму для создания новых объектов `City`, следуя тому же самому подходу, что и в примерах из предшествующих глав.

Представления в данном приложении будут использовать разделяемую компоновку. Создадим папку `Views/Shared` и поместим в нее файл по имени `_Layout.cshtml` с разметкой из листинга 23.4.

На заметку! Поскольку цель главы заключается в демонстрации работы вспомогательных функций дескрипторов, компоновка и представления в примере приложения реализованы с применением только стандартных HTML-элементов, которые будут заменяться по мере рассмотрения различных вспомогательных функций дескрипторов.

Листинг 23.4. Содержимое файла _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

Создадим папку `Views/Home` и добавим в нее файл по имени `Index.cshtml`, содержимое которого показано в листинге 23.5.

Листинг 23.5. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<City>

@{ Layout = "_Layout"; }

<table class="table table-sm table-bordered">
  <thead class="bg-primary text-white">
    <tr>
      <th>Name</th>
      <th>Country</th>
      <th class="text-right">Population</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var city in Model) {
      <tr>
        <td>@city.Name</td>
        <td>@city.Country</td>
        <td class="text-right">@city.Population?.ToString("#,###")</td>
      </tr>
    }
  </tbody>
</table>
<a href="/Home/Create" class="btn btn-primary">Create</a>
```

В представлении используется последовательность объектов `City` для заполнения таблицы и включается элемент `a`, нацеленный на URL вида `/Home/Create`, который с помощью Bootstrap стилизован под кнопку. Чтобы создать второе представление, добавим в папку `Views/Home` файл по имени `Create.cshtml` с разметкой из листинга 23.6.

Листинг 23.6. Содержимое файла `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Создадим в папке Views файл импортирования представлений по имени _ViewImports.cshtml и поместим в него выражение, приведенное в листинге 23.7. Оно позволит ссылаться на классы из папки Models, не указывая пространство имен.

Листинг 23.7. Содержимое файла _ViewImports.cshtml из папки Views

```
@using Cities.Models
```

Представления в рассматриваемом примере полагаются на CSS-пакет Bootstrap. Создадим в корневой папке проекта файл bower.json с применением шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел dependencies (листинг 23.8).

Листинг 23.8. Добавление пакета Bootstrap в файле bower.json из папки Cities

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Конфигурирование приложения

Последний подготовительный шаг связан с конфигурированием приложения, как показано в листинге 23.9. Это та же самая базовая конфигурация, используемая во всех проектах во второй части книги, но с дополнительным оператором, который регистрирует хранилище как службу с применением жизненного цикла одиночки.

Листинг 23.9. Содержимое файла Startup.cs из папки Cities

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Cities.Models;
namespace Cities {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IRepository, MemoryRepository>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

В результате запуска приложения отображается список объектов City, который хранилище создает по умолчанию. Щелкнем на кнопке Create (Создать), заполним форму и щелкнем на кнопке Add (Добавить); в хранилище добавится новый объект (рис. 23.1).



Рис. 23.1. Выполнение примера приложения

Создание вспомогательной функции дескриптора

Как и со многими средствами MVC, пониманию вспомогательных функций дескрипторов лучше всего способствует создание одного такого класса, что позволит выявить особенности их работы и согласование с остальными частями приложения. В приведенных далее разделах вы ознакомитесь с процессом создания и использования вспомогательной функции дескриптора, которая будет применять CSS-классы Bootstrap к элементу button, так что элемент вида:

```
...
<button type="submit" bs-button-color="danger">Add</button>
...
```

трансформируется следующим образом:

```
...
<button type="submit" class="btn btn-danger">Add</button>
...
```

Вспомогательная функция дескриптора будет распознавать атрибут bs-button-color и использовать его значение для установки атрибута class элемента, отправляемого браузеру. Это не особо впечатляющая или сколько-нибудь полезная трансформация, но она формирует основу для объяснения работы вспомогательных функций дескрипторов.

Определение класса вспомогательной функции дескриптора

Вспомогательные функции дескрипторов можно определять где угодно в проекте, но их удобно держать вместе, т.к. в отличие от большинства компонентов MVC перед применением они нуждаются в регистрации. Добавим в проект папку Infrastructure/TagHelpers, в которой будут создаваться вспомогательные функции дескрипторов.

Вспомогательные функции дескрипторов являются производными от класса `TagHelper`, который определен в пространстве имен `Microsoft.AspNetCore.Razor.TagHelpers`. Чтобы создать вспомогательную функцию дескриптора, добавим в папку `Infrastructure/TagHelpers` файл по имени `ButtonTagHelper.cs` и определим в нем класс, показанный в листинге 23.10.

Листинг 23.10. Содержимое файла `ButtonTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

  public class ButtonTagHelper : TagHelper {

    public string BsButtonColor { get; set; }

    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      output.Attributes.SetAttribute("class", $"btn btn-{BsButtonColor}");
    }
  }
}
```

В классе `TagHelper` определен метод `Process()`, который переопределяется подклассами для реализации поведения, трансформирующего элементы. Имя вспомогательной функции дескриптора образуется из имени трансформируемого элемента и суффикса `TagHelper`. В рассматриваемом примере имя класса `ButtonTagHelper` сообщает инфраструктуре MVC о том, что это вспомогательная функция дескриптора, которая оперирует на элементах `button`. Область действия вспомогательной функции дескриптора может быть расширена или сужена с использованием атрибутов, описанных в разделе "Управление областью действия вспомогательной функции дескриптора" далее в главе, но таково его стандартное поведение.

Совет. Переопределяя метод `ProcessAsync()` вместо `Process()`, можно создавать асинхронные вспомогательные функции дескрипторов, но для большинства вспомогательных функций дескрипторов подобное не требуется, потому что они обычно вносят в HTML-элементы небольшие и специализированные изменения. Стандартная реализация `ProcessAsync()` в любом случае вызывает метод `Process()`. Пример асинхронной вспомогательной функции дескриптора приведен в главе 24.

Получение данных контекста

Вспомогательные функции дескрипторов получают информацию о трансформируемом элементе через экземпляр класса `TagHelperContext`, который передается в качестве аргумента методу `Process()` и определяет свойства, перечисленные в табл. 23.3.

Хотя получать доступ к деталям атрибутов элемента можно через словарь `AllAttributes`, более удобный подход предусматривает определение свойства, имя которого соответствует интересующему атрибуту, например:

```
...
public string BsButtonColor { get; set; }
...
```

Таблица 23.3. Свойства класса `TagHelperContext`

Имя	Описание
`AllAttributes`	Это свойство возвращает словарь только для чтения, индексированный по имени и по числовому индексу, с атрибутами, которые были применены к элементу, подвергающемуся трансформации
`Items`	Это свойство возвращает словарь, который используется для согласования вспомогательных функций дескрипторов, как описано в разделе "Согласование вспомогательных функций дескрипторов" далее в главе
`UniqueId`	Это свойство возвращает уникальный идентификатор для трансформируемого элемента

Когда вспомогательная функция дескриптора применяется, инфраструктура MVC инспектирует определенные ею свойства и устанавливает значения любых из них, имена которых совпадают с атрибутами, примененными к HTML-элементу. В качестве части процесса MVC будет пытаться преобразовать значение атрибута с целью соответствия типу свойства C#, так что свойства `bool` могут использоваться для получения значений булевских атрибутов (`true` и `false`), а свойства `int` — для получения значений числовых атрибутов (наподобие `1` и `2`).

Что случилось со вспомогательными методами HTML?

В ранних версиях ASP.NET MVC для генерации элементов форм применялись вспомогательные методы HTML. Они представляли собой методы, обращение к которым осуществлялось посредством выражений Razor, начинающихся с `@Html`. Таким образом, элемент `input` для свойства `Population` создавался бы следующим образом:

```
...
@Html.TextBoxFor(m => m.Population)
...
```

Проблема выражений со вспомогательными методами HTML в том, что они не соответствуют структуре HTML-элементов; в итоге появляются неуклюжие конструкции вроде показанной ниже, которая добавляет стили Bootstrap к генерируемому элементу:

```
...
@Html.TextBoxFor(m => m.Population, new { @class = "form-control" })
...
```

Атрибуты должны выражаться в динамическом объекте и предваряться префиксом `@`, когда они совпадают по написанию с зарезервированными словами C#, такими как `class`. По мере усложнения требуемых HTML-элементов выражения со вспомогательными методами HTML становятся все более неудобными. Вспомогательные функции дескрипторов избавляют от такой неуклюжести за счет использования атрибутов HTML:

```
...
<input class="form-control" asp-for="Population" />
...
```

Результатом будет более естественное соответствие природе HTML и возможность построения представлений, которые легче в чтении и понимании. Инфраструктура MVC по-прежнему поддерживает вспомогательные методы HTML (на самом деле вспомогательные функции дескрипторов внутренне применяют вспомогательные методы HTML). Это означает, что вы можете использовать их для обратной совместимости в представлениях, которые первоначально разрабатывались для версии MVC 5, однако новые представления должны задействовать в своих интересах более естественный подход, предлагаемый вспомогательными функциями дескрипторов.

Имя атрибута автоматически преобразуется из стандартного стиля HTML, bs-button-color, в стиль C#, т.е. BsButtonColor. Можно применять любой префикс атрибута кроме asp- (который использует Microsoft) и data- (зарезервирован для специальных атрибутов, отправляемых клиенту). В текущем примере посредством атрибута BsButtonColor получается цветовая схема для применения к элементу button в методе Process():

```
...
output.Attributes.SetAttribute("class", $"btn btn-{BsButtonColor}");
...
```

Свойства, для которых отсутствуют соответствующие атрибуты HTML-элемента, не устанавливаются, т.е. вы должны проверять, что не работаете с null или стандартными значениями. В разделе "Управление областью действия вспомогательной функции дескриптора" далее в главе объясняется, как изменять область действия вспомогательной функции дескриптора, чтобы он использовался только в отношении элементов, определяющих атрибуты, от которых вы зависите.

Совет. Применение имен атрибутов HTML для свойств вспомогательной функции дескриптора далеко не всегда приводит к получению читабельных или понятных классов. Вы можете разорвать связь между именем свойства и атрибутом, которое оно представляет, с использованием атрибута HtmlAttributeName, позволяющего указывать представляемый свойством атрибут HTML.

Генерирование вывода

Метод Process() трансформирует элемент путем конфигурирования объекта TagHelperOutput, который получается в аргументе. Объект TagHelperOuput начинает свое существование с описания HTML-элемента в том виде, как он появляется в представлении Razor, и модифицирует его посредством свойств и метода, которые описаны в табл. 23.4.

Таблица 23.4. Свойства и метод класса TagHelperOutput

Имя	Описание
TagName	Это свойство применяется для получения и установки имени дескриптора в выходном элементе
Attributes	Это свойство возвращает словарь, содержащий атрибуты для выходного элемента
Content	Это свойство возвращает объект TagHelperContent, который используется для установки содержимого элемента
PreElement	Это свойство возвращает объект TagHelperContext, который применяется для вставки содержимого в представление перед выходным элементом. См. раздел "Вставка перед и после содержимого и элементов" далее в главе
PostElement	Это свойство возвращает объект TagHelperContext, который используется для вставки содержимого в представление после выходного элемента. См. раздел "Вставка перед и после содержимого и элементов" далее в главе

Окончание табл. 23.4

Имя	Описание
PreContent	Это свойство возвращает объект TagHelperContext, который применяется для вставки содержимого перед содержимым выходного элемента. См. раздел "Вставка перед и после содержимого и элементов" далее в главе
PostContent	Это свойство возвращает объект TagHelperContext, который используется для вставки содержимого после содержимого выходного элемента. См. раздел "Вставка перед и после содержимого и элементов" далее в главе
TagMode	Это свойство указывает, как будет записываться выходной элемент, с применением значения из перечисления TagMode. См. раздел "Создание сокращающих элементов" далее в главе
SupressOuput()	Вызов этого метода предотвращает включение элемента в представление. См. раздел "Подавление выходного элемента" далее в главе

В классе ButtonTagHelper с помощью словаря Attributes к HTML-элементу добавляется атрибут class, который указывает стили Bootstrap для кнопки, включая значение свойства BsButtonColor, что означает возможность указания разных цветов с использованием имен Bootstrap, таких как primary, info и danger.

Регистрация вспомогательных функций дескрипторов

Вспомогательные функции дескрипторов можно применять только после того, как они были зарегистрированы с использованием Razor-выражения @addTagHelper. Набор представлений, к которым будет применяться вспомогательная функция дескриптора, зависит от того, где используется выражение @addTagHelper. Для отдельного преставления упомянутое выражение присутствует в самом представлении. Для подмножества представлений в приложении выражение @addTagHelper находится внутри файла _ViewImports.cshtml в папке, которая содержит представления, или в родительской папке. Таким образом, выражение @addTagHelper в файле /Views/Home/_ViewImports.cshtml регистрирует вспомогательные функции дескрипторов для всех представлений, ассоциированных с контроллером Home. Нам нужно, чтобы вспомогательные функции дескрипторов были доступны во всех представлениях приложения, поэтому добавим выражение @addTagHelper в файл Views/_ViewImports.cshtml, как показано в листинге 23.11.

Листинг 23.11. Регистрация вспомогательных функций дескрипторов в файле _ViewImports.cshtml из папки Views

```
@using Cities.Models
@addTagHelper Cities.Infrastructure.TagHelpers.*, Cities
```

В первой части аргумента задаются имена классов вспомогательных функций дескрипторов (поддерживаются групповые символы), а во второй части указывается имя сборки, где они определены. В листинге 23.11 зарегистрированы все вспомогательные функции дескрипторов в пространстве имен Cities.Infrastructure.TagHelpers из сборки Cities.

Использование вспомогательной функции дескриптора

Наконец, вспомогательную функцию дескриптора можно применить для трансформации элемента. Удалим атрибут `class` из элемента `button` в представлении `Create.cshtml` и заменим его атрибутом, который ожидает класс `ButtonTagHelper` (листинг 23.12).

Листинг 23.12. Использование вспомогательной функции дескриптора в файле `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>

  <button type="submit" bs-button-color="danger">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

После запуска приложения и щелчка на кнопке Create браузер запросит URL вида `/Home/Create`, в результате чего стиль и цвет кнопки Add изменятся (рис. 23.2).

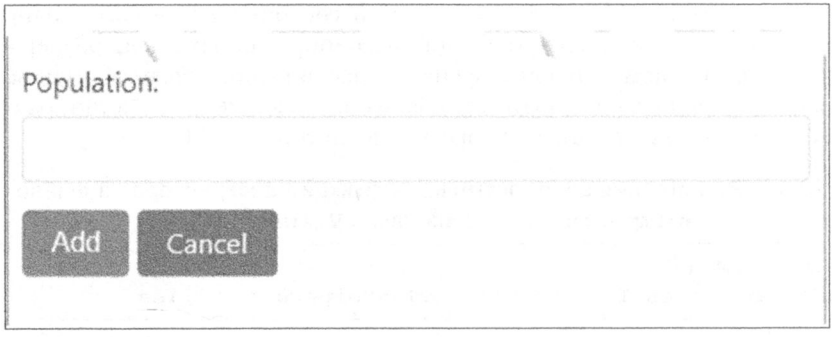


Рис. 23.2. Применение вспомогательной функции дескриптора для стилизации кнопки

Модульное тестирование вспомогательной функции дескриптора

Модульное тестирование вспомогательной функции дескриптора — относительно простой процесс, который основан на снабжении метода `Process()` значащим содержимым, подлежащим обработке. Вот пример модульного теста для вспомогательной функции дескриптора из листинга 23.10:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Cities.Infrastructure.TagHelpers;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Xunit;

namespace Cities.Tests {

  public class TagHelperTests {

    [Fact]
    public void TestTagHelper() {
      // Организация
      var context = new TagHelperContext(
        new TagHelperAttributeList(),
        new Dictionary<object, object>(),
        "myuniqueid");

      var output = new TagHelperOutput("button",
        new TagHelperAttributeList(), (cache, encoder) =>
          Task.FromResult<TagHelperContent>
            (new DefaultTagHelperContent()));

      // Действие
      var tagHelper = new ButtonTagHelper {
        BsButtonColor = "testValue"
      };
      tagHelper.Process(context, output);

      // Утверждение
      Assert.Equal($"btn btn-{tagHelper.BsButtonColor}",
        output.Attributes["class"].Value);
    }
  }
}
```

Большая часть работы этого модульного теста связана с настройкой объектов `TagHelperContext` и `TagHelperOutput`, чтобы их можно было передать методу `Process()` класса вспомогательной функции дескриптора и удостовериться в том, что HTML-элемент был корректно трансформирован. Объем работ, требуемых для подготовки вспомогательной функции дескриптора к тестированию, вполне естественно зависит от сложности HTML-разметки, которой она оперирует, и степени ее трансформации. Тем не менее, большинство вспомогательных функций дескрипторов будут относительно простыми и могут тестироваться в соответствии с приведенным ранее базовым шаблоном.

Управление областью действия вспомогательной функции дескриптора

Вспомогательные функции дескрипторов применяются ко всем элементам заданного типа, т.е. метод `Process()` класса `ButtonTagHelper`, созданного в предыдущем разделе, будет вызываться для каждого элемента `button` в каждом представлении внутри приложения. Это не всегда пригодно. Чтобы более пристально взглянуть на проблему, добавим в представление `Create.cshtml` еще один элемент `button` (листинг 23.13).

Листинг 23.13. Добавление элемента `button` в файле `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>

  <button type="submit" bs-button-color="danger">Add</button>
  <button type="reset" class="btn btn-primary">Reset</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Новый элемент `button` уже имеет атрибут `class` и не требует трансформации, выполняемой классом `ButtonTagHelper`. Но после запуска приложения и запрашивания URL вида `/Home/Create` возникает проблема, как иллюстрируется на рис. 23.3.

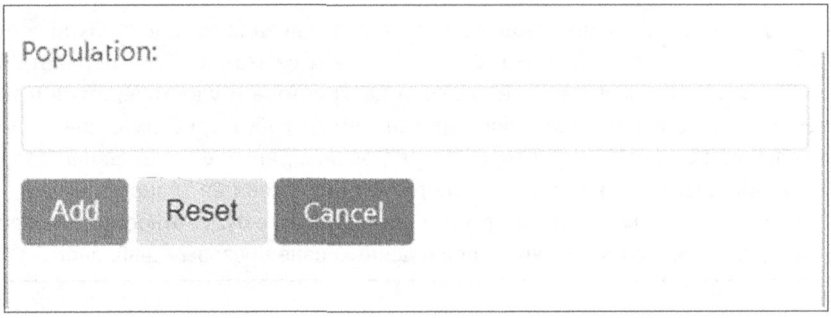


Рис. 23.3. Эффект от стандартной области действия вспомогательной функции дескриптора

Выяснить причину неудовлетворительного форматирования можно, просмотрев HTML-разметку, которая отправлена браузеру, где обнаруживается проблема с атрибутом `class`:

```
<button type="reset" class="btn btn-">Reset</button>
```

Инфраструктура MVC применила класс `ButtonTagHelper` к новому элементу `button`, но не установила значение свойства `BsButtonColor` из-за отсутствия соответствующего атрибута `bs-button-color` в HTML-элементе. Это привело к тому, что вспомогательная функция дескриптора заменила атрибут `class` атрибутом, в котором некорректно указаны стили Bootstrap, и в итоге получился неправильно сформатированный элемент.

Сужение области действия для вспомогательной функции дескриптора

Для решения проблемы существуют два подхода. Первый предусматривает модификацию класса `ButtonTagHelper`, чтобы сделать его чувствительным к разным элементам `button`, с которыми он может столкнуться. В рассматриваемом примере приложения это потребовало бы проверки, есть ли атрибут `bs-button-color`, и отказа от замены атрибута `class`, если он был определен. Проблема такого подхода в том, что по мере добавления в приложение представлений, содержащих элементы `button`, вспомогательная функция дескриптора будет становиться все более и более сложной, причем вся привносимая сложность связана с описанием условий, при которых класс `ButtonTagHelper` не должен выполнять свою трансформацию.

Второй подход заключается в том, чтобы позволить вспомогательной функции дескриптора определять ограничения на ее использование, сужая область действия, в которой она будет применяться. Ограничения вспомогательной функции дескриптора указываются с использованием атрибута `HtmlTargetElement` (листинг 23.14).

Листинг 23.14. Сужение области действия для вспомогательной функции дескриптора в файле `ButtonTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

  [HtmlTargetElement("button", Attributes = "bs-button-color",
                     ParentTag = "form")]
  public class ButtonTagHelper : TagHelper {

    public string BsButtonColor { get; set; }

    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      output.Attributes.SetAttribute("class", $"btn btn-{BsButtonColor}");
    }
  }
}
```

Атрибут `HtmlTargetElement` описывает элементы, к которым применяется вспомогательная функция дескриптора. Первый аргумент указывает тип элементов и поддерживает дополнительные именованные свойства, перечисленные в табл. 23.5.

Таблица 23.5. Свойства атрибута `HtmlTargetElement`

Имя	Описание
`Attributes`	Это свойство используется для указания на то, что вспомогательная функция дескриптора должна применяться только к элементам, которые имеют заданный набор атрибутов, предоставляемый в виде списка с разделителями-запятыми. Элемент должен располагать всеми указанными атрибутами. Имя атрибута, которое заканчивается символом звездочки, будет трактоваться подобно префиксу, так что `bs-button-*` соответствует `bs-button-color`, `bs-button-size` и т.д.
`ParentTag`	Это свойство используется для указания на то, что вспомогательная функция дескриптора должна применяться только к элементам, которые содержатся внутри элемента заданного типа
`TagStructure`	Это свойство используется для указания на то, что вспомогательная функция дескриптора должна применяться только к элементам, чья структура дескриптора соответствует заданному значению из перечисления `TagStructure`, которое определено как `Unspecified`, `NormalOrSelfClosing` и `WithoutEndTag`

В листинге 23.14 класс `ButtonTagHelper` ограничен так, что он применяется только к элементам `button`, которые имеют атрибут `bs-button-color`, а их родительским элементом является `form`. Запустив приложение и запросив URL вида `/Home/Create`, можно заметить, что кнопка Reset (Сброс) больше не трансформируется, т.к. в ней отсутствует требуемый атрибут (рис. 23.4).

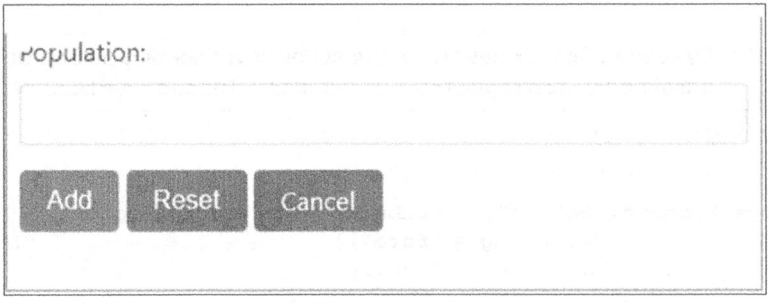


Рис. 23.4. Результат сужения области действия для вспомогательной функции дескриптора

Расширение области действия для вспомогательной функции дескриптора

Атрибут `HtmlTargetElement` может также использоваться для расширения области действия вспомогательной функции дескриптора, чтобы она охватывала более обширный диапазон элементов. Это полезно, когда одну и ту же трансформацию необходимо выполнять для нескольких типов элементов, что идет вразрез с исходным условием сопоставления элементов на основе имени вспомогательной функции дескриптора (листинг 23.15).

Листинг 23.15. Расширение области действия для вспомогательной функции дескриптора в файле ButtonTagHelper.cs из папки Infrastructure/TagHelpers

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

  [HtmlTargetElement(Attributes = "bs-button-color", ParentTag = "form")]
  public class ButtonTagHelper : TagHelper {

    public string BsButtonColor { get; set; }

    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      output.Attributes.SetAttribute("class", $"btn btn-{BsButtonColor}");
    }
  }
}
```

В листинге 23.15 для атрибута `HtmlTargetElement` тип элемента не указан, а потому вспомогательная функция дескриптора будет применяться к любому элементу, который имеет атрибут `bs-button-color`, независимо от типа элемента.

В листинге 23.16 элемент `a` внутри формы модифицирован так, что он использует такой же набор стилей Bootstrap, как и элементы `button`, поэтому он будет трансформирован вспомогательной функцией дескриптора.

Листинг 23.16. Модификация элемента a в файле Create.cshtml из папки Views/Home

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>

  <button type="submit" bs-button-color="danger">Add</button>
  <button type="reset" class="btn btn-primary" >Reset</button>
  <a bs-button-color="primary" href="/Home/Index">Cancel</a>
</form>
```

Возможность расширения области действия для вспомогательной функции дескриптора означает, что вы не обязаны создавать вспомогательные функции дескрипторов, которые повторяют одну и ту же операцию в отношении разных типов элементов. Однако требуется определенная осторожность, поскольку довольно легко

создать вспомогательную функцию дескриптора, которая по мере будущего развития содержимого представлений в приложении начнет давать совпадения с элементами слишком широко. Более сбалансированный подход предусматривает применение атрибута `HtmlTargetElement` несколько раз с указанием полного набора элементов, которые будут трансформироваться, в виде комбинации узко определенных сопоставлений (листинг 23.17).

Листинг 23.17. Балансировка области действия в файле `ButtonTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

  [HtmlTargetElement("button", Attributes = "bs-button-color", ParentTag = "form")]
  [HtmlTargetElement("a", Attributes = "bs-button-color", ParentTag = "form")]
  public class ButtonTagHelper : TagHelper {

    public string BsButtonColor { get; set; }

    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      output.Attributes.SetAttribute("class", $"btn btn-{BsButtonColor}");
    }
  }
}
```

Такая конфигурация обеспечивает тот же самый результат в приложении, но гарантирует, что вспомогательная функция дескриптора не будет вызывать проблем, если позже в процессе разработки по ряду причин атрибуты `bs-button-color` начнут добавляться к другим типам элементов.

Упорядочивание выполнения вспомогательных функций дескрипторов

В качестве общего правила рекомендуется использовать только одну вспомогательную функцию дескриптора с любым заданным HTML-элементом. Причина в том, что совсем несложно попасть в ситуацию, когда одна вспомогательная функция дескриптора не считается с трансформацией, примененной другой вспомогательной функцией дескриптора, переопределяя значения атрибутов или содержимое. Если вы нуждаетесь в применении нескольких вспомогательных функций дескрипторов, тогда можете управлять последовательностью их выполнения, устанавливая свойство `Order`, которое унаследовано из базового класса `TagHelper`. Управление последовательностью выполнения может помочь свести к минимуму конфликты между вспомогательными функциями дескрипторов, хотя столкнуться с проблемами по-прежнему довольно легко.

Усовершенствованные возможности вспомогательных функций дескрипторов

В предыдущем разделе было продемонстрировано, каким образом создавать и использовать базовую вспомогательную функцию дескриптора, но это лишь малая толика доступных возможностей. В последующих разделах будут показаны более сложные случаи применения вспомогательных функций дескрипторов и средств, которые они предоставляют.

Создание сокращающих элементов

Вспомогательные функции дескрипторов не ограничиваются трансформированием стандартных HTML-элементов и могут также использоваться для замены специальных элементов ходовым содержимым. Данная возможность позволяет сделать представления более краткими и прояснить их замысел. Заменим элементы `button` в представлении `Create.cshtml` специальными элементами, как показано в листинге 23.18.

Листинг 23.18. Добавление специальных элементов в файле `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }

<form method="post" action="/Home/Create">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>
  <formbutton type="submit" bg-color="danger" />
  <formbutton type="reset" />
  <a bs-button-color="primary" href="/Home/Index">Cancel</a>
</form>
```

Элемент `formbutton` не входит в спецификацию HTML и не распознается браузерами. Взамен мы собираемся применять элементы `formbutton` как сокращения для генерации элементов `button`, которые требуются форме. Добавим в папку `Infrastructure/TagHelper` файл класса по имени `FormButtonTagHelper.cs` с определением из листинга 23.19.

Совет. Когда приходится иметь дело со специальными элементами, которые не являются частью спецификации HTML, должен применяться атрибут `HtmlTargetElement` и указываться имя элемента (см. листинг 23.19). Соглашение о применении вспомогательных функций дескрипторов к элементам на основе имени класса работает только для имен стандартных элементов.

Листинг 23.19. Содержимое файла `FormButtonTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

  [HtmlTargetElement("formbutton")]
  public class FormButtonTagHelper : TagHelper {
```

```
    public string Type { get; set; } = "Submit";
    public string BgColor { get; set; } = "primary";
    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      output.TagName = "button";
      output.TagMode = TagMode.StartTagAndEndTag;
      output.Attributes.SetAttribute("class", $"btn btn-{BgColor}");
      output.Attributes.SetAttribute("type", Type);
      output.Content.SetContent(Type == "submit" ? "Add" : "Reset");
    }
  }
}
```

Метод `Process()` использует свойства объекта `TagHelperOuput` для генерации совершенно другого элемента. Свойство `TagName` применяется для указания элемента `button`, свойство `TagMode` — для указания на то, что элемент записывается с использованием открывающего и закрывающего дескрипторов, метод `Attributes.SetAttribute()` применяется для определения атрибута `class` со стилями Bootstrap, а свойство `Content` используется для установки содержимого элемента.

Совет. Обратите внимание в листинге 23.19 на установку атрибута `type` выходного элемента. Причина в том, что любой атрибут, для которого имеется свойство, определенное вспомогательной функцией дескриптора, в выходном элементе опускается. Обычно это хорошая идея, т.к. она предотвращает появление в отправляемой браузеру HTML-разметке атрибутов, которые предназначены для конфигурирования вспомогательных функций дескрипторов. Тем не менее, в данном случае атрибут `type` используется для конфигурирования вспомогательной функции дескриптора, а потому необходимо, чтобы он присутствовал также в выходном элементе.

Установка свойства `TagName` важна из-за того, что выходной элемент по умолчанию записывается в таком же стиле, как специальный элемент. В листинге 23.18 применялся самозакрывающийся дескриптор:

```
...
<formbutton type="submit" bg-color="danger" />
...
```

Поскольку выходной элемент должен включать содержимое, нужно явно указать значение перечисления `TagMode.StartTagAndEndTag`, чтобы использовались отдельные открывающий и закрывающий дескрипторы.

Свойство `Content` возвращает экземпляр класса `TagHelperContent`, который применяется для установки содержимого элементов. В табл. 23.6 описаны самые важные методы класса `TagHelperContent`.

В листинге 23.19 вспомогательная функция дескриптора использует метод `SetContent()` для установки содержимого выходного элемента на основе значения атрибута `type`, которое предоставляется через свойство `Type`. Запустив приложение и запросив URL вида `/Home/Create`, можно обнаружить, что специальные элементы `formbutton` были заменены стандартными HTML-элементами. В итоге приведенные ниже элементы:

```
...
<formbutton type="submit" bg-color="danger" />
<formbutton type="reset" />
...
```

Таблица 23.6. Полезные методы класса `TagHelperContent`

Имя	Описание
`SetContent(text)`	Этот метод устанавливает содержимое выходного элемента. Аргумент `string` кодируется так, чтобы безопасно вставляться в HTML-элемент
`SetHtmlContent(html)`	Этот метод устанавливает содержимое выходного элемента. Предполагается, что аргумент `string` закодирован как безопасный. Должен использоваться с осторожностью
`Append(text)`	Этот метод кодирует безопасным образом указанный аргумент `string` и добавляет результат к содержимому выходного элемента
`AppendHtml(html)`	Этот метод добавляет указанный аргумент `string` к содержимому выходного элемента, не выполняя кодирование. Должен применяться с осторожностью
`Clear()`	Этот метод удаляет содержимое выходного элемента

трансформировались следующим образом:

```
<button class="btn btn-danger" type="submit">Add</button>
<button class="btn btn-primary" type="reset">Reset</button>
```

Вставка перед и после содержимого и элементов

Класс `TagHelperOutput` предлагает четыре свойства, которые облегчают внедрение нового содержимого в представление, так что оно окружает элемент или его содержимое (табл. 23.7). Далее будет показано, как добавлять содержимое вокруг и внутрь целевого элемента.

Таблица 23.7. Свойства класса `TagHelperOutput`, предназначенные для дополнения содержимого и элементов

Имя	Описание
`PreElement`	Это свойство применяется для вставки элементов в представление перед целевым элементом
`PostElement`	Это свойство используется для вставки элементов в представление после целевого элемента
`PreContent`	Это свойство применяется для вставки содержимого в целевой элемент перед существующим содержимым
`PostContent`	Это свойство используется для вставки содержимого в целевой элемент после существующего содержимого

Добавление содержимого вокруг выходного элемента

Первые два свойства класса `TagHelperOuput`, `PreElement` и `PostElement`, применяются для вставки элементов в представление перед и после выходного элемента. Добавим в папку `Infrastructure/TagHelpers` файл класса по имени `ContentWrapperTagHelper.cs` и определим в нем класс вспомогательной функции дескриптора, как показано в листинге 23.20.

Листинг 23.20. Содержимое файла `ContentWrapperTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

  [HtmlTargetElement("div", Attributes = "title")]
  public class ContentWrapperTagHelper : TagHelper {

    public bool IncludeHeader { get; set; } = true;
    public bool IncludeFooter { get; set; } = true;

    public string Title { get; set; }

    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {

      output.Attributes.SetAttribute("class", "m-1 p-1");

      TagBuilder title = new TagBuilder("h1");
      title.InnerHtml.Append(Title);

      TagBuilder container = new TagBuilder("div");
      container.Attributes["class"] = "bg-info m-1 p-1";

      container.InnerHtml.AppendHtml(title);
      if (IncludeHeader) {
        output.PreElement.SetHtmlContent(container);
      }
      if (IncludeFooter) {
        output.PostElement.SetHtmlContent(container);
      }
    }
  }
}
```

Созданная вспомогательная функция дескриптора трансформирует элементы `div`, которые имеют атрибут `title`, с использованием свойств `PreElement` и `PostElement` для добавления верхнего и нижнего колонтитулов, окружающих выходной элемент.

При генерировании новых HTML-элементов можно применять стандартное форматирование строк C# для создания требуемого содержимого, но это неудобный и чреватый ошибками процесс за исключением разве что самых простых элементов. Более надежный подход предусматривает использование класса `TagBuilder`, который определен в пространстве имен `Microsoft.AspNetCore.Mvc.Rendering` и позволяет создавать элементы в лучше структурированной манере. Класс `TagHelperContent` определяет методы, принимающие объекты `TagBuilder`, которые облегчают создание HTML-содержимого во вспомогательных функциях дескрипторов.

В классе вспомогательной функции дескриптора `ContentWrapperTagHelper` с помощью класса `TagBuilder` создается элемент `h1`, содержащийся внутри элемента `div`, который был стилизован посредством классов Bootstrap. Имеются также необязательные атрибуты `include-header` и `include-footer` типа `bool`, применяемые для указания, куда внедрять содержимое; по умолчанию оно добавляется перед и после выходного элемента. В листинге 23.21 приведена обновленная компоновка, содержащая элемент, который будет трансформироваться вспомогательной функцией дескриптора.

Листинг 23.21. Включение вспомогательной функции дескриптора в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div title="Cities">@RenderBody()</div>
</body>
</html>
```

Запуск приложения приводит к тому, что вспомогательная функция дескриптора применяется повсюду в приложении, добавляя верхний и нижний колонтитулы к каждой странице (рис. 23.5).

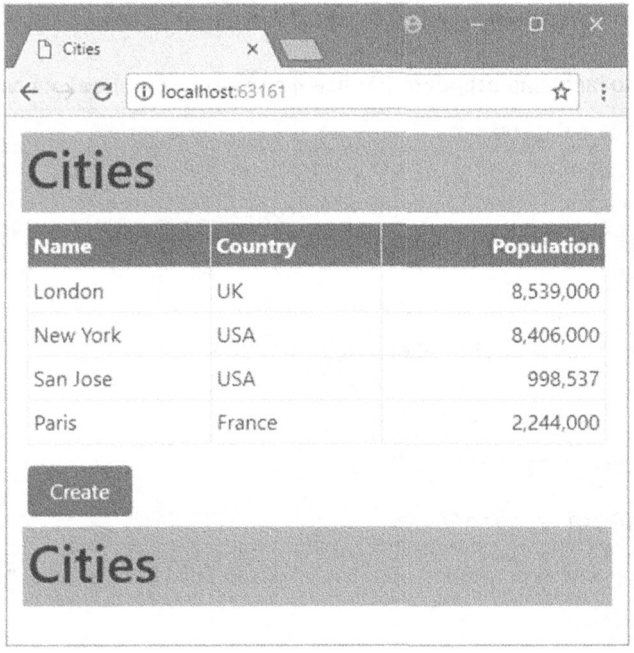


Рис. 23.5. Вставка HTML-элементов с помощью вспомогательной функции дескриптора

Вставка содержимого внутрь выходного элемента

Свойства `PreContent` и `PostContent` используются для вставки содержимого внутрь выходного элемента, окружая его исходное содержимое. Добавим в папку `Infrastructure/TagHelpers` файл класса по имени `TableCellTagHelper.cs` с определением из листинга 23.22.

Листинг 23.22. Содержимое файла `TableCellTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

  [HtmlTargetElement("td", Attributes = "wrap")]
  public class TableCellTagHelper : TagHelper {

    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      output.PreContent.SetHtmlContent("<b><i>");
      output.PostContent.SetHtmlContent("</i></b>");
    }
  }
}
```

Класс вспомогательной функции дескриптора `TableCellTagHelper` оперирует на элементах `td` с атрибутом `wrap` и помещает элементы `b` и `i` вокруг содержимого выходного элемента. В листинге 23.23 демонстрируется добавление атрибута `wrap` к одной из ячеек таблицы внутри файла представления `Index.cshtml`.

Листинг 23.23. Добавление атрибута HTML в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<City>

@{ Layout = "_Layout"; }

<table class="table table-sm table-bordered">
  <thead class="bg-primary text-white">
    <tr>
      <th>Name</th>
      <th>Country</th>
      <th class="text-right">Population</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var city in Model) {
      <tr>
        <td wrap>@city.Name</td>
        <td>@city.Country</td>
        <td class="text-right">@city.Population?.ToString("#,###")</td>
      </tr>
    }
  </tbody>
</table>
<a href="/Home/Create" class="btn btn-primary">Create</a>
```

После запуска приложения будет видно, что первая колонка ячеек в таблице, в которой перечислены объекты `City`, отображается полужирным курсивом. Заглянув в HTML-разметку, отправленную браузеру, можно заметить, что содержимое, добавленное через свойства `PreContent` и `PostContent`, находится с обеих сторон исходного содержимого элемента:

```
...
<tr>
  <td wrap><b><i>London</i></b></td>
  <td>UK</td>
  <td class="text-right">8,539,000</td>
</tr>
...
```

Совет. Обратите внимание, что атрибут `wrap` остался в выходном элементе. Причина в том, что во вспомогательной функции дескриптора не было определено свойство, соответствующее упомянутому атрибуту. Если вы хотите предотвратить попадание атрибутов в вывод, тогда определите для них свойства во вспомогательной функции дескриптора, даже когда не собираетесь применять их значения.

Получение данных контекста представления и использование внедрения зависимостей

Одним из наиболее распространенных применений классов вспомогательных функций дескрипторов, включая встроенные классы, которые рассматриваются в главах 24 и 25, является трансформация элементов так, чтобы они содержали детали текущего запроса или текущей модели представления. Добавим в папку `Infrastructure/TagHelpers` файл класса по имени `FormTagHelper.cs`, содержимое которого приведено в листинге 23.24.

Листинг 23.24. Содержимое файла `FormTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
namespace Cities.Infrastructure.TagHelpers {
  public class FormTagHelper : TagHelper {
    private IUrlHelperFactory urlHelperFactory;
    public FormTagHelper(IUrlHelperFactory factory) {
      urlHelperFactory = factory;
    }
    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewContextData { get; set; }
    public string Controller { get; set; }
    public string Action { get; set; }
    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContextData);
      output.Attributes.SetAttribute("action", urlHelper.Action(
        Action ??
          ViewContextData.RouteData.Values["action"].ToString(),
        Controller ??
          ViewContextData.RouteData.Values["controller"].ToString()));
    }
  }
}
```

Как и можно было предположить по имени, класс FormTagHelper оперирует на элементах form, устанавливая их атрибуты action для указания, куда будут отправляться данные формы. Если элемент form имеет атрибуты controller и action, то их значения будут использоваться для генерации целевого URL; в противном случае будут применяться значения controller и action из данных маршрутизации для текущего запроса. Для получения данных контекста понадобится добавить свойство по имени ViewContextData и декорировать его двумя атрибутами:

```
...
[ViewContext]
[HtmlAttributeNotBound]
public ViewContext ViewContextData { get; set; }
...
```

Атрибут ViewContext указывает на то, что значение свойства ViewContextData должно присваиваться объекту ViewContext, когда создается новый экземпляр класса FormTagHelper, как объяснялось в главе 18. Класс ViewContext предоставляет детали о визуализируемом представлении, о данных маршрутизации и о текущем HTTP-запросе (см. главу 21).

Атрибут HtmlAttributeNotBound предотвращает присваивание инфраструктурой MVC значения свойству ViewContextData, если HTML-элемент input имеет атрибут view-context. Это рекомендуемый прием, особенно в случае создания вспомогательных функций дескрипторов для использования другими разработчиками.

Совет. Существует встроенная вспомогательная функция дескриптора для элемента form, которая может применяться при нацеливании на методы действий и должна использоваться в реальных проектах. Вспомогательная функция дескриптора, рассматриваемая в настоящем разделе, предназначена просто для демонстрации, как можно применять данные контекста. За деталями об этой встроенной вспомогательной функции дескриптора обращайтесь в главу 24.

Вспомогательные функции дескрипторов могут объявлять в своих конструкторах зависимости от служб, которые распознаются с использованием средства внедрения зависимостей. В настоящем примере объявлена зависимость от службы IUrlHelperFactory, которая позволяет создавать исходящие URL из данных маршрутизации (и является службой, лежащей в основе свойства Url, которое предоставляется классом Controller, описанным в главе 16). Внутри метода Process() вспомогательная функция дескриптора применяет метод IUrlHelperFactory.GetUrlHelper() для получения объекта реализации IUrlHelper, который конфигурируется с помощью объекта ViewContext и затем используется при создании URL для атрибута action выходного элемента. В листинге 23.25 демонстрируется подготовленное представление, из которого удален атрибут action, так что его можно устанавливать посредством вспомогательной функции дескриптора.

Листинг 23.25. Удаление атрибута action из элемента form в файле Create.cshtml из папки Views/Home

```
@model City
@{ Layout = "_Layout"; }
<form method="post">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
```

```
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>
  <formbutton type="submit" bg-color="danger" />
  <formbutton type="reset" />
  <a bs-button-color="primary" href="/Home/Index">Cancel</a>
</form>
```

Запустив приложение, запросив URL вида `/Home/Create` и просмотрев HTML-разметку, отправленную браузеру, можно заметить, что элемент `form` имеет атрибут `action`, значение которого получено с применением данных контекста:

```
...
<form method="post" action="/Home/Create">
...
```

Работа с моделью представления

Вспомогательные функции дескрипторов могут оперировать на модели представления, подгоняя выполняемые ими трансформации или создаваемый вывод. Добавим в папку `Infrastructure/TagHelpers` файл класса по имени `LabelAndInputTagHelper.cs` с содержимым из листинга 23.26.

Листинг 23.26. Содержимое файла `LabelAndInputTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
namespace Cities.Infrastructure.TagHelpers {
  [HtmlTargetElement("label", Attributes = "helper-for")]
  [HtmlTargetElement("input", Attributes = "helper-for")]
  public class LabelAndInputTagHelper : TagHelper {
    public ModelExpression HelperFor { get; set; }
    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      if (output.TagName == "label") {
        output.TagMode = TagMode.StartTagAndEndTag;
        output.Content.Append(HelperFor.Name);
        output.Attributes.SetAttribute("for", HelperFor.Name);
      } else if (output.TagName == "input") {
        output.TagMode = TagMode.SelfClosing;
        output.Attributes.SetAttribute("name", HelperFor.Name);
        output.Attributes.SetAttribute("class", "form-control");
        if (HelperFor.Metadata.ModelType == typeof(int?)) {
          output.Attributes.SetAttribute("type", "number");
        }
      }
    }
  }
}
```

Класс вспомогательной функции дескриптора `LabelAndInputTagHelper` трансформирует элементы `label` и `input`, которые имеют атрибут `helper-for`. Важной частью класса `LabelAndInputTagHelper` является тип свойства `HelperFor`, которое используется для получения значения атрибута `helper-for`:

```
...
public ModelExpression HelperFor { get; set; }
...
```

Класс `ModelExpression` применяется, когда нужно оперировать с частью модели представления, что легче всего объяснить, продвинувшись дальше и показав, как вспомогательная функция дескриптора используется в представлении (листинг 23.27).

Листинг 23.27. Применение вспомогательной функции дескриптора в файле `Create.cshtml` из папки `Views/Home`

```
@model Cities.Models.City

@{ Layout = "_Layout"; }

<form method="post">
  <div class="form-group">
    <label helper-for="Name" />
    <input helper-for="Name" />
  </div>
  <div class="form-group">
    <label helper-for="Country" />
    <input helper-for="Country" />
  </div>
  <div class="form-group">
    <label helper-for="Population"/>
    <input helper-for="Population" />
  </div>
  <formbutton type="submit" bg-color="danger" />
  <formbutton type="reset" />
  <a bs-button-color="primary" href="/Home/Index">Cancel</a>
</form>
```

Значение атрибута `helper-for` — это свойство из класса `Model`, которое обнаруживается инфраструктурой MVC и предоставляется вспомогательной функции дескриптора в виде объекта `ModelExpression`.

Класс `ModelExpression` здесь подробно не рассматривается, потому что любой анализ типов приводит к бесконечным спискам классов и свойств. Кроме того, в MVC имеется удобный набор встроенных вспомогательных функций дескрипторов, используемых моделью представления для трансформации элементов, как описано в главе 24, т.е. создавать собственные классы такого рода вам не придется.

В примере вспомогательной функции дескриптора задействованы две базовые возможности, которые полезно рассмотреть. Первая из них — получение имени свойства модели, так что его можно включить в выходной элемент:

```
...
output.Content.Append(HelperFor.Name);
output.Attributes.SetAttribute("for", HelperFor.Name);
...
```

Свойство Name возвращает имя свойства модели. Вторая возможность связана с получением типа свойства модели, что позволяет изменить значение атрибута type элементов input:

```
...
if (HelperFor.Metadata.ModelType == typeof(int?)) {
  output.Attributes.SetAttribute("type", "number");
}
...
```

Если запустить приложение, запросить URL вида /Home/Create и просмотреть HTML-разметку, которая была отправлена браузеру, то в ней обнаружатся следующие элементы:

```
<div class="form-group">
  <label for="Name">Name</label>
  <input name="Name" class="form-control" />
</div>
<div class="form-group">
  <label for="Country">Country</label>
  <input name="Country" class="form-control" />
</div>
<div class="form-group">
  <label for="Population">Population</label>
  <input name="Population" class="form-control" type="number" />
</div>
```

Атрибут type элемента input по имени Population был установлен в number, подчеркивая тот факт, что свойство City.Population в классе C# имеет тип int; это демонстрирует возможность отражения различных характеристик модели в HTML-разметке, генерируемой вспомогательной функцией дескриптора. В зависимости от применяемого браузера элемент input будет разрешать ввод только чисел.

Согласование вспомогательных функций дескрипторов

Свойство TagHelperContext.Items предоставляет словарь, который используется для согласования вспомогательных функций дескрипторов, оперирующих на элементах и их дочерних элементах. Чтобы ознакомиться с применением коллекции Items, добавим в папку Infrastructure/TagHelpers файл класса по имени CoordinatingTagHelpers.cs и определим в нем пару классов вспомогательных функций дескрипторов, как показано в листинге 23.28.

Листинг 23.28. Содержимое файла CoordinatingTagHelpers.cs из папки Infrastructure/TagHelpers

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {

  [HtmlTargetElement("div", Attributes = "theme")]
  public class ButtonGroupThemeTagHelper : TagHelper {
    public string Theme { get; set; }
    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      context.Items["theme"] = Theme;
    }
  }
```

```
  [HtmlTargetElement("button", ParentTag = "div")]
  [HtmlTargetElement("a", ParentTag = "div")]
  public class ButtonThemeTagHelper : TagHelper {

    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      if (context.Items.ContainsKey("theme")) {
        output.Attributes.SetAttribute("class",
          $"btn btn-{context.Items["theme"]}");
      }
    }
  }
}
```

Первый класс вспомогательной функции дескриптора, `ButtonGroupThemeTagHelper`, оперирует на элементах `div`, которые имеют атрибут `theme`. Согласование вспомогательных функций дескрипторов может трансформировать их собственные элементы, но в рассматриваемом примере оно просто добавляет значение атрибута `theme` в словарь `Items`, так что данное значение становится доступным вспомогательным функциям дескрипторов, которые оперируют на элементах, содержащихся внутри элемента `div`.

Второй класс вспомогательной функции дескриптора, `ButtonThemeTagHelper`, оперирует на элементах `button` и `a`, находящихся внутри элемента `div`. Он использует значение атрибута `theme` из словаря `Items`, чтобы установить стиль Bootstrap для своего выходного элемента. В листинге 23.20 приведен набор элементов, к которым будут применены только что созданные вспомогательные функции дескрипторов.

Листинг 23.29. Применение согласованных вспомогательных функций дескрипторов в файле `Create.cshtml` из папки `Views/Home`

```
@model Cities.Models.City

@{ Layout = "_Layout"; }

<form method="post">
  <div class="form-group">
    <label helper-for="Name" />
    <input helper-for="Name" />
  </div>
  <div class="form-group">
    <label helper-for="Country" />
    <input helper-for="Country" />
  </div>
  <div class="form-group">
    <label helper-for="Population" />
    <input helper-for="Population" />
  </div>
  <div theme="primary">
    <button type="submit">Add</button>
    <button type="reset">Reset</button>
    <a href="/Home/Index">Cancel</a>
  </div>
</form>
```

Запустив приложение и запросив URL вида `/Home/Create`, легко заметить, что группа кнопок стилизована одинаково. Если заменить значение атрибута `theme` элемента `div` другой настройкой темы Bootstrap, такой как `info`, `danger` или `primary`, и затем перезагрузить страницу, то изменение отразится в стилях кнопок (рис. 23.6).

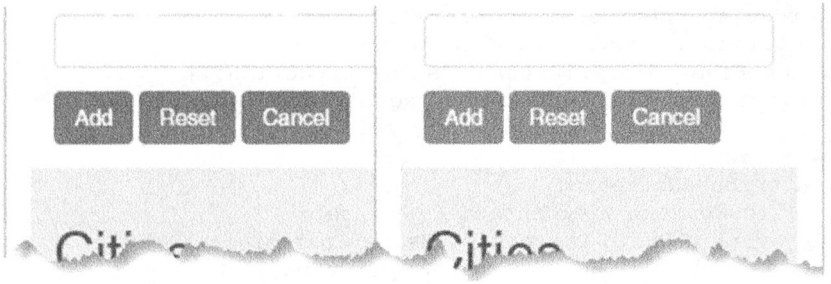


Рис. 23.6. Согласование вспомогательных функций дескрипторов

Подавление выходного элемента

Вспомогательные функции дескрипторов могут использоваться для того, чтобы предотвратить включение элемента в HTML-разметку, отправляемую браузеру, за счет вызова метода `SuppressOuput()` на объекте `TagHelperOutput`, который получается в качестве аргумента метода `Process()`. В листинге 23.30 к разделяемой компоновке добавляется элемент, который отображает хорошо видимое сообщение, но только для запросов, направленных на заданное действие.

Листинг 23.30. Добавление хорошо видимого сообщения в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div show-for-action="Index" class="m-1 p-1 bg-danger">
    <h2>Important Message</h2>
  </div>
  <div title="Cities">@RenderBody()</div>
</body>
</html>
```

Атрибут `show-for-action` указывает имя действия, для которого необходимо отобразить сообщение. Такой подход к управлению включением содержимого нельзя считать приемлемым в реальном приложении, но его вполне достаточно для примера приложения с единственным контроллером и двумя действиями. Добавим в папку `Infrastructure/TagHelpers` файл класса по имени `SelectiveTagHelper.cs` с содержимым из листинга 23.31.

Листинг 23.31. Содержимое файла `SelectiveTagHelper.cs` из папки `Infrastructure/TagHelpers`

```
using System;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
namespace Cities.Infrastructure.TagHelpers {
  [HtmlTargetElement(Attributes = "show-for-action")]
  public class SelectiveTagHelper : TagHelper {
    public string ShowForAction { get; set; }
    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewContext { get; set; }
    public override void Process(TagHelperContext context,
                                 TagHelperOutput output) {
      if (!ViewContext.RouteData.Values["action"].ToString()
           .Equals(ShowForAction, StringComparison.OrdinalIgnoreCase)) {
        output.SuppressOutput();
      }
    }
  }
}
```

С помощью объекта `ViewContext` класс вспомогательной функции дескриптора получает значение `action` из данных маршрутизации и сравнивает его со значением атрибута `show-for-action` в HTML-элементе. Если они не совпадают, тогда вызывается метод `SuppressOutput()`. Чтобы взглянуть на результат, запустим приложение и запросим URL вида `/Home/Index` и `/Home/Create`. Как показано на рис. 23.7, сообщение отображается только в случае нацеливания на действие `Index`.

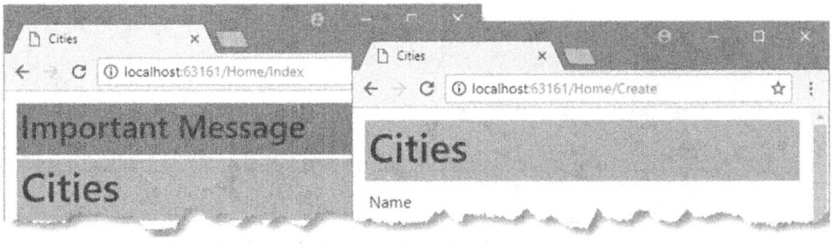


Рис. 23.7. Подавление элементов с применением вспомогательной функции дескриптора

Резюме

В главе рассматривалось использование вспомогательных функций дескрипторов, которые являются новым дополнением инфраструктуры ASP.NET Core MVC. Была объяснена роль, которую они играют в представлении Razor, а также продемонстрировано создание, регистрация и применение специальных вспомогательных функций дескрипторов. Вы узнали, как управлять областью действия вспомогательной функции дескриптора, и ознакомились с разнообразными способами, которыми вспомогательные функции дескрипторов могут трансформировать HTML-элементы. В следующей главе будут описаны вспомогательные функции дескрипторов, которые используются для работы с элементами HTML-форм.

ГЛАВА 24

Использование вспомогательных функций дескрипторов для форм

Инфраструктура MVC предлагает набор встроенных вспомогательных функций дескрипторов, которые используются для выполнения часто требующихся трансформаций в отношении HTML-элементов. В настоящей главе рассматриваются вспомогательные функции дескрипторов, которые оперируют на HTML-формах, включая элементы `form`, `input`, `label`, `select`, `option` и `textarea`. В главе 25 будут описаны другие встроенные вспомогательные функции дескрипторов, предлагающие средства, которые не относятся к формам. В табл. 24.1 приведена сводка, позволяющая поместить вспомогательные функции дескрипторов для форм в контекст.

Таблица 24.1. Помещение вспомогательных функций дескрипторов для форм в контекст

Вопрос	Ответ
Что это такое?	Вспомогательные функции дескрипторов для форм применяются при трансформации элементов HTML-форм, поэтому вам не придется писать собственные классы такого рода, чтобы решать наиболее распространенные задачи
Чем они полезны?	Вспомогательные функции дескрипторов для форм обеспечивают согласованную генерацию элементов внутри HTML-форм, таких как `label` и `input`. Большей частью вспомогательные функции дескрипторов гарантируют, что важные атрибуты наподобие `id`, `name` и `for` устанавливаются напрямую с использованием классов моделей представлений, но некоторые вспомогательные функции дескрипторов способны также генерировать содержимое, скажем, заполняя элементы `select` элементами `option`
Как они используются?	Встроенные вспомогательные функции дескрипторов ищут атрибуты с префиксом `asp-`, такие как `asp-for`

Окончание табл. 24.1

Вопрос	Ответ
Существуют ли какие-то скрытые ловушки или ограничения?	Единственное ограничение связано со способом, которым данные модели должны предоставляться вспомогательной функции дескриптора, генерирующей элементы `option` внутри элементов `select`. Описание проблемы и специальной вспомогательной функции дескриптора, которая ее решает, приведено в разделе "Работа с элементами `select` и `option`" далее в главе
Существуют ли альтернативы?	Создавать HTML-формы в представлениях возможно вообще без применения атрибутов вспомогательных функций дескрипторов. Можно также разрабатывать собственные вспомогательные функции дескрипторов, используя приемы, которые были описаны в главе 23

В табл. 24.2 приведена сводка по главе.

Таблица 24.2. Сводка по главе

Задача	Решение	Листинг
Установка атрибута `action` в элементе `form`	Используйте вспомогательную функцию дескриптора для элемента `form`	24.5
Препятствование подделке межсайтовых запросов	Примените атрибут `ValidateAntiForgeryToken` к методу действия и дополнительно установите атрибут `asp-antiforgery` в `true` для элемента `form`	24.6, 24.7
Установка атрибутов `id`, `name` и `value` в элементе `input`	Используйте атрибут `asp-for`	24.8
Форматирование значения, отображаемого элементом `input`	Примените атрибут `asp-format` к элементу `input` или атрибут `DisplayFormat` в классе модели	24.9–24.12
Установка атрибута `for` и содержимого элемента `label`	Используйте атрибут `asp-for`	24.13
Изменение содержимого элемента `label`, к которому был применен атрибут `asp-for`	Примените атрибут `Display` к свойству класса модели и с помощью свойства `Name` укажите содержимое	24.14
Установка атрибутов `id` и `name` в элементе `select`	Используйте атрибут `asp-for`	24.15
Генерация элементов `option`	Примените атрибут `asp-items`	24.16–24.21
Установка атрибутов `id` и `name` в элементе `textarea`	Используйте атрибут `asp-for`	24.22, 24.23

Подготовка проекта для примера

Мы продолжим работать с проектом `Cities`, созданным в главе 23. Для целей текущей главы необходимо включить встроенные вспомогательные функции дескрипторов, поступающие с инфраструктурой MVC, и отключить специальные вспомога-

тельные функции дескрипторов, которые были созданы в главе 23. В листинге 24.1 показаны изменения, внесенные в файл импортирования представлений, где выражение `@addTagHelper` для классов вспомогательных функций дескрипторов из сборки `Cities` заменено выражением, устанавливающим вместо них классы вспомогательных функций дескрипторов MVC, которые определены в сборке по имени `Microsoft.AspNetCore.Mvc.TagHelpers`.

Листинг 24.1. Изменение вспомогательных функций дескрипторов в файле `_ViewImports.cshtml` из папки `Views`

```
@using Cities.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Переустановка представлений и компоновки

В листинге 24.2 приведено содержимое представления `Index.cshtml`, из которого удалены атрибуты, используемые специальными вспомогательными функциями дескрипторов.

Листинг 24.2. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<City>

@{ Layout = "_Layout"; }

<table class="table table-sm table-bordered">
  <thead class="bg-primary text-white">
    <tr>
      <th>Name</th>
      <th>Country</th>
      <th class="text-right">Population</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var city in Model) {
      <tr>
        <td>@city.Name</td>
        <td>@city.Country</td>
        <td class="text-right">@city.Population?.ToString("#,###")</td>
      </tr>
    }
  </tbody>
</table>
<a href="/Home/Create" class="btn btn-primary">Create</a>
```

В листинге 24.3 показаны соответствующие изменения внутри файла `Create.cshtml`, которые возвращают использование стандартных HTML-элементов без атрибутов, применяемых в главе 23.

Листинг 24.3. Содержимое файла `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }
```

```
<form method="post" action="/Home/Create">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Последнее изменение касается разделяемой компоновки (листинг 24.4).

Листинг 24.4. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

В результате запуска приложения отобразится список городов; щелкнув на кнопке Create (Создать), можно заполнить форму и отправить новые данные серверу (рис. 24.1).

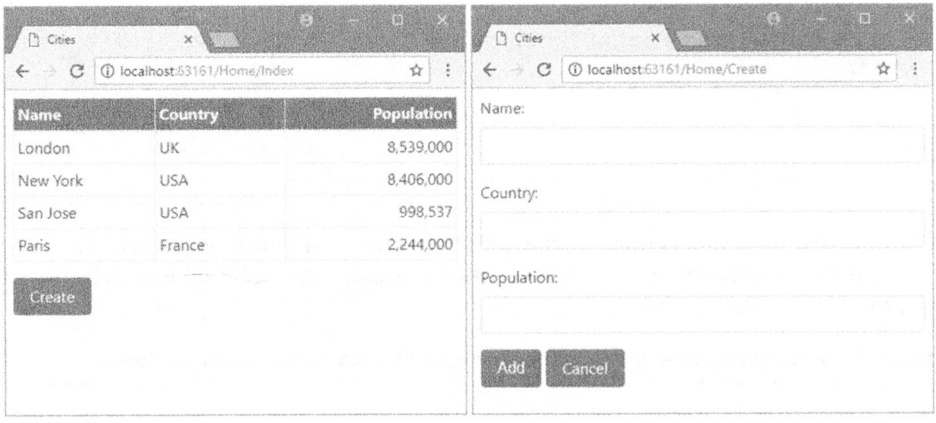


Рис. 24.1. Выполнение примера приложения

Работа с элементами form

Класс FormTagHelper реализует встроенную вспомогательную функцию дескриптора для элементов form, которая используется при управлении конфигурацией HTML-форм, так что их можно нацеливать на правильный метод действия, основываясь на конфигурации маршрутизации приложения. Класс FormTagHelper поддерживает атрибуты, описанные в табл. 24.3.

Таблица 24.3. Атрибуты встроенной вспомогательной функции дескриптора для элементов form

Имя	Описание
asp-controller	Этот атрибут применяется, чтобы указать системе маршрутизации значение controller для URL в атрибуте action. Если он опущен, тогда будет использоваться контроллер, визуализирующий представление
asp-action	Этот атрибут применяется, чтобы указать системе маршрутизации метод действия для URL в атрибуте action. Если он опущен, тогда будет использоваться действие, визуализирующее представление
asp-route-*	Атрибуты с именами, начинающимися с asp-route-, применяются для указания дополнительных значений URL в атрибуте action; таким образом, атрибут asp-route-id используется для предоставления системе маршрутизации значения сегмента id
asp-route	Этот атрибут применяется для указания имени маршрута, который будет использоваться при генерации URL в атрибуте action
asp-area	Этот атрибут применяется для указания имени области, которая будет использоваться при генерации URL в атрибуте action
asp-antiforgery	Этот атрибут управляет добавлением информации, противодействующей подделке, как объясняется в разделе "Использование средства противодействия подделке" далее в главе

Установка цели формы

Основным назначением класса FormTagHelper является установка атрибута action элемента form с применением конфигурации маршрутизации приложения, гарантируя тем самым, что данные формы всегда посылаются корректному URL, даже когда схема маршрутизации изменяется. В листинге 24.5 атрибуты asp-action и asp-controller используются для нацеливания на метод действия Create() контроллера Home.

На заметку! Вспомогательная функция дескриптора не устанавливает атрибут method, и если он отсутствует в элементе form, тогда для отправки данных формы клиенту браузер будет применять запрос GET. Как объяснялось в главе 17, это может вызвать проблемы, если данные формы используются для модификации данных в приложении. Рекомендуется устанавливать атрибут method, даже если нужны запросы GET, чтобы сделать очевидным тот факт, что вы не забыли выбрать тип метода.

Листинг 24.5. Установка цели формы в файле `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Запустив приложение, запросив URL вида `/Home/Create` и просмотрев HTML-разметку, которая была отправлена клиенту, можно заметить, что вспомогательная функция дескриптора добавила к элементу `form` атрибут `action` и установила его значение с применением системы маршрутизации:

```
<form method="post" action="/Home/Create">
```

Использование средства противодействия подделке

Подделка межсайтовых запросов (cross-site request forgery — CSRF) — это способ злоумышленной эксплуатации веб-приложения, направленный на то, чтобы задействовать в своих интересах метод аутентификации пользовательских запросов. В большинстве веб-приложений, включая создаваемые с использованием ASP.NET Core, идентификация запросов, относящихся к специфическому сеансу, с которым обычно ассоциирован пользователь, осуществляется с применением cookie-наборов.

Атака CSRF (также называемая *мистификацией сеанса*) опирается на ситуацию, когда пользователь посещает злоумышленный веб-сайт после работы со своим веб-приложением, явно не закончив свои сеансы щелчком на кнопке выхода из приложения. Приложение по-прежнему считает сеанс пользователя активным, а срок действия cookie-набора, который браузер сохранил, пока еще не истек. Злоумышленный веб-сайт содержит код JavaScript, отправляющий приложению запрос формы, который выполняет операцию без согласия пользователя; при этом природа операции будет зависеть от приложения, подвергаемого атаке. Поскольку код JavaScript выполняется браузером пользователя, запрос к приложению включает cookie-набор сеанса и приложение выполняет операцию безо всякого уведомления пользователя или согласия с его стороны.

Если элемент `form` не содержит атрибут `action` (из-за того, что он генерируется системой маршрутизации с помощью атрибутов `asp-controller` и `asp-acton`), то класс `FormTagHelper` автоматически включает средство противостояния атакам CSRF. Указанное средство добавляет к форме скрытый элемент `input` с марке-

ром безопасности внутри HTML-разметки, которая отправляется клиенту наряду с cookie-набором. Приложение будет обрабатывать запрос, только если он содержит и cookie-набор, и скрытое значение формы, к которому злоумышленный веб-сайт не имеет доступа. Каждый запрос к форме генерирует новый уникальный набор маркеров безопасности.

Запустим приложение, запросим URL вида `/Home/Create` и просмотрим HTML-разметку, отправленную браузеру; в ней обнаружится скрытый элемент `input` вроде показанного ниже:

```
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8KuVkH8hFlRApeFBxTrhCFTKZe0B9BKwnWDJqLRUDk__
PrEwaeCJmiBbGkwW1ZI816c_TrM5XQkJBeqNI5IL8FhuORvjZuYIL-
GZvnWZ62OThsZYT02HNX_Lu5LWDNWDdVoS5O5hZtzaoHLeY5lNto" />
```

Воспользовавшись инструментами браузера, доступными по нажатию клавиши <F12>, можно также увидеть соответствующий cookie-набор, который добавляется к ответу. Добавление маркеров безопасности к HTML-ответам — лишь часть процесса; они должны также проверяться контроллером, как продемонстрировано в листинге 24.6.

Листинг 24.6. Проверка маркеров противодействия подделке в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using Cities.Models;

namespace Cities.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }
    public ViewResult Index() => View(repository.Cities);
    public ViewResult Create() => View();

    [HttpPost]
    [ValidateAntiForgeryToken]
    public IActionResult Create(City city) {
      repository.AddCity(city);
      return RedirectToAction("Index");
    }
  }
}
```

Атрибут `ValidateAntoForgeryToken` гарантирует, что запрос содержит допустимые маркеры противодействия атакам CSRF, и будет генерировать исключение, если они отсутствуют или не имеют ожидаемые значения.

Класс `FormTagHelper` предлагает атрибут `asp-antiforgery` для переопределения стандартного поведения противодействия атакам CSRF. При установке этого атрибута в `true` маркеры безопасности будут включаться в запросы, даже если элемент `form` имеет атрибут `action`. Когда значением атрибута `asp-antiforgery` является `false`, маркеры безопасности будут отключены. В листинге 24.7 средство противодействия атакам CSRF включается явно, хотя маркеры безопасности были бы добавлены в любом случае, потому что для элемента `form` не определен атрибут `action`.

Листинг 24.7. Включение средства противостояния атакам CSRF в файле `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" name="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" name="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" name="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Совет. Тестирование средства противодействия атакам CSRF связано с некоторыми ухищрениями. Оно делается путем запрашивания URL, который содержит форму (`/Home/Create` в рассматриваемом примере), и последующего применения инструментов <F12> браузера для нахождения и удаления из формы скрытого элемента `input` (либо изменения его значения). После заполнения формы и ее отправки приложению браузер не будет иметь одну часть требующихся данных, в результате чего запрос должен завершиться неудачей с отображением страницы ошибки.

Работа с элементами `input`

Элемент `input` является основой HTML-форм и предоставляет главные средства, с помощью которых пользователь может снабжать приложение неструктурированными данными. Класс `InputTagHelper` используется для трансформирования элементов `input`, чтобы они отражали тип данных и формат свойства модели представления, применяемого для сбора информации, с использованием атрибутов из табл. 24.4.

Таблица 24.4. Атрибуты встроенной вспомогательной функции дескриптора для элементов `input`

Имя	Описание
`asp-for`	Этот атрибут применяется для указания свойства модели представления, которое олицетворяет элемент `input`
`asp-format`	Этот атрибут используется для указания формата, применяемого при отображении значения свойства модели представления, которое олицетворяет элемент `input`

Конфигурирование элементов input

Атрибут asp-for устанавливается в имя свойства модели представления, которое затем используется для установки атрибутов name, id, type и value элемента input. В листинге 24.8 демонстрируется применение атрибута asp-for к элементам input внутри представления Create.cshtml.

Листинг 24.8. Конфигурирование элементов input в файле Create.cshtml из папки Views/Home

```
@model City
@{ Layout = "_Layout"; }
<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" asp-for="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" asp-for="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

После запуска приложения и запрашивания URL вида /Home/Create обнаружится, что вспомогательная функция дескриптора использовала свойство, указанное в атрибуте asp-for, для подстройки каждого элемента input, как показано в следующем фрагменте (маркер безопасности, защищающий от атак CSRF, опущен):

```
<form method="post" action="/Home/Create">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" type="text" id="Name"
           name="Name" value="" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" type="text" id="Country"
           name="Country" value="" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" type="number" id="Population"
          name="Population" value="" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Атрибут type элемента input сообщает браузеру о том, как отображать элемент в форме. Простой результат этого процесса можно просмотреть в элементе input для свойства Population, атрибут type которого был установлен в number. Дело в том, что типом C# свойства Population является int?; таким образом, вспомогательная функция дескриптора применяет атрибут type, чтобы указать браузеру о принятии только числовых значений.

На заметку! Способ интерпретации атрибута type возлагается на браузер. Не все браузеры реагируют на все значения атрибута type, которые определены в спецификации HTML5, и даже если они реагируют, то существуют отличия в том, каким образом они это делают. Атрибут type может быть удобной подсказкой о разновидности данных, которые ожидаются внутри формы, но чтобы гарантировать предоставление пользователями пригодных данных, обязательно должно использоваться средство проверки достоверности модели, как объясняется в главе 27.

В табл. 24.5 описан способ применения различных типов свойств C# для установки атрибута type элементов input.

Таблица 24.5. Типы свойств C# и атрибуты type элементов input, которые они генерируют

Тип C#	Атрибут type элемента input
byte, sbyte, int, uint, short, ushort, long, ulong	number
float, double, decimal	text с дополнительными атрибутами для проверки достоверности модели, как будет показано далее в главе
bool	checkbox
string	text
DateTime	datetime

Типы float, double и decimal производят элементы input с атрибутом type, установленным в text, т.к. не все браузеры разрешают использовать полный диапазон символов для выражения допустимых значений указанных типов. Чтобы оказать содействие пользователю, вспомогательная функция дескриптора добавляет к элементу input атрибуты, которые применяются со средством проверки достоверности модели (глава 27).

Стандартные отображения из табл. 24.5 можно переопределить, определив атрибут type элемента input. Вспомогательная функция дескриптора не будет переопределять указанное в type значение, что позволит задействовать разнообразные доступные типы элементов input вроде password или hidden либо новые типы, появившиеся в HTML5, наподобие number.

Недостаток такого подхода в том, что приходится помнить о необходимости установки атрибута type во всех представлениях, где элементы input генерируются для заданного свойства модели. Если нужно переопределить стандартное отображение во множестве представлений, тогда можно применить атрибут UIHint к свойству в классе модели C# и указать в качестве его аргумента одно из значений, перечисленных в табл. 24.6.

Совет. Вспомогательная функция дескриптора будет устанавливать атрибут `type` элементов `input` в `text`, когда тип свойства модели не является одним из описанных в табл. 24.5 и не декорирован атрибутом `UIHint`.

Таблица 24.6. Аргументы атрибута `UIHint` и генерируемые ими значения атрибута `type` элементов `input`

Значение	Атрибут `type` элемента `input`
`HiddenInput`	`hidden`
`Password`	`password`
`Text`	`text`
`PhoneNumber`	`tel`
`Url`	`url`
`EmailAddress`	`email`
`Time`	`time` (это значение используется для отображения компонента времени объекта `DateTime`)
`Date`	`date` (это значение применяется для отображения компонента даты объекта `DateTime`)
`DateTime-local`	`datetime-local` (это значение используется для отображения объекта `DateTime` без информации о временном поясе)

Форматирование значений данных

Когда метод действия снабжает представление объектом модели представления, вспомогательная функция дескриптора применяет значение свойства, указанное в атрибуте `asp-for`, для установки атрибута `value` элемента `input`. Атрибут `asp-format` используется для указания, каким образом форматировать это значение данных.

Чтобы продемонстрировать сказанное, добавим в контроллер `Home` новый метод действия (листинг 24.9), который выбирает из хранилища первый объект `City` и применяет его в качестве модели представления для `Create.cshtml`.

Листинг 24.9. Добавление метода действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Cities.Models;
using System.Linq;

namespace Cities.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public ViewResult Index() => View(repository.Cities);

    public ViewResult Edit() => View("Create", repository.Cities.First());
```

```
    public ViewResult Create() => View();

    [HttpPost]
    [ValidateAntiForgeryToken]
    public IActionResult Create(City city) {
      repository.AddCity(city);
      return RedirectToAction("Index");
    }
  }
}
```

Запустив приложение, запросив URL вида `/Home/Edit` и просмотрев HTML-разметку, которая была отправлена браузеру, легко заметить, что атрибуты `value` заполнились с использованием объекта модели представления, например:

```
<input class="form-control" type="number" id="Population"
       name="Population" value="8539000" />
```

Атрибут `asp-format` принимает значение, которое будет передано стандартной системе форматирования строк C# (листинг 24.10).

Листинг 24.10. Форматирование значения данных в файле `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" asp-for="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" asp-for="Population" asp-format="{0:#,###}" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Значение атрибута применяется дословно, т.е. потребуется включить фигурные скобки и ссылку `0:`, а также желаемый формат. В результате запуска приложения и запроса URL вида `/Home/Edit`, можно увидеть, что значение `Population` было сформатировано следующим образом:

```
<input class="form-control" type="number" id="Population"
       name="Population" value="8,539,000" />
```

Данное средство должно использоваться с осторожностью, т.к. необходимо гарантировать, что остальная часть приложения сконфигурирована для поддержки подоб-

ного формата. В рассматриваемом случае возникает проблема из-за форматирования значения Population. Вспомогательная функция дескриптора установила атрибут type элемента input для свойства Population в number с применением стандартных отображений, описанных в табл. 24.5, но указанная строка формата сгенерировала атрибут value, который содержит нецифровые символы. В результате браузеры, поддерживающие тип элемента number (вспомните, что не все браузеры это делают), могут вообще не отобразить какое-либо значение в элементе.

В приложении также понадобится обеспечить возможность разбора значений в используемом формате. Пример приложения ожидает получать значение Population, которое может быть разобрано в int, так что значения, содержащие нецифровые символы, приведут к ошибкам проверки достоверности (глава 27).

Применение форматирования через класс модели

Если вы хотите всегда использовать для свойства модели одно и то же форматирование, тогда можете декорировать класс C# атрибутом DisplayFormat, который определен в пространстве имен System.ComponentModel.DataAnnotations. Для форматирования значения данных атрибут DisplayFormat требует двух аргументов: аргумент DataFormatString указывает строку формата, а аргумент ApplyFormatInEditMode — что форматирование должно применяться при редактировании значений. В листинге 24.11 свойство Population декорировано атрибутом DisplayFormat с использованием формата, который может быть обработан приложением и браузером как числовой.

Листинг 24.11. Применение атрибута форматирования к классу модели в файле City.cs из папки Models

```
using System.ComponentModel.DataAnnotations;

namespace Cities.Models {

  public class City {
    public string Name { get; set; }
    public string Country { get; set; }
    [DisplayFormat(DataFormatString = "{0:F2}", ApplyFormatInEditMode = true)]
    public int? Population { get; set; }
  }
}
```

Атрибут asp-format имеет преимущество перед атрибутом DisplayFormat, поэтому он удален из представления (листинг 24.12).

Листинг 24.12. Удаление атрибута форматирования в файле Create.cshtml из папки Views/Home

```
@model City

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
  <div class="form-group">
    <label for="Name">Name:</label>
    <input class="form-control" asp-for="Name" />
  </div>
```

```
  <div class="form-group">
    <label for="Country">Country:</label>
    <input class="form-control" asp-for="Country" />
  </div>
  <div class="form-group">
    <label for="Population">Population:</label>
    <input class="form-control" asp-for="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

В результате запуска приложения и запрашивания URL вида `/Home/Edit` обнаружится, что значение `Population` было сформатировано с двумя знаками после десятичной точки:

```
<input class="form-control" type="number" id="Population"
       name="Population" value="8539000.00" />
```

Работа с элементами `label`

Элемент `label` трансформируется классом `LabelTagHelper`, который использует класс модели представления, чтобы удостовериться в том, что метки не содержат опечаток и являются согласованными. Существует только один поддерживаемый атрибут, который описан в табл. 24.7.

Таблица 24.7. Атрибут встроенной вспомогательной функции дескриптора для элементов `label`

Имя	Описание
`asp-for`	Этот атрибут применяется для указания свойства модели представления, которое олицетворяет элемент `label`

Вспомогательная функция дескриптора будет использовать имя свойства модели представления для установки значения атрибута `for` и содержимого элемента `label`. В листинге 24.13 атрибут `asp-for` применяется к элементам `label` формы, которые будут трансформированы вспомогательной функцией дескриптора.

Листинг 24.13. Использование вспомогательной функции дескриптора в файле `Create.cshtml` из папки `Views/Home`

```
@model City

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
  <div class="form-group">
    <label asp-for="Name"></label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label asp-for="Country"></label>
```

```
    <input class="form-control" asp-for="Country" />
  </div>
  <div class="form-group">
    <label asp-for="Population"></label>
    <input class="form-control" asp-for="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Поскольку элементы `label` пусты, вспомогательная функция дескриптора будет применять имена свойств модели в качестве содержимого этих элементов и установит атрибут `for`, который сообщает браузеру о том, с каким элементом `input` ассоциирован каждый элемент `label`. Если запустить приложение, запросить URL вида `/Home/Create` или `/Home/Edit` и просмотреть HTML-разметку, отправленную браузеру, то обнаружатся следующие выходные элементы:

```
<form method="post" action="/Home/Create">
  <div class="form-group">
    <label for="Name">Name</label>
    <input class="form-control" type="text" id="Name"
           name="Name" value="London" />
  </div>
  <div class="form-group">
    <label for="Country">Country</label>
    <input class="form-control" type="text" id="Country"
           name="Country" value="UK" />
  </div>
  <div class="form-group">
    <label for="Population">Population</label>
    <input class="form-control" type="number" id="Population"
           name="Population" value="8539000.00" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Переопределить значение, используемое как содержимое элемента `label`, можно путем применения атрибута `Display` к свойству класса модели (листинг 24.14).

Листинг 24.14. Изменение описания для свойства модели в файле `City.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;

namespace Cities.Models {

  public class City {
    [Display(Name = "City")]
    public string Name { get; set; }
    public string Country { get; set; }
    [DisplayFormat(DataFormatString = "{0:F2}", ApplyFormatInEditMode = true)]
    public int? Population { get; set; }
  }
}
```

Аргумент Name указывает значение для использования вместо имени свойства. В результате запуска приложения, запрашивания URL вида /Home/Create и просмотра HTML-разметки, отправленной браузеру, выяснится, что содержимое элемента label изменилось:

```
<div class="form-group">
  <label for="Name">City</label>
  <input class="form-control" type="text" id="Name"
         name="Name" value="London" />
</div>
```

Обратите внимание, что значение атрибута for не изменилось, а потому браузеру известно об ассоциировании элемента label со специфическим элементом input, на который атрибут Display не влияет.

Совет. Вы можете предотвратить установку содержимого элемента label вспомогательной функцией дескриптора, определив содержимое самостоятельно. Это полезно, если вы хотите, чтобы элементы label содержали нечто большее, чем просто имя свойства, которое способна обеспечить встроенная вспомогательная функция дескриптора.

Работа с элементами select и option

Элементы select и option применяются для предоставления пользователю фиксированного набора вариантов вместо открытой записи данных, которая возможна посредством элемента input. Класс SelectTagHelper отвечает за трансформирование элементов select и поддерживает атрибуты, описанные в табл. 24.8.

Таблица 24.8. Атрибуты встроенной вспомогательной функции дескриптора для элементов select

Имя	Описание
asp-for	Этот атрибут используется для указания свойства модели представления, которое олицетворяет элемент select
asp-items	Этот атрибут позволяет указать источник значений для элементов option, содержащихся внутри элемента select

Атрибут asp-for устанавливает значения атрибутов for и id, чтобы отразить свойство модели, которое он получает. В листинге 24.15 элемент input для свойства Country заменен элементом select, определяющим атрибут asp-for.

Листинг 24.15. Использование элемента select в файле Create.cshtml из папки Views/Home

```
@model City
@{ Layout = "_Layout"; }
<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
  <div class="form-group">
    <label asp-for="Name"></label>
    <input class="form-control" asp-for="Name" />
  </div>
```

```
<div class="form-group">
  <label asp-for="Country"></label>
  <select class="form-control" asp-for="Country">
    <option disabled selected value="">Select a Country</option>
    <option>UK</option>
    <option>USA</option>
    <option>France</option>
    <option>China</option>
  </select>
</div>
<div class="form-group">
  <label asp-for="Population"></label>
  <input class="form-control" asp-for="Population" />
</div>
<button type="submit" class="btn btn-primary">Add</button>
<a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Здесь элемент `select` вручную заполняется элементами `option`, которые предлагают пользователю для выбора диапазон стран. Запустив приложение и запросив URL вида `/Home/Create`, легко заметить, что HTML-разметка, отправленная клиенту, содержит следующий элемент `select`:

```
<select class="form-control" id="Country" name="Country">
  <option disabled selected value="">Select a Country</option>
  <option>UK</option>
  <option>USA</option>
  <option>France</option>
  <option>China</option>
</select>
```

Если запросить URL вида `/Home/Edit` и просмотреть HTML-разметку, посланную браузеру, то обнаружится, что значение свойства `Country` объекта модели было применено для изменения выбранного элемента `option`:

```
<select class="form-control" id="Country" name="Country">
  <option disabled selected value="">Select a Country</option>
  <option selected="selected">UK</option>
  <option>USA</option>
  <option>France</option>
  <option>China</option>
</select>
```

Задача выбора элемента `option` решается классом `OptionTagHelper`, который получает инструкции от `SelectTagHelper` через коллекцию `TagHelperContext.Items`. Как объяснялось в главе 23, эта коллекция используется вспомогательными функциями дескрипторов, которым нужно работать вместе. Данные, добавляемые в коллекцию `Items` классом `SelectTagHelper`, будут задействованы в следующем разделе при создании специальной вспомогательной функции дескриптора, предназначенной для обхода ограничения встроенного класса.

Использование источника данных для заполнения элемента `select`

Явное определение элементов `option` для элемента `select` — удобный подход для вариантов выбора, которые всегда имеют одни и те же значения. Однако он ничем не

поможет в ситуации, когда необходимо предоставить варианты, берущиеся из данных модели, или когда нужен одинаковый набор вариантов в нескольких представлениях, но нежелательно вручную поддерживать дублированное содержимое.

Генерирование элементов option из перечисления

Если есть фиксированный набор вариантов для предоставления пользователю, но важно не дублировать его в представлениях повсюду в приложении, тогда можно применить перечисление. Добавим в папку Models файл класса по имени CountryNames.cs и определим в нем перечисление, как показано в листинге 24.16.

Листинг 24.16. Содержимое файла CountryNames.cs из папки Models

```
namespace Cities.Models {
  public enum CountryNames {
    UK,
    USA,
    France,
    China
  }
}
```

Использовать перечисление в атрибуте asp-items напрямую нельзя, т.к. вспомогательная функция дескриптора ожидает получить последовательность объектов SelectListItem. Тем не менее, доступен удобный вспомогательный метод, который выполняет требуемое преобразование (листинг 24.17).

Листинг 24.17. Применение перечисления для генерации элементов option в файле Create.cshtml из папки Views/Home

```
@model City

@{ Layout = "_Layout"; }
<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
  <div class="form-group">
    <label asp-for="Name"></label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label asp-for="Country"></label>
    <select class="form-control" asp-for="Country"
        asp-items="@new SelectList(Enum.GetNames(typeof(CountryNames)))">
      <option disabled selected value="">Select a Country</option>
    </select>
  </div>
  <div class="form-group">
    <label asp-for="Population"></label>
    <input class="form-control" asp-for="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

В случае использования перечисления лучший способ генерации элементов option предусматривает снабжение атрибута asp-items объектом SelectList, который заполнен именами значений перечисления. "За кулисами" класс SelectTagHelper генерирует элементы option из коллекции IEnumerable<SelectListItem>, а класс SelectList реализует этот интерфейс. Запустив приложение и запросив URL вида /Home/Create или /Home/Edit, можно увидеть, что HTML-разметка, отправленная браузеру, содержит набор элементов option, которые соответствуют значениям в перечислении:

```
<select class="form-control" id="Country" name="Country">
  <option disabled selected value="">Select a Country</option>
  <option>UK</option>
  <option>USA</option>
  <option>France</option>
  <option>China</option>
</select>
```

Обратите внимание, что вспомогательная функция дескриптора оставила один элемент-заполнитель option. Любые элементы option, которые определяются явным образом, остаются на месте, т.е. вам не придется смешивать заполнители и значения данных.

Генерирование элементов option из модели

Если необходимо генерировать элементы option для отражения данных в модели, то самый простой подход заключается в предоставлении данных, требуемых при генерации элементов, посредством объекта ViewBag (листинг 24.18).

Листинг 24.18. Предоставление данных для выбора через объект ViewBag в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using Cities.Models;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace Cities.Controllers {
  public class HomeController : Controller {
    private IRepository repository;
    public HomeController(IRepository repo) {
      repository = repo;
    }
    public ViewResult Index() => View(repository.Cities);
    public ViewResult Edit() {
      ViewBag.Countries = new SelectList(repository.Cities
        .Select(c => c.Country).Distinct());
      return View("Create", repository.Cities.First());
    }

    public ViewResult Create() {
      ViewBag.Countries = new SelectList(repository.Cities
        .Select(c => c.Country).Distinct());
      return View();
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
```

```
    public IActionResult Create(City city) {
      repository.AddCity(city);
      return RedirectToAction("Index");
    }
  }
}
```

Методы действий `Edit()` и `Create()` устанавливают свойство `ViewBag.Countries` в объект `SelectList`, который заполняется уникальными значениями для свойства `City.Country` в хранилище. В листинге 24.19 посредством атрибута `asp-items` вспомогательной функции дескриптора сообщается о необходимости получения данных для элементов `option` из свойства `ViewBag.Countries`.

Листинг 24.19. Применение объекта `ViewBag` для заполнения элементов `option` в файле `Create.cshtml` из папки `Views/Home`

```
@model City
@{ Layout = "_Layout"; }
<form method="post" asp-controller="Home" asp-action="Create"
                    asp-antiforgery="true">
  <div class="form-group">
    <label asp-for="Name"></label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label asp-for="Country"></label>
    <select class="form-control" asp-for="Country"
            asp-items="ViewBag.Countries">
      <option disabled selected value="">Select a Country</option>
    </select>
  </div>
  <div class="form-group">
    <label asp-for="Population"></label>
    <input class="form-control" asp-for="Population" />
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Запуск приложения и запрашивание URL вида `/Home/Create`or `/Home/Edit` приведет к созданию таких элементов `option`:

```
<select class="form-control" id="Country" name="Country">
  <option disabled selected value="">Select a Country</option>
  <option selected>UK</option>
  <option>USA</option>
  <option>France</option>
</select>
```

Использование специальной вспомогательной функции дескриптора для генерации элементов `option` из модели

Проблема с передачей данных для элементов `option` через объект `ViewBag` связана с тем, что нужно не забыть генерировать данные в каждом методе действия, который визуализирует представление, где применяется вспомогательная функция

дескриптора. Это приводит к дублированию кода, как видно в листинге 24.18, и затрудняет тестирование и сопровождение контроллера.

Более эффективный подход предусматривает создание специальной вспомогательной функции дескриптора, которая дополняет встроенный класс `SelectTagHelper`. Добавим в папку `Infrastructure/TagHelper` файл класса по имени `SelectOptionTagHelper.cs` с определением из листинга 24.20.

Листинг 24.20. Содержимое файла `SelectOptionTagHelper.cs` из папки `Infrastructure/TagHelper`

```
using System;
using System.Linq;
using System.Reflection;
using System.Threading.Tasks;
using Cities.Models;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace Cities.Infrastructure.TagHelpers {
  [HtmlTargetElement("select", Attributes = "model-for")]
  public class SelectOptionTagHelper : TagHelper {
    private IRepository repository;
    public SelectOptionTagHelper(IRepository repo) {
      repository = repo;
    }
    public ModelExpression ModelFor { get; set; }
    public override async Task ProcessAsync(TagHelperContext context,
        TagHelperOutput output) {
      output.Content.AppendHtml(
        (await output.GetChildContentAsync(false)).GetContent());
      string selected = ModelFor.Model as string;
      PropertyInfo property = typeof(City)
        .GetTypeInfo().GetDeclaredProperty(ModelFor.Name);
      foreach (string country in repository.Cities
          .Select(c => property.GetValue(c)).Distinct()) {
        if (selected != null && selected.Equals(country,
            StringComparison.OrdinalIgnoreCase)) {
          output.Content.AppendHtml($"<option selected>{country}</option>");
        } else {
          output.Content.AppendHtml($"<option>{country}</option>");
        }
      }
      output.Attributes.SetAttribute("Name", ModelFor.Name);
      output.Attributes.SetAttribute("Id", ModelFor.Name);
    }
  }
}
```

Специальная вспомогательная функция дескриптора оперирует на элементах `select` с атрибутом `model-for` и использует внедрение зависимостей, чтобы получить объект хранилища, который можно применять для доступа к данным модели независимо от контроллера, визуализирующего представление. В классе вспомогательной функции дескриптора определен асинхронный метод `ProcessAsync()`, т.к.

он упрощает процесс получения и сохранения любого существующего содержимого элемента select, что делается посредством метода GetChildContentAsync().

Класс SelectTagHelper указывает имена элементов option, которые должны быть выбраны, через запись в коллекции Items, используя в качестве ключа собственный тип. Вспомогательная функция дескриптора получает список выбранных элементов и применяет его в сочетании с результатами запроса LINQ при генерации элементов option для каждого уникального значения в хранилище. В листинге 24.21 элемент select был обновлен с целью замены атрибута asp-items атрибутом model-for, а также было добавлено выражение @addTagHelper, которое включает специальную вспомогательную функцию дескриптора только для этого представления.

Листинг 24.21. Включение специальной вспомогательной функции дескриптора в файле Create.cshtml из папки Views/Home

```
@model City
@addTagHelper Cities.Infrastructure.TagHelpers.SelectOptionTagHelper, Cities

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
      asp-antiforgery="true">
  <div class="form-group">
    <label asp-for="Name"></label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label asp-for="Country"></label>
    <select class="form-control" model-for="Country">
      <option disabled selected value="">Select a Country</option>
    </select>
  </div>

  <div class="form-group">
    <label asp-for="Population"></label>
    <input class="form-control" asp-for="Population" />
  </div>

  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

Новая вспомогательная функция дескриптора генерирует тот же самый вывод, но делает это без использования данных ViewBag, которые требуются встроенному вспомогательному классу. Такой подход предпочтителен тем, что он сохраняет методы действий сфокусированными на свои специфические задачи и сберегает общую форму приложения.

Работа с элементами textarea

Элемент textarea применяется для запроса у пользователя крупного объема текста и обычно используется для неструктурированных данных, таких как комментарии или замечания. Класс TextAreaTagHelper отвечает за трансформирование элементов textarea и поддерживает единственный атрибут, описанный в табл. 24.9.

Таблица 24.9. Атрибут встроенной вспомогательной функции дескриптора для элементов `textarea`

Имя	Описание
`asp-for`	Этот атрибут применяется для указания свойства модели представления, которое олицетворяет элемент `textarea`

Класс `TextAreaTagHelper` относительно прост; значение, предоставляемое в атрибуте `asp-for`, используется для установки атрибутов `id` и `name` элемента `textarea`. Чтобы взглянуть на работу данной вспомогательной функции дескриптора, добавим в класс модели `City` новое свойство (листинг 24.22).

Листинг 24.22. Добавление свойства в файле `City.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;

namespace Cities.Models {
  public class City {
    [Display(Name = "City")]
    public string Name { get; set; }
    public string Country { get; set; }
    [DisplayFormat(DataFormatString = "{0:F2}", ApplyFormatInEditMode = true)]
    public int? Population { get; set; }
    public string Notes { get; set; }
  }
}
```

Добавим к представлению `Create.cshtml` элемент `textarea` с применением атрибута `asp-for` для ассоциирования элемента со свойством `Notes` класса `City` (листинг 24.23).

Листинг 24.23. Добавление элемента `textarea` в файле `Create.cshtml` из папки `Views/Home`

```
@model City
@addTagHelper Cities.Infrastructure.TagHelpers.SelectOptionTagHelper, Cities

@{ Layout = "_Layout"; }

<form method="post" asp-controller="Home" asp-action="Create"
      asp-antiforgery="true">
  <div class="form-group">
    <label asp-for="Name"></label>
    <input class="form-control" asp-for="Name" />
  </div>
  <div class="form-group">
    <label asp-for="Country"></label>
    <select class="form-control" asp-for="Country"
            asp-items="ViewBag.Countries">
      <option disabled selected value="">Select a Country</option>
    </select>
  </div>
  <div class="form-group">
    <label asp-for="Population"></label>
```

```
    <input class="form-control" asp-for="Population" />
  </div>
  <div class="form-group">
    <label asp-for="Notes"></label>
    <textarea class="form-control" asp-for="Notes"></textarea>
  </div>
  <button type="submit" class="btn btn-primary">Add</button>
  <a class="btn btn-primary" href="/Home/Index">Cancel</a>
</form>
```

После запуска приложения и запрашивания URL вида `/Home/Create` или `/Home/Edit` можно заметить, что HTML-разметка, отправленная браузеру, включает следующий элемент `textarea`:

```
<div class="form-group">
  <label for="Notes">Notes</label>
  <textarea id="Notes" name="Notes"></textarea>
</div>
```

Несмотря на простоту, класс `TextAreaTagHelper` обеспечивает согласованность с остальными вспомогательными функциями дескрипторов для форм, которые рассматривались в настоящей главе.

Вспомогательные функции дескрипторов для проверки достоверности форм

Есть еще две вспомогательные функции дескрипторов, которые имеют отношение к HTML-формам; ради полноты они описаны в табл. 24.10, но подробно объясняются в главе 27. Эти вспомогательные классы используются для предоставления пользователю отклика, когда введенные им данные не удовлетворяют ожиданиям приложения.

Таблица 24.10. Вспомогательные функции дескрипторов для проверки достоверности

Имя	Описание
`ValidationMessage`	Эта вспомогательная функция дескриптора применяется для предоставления пользователю отклика проверки достоверности, касающегося одиночного элемента формы
`ValidationSummary`	Эта вспомогательная функция дескриптора используется для предоставления пользователю отклика проверки достоверности, касающегося всех элементов в форме

Резюме

В главе были описаны встроенные вспомогательные функции дескрипторов, которые применяются для трансформации элементов HTML-форм. Такие вспомогательные функции дескрипторов гарантируют, что формы генерируются напрямую из класса модели, сокращая возможности возникновения ошибок и предлагая согласованный подход к написанию представлений Razor. В следующей главе рассматриваются оставшиеся встроенные вспомогательные функции дескрипторов, которые имеют дело с набором разных HTML-элементов.

ГЛАВА 25

Использование других встроенных вспомогательных функций дескрипторов

Вспомогательные функции дескрипторов, которые были описаны в главе 24, сосредоточены на выпуске HTML-форм, но они не являются единственными встроенными вспомогательными функциями дескрипторов, предлагаемыми инфраструктурой ASP.NET Core MVC. В настоящей главе рассматриваются вспомогательные функции дескрипторов, которые управляют файлами JavaScript и таблицами стилей CSS, создают URL для якорных элементов, предоставляют возможность аннулирования кеша для элементов изображений и поддерживают кеширование данных. Вдобавок будет описана вспомогательная функция дескриптора, обеспечивающая доступ для URL, относительных к приложению, которые помогают гарантировать, что браузеры могут обращаться к статическому содержимому, когда приложение развернуто в среде, разделяемой с другими приложениями.

В табл. 25.1 приведена сводка по главе.

Таблица 25.1. Сводка по главе

Задача	Решение	Листинг
Включение содержимого на основе среды размещения	Используйте элемент `environment`	25.2, 25.8
Выбор файлов JavaScript	Примените атрибуты `asp-src-include` и `asp-src-exclude` к элементу `script`	25.3–25.7
Использование сети доставки содержимого для файлов JavaScript	Примените атрибуты `asp-fallback` к элементу `script`	25.9, 25.10
Выбор файлов CSS	Примените атрибуты `asp-href-include` и `asp-href-exclude` к элементу `link`	25.11
Использование сети доставки содержимого для файлов CSS	Примените атрибуты `asp-fallback` к элементу `link`	25.12

Окончание табл. 25.1

Задача	Решение	Листинг
Генерация URL для якорного элемента	Используйте вспомогательный класс `AnchorTagHelper`	25.13
Обеспечение обнаружения изменений в изображениях	Примените атрибут `asp-append-version` к элементу `img`	25.14
Кеширование данных	Используйте элемент `cache`	25.15–25.23
Создание URL, относительных к приложению	Снабдите URL префиксом ~	25.24–25.26

Подготовка проекта для примера

Мы продолжим работать с проектом `Cities` из главы 24. Подготовка для текущей главы предусматривает создание папки `wwwroot/images` и добавление в нее файла изображения по имени `city.png`. В нем находится общедоступная фотография панорамы Нью-Йорка (рис. 25.1).



Рис. 25.1. Добавление изображения в проект

Файл изображения `city.png` входит в состав кода примеров для книги, доступного в хранилище GitHub для книги (`https://github.com/apress/pro-asp.net-core-mvc-2`). При желании можете воспользоваться собственным изображением.

Еще одно изменение, которое потребуется сделать, связано с добавлением в проект пакета jQuery, используя Bower (листинг 25.1).

Листинг 25.1. Добавление пакета jQuery в файле `bower.json` из папки `Cities`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

В результате запуска приложения отобразится список объектов в хранилище и появится возможность создания новых объектов (рис. 25.2).

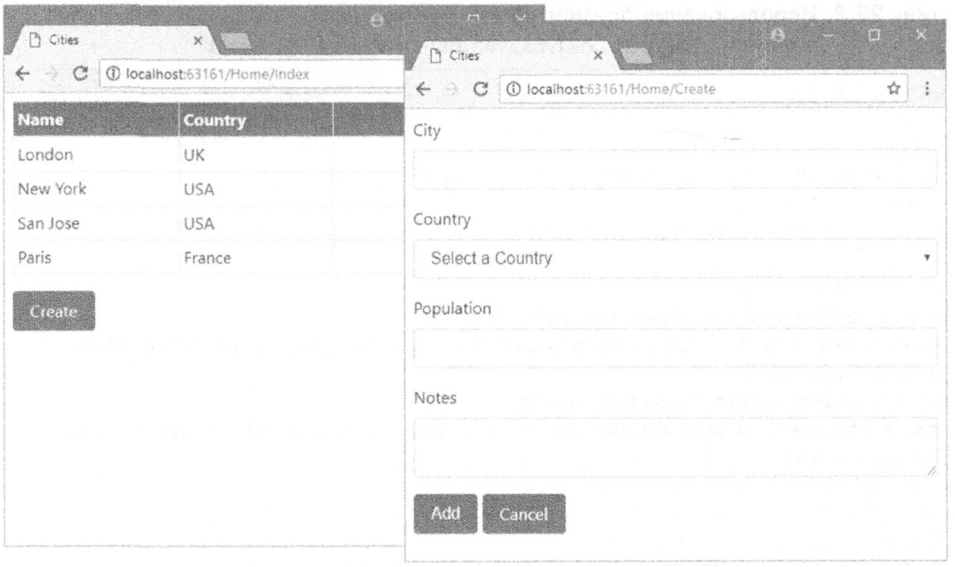


Рис. 25.2. Выполнение примера приложения

Использование вспомогательной функции дескриптора для среды размещения

Класс `EnvironmentTagHelper` применяется к специальному элементу `environment` и определяет, включается ли область содержимого в HTML-разметку, отправляемую браузеру, на основе среды размещения, как было показано в главе 14. Это может не казаться самым захватывающим местом, чтобы начинать с него, но данный вспомогательный класс необходим для эффективного использования ряда связанных средств, которые будут описаны позже. Элемент `environment` полагается на атрибут `names`, который описан в табл. 25.2 для ссылки в будущем.

Таблица 25.2. Атрибут встроенной вспомогательной функции дескриптора для элементов `environment`

Имя	Описание
`names`	Этот атрибут применяется для указания списка разделенных запятыми имен сред размещения, для которых содержимое, находящееся внутри элемента `environment`, будет включаться в HTML-разметку, отправляемую клиенту

Добавим в разделяемую компоновку элементы `environment`, включающие разное содержимое в представление для среды разработки и производственной среды (листинг 25.2).

На рис. 25.3 показаны результаты запуска приложения в среде разработки и производственной среде. Элемент `environment` проверяет имя текущей среды размещения и либо включает имеющееся в нем содержимое, либо пропускает его (сам элемент `environment` всегда опускается из HTML-разметки, отправляемой клиенту).

Листинг 25.2. Использование элементов `environment` в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <environment names="development">
    <div class="m-1 p-1 bg-info"><h2>This is Development</h2></div>
  </environment>
  <environment names="production">
    <div class="m-1 p-1 bg-danger"><h2>This is Production</h2></div>
  </environment>
  <div>@RenderBody()</div>
</body>
</html>
```



Рис. 25.3. Управление содержимым с применением среды размещения

Использование вспомогательных функций дескрипторов для JavaScript и CSS

Следующая категория встроенных вспомогательных функций дескрипторов применяется для управления файлами JavaScript и таблицами стилей CSS посредством элементов `script` и `link`, которые обычно включаются в разделяемую компоновку. Как будет показано в позже, такие вспомогательные функции дескрипторов характеризуются высокой мощностью и гибкостью, но требуют пристального внимания во время использования, чтобы избежать создания непредсказуемых результатов.

Управление файлами JavaScript

Класс `ScriptTagHelper` — это встроенная вспомогательная функция дескриптора для элементов `script`. Класс `ScriptTagHelper` предназначен для управления включением файлов JavaScript в представления с применением атрибутов, которые кратко описаны в табл. 25.3 и подробно рассматриваются далее в главе.

Таблица 25.3. Атрибуты встроенной вспомогательной функции дескриптора для элементов `script`

Имя	Описание
`asp-src-include`	Этот атрибут используется для указания файлов JavaScript, которые будут включены в представление
`asp-src-exclude`	Этот атрибут применяется для указания файлов JavaScript, которые будут исключены из представления
`asp-append-version`	Этот атрибут используется для аннулирования кеша, как описано во врезке "Понятие аннулирования кеша" далее в главе
`asp-fallback-src`	Этот атрибут применяется для указания запасного файла JavaScript, подлежащего использованию в случае возникновения проблемы с сетью доставки содержимого
`asp-fallback-src-include`	Этот атрибут применяется для выбора файлов JavaScript, которые будут использоваться при наличии проблемы с сетью доставки содержимого
`asp-fallback-src-exclude`	Этот атрибут применяется для исключения файлов JavaScript, чтобы предотвратить их использование в случае возникновения проблемы с сетью доставки содержимого
`asp-fallback-test`	Этот атрибут применяется для указания фрагмента JavaScript, который будет использоваться для определения, корректно ли бы загружен код JavaScript из сети доставки содержимого

Выбор файлов JavaScript

Атрибут `asp-src-include` применяется для включения файлов JavaScript в представление с использованием шаблонов универсализации имен, которые поддерживают набор групповых символов, применяемых для сопоставления с именами файлов. В табл. 25.4 описаны наиболее распространенные шаблоны универсализации имен.

Таблица 25.4. Распространенные шаблоны универсализации имен

Шаблон	Пример	Описание
`?`	`js/src?.js`	Этот шаблон дает совпадение с любым одиночным символом кроме `/`. Пример шаблона соответствует любому файлу из каталога `js` с именем, состоящим из `src`, за которым следует любой символ, а после него расширение `.js`, таким как `js/src1.js` и `js/srcX.js`, но не `js/src123.js` или `js/mydir/src1.js`
`*`	`js/*.js`	Этот шаблон дает совпадение с любым количеством символов кроме `/`. Пример шаблона соответствует любому файлу из каталога `js` с именем, которое заканчивается расширением `.js`, таким как `js/src1.js` и `js/src123.js`, но не `js/mydir/src1.js`
`**`	`js/**/*.js`	Этот шаблон дает совпадение с любым количеством символов, включая `/`. Пример шаблона соответствует любому файлу с расширением `.js`, который содержится внутри каталога `js` или любого его подкаталога, например, `/js/src1.js` и `/js/mydir/src1.js`

Использование шаблона универсализации имен в атрибуте `asp-src-include` означает, что представление будет всегда включать файлы JavaScript в приложении, даже если имя или путь к файлам изменяются либо файлы добавляются или удаляются. В листинге 25.3 демонстрируется выбор файлов JavaScript для пакета jQuery, который Bower устанавливает в папку `wwwroot/lib/jquery/dist`.

Листинг 25.3. Выбор файлов JavaScript в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script asp-src-include="/lib/jquery/dist/**/*.js"></script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

В примере применяется распространенный шаблон. Шаблоны оцениваются внутри папки `wwwroot`, а библиотека jQuery доставляется как одиночный файл JavaScript по имени `jquery.js`.

Шаблон универсализации имен пытается выбрать файл jQuery, одновременно приспосабливаясь к любым будущим изменениям в способе распространения jQuery, таким как изменение имени файла JavaScript. Запустив приложение и просмотрев HTML-разметку, отправленную клиенту, можно увидеть, что в ней присутствует проблема:

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script src="/lib/jquery/dist/core.js"></script>
  <script src="/lib/jquery/dist/jquery.js"></script>
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script src="/lib/jquery/dist/jquery.slim.js"></script>
  <script src="/lib/jquery/dist/jquery.slim.min.js"></script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
```

Класс `ScriptTagHelper` генерирует элемент `script` для каждого файла, имя которого соответствует шаблону, указанному в атрибуте `asp-src-include`. Вместо выбора только файла `jquery.js` имеются также элементы `script` для файла `jquery.min.js`, который является минифицированной версией файла `jquery.js`, а также для обычного и минифицированного вариантов основной и сокращенной (slim) версий библиотеки jQuery.

Вы можете даже не подозревать, что дистрибутив jQuery содержит настолько много файлов, т.к. Visual Studio по умолчанию их скрывает. Чтобы отобразить полное содержимое папки `wwwroot/lib/jquery/dist`, понадобится раскрыть элемент `jquery.js` в окне Solution Explorer и затем сделать то же самое с находящимися в нем элементами (рис. 25.4).

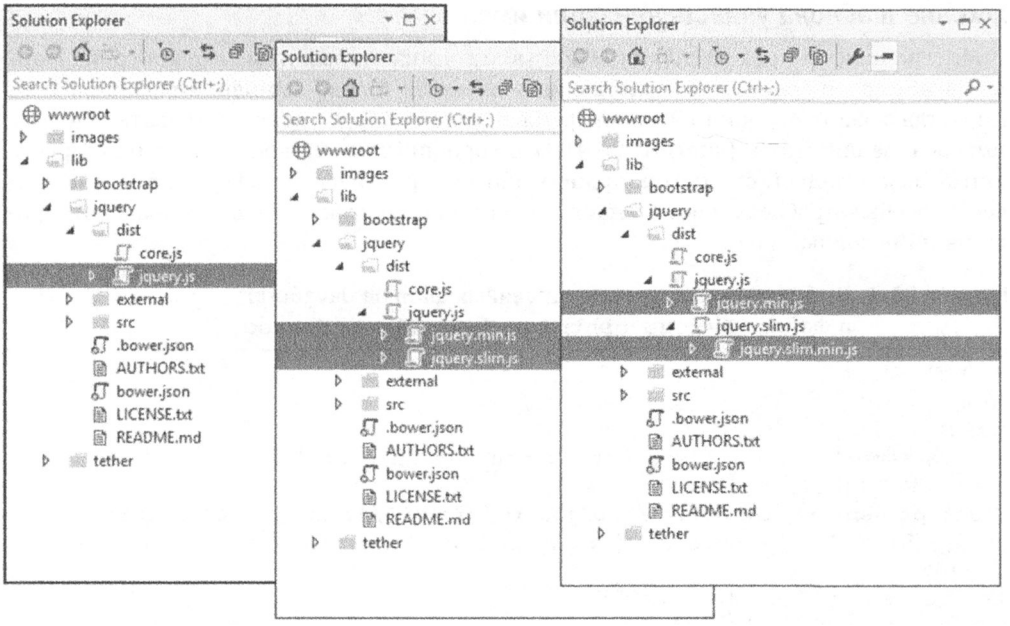


Рис. 25.4. Отображение полного содержимого папки в окне Solution Explorer

Шаблон, примененный в листинге 25.3, вызывает многократную отправку браузеру кода jQuery, что приводит к излишней трате полосы пропускания и замедлению работы приложения. В случае некоторых библиотек результатом могут стать ошибки или непредсказуемое поведение. Решить проблему можно тремя способами, которые обсуждаются в последующих разделах.

Использование отображений на исходный код

Файлы JavaScript минифицируются с целью уменьшения размеров, что означает возможность их доставки клиенту быстрее и с меньшим расходом полосы пропускания. Процесс минификации удаляет из файла все пробельные символы, а также переименовывает функции и переменные так, что значащие имена вроде `myHelpfullyNamedFunction` будут представлены меньшим количеством символов, скажем, `x1`. В случае применения отладчика JavaScript браузера для поиска источника проблем в минифицированном коде имена, подобные `x1`, сделают почти невозможным прохождение по коду.

Файл `jquery.min.map` является *отображением на исходный код*, которое используется рядом браузеров для того, чтобы помочь в отладке минифицированного кода, обеспечивая соответствие между минифицированным кодом и читабельным полным исходным кодом.

На момент написания книги отображения на исходный код не были универсально поддерживаемым средством, но их можно было применять в большинстве последних версий браузеров Chrome и Edge. Например, браузер Chrome будет автоматически запрашивать файл отображений на исходный код, если открыто окно инструментов разработчика, а это значит, что браузеру можно отправлять минифицированные версии файлов JavaScript и по-прежнему иметь возможность их легкой отладки.

Сужение шаблона универсализации имен

Многие пакеты предоставляют обычные и минифицированные версии своих файлов JavaScript, и если вы собираетесь использовать только минифицированные версии, тогда можете ограничить набор файлов, которые будут соответствовать шаблону универсализации имен (листинг 25.4). Это хороший подход, когда вы не планируете заниматься отладкой библиотеки jQuery, которая реализована эффективно и не вызывает особых проблем, или же знаете, что целевые браузеры поддерживают отображения на исходный код.

Листинг 25.4. Выбор только минифицированных файлов JavaScript в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script asp-src-include="/lib/jquery/dist/**/*.min.js"></script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

После запуска приложения и просмотра HTML-разметки, отправленной браузеру, обнаружится, что были включены только минифицированные файлы:

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script src="/lib/jquery/dist/jquery.slim.min.js"></script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
```

Сужение шаблона для файлов JavaScript помогло, но браузеру по-прежнему будет отправляться нормальная и сокращенная версии библиотеки jQuery (сокращенная версия лишена менее распространенных функций; за деталями обращайтесь на вебсайт `jquery.com`). Чтобы еще больше сузить выбор, в шаблон универсализации имен можно включить сокращенную версию (листинг 25.5).

Листинг 25.5. Сужение выбора в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script asp-src-include="/lib/jquery/dist/**/*.slim.min.js"></script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
```

```
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

В результате браузеру будет посылаться только одна версия файла jQuery с сохранением гибкости в плане местоположения этого файла:

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script src="/lib/jquery/dist/jquery.slim.min.js"></script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
```

Исключение файлов

Сужение шаблона для файлов JavaScript помогает, когда нужно выбрать файл, имя которого содержит специфический элемент вроде `slim`. Оно бесполезно, если имя файла не имеет такого элемента, как в ситуации, когда необходима полная версии минифицированного файла. К счастью, с помощью атрибута `asp-src-exclude` можно удалять файлы из списка, построенного атрибутом `asp-src-include`, как показано в листинге 25.6.

Листинг 25.6. Исключение файлов JavaScript в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script asp-src-include="/lib/jquery/dist/**/*.min.js"
          asp-src-exclude="**.slim.**">
  </script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

Если запустить приложение и заглянуть в HTML-разметку, отправленную браузеру, то можно заметить, что была включена только полная минифицированная версия библиотеки jQuery:

```
<head>
<meta name="viewport" content="width=device-width" />
<title>Cities</title>
<script src="/lib/jquery/dist/jquery.min.js"></script>
<link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
```

Тот же самый прием можно применять, когда интересуют неминифицированные версии файлов, которые могут быть удобны на стадии разработки (листинг 25.7).

Листинг 25.7. Выбор неминифицированной версии файла в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script asp-src-include="/lib/jquery/dist/**/j*.js"
          asp-src-exclude="**.slim.**,**.min.**">
  </script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

Обратите внимание на возможность указания множества элементов, разделяя их запятыми. Запустив приложение и просмотрев HTML-разметку, отправленную браузеру, можно увидеть, что была включена только неминифицированная версия файла JavaScript:

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script src="/lib/jquery/dist/jquery.js"></script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
```

Использование среды размещения для выбора файлов

Общий подход предусматривает применение обычных файлов JavaScript на этапе разработки, что облегчает отладку, и использование минифицированных файлов в производственной среде для сокращения расхода полосы пропускания. В таком случае можно задействовать элемент `environment`, чтобы избирательно включать элементы `script`, основываясь на среде размещения (листинг 25.8).

Листинг 25.8. Выбор файлов на основе среды размещения в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <environment names="development">
    <script asp-src-include="/lib/jquery/dist/**/j*.js"
            asp-src-exclude="**.slim.**,**.min.**">
    </script>
  </environment>
  <environment names="staging, production">
    <script asp-src-include="/lib/jquery/dist/**/*.min.js"
            asp-src-exclude="**.slim.**">
    </script>
  </environment>
```

```
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

Преимущество такого подхода в том, что он адаптирует приложение к среде размещения, но при этом придется писать и сопровождать многочисленные наборы элементов `script`.

Понятие аннулирования кеша

Статическое содержимое, такое как изображения, таблицы стилей CSS и файлы JavaScript, часто кешируется, чтобы предотвратить попадание на серверы приложений запросов к редко изменяющемуся содержимому. Кеширование делается разными способами: браузер может сообщить серверу о необходимости кеширования содержимого, приложение может использовать серверы кешей как дополнение к серверам приложений или содержимое может распространяться с использованием сети доставки содержимого. Не все кеширование будет находиться под вашим контролем. Например, в крупных корпорациях кеши часто устанавливаются для снижения требований к полосе пропускания, поскольку значительный процент запросов направляется к одним и тем же сайтам или приложениям.

С кешированием связана одна проблема: клиенты не получают новые версии статических файлов немедленно после их развертывания, потому что запросы клиентов продолжают обслуживаться с применением ранее кешированного содержимого. В конце концов, время существования кешированного содержимого истечет и начнет использоваться новое содержимое, но остается промежуток, когда динамическое содержимое, генерируемое контроллерами приложения, не согласовано со статическим содержимым, доставляемым кешами. В зависимости от содержимого, которое было обновлено, могут возникнуть проблемы с компоновкой или непредсказуемое поведение приложения.

Упомянутая проблема решается с помощью *аннулирования кеша*. Идея состоит в том, чтобы позволить кешам обрабатывать статическое содержимое, но незамедлительно отражать любые изменения, которые произошли на сервере. Вспомогательные функции дескрипторов поддерживают аннулирование кеша за счет добавления к URL для статического содержимого строки запроса, которая включает контрольную сумму, действующую в качестве номера версии. Скажем, для файлов JavaScript класс `ScriptTagHelper` поддерживает аннулирование кеша через атрибут `asp-append-version`:

```
...
<script asp-src-include="/lib/jquery/dist/**/j*.js"
        asp-src-exclude="**.slim.**,**.min.**"
        asp-append-version="true">
</script>
...
```

Включение средства аннулирования кеша приводит к появлению в HTML-разметке, отправляемой браузеру, элемента следующего вида:

```
...
<script src="/lib/jquery/dist/jquery.min.js?v=3zRSQ1HF-ocUiVcdv9yKTXqM">
</script>
...
```

Тот же самый номер версии будет применяться вспомогательной функцией дескриптора до тех пор, пока содержимое файла не изменится, например, из-за обновления библиотеки JavaScript, в результате чего будет вычислена другая контрольная сумма. Добавление номера версии означает, что всякий раз, когда изменяется файл, клиент будет запрашивать отличающийся URL, трактуемый кешами как запрос для нового содержимого, который не может быть удовлетворен с использованием ранее кешированного содержимого, поэтому он передается серверу приложений. Затем содержимое кешируется обычным образом до следующего обновления, которое приведет к генерации еще одного URL с другим номером версии.

Работа с сетями доставки содержимого

Сеть доставки содержимого (content delivery network — CDN) применяется для передачи запросов к содержимому приложения серверам, которые находятся поблизости от пользователя. Вместо запрашивания файла JavaScript у ваших серверов браузер запрашивает его у имени хоста, которое преобразуется в географически местный сервер, что уменьшает время загрузки файлов и сокращает ширину полосы пропускания, необходимую приложению. При наличии крупной географически распределенной базы пользователей может оказаться целесообразной коммерческая подписка на CDN, но даже небольшое и простое приложение способно извлечь выгоду от использования бесплатных CDN, приводимых в действие ведущими технологическими компаниями для доставки общих пакетов JavaScript, таких как jQuery.

В настоящей главе будет применяться сеть Microsoft CDN, которая предлагает бесплатный доступ к популярным пакетам, со списком которых можно ознакомиться по адресу `www.asp.net/ajax/cdn`. Здесь используется пакет jQuery 3.2.1, и вот шесть URL в сети Microsoft CDN, указывающие на этот выпуск:

- `http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.js`
- `http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.min.js`
- `http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.min.map`
- `http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.slim.js`
- `http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.slim.min.js`
- `http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.slim.min.map`

Приведенные URL предоставляют обычный файл JavaScript, минифицированный файл JavaScript и отображения на исходный код, относящееся к минифицированным файлам для полной и сокращенной версий jQuery. В листинге 25.9 показана модифицированная компоновка для примера приложения, в которой локальные файлы заменены файлами, получаемыми из сети CDN.

Листинг 25.9. Применение сети CDN в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.min.js">
  </script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
```

```
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

Указание CDN означает, что никакие запросы для jQuery не будут попадать на серверы приложения. Проблема с сетями CDN связана с тем, что они не находятся под контролем вашей организации, т.е. такие сети могут отказывать, оставляя приложение функционирующим, но неспособным работать ожидаемым образом из-за недоступности содержимого CDN. Чтобы решить проблему, класс `ScriptTagHelper` предлагает возможность обратиться за помощью к локальным файлам, когда клиент не может загрузить содержимое из CDN (листинг 25.10).

Листинг 25.10. Использование обхода загрузки из CDN в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.min.js"
    asp-fallback-src-include="/lib/jquery/dist/**/*.min.js"
    asp-fallback-src-exclude="**.slim.**" asp-fallback-test="window.jQuery">
  </script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

Атрибуты `asp-fallback-src-include` и `asp-fallback-src-exclude` применяются для выбора и исключения локальных файлов, которые будут использоваться, если сеть CDN не способна доставить файл, указанный посредством обычного атрибута `src`. Чтобы выяснить, работает ли CDN, с помощью атрибута `asp-fallback-test` определяется фрагмент кода JavaScript, который будет оцениваться в браузере. Если фрагмент оценивается как `false`, тогда будут запрашиваться запасные файлы.

Запустим приложение и просмотрим HTML-разметку, которая была отправлена клиенту. Будет видно, что класс `ScriptTagHelper` взял фрагмент из атрибута `asp-fallback-test` и применил его для создания еще одного элемента `script`:

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.min.js">
  </script>
  <script>
    (window.jQuery||document.write("\u003Cscript
      src=\u0022\/lib\/jquery\/dist\/jquery.min.js
      \u0022\u003E\u003C\/script\u003E"));
  </script>
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
```

Фрагмент кода JavaScript, указываемый в атрибуте `asp-fallback-test`, должен возвращать `true`, если файл из CDN был загружен, и `false` в противном случае. Простейший подход обычно предусматривает проверку точки входа в функциональность, предлагаемую той или иной библиотекой. Библиотека jQuery создает функцию по имени `jQuery` на глобальном объекте `window`, которая и проверяется в листинге 25.10. Понадобится найти эквивалентные тесты для всех файлов, загружаемых из CDN.

Важно тестировать настройки обхода, т.к. их отказ не может быть обнаружен до тех пор, пока сеть CDN не прекратит работу, а пользователи не утратят доступ к приложению. Проверить обход проще всего путем изменения имени файла, указанного в атрибуте `src`, на такое, о котором доподлинно известно, что оно не существует (я дополняю имя файла словом `FAIL`), и затем с помощью инструментов <F12> просмотреть сетевые запросы, выполняемые браузером. Должна быть видна ошибка для файла из сети CDN, за которой следует запрос запасного файла.

Внимание! Средство обхода CDN полагается на то, что браузеры загружают и выполняют содержимое элементов `script` синхронным образом в порядке, в котором они определены. Существует несколько приемов ускорения загрузки и выполнения сценариев JavaScript за счет того, что они делают процесс асинхронным. Однако такой подход может привести к тому, что проверка обхода выполнится перед тем, как браузер извлечет файл из CDN и выполнит его содержимое, в результате порождая запросы для запасных файлов даже при нормальной работе CDN и ставя под сомнение само использование CDN. Не смешивайте асинхронную загрузку сценариев со средством обхода CDN.

Управление таблицами стилей CSS

Класс `LinkTagHelper` — это встроенная вспомогательная функция дескриптора для элементов `link`, которая применяется при управлении включением таблиц стилей CSS в представление. Класс `LinkTagHelper` поддерживает атрибуты, описанные в табл. 25.5, использование которых демонстрируется в последующих разделах.

Таблица 25.5. Атрибуты встроенной вспомогательной функции дескриптора для элементов `link`

Имя	Описание
`asp-href-include`	Этот атрибут применяется для выбора файлов, предназначенных атрибуту `href` выходного элемента
`asp-href-exclude`	Этот атрибут используется для исключения файлов из атрибута `href` выходного элемента
`asp-append-version`	Этот атрибут применяется для включения средства аннулирования кеша, как описано во врезке "Понятие аннулирования кеша" ранее в главе
`asp-fallback-href`	Этот атрибут используется для указания запасного файла на случай возникновения проблемы с сетью CDN
`asp-fallback-href-include`	Этот атрибут применяется для выбора файлов, которые будут использоваться при наличии проблемы с CDN
`asp-fallback-href-exclude`	Этот атрибут применяется для исключения файлов из набора, который будет использоваться при наличии проблемы с CDN

Окончание табл. 25.5

Имя	Описание
`asp-fallback-href-test-class`	Этот атрибут применяется для указания класса CSS, который будет использоваться при тестировании работоспособности сети CDN
`asp-fallback-href-test-property`	Этот атрибут применяется для указания свойства CSS, которое будет использоваться при тестировании работоспособности сети CDN
`asp-fallback-href-test-value`	Этот атрибут применяется для указания значения CSS, которое будет использоваться при тестировании работоспособности сети CDN

Выбор таблиц стилей

Класс `LinkTagHelper` разделяет с классом `ScriptTagHelper` многие средства, в том числе поддержку шаблонов универсализации имен для выбора либо исключения файлов CSS, так что их не нужно указывать по отдельности. Наличие возможности точного выбора файлов CSS имеет такую же важность, как и для файлов JavaScript, поскольку таблицы стилей тоже поступают в виде обычных и минифицированных версий, дополнительно поддерживая отображения на исходный код. Популярный пакет Bootstrap, который повсеместно в книге применялся для стилизации HTML-элементов, включает свои таблицы стилей CSS в папке `wwwroot/lib/bootstrap/dist/css`, и если раскрыть все элементы в окне Solution Explorer, то можно заметить, что доступно несколько файлов (рис. 25.5).

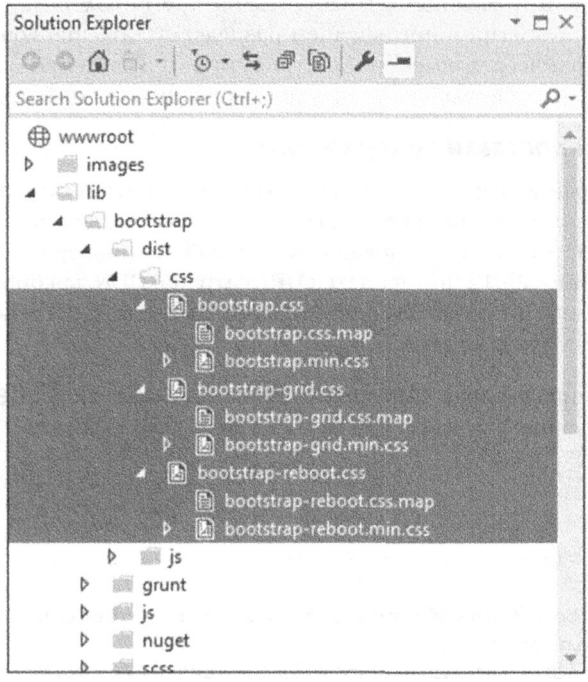


Рис. 25.5. Файлы дистрибутива Bootstrap

Файл bootstrap.css является обычной таблицей стилей, файл bootstrap.min.css — ее минифицированной версией, а файл bootstrap.css.map — отображением на исходный код. Остальные файлы предоставляют специфические средства, рассмотрение которых выходит за рамки настоящей главы. В листинге 25.11 используется атрибут asp-href-include элемента link для выбора минифицированной таблицы стилей. (Кроме того, удален элемент script для загрузки библиотеки jQuery, которая больше не требуется.)

Листинг 25.11. Выбор таблицы стилей в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link rel="stylesheet"
        asp-href-include="/lib/bootstrap/dist/**/*.min.css"
        asp-href-exclude="**/*-reboot*,**/*-grid*"/>
  </head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

Как и при выборе файлов JavaScript, здесь должно уделяться аналогичное внимание деталям, потому что довольно легко сгенерировать элементы link для нескольких версий того же самого файла или файлов, которые не нужны. Чтобы управлять выбираемыми файлами, можно следовать тем же трем подходам, которые были описаны для файлов JavaScript в предыдущем разделе: сужать шаблон универсализации имен, исключать файлы с применением атрибута asp-href-exclude и использовать элемент environment для выбора между дублирующимися наборами файлов.

Работа с сетями доставки содержимого

Вспомогательный класс LinkTag предлагает набор атрибутов для возвращения к локальному содержимому, когда сеть CDN не доступна, хотя процесс проверки, загрузилась ли таблица стилей, чуть сложнее, чем подобная проверка в отношении файла JavaScript. В листинге 25.12 применяется URL сети MaxCDN для библиотеки Bootstrap, просто чтобы продемонстрировать альтернативу платформе Microsoft (MaxCDN — это сеть CDN, рекомендуемая проектом Bootstrap).

Листинг 25.12. Использование CDN для загрузки таблицы стилей CSS в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
        asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
        asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
```

```
        asp-fallback-test-class="btn"
        asp-fallback-test-property="display"
        asp-fallback-test-value="inline-block"
        rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <div>@RenderBody()</div>
</body>
</html>
```

В атрибуте `href` указывается URL сети CDN, а с помощью атрибутов `asp-fallback-href-include` и `asp-fallback-href-exclude` выбирается файл, который будет применяться, если сеть CDN окажется недоступной. Тем не менее, проверка, работает ли сеть CDN, требует использования трех разных атрибутов и понимания классов CSS, которые определяются необходимой таблицей стилей CSS.

Средство перехода на запасные таблицы стилей CSS инициируется путем добавления к отправляемому документу элемента `meta` для класса, определенного в атрибуте `asp-fallback-test-class`. В листинге 25.12 был указан класс `btn`, т.е. к HTML-разметке, отправляемой браузеру, будет добавлен элемент `meta` следующего вида:

```
<meta name="x-stylesheet-fallback-test" class="btn" />
```

Указываемый класс CSS должен быть определен в таблице стилей, которая подлежит загрузке из CDN. Заданный здесь класс `btn` обеспечивает базовое форматирование для кнопочных элементов Bootstrap.

Атрибут `asp-fallback-test-property` применяется для указания свойства CSS, которое устанавливается классом CSS, а атрибут `asp-fallback-test-value` используется для указания устанавливаемого им значения. Вспомогательная функция дескриптора добавляет к представлению код JavaScript, который проверяет значение свойства CSS из элемента `meta` с целью выяснения, загрузилась ли таблица стилей, и на случай, если не загрузилась, предусматривает элементы `link` для запасных файлов. Класс `btn` из Bootstrap устанавливает свойство `display` в `inline-block`, что является тестом, который проверяет, удалось ли браузеру загрузить таблицу стилей Bootstrap из сети CDN.

Совет. Простейший способ понять, каким образом проводить проверку для сторонних пакетов вроде Bootstrap, предполагает применение инструментов <F12> браузера. Чтобы определить тест в листинге 25.12, элементу назначается класс `btn`, после чего в браузере инспектируются индивидуальные свойства CSS, которые этот класс изменил. Такой прием проще попытки просмотра длинных и сложных таблиц стилей.

Работа с якорными элементами

Элемент `a` является базовым инструментом для навигации в рамках приложения и отправки запросов `GET` приложению с целью получения разнообразного содержимого. Класс `AnchorTagHelper` используется для трансформации атрибута `href` элементов `a`, чтобы целевые URL генерировались при участии системы маршрутизации, с применением атрибутов из табл. 25.6.

Таблица 25.6. Атрибуты встроенной вспомогательной функции дескриптора для якорных элементов

Имя	Описание
asp-action	Этот атрибут указывает метод действия, на который будет нацелен URL
asp-controller	Этот атрибут указывает контроллер, на который будет нацелен URL
asp-area	Этот атрибут указывает область, на которую будет нацелен URL
asp-fragment	Этот атрибут используется для указания фрагмента URL (который находится после символа #)
asp-host	Этот атрибут указывает имя хоста, на который будет нацелен URL
asp-protocol	Этот атрибут указывает протокол, который будет применять URL
asp-route	Этот атрибут указывает имя маршрута, который будет использоваться для генерации URL
asp-route-*	Атрибуты с именами, начинающимися на asp-route-, применяются для указания дополнительных значений, относящихся к URL, так что атрибут asp-route-id используется для предоставления системе маршрутизации значения сегмента id

Класс AnchorTagHelper прост и предсказуем; он облегчает генерацию URL в элементах a, которые задействуют конфигурацию маршрутизации приложения. В листинге 25.13 элемент a внутри представления Index.cshtml обновлен, чтобы его атрибут href выпускался вспомогательной функцией дескриптора.

Листинг 25.13. Трансформация якорного элемента в файле Index.cshtml из папки Views/Home

```
@model IEnumerable<City>

@{ Layout = "_Layout"; }

<table class="table table-sm table-bordered">
  <thead class="bg-primary text-white">
    <tr>
      <th>Name</th>
      <th>Country</th>
      <th class="text-right">Population</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var city in Model) {
      <tr>
        <td>@city.Name</td>
        <td>@city.Country</td>
        <td class="text-right">@city.Population?.ToString("#,###")</td>
      </tr>
    }
  </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create</a>
```

Запустив приложение и запросив URL вида /Home/Index, легко заметить, что вспомогательная функция дескриптора трансформировала элемент a следующим образом:

```
<a class="btn btn-primary" href="/Home/Create">Create</a>
```

Работа с элементами img

Класс ImageTagHelper предлагает средство аннулирования кеша для изображений через атрибут src элементов img, давая приложению возможность применять в своих интересах кеширование и гарантируя немедленное отражение любых изменений в изображениях. Класс ImageTagHelper оперирует с элементами img, которые определяют атрибут asp-append-version, описанный в табл. 25.7.

Таблица 25.7. Атрибут встроенной вспомогательной функции дескриптора для элементов img

Имя	Описание
asp-append-version	Этот атрибут используется для включения аннулирования кеша, как объяснялось во врезке "Понятие аннулирования кеша" ранее в главе

В листинге 25.14 к разделяемой компоновке добавлен элемент img с изображением панорамы Нью-Йорка, которое было включено в проект в начале главы. (Кроме того, ради простоты обеспечено применение локальных файлов с таблицами стилей.)

Листинг 25.14. Добавление изображения в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
        asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
        asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
        asp-fallback-test-class="btn"
        asp-fallback-test-property="display"
        asp-fallback-test-value="inline-block"
        rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <img src="/images/city.png" asp-append-version="true" />
  <div>@RenderBody()</div>
</body>
</html>
```

После запуска приложения изображение будет отображаться в верхней части каждой страницы. Если просмотреть HTML-разметку, которая была отправлена браузеру, то обнаружится, что URL, используемый для запроса файла изображения, содержит контрольную сумму версии:

```
<img src="/images/city.png?v=KaMNDSZFbzNpE8Pkb30EXcAJufRcRDpKh0K_IIPNc7E"/>
```

Как и со средствами аннулирования кеша для файлов JavaScript и таблиц стилей CSS, включенная в URL контрольная сумма остается неизменной до тех пор, пока файл не будет модифицирован.

Использование кеша данных

Инфраструктура MVC поддерживает в памяти кеш, который может применяться для кеширования фрагментов содержимого с целью ускорения визуализации представлений. Содержимое, подлежащее кешированию, обозначается в файле представления с использованием элемента `cache`, который обрабатывается классом `CacheTagHelper` через атрибуты, описанные в табл. 25.8.

Таблица 25.8. Атрибуты встроенной вспомогательной функции дескриптора для элементов `cache`

Имя	Описание
`enabled`	Этот атрибут типа `bool` используется для управления тем, будет ли кешироваться содержимое элемента `cache`. Отсутствие данного атрибута означает включение кеширования
`expires-on`	Этот атрибут применяется для указания абсолютного времени, когда срок существования кешированного содержимого истечет, в виде значения `DateTime`
`expires-after`	Этот атрибут используется для указания относительного времени, когда срок существования кешированного содержимого истечет, в виде значения `TimeSpan`
`expires-sliding`	Этот атрибут применяется для указания промежутка времени с момента последнего использования, когда срок существования кешированного содержимого истечет, в виде значения `TimeSpan`
`vary-by-header`	Этот атрибут применяется для указания имени заголовка запроса, который будет использоваться при управлении разными версиями кешированного содержимого
`vary-by-query`	Этот атрибут применяется для указания имени ключа строки запроса, который будет использоваться при управлении разными версиями кешированного содержимого
`vary-by-route`	Этот атрибут применяется для указания имени переменной маршрутизации, которая будет использоваться при управлении разными версиями кешированного содержимого
`vary-by-cookie`	Этот атрибут применяется для указания имени cookie-набора, который будет использоваться при управлении разными версиями кешированного содержимого
`vary-by-user`	Этот атрибут типа `bool` применяется для указания, будет ли имя аутентифицированного пользователя использоваться при управлении разными версиями кешированного содержимого
`vary-by`	Этот атрибут оценивается для предоставления ключа, применяемого при управлении разными версиями кешированного содержимого
`priority`	Этот атрибут используется для указания относительного приоритета, учитываемого в случае исчерпания пространства кеша в памяти и очистки кешированного содержимого, срок существования которого еще не истек

На заметку! Кеширование — удобный инструмент для многократного использования разделов содержимого, чтобы их не приходилось генерировать при каждом запросе. Но эффективное применение кеширования требует тщательного обдумывания и планирования. Хотя кеширование способно улучшить производительность приложения, оно также создает необычные эффекты, такие как получение пользователями устаревшего содержимого, наличие множества кешей с разными версиями содержимого и нарушение работы развертываемых обновлений из-за того, что содержимое, кешированное из предыдущей версии приложения, смешалось с содержимым из новой версии. Не включайте кеширование, если только не имеете дело с четко определенной проблемой низкой производительности, и удостоверьтесь в том, что хорошо понимаете влияние, которое оно будет оказывать.

Чтобы посмотреть, как работают атрибуты кеша, создадим папку `Components`, добавим в нее файл класса по имени `TimeViewComponent.cs` и определим компонент представления, приведенный в листинге 25.15.

Листинг 25.15. Содержимое файла `TimeViewComponent.cs` из папки `Components`

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace Cities.Components {

  public class TimeViewComponent : ViewComponent {

    public IViewComponentResult Invoke() {
      return View(DateTime.Now);
    }
  }
}
```

Метод `Invoke()` выбирает стандартное представление и передает ему объект `DateTime` в качестве модели представления. Для того чтобы обеспечить компонент представления необходимым ему представлением, создадим папку `Views/Home/Components/Time` и добавим в нее файл представления по имени `Default.cshtml` с разметкой из листинга 25.16.

Листинг 25.16. Содержимое файла `Default.cshtml` из папки `Views/Home/Components/Time`

```
@model DateTime

<div class="m-1 p-1 bg-info text-white">
  Rendered at @Model.ToString("HH:mm:ss")
</div>
```

Объект модели `DateTime` применяется для отображения текущего времени с точностью до секунды. В листинге 25.17 элемент `img`, добавленный в предыдущем разделе, заменяется выражением `@await Component.InvokeAsync()`, которое обращается к компоненту представления.

Листинг 25.17. Использование компонента представления в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
        asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
        asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
        asp-fallback-test-class="btn"
        asp-fallback-test-property="display"
        asp-fallback-test-value="inline-block"
        rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @await Component.InvokeAsync("Time")
  <div>@RenderBody()</div>
</body>
</html>
```

После запуска приложения появится баннер, отображающий время, когда содержимое было визуализировано. Подождав несколько секунд, перезагрузим страницу; легко заметить, что отображаемое время изменилось (рис. 25.6).

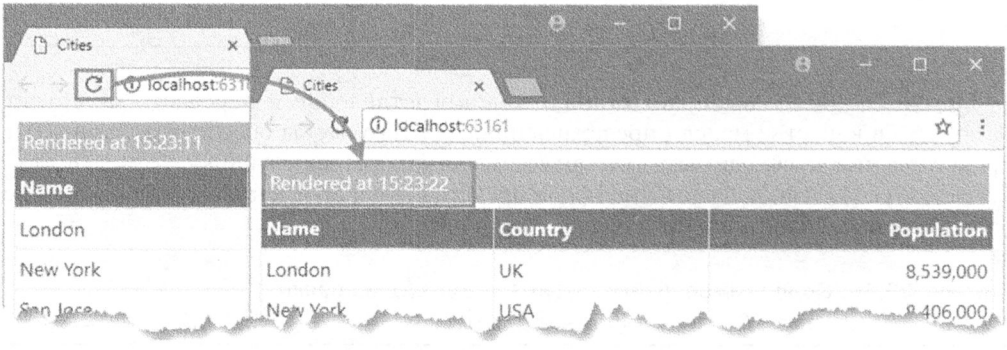


Рис. 25.6. Отображение времени в примере приложения

Элемент cache применяется для окружения содержимого, которое должно быть добавлено в кеш. В листинге 25.18 элемент cache используется для добавления в кеш вывода из компонента представления.

Листинг 25.18. Кеширование содержимого в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
```

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
        asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
        asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
        asp-fallback-test-class="btn"
        asp-fallback-test-property="display"
        asp-fallback-test-value="inline-block"
        rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <cache>
    @await Component.InvokeAsync("Time")
  </cache>
  <div>@RenderBody()</div>
</body>
</html>
```

Применение элемента `cache` без каких-либо атрибутов сообщает инфраструктуре MVC о том, что для удовлетворения всех будущих запросов необходимо повторно использовать отмеченное содержимое. Теперь после запуска приложения содержимое, сгенерированное компонентом представления, кешируется, поэтому даже при перезагрузке страницы будет отображаться то же самое показание времени.

Совет. Применяемый классом `CacheTagHelper` кеш основан на памяти, т.е. его емкость ограничивается доступным объемом ОЗУ. При нехватке пространства содержимое из кеша начинает удаляться, а в случае останова или перезапуска приложения все содержимое кеша утрачивается.

Установка времени истечения для кеша

Атрибуты `expires-*` позволяют указывать время, когда истекает срок существования кешированного содержимого, выражаемое как абсолютное время, время относительно текущего времени или промежуток времени, в течение которого кешированное содержимое не запрашивается. В листинге 25.19 с помощью атрибута `expires-after` указывается, что содержимое должно кешироваться на протяжении 15 секунд.

Листинг 25.19. Установка времени истечения для кеша в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
        asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
```

```
      asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
      asp-fallback-test-class="btn"
      asp-fallback-test-property="display"
      asp-fallback-test-value="inline-block"
      rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <cache expires-after="@TimeSpan.FromSeconds(15)">
    @await Component.InvokeAsync("Time")
  </cache>
  <div>@RenderBody()</div>
</body>
</html>
```

В результате запуска приложения обнаружится, что срок существования кешированных данных истекает спустя 15 секунд, после чего перезагрузка страницы приводит к вызову компонента представления и кешированию нового содержимого на протяжении следующих 15 секунд.

Установка фиксированного момента истечения

Посредством атрибута `expires-on`, принимающего значение `DateTime`, можно указывать фиксированный момент времени, при наступлении которого истечет срок существования кешированного содержимого (листинг 25.20).

Листинг 25.20. Указание фиксированного момента истечения в файле _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
      asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
      asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
      asp-fallback-test-class="btn"
      asp-fallback-test-property="display"
      asp-fallback-test-value="inline-block"
      rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <cache expires-on="@DateTime.Parse("2100-01-01")">
    @await Component.InvokeAsync("Time")
  </cache>
  <div>@RenderBody()</div>
</body>
</html>
```

Здесь указано, что данные должны кешироваться до 2100 года. Это не особенно полезная стратегия кеширования, поскольку приложение наверняка будет запущено заново до того, как начнется следующее столетие, но иллюстрирует способ указания фиксированного момента в будущем вместо выражения времени истечения относительно момента, когда содержимое попадает в кеш.

Установка периода истечения с момента последнего использования

Атрибут `expires-sliding` применяется для указания промежутка времени, спустя который срок существования содержимого истекает, если оно не извлекалось из кеша. В листинге 25.21 задан скользящий период истечения, составляющий 10 секунд.

Листинг 25.21. Указание периода истечения с момента последнего использования кешированного содержимого в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
        asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
        asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
        asp-fallback-test-class="btn"
        asp-fallback-test-property="display"
        asp-fallback-test-value="inline-block"
        rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <cache expires-sliding="@TimeSpan.FromSeconds(10)">
    @await Component.InvokeAsync("Time")
  </cache>
  <div>@RenderBody()</div>
</body>
</html>
```

Для оценки эффекта от наличия атрибута `express-sliding` понадобится запустить приложение и периодически перезагружать страницу. До тех пор, пока страница перезагружается в пределах 10-секундного интервала, будет применяться кешированное содержимое. В случае ожидания свыше 10 секунд перед перезагрузкой страницы кешированное содержимое отбрасывается, с помощью компонента представления генерируется новое содержимое, после чего процесс начинается заново.

Использование вариаций кеша

По умолчанию все запросы получают одно и то же кешированное содержимое. Класс `CacheTagHelper` способен поддерживать разные версии кешированного содержимого и применять их для удовлетворения различных типов HTTP-запросов, указываемых с использованием одного из атрибутов, имена которых начинаются с `vary-by`. В листинге 25.22 демонстрируется применение атрибута `vary-by-route` для создания вариаций кеша на основе значения `action`, предоставляемого системой маршрутизации.

Листинг 25.22. Создание вариации кеша в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
```

```
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
        asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
        asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
        asp-fallback-test-class="btn"
        asp-fallback-test-property="display"
        asp-fallback-test-value="inline-block"
        rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <cache expires-sliding="@TimeSpan.FromSeconds(10)" vary-by-route="action">
    @await Component.InvokeAsync("Time")
  </cache>
  <div>@RenderBody()</div>
</body>
</html>
```

Запустив приложение и воспользовавшись двумя вкладками или окнами браузера для запроса URL вида `/Home/Index` и `/Home/Create`, можно заметить, что каждая вкладка или окно получает собственное кешированное содержимое со своим сроком истечения, т.к. каждый запрос генерирует разное значение маршрутизации `action`. Класс `CacheTagHelper` поддерживает ряд атрибутов, которые определяют разнообразные вариации, включая кеширование содержимого для индивидуальных пользователей.

Существует также атрибут `vary-by`, который позволяет определять произвольные вариации кеша с применением любого значения данных. В листинге 25.23 воссоздается эффект от атрибута `vary-by-route` за счет указания значения, получаемого прямо из данных маршрута.

Листинг 25.23. Указание специальной вариации кеша в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
css/bootstrap.min.css"
        asp-fallback-href-include="/lib/bootstrap/dist/**/*.min.css"
        asp-fallback-href-exclude="**/*-reboot*,**/*-grid*"
        asp-fallback-test-class="btn"
        asp-fallback-test-property="display"
        asp-fallback-test-value="inline-block"
        rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <cache expires-sliding="@TimeSpan.FromSeconds(10)"
         vary-by="@ViewContext.RouteData.Values["action"]">
    @await Component.InvokeAsync("Time")
  </cache>
```

```
  <div>@RenderBody()</div>
</body>
</html>
```

Атрибут `vary-by` может использоваться для создания более сложных вариаций кеша, хотя его применение требует осторожности, поскольку довольно легко ослабить внимание и создать в итоге вариации, которые настолько специфичны, что содержимое в кеше никогда не будет задействовано повторно до истечения срока его существования.

Использование URL, относительных к приложению

Последняя встроенная вспомогательная функция дескриптора — класс `UrlResolutionTagHelper` — применяется для обеспечения поддержки URL, относительных к приложению, которые представляют собой URL, снабженные префиксом в виде тильды (символа ~). В листинге 25.24 элемента `link` внутри разделяемой компоновки изменен, чтобы он использовал явно определенный URL вместо URL, сгенерированного из системы маршрутизации с помощью вспомогательных функций дескрипторов.

Листинг 25.24. Применение явного URL в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <cache expires-sliding="@TimeSpan.FromSeconds(10)"
         vary-by="@ViewContext.RouteData.Values["action"]">
    @await Component.InvokeAsync("Time")
  </cache>
  <div>@RenderBody()</div>
</body>
</html>
```

Явные URL прекрасно подходят при условии, что вы осознаете необходимость их обновления при изменении схемы URL приложения. И для многих приложений это единственный фактор, который следует учитывать.

Однако есть приложения, которые будут предназначены для разделяемой среды, где единственный сервер поддерживает множество приложений, различаемых путем добавления префикса к URL. В листинге 25.25 приведена модифицированная конфигурация приложения, которая настраивает конвейер запросов для обработки запросов с префиксом `mvcpp`, эмулируя разделяемую среду.

Листинг 25.25. Добавление префикса URL в файле `Startup.cs` из папки `Cities`

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Cities.Models;
namespace Cities {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IRepository, MemoryRepository>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.Map("/mvcapp", appBuilder => {
        appBuilder.UseStatusCodePages();
        appBuilder.UseDeveloperExceptionPage();
        appBuilder.UseStaticFiles();
        appBuilder.UseMvcWithDefaultRoute();
      });
    }
  }
}
```

Метод `Map()` позволяет настраивать несколько конвейеров запросов с разными префиксами. Такая возможность не особенно полезна при повседневной разработке приложений MVC, потому что создавать префиксы URL внутри приложения MVC можно с использованием системы маршрутизации. Но для настоящей главы это удобное средство, т.к. оно означает, что каждый URL запрашивается клиентами, включая запросы к статическому содержимому.

Возникшую проблему можно заметить, запустив приложение и запросив URL вида `/mvcapp`, который теперь является стандартным URL для приложения и нацелен на действие `Index` контроллера `Home`. Теперь, когда все URL должны начинаться с `/mvcapp`, явный URL для таблицы стилей в элементе `link` не работает, из-за чего содержимое в приложении не может быть стилизовано (рис. 25.7). (Чтобы увидеть проблему, может потребоваться очистка кеша браузера.)

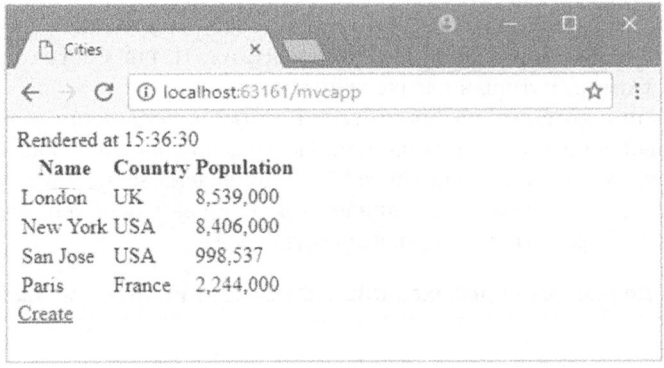


Рис. 25.7. Эффект от явно определенного URL

Проблему можно было бы устранить, обновив явный URL для включения префикса, но такой прием не всегда удается, потому что префикс может измениться при развертывании или не быть известным на этапе разработки. Более удачное решение предусматривает применение URL, относительных к приложению, когда путь к статическому содержимому выражается относительно любого префикса, который может быть сконфигурирован (листинг 25.26).

Листинг 25.26. Использование URL, относительного к приложению, в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Cities</title>
  <link href="~/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  <cache expires-sliding="@TimeSpan.FromSeconds(10)"
         vary-by="@ViewContext.RouteData.Values["action"]">
    @await Component.InvokeAsync("Time")
  </cache>
  <div>@RenderBody()</div>
</body>
</html>
```

Символ тильды обнаруживается классом `UrlResolutionTagHelper`, который заменяет его путем, требующимся для достижения содержимого папки `wwwroot`. Запустив приложение, можно заметить, что содержимое стилизовано, а при просмотре HTML-разметки, отправляемой браузеру, выяснится, что элемент `link` содержит URL, который включает префикс `mvcapp`:

```
<link href="/mvcapp/lib/bootstrap/dist/css/bootstrap.min.css"
      rel="stylesheet" />
```

Вспомогательный класс `UrlResolutionTag` ищет в URL широкий диапазон элементов, описанных в табл. 25.9.

Таблица 25.9. Элементы и атрибуты, трансформируемые классом `UrlResolutionTagHelper`

Элемент	Атрибуты	Элемент	Атрибуты
a	href	img	src, srcset
applet	archive	input	src, formaction
area	href	ins	cite
audio	src	link	href
base	href	menuitem	icon
blockquote	cite	object	archive, data
button	formaction	q	cite
del	cite	script	src
embed	src	source	src, srcset
form	action	track	src
html	manifest	video	src, poster
iframe	src		

Совет. Если вы применяете другую встроенную вспомогательную функцию дескриптора для генерации URL из системы маршрутизации, тогда генерируемая им HTML-разметка будет автоматически включать любой необходимый префикс, который получается из свойства контекста `HttpRequest.PathBase`, а его значение предоставляется сервером, размещающим приложение.

Резюме

В главе рассматривались встроенные вспомогательные функции дескрипторов, которые не связаны с HTML-формами. Такие вспомогательные функции дескрипторов помогают управлять доступом к файлам JavaScript и таблицам стилей CSS, создавать URL для якорных элементов, выполнять аннулирование кеша для изображений, кешировать данные и трансформировать URL, относительные к приложению. В следующей главе будет представлена система привязки моделей, которая используется для обработки данных в HTTP-запросах, так что их можно легко потреблять внутри приложения MVC.

ГЛАВА 26

Привязка моделей

Привязка моделей — это процесс создания объектов .NET с использованием данных из HTTP-запроса для снабжения методов действий аргументами, в которых они нуждаются. В настоящей главе будет описана работа системы привязки моделей, объяснено, как она привязывает простые типы, сложные типы и коллекции, а также продемонстрировано, каким образом взять на себя контроль над процессом, чтобы указывать часть запроса, которая предоставляет значения данных, требующиеся методам действий. В табл. 26.1 приведена сводка, позволяющая поместить привязку моделей в контекст.

Таблица 26.1. Помещение привязки моделей в контекст

Вопрос	Ответ
Что это такое?	Привязка моделей представляет собой процесс создания объектов, которые требуются методам действий в качестве аргументов, с применением значений данных, получаемых из HTTP-запроса
Чем она полезна?	Привязка моделей позволяет методам действий объявлять параметры, используя типы C#, и автоматически получать данные из запроса без необходимости в инспектировании, разборе и обработке данных напрямую
Как она используются?	В своей простейшей форме методы действий объявляют параметры, имена которых применяются для извлечения значений данных из HTTP-запроса. Часть запроса, используемая для получения данных, может быть сконфигурирована за счет применения атрибутов к параметрам методов действий
Существуют ли какие-то скрытые ловушки или ограничения?	Основная ловушка — получение данных из неправильной части запроса. Способ поиска данных внутри запросов объясняется в разделе "Понятие привязки моделей" далее в главе, а места для поиска могут быть указаны явно с использованием атрибутов, которые описаны позже в разделе "Указание источника данных привязки моделей"
Существуют ли альтернативы?	Методы действий вообще не обязаны объявлять параметры, и могут получать данные прямо из HTTP-запроса с применением объектов контекста, которые были описаны в главе 17. Однако результатом будет более сложный код, который труднее читать и сопровождать

В табл. 26.2 приведена сводка по главе.

Таблица 26.2. Сводка по главе

Задача	Решение	Листинг
Привязка простого типа или коллекции	Добавьте параметр к методу действия	26.1–26.10, 26.23–26.29
Привязка сложного типа	Удостоверьтесь в том, что HTML-разметка, генерируемая представлением, хорошо структурирована	26.11–26.19
Избирательная привязка свойств	Укажите имена значений данных с применением атрибута `Bind` либо используйте атрибут `BindNever` для исключения свойств модели из процесса привязки	26.20–26.22
Указание источника значения привязки данных	Примените к аргументу метода действия или к свойству модели атрибут, который идентифицирует, откуда должно поступать значение привязки	26.30–26.38

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `MvcModels` с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core.

Создание модели и хранилища

Создадим папку `Models` и добавим в нее файл класса по имени `Person.cs` с определением классов и перечисления из листинга 26.1.

Листинг 26.1. Содержимое файла `Person.cs` из папки `Models`

```
using System;

namespace MvcModels.Models {
  public class Person {
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
    public Address HomeAddress { get; set; }
    public bool IsApproved { get; set; }
    public Role Role { get; set; }
  }
  public class Address {
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
  }
```

```
  public enum Role {
    Admin,
    User,
    Guest
  }
}
```

Добавим в папку `Models` файл класса по имени `Repository.cs` и поместим в него определения интерфейса и класса реализации, приведенные в листинге 26.2.

Листинг 26.2. Содержимое файла `Repository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace MvcModels.Models {

  public interface IRepository {
    IEnumerable<Person> People { get; }

    Person this[int id] { get; set; }
  }

  public class MemoryRepository : IRepository {
    private Dictionary<int, Person> people
      = new Dictionary<int, Person> {
        [1] = new Person {PersonId = 1, FirstName = "Bob",
          LastName = "Smith", Role = Role.Admin},
        [2] = new Person {PersonId = 2, FirstName = "Anne",
          LastName = "Douglas", Role = Role.User},
        [3] = new Person {PersonId = 3, FirstName = "Joe",
          LastName = "Able", Role = Role.User},
        [4] = new Person {PersonId = 4, FirstName = "Mary",
          LastName = "Peters", Role = Role.Guest}
    };

    public IEnumerable<Person> People => people.Values;

    public Person this[int id] {
      get {
        return people.ContainsKey(id) ? people[id] : null;
      }
      set {
        people[id] = value;
      }
    }
  }
}
```

В интерфейсе `IRepository` определено свойство `People`, предназначенное для извлечения всех объектов модели, и индексатор, который позволяет извлекать или сохранять отдельные объекты `Person`. Класс `MemoryRepository` реализует этот интерфейс и применяет словарь со стандартным содержимым. Реализация хранилища не поддерживает постоянство, так что после останова или перезапуска состояние приложения возвращается к стандартному содержимому.

Создание контроллера и представления

Создадим папку `Controllers`, добавим в нее файл класса по имени `HomeController.cs` и определим контроллер, код которого показан в листинге 26.3. Класс контроллера полагается на внедрение зависимостей, чтобы получить хранилище, которое он использует в методе `Index()` для выбора одиночного объекта `Person` с применением значения его свойства `PersonId`.

Листинг 26.3. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
    public HomeController(IRepository repo) {
      repository = repo;
    }
    public ViewResult Index(int id) => View(repository[id]);
  }
}
```

Чтобы снабдить метод действия представлением, создадим папку `Views/Home` и добавим в нее файл представления Razor по имени `Index.cshtml` с разметкой из листинга 26.4, которая отображает ряд свойств объекта модели в таблице.

Листинг 26.4. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model Person
@{ Layout = "_Layout"; }

<div class="bg-primary m-1 p-1 text-white"><h2>Person</h2></div>

<table class="table table-sm table-bordered table-striped">
  <tr><th>PersonId:</th><td>@Model.PersonId</td></tr>
  <tr><th>First Name:</th><td>@Model.FirstName</td></tr>
  <tr><th>Last Name:</th><td>@Model.LastName</td></tr>
  <tr><th>Role:</th><td>@Model.Role</td></tr>
</table>
```

Представление `Index.cshtml` рассчитывает на разделяемую компоновку. Создадим папку `Views/Shared` и поместим в нее файл по имени `_Layout.cshtml`, содержимое которого приведено в листинге 26.5.

Листинг 26.5. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link asp-href-include="/lib/bootstrap/dist/**/*.min.css" rel="stylesheet" />
  @RenderSection("scripts", false)
</head>
```

```
<body class="m-1 p-1">
  @RenderBody()
</body>
</html>
```

Компоновка включает элемент `link` для таблицы стилей Bootstrap и визуализирует содержимое представления. Имеется также дополнительный раздел `scripts`, который будет использоваться позже в главе. Для упрощения представлений, применяемых в главе, добавим в файл `_ViewImports.cshtml` из папки `Views` пространство имен, содержащее классы модели (листинг 26.6).

Листинг 26.6. Импортирование пространства имен в файле `_ViewImports.cshtml` из папки `Views`

```
@using MvcModels.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Представления в примере опираются на CSS-пакет Bootstrap. Создадим в корневой папке проекта файл `bower.json` с применением шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел `dependencies` (листинг 26.7).

Листинг 26.7. Добавление пакета Bootstrap в файле `bower.json` из папки `MvcModels`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Конфигурирование приложения

Чтобы завершить начальную настройку примера приложения, включим в классе `Startup` инфраструктуру MVC и другие компоненты промежуточного ПО, нужные при разработке (листинг 26.8). Кроме того, создадим службу для хранилища, позволив контроллеру получать доступ к модели данных.

Листинг 26.8. Содержимое файла `Startup.cs` из папки `MvcModels`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using MvcModels.Models;

namespace MvcModels {
  public class Startup {
```

```
    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<IRepository, MemoryRepository>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
  }
}
```

Запустив приложение и запросив URL вида `/Home/Index/1`, мы получим результат, представленный на рис. 26.1. (В текущий момент запрос стандартного URL вызовет ошибку.)

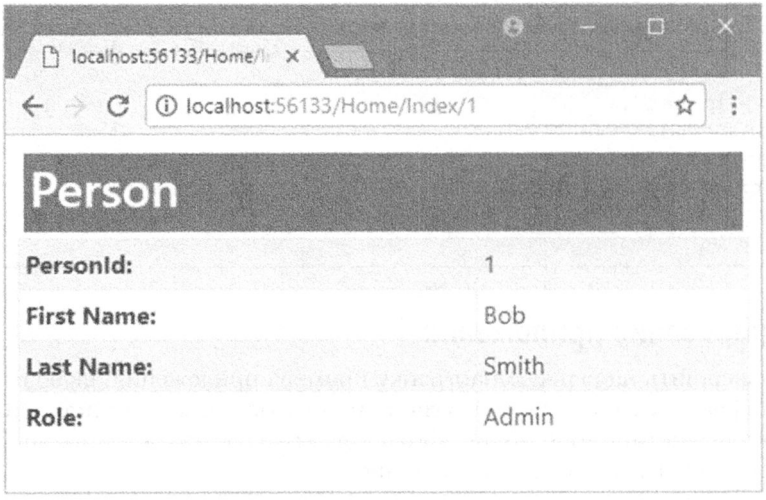


Рис. 26.1. Выполнение примера приложения

Понятие привязки моделей

Привязка моделей — элегантный мост между HTTP-запросом и методами действий C#. На привязку моделей в той или иной степени полагается большинство приложений MVC Framework, в том числе рассматриваемый в главе пример приложения. Привязка моделей использовалась при тестировании примера приложения из предыдущего раздела. Запрошенный URL содержал значение свойства `PersonId` объекта `Person`, подлежащего отображению:

```
/Home/Index/1
```

Инфраструктура MVC транслировала указанную часть URL и применила ее в качестве аргумента, когда вызывала метод `Index()` контроллера `Home`, чтобы обслужить запрос:

```
...
public ViewResult Index(int id) => View(repository[id]);
...
```

Чтобы иметь возможность вызывать метод `Index()`, инфраструктуре MVC необходимо значение для аргумента `id`. Снабжение этим значением входит в число обязанностей системы привязки моделей, отвечающей за предоставление значений данных, которые могут использоваться при вызове методов действий.

Система привязки моделей опирается на *связыватели моделей*, которые являются компонентами, ответственными за предоставление значений данных из одной части запроса или приложения. Стандартные связыватели моделей ищут значения данных в следующих трех местах:

- отправленные данные формы;
- переменные маршрутизации;
- строки запросов.

Каждый источник данных инспектируется по порядку до тех пор, пока не будет обнаружено значение для аргумента. В рассматриваемом примере данные формы отсутствуют, поэтому значение в них не может быть найдено. Но в конфигурации приложения, приведенной в листинге 26.8, предусмотрен сегмент маршрутизации по имени `id`, который позволит системе привязки моделей снабдить инфраструктуру MVC значением, подлежащим применению при вызове метода `Index()`. После того, как найдено подходящее значение, поиск останавливается, т.е. строка запроса не будет просматриваться на предмет наличия в ней значения данных.

Совет. В разделе "Указание источника данных привязки моделей" далее в главе объясняется, как можно указать источник данных привязки моделей с использованием атрибутов. Это позволяет установить, что значение данных должно получаться, скажем, из строки запроса, даже если подходящее значение имеется также и в данных формы или переменных маршрутизации.

Знание порядка поиска значений данных важно из-за того, что запрос может содержать несколько значений, например:

```
/Home/Index/3?id=1
```

Система маршрутизации обработает запрос и сопоставит сегмент `id` в шаблоне URL со значением `3`, но строка запроса содержит значение `id`, равное `1`. Поскольку данные маршрутизации просматриваются раньше строки запроса, метод действия `Index()` получит значение `3`, а значение из строки запроса будет проигнорировано.

С другой стороны, в случае запроса URL, который не имеет сегмента `id`, будет исследоваться строка запроса, т.е. URL вроде приведенного ниже также позволит системе привязки моделей предоставить инфраструктуре MVC значение аргумента `id`, чтобы она смогла вызывать метод `Index()`:

```
/Home/Index?id=1
```

Результаты запросов обоих URL показаны на рис. 26.2.

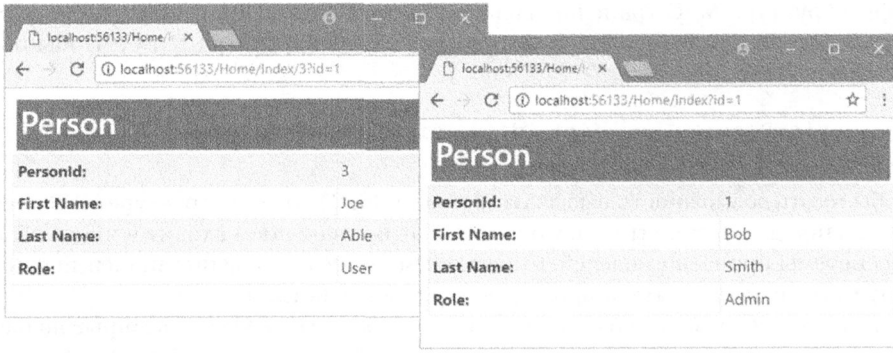


Рис. 26.2. Эффект от упорядочивания источников данных привязки моделей

Стандартные значения привязки

Привязка моделей является негарантированным средством, т.е. MVC будет применять его в попытке получить значения, необходимые для вызова метода действия, но все равно вызовет метод, даже если значения данных предоставить не удалось. Это может приводить к непредсказуемому поведению. Например, запрос URL вида `/Home/Index` генерирует исключение (рис. 26.3).

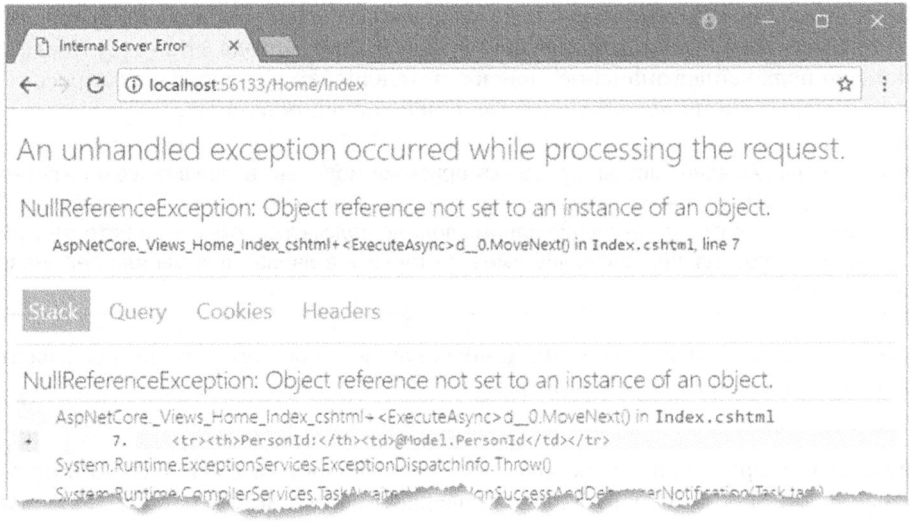


Рис. 26.3. Ошибка обработки свойства модели

Исключение было инициировано не системой привязки моделей. Взамен оно произошло при обработке представления `Index`, выбранного методом действия `Index()`. Чтобы вызвать метод `Index()`, инфраструктура MVC должна обеспечить значение для аргумента `id`, а потому она предлагает каждому связывателю модели проинспектировать свою часть запроса и предоставить значение.

В примере отсутствуют данные формы, не указано значение для сегмента маршрутизации id и нет строки запроса в URL, так что система привязки моделей не в состоянии предоставить значение данных. Чтобы вызвать метод Index(), инфраструктура MVC обязана предоставить *какое-нибудь* значение для аргумента id, поэтому она использует стандартное значение и надеется на лучшее. Стандартным значением для аргументов int является 0, что и приводит к генерации исключения. В методе Index() значение аргумента id применяется при извлечении объекта модели из хранилища:

```
...
public ViewResult Index(int id) => View(repository[id]);
...
```

Когда MVC использует стандартное значение, метод действия пытается извлечь объект модели с id, равным 0. Такой объект не существует, из-за чего хранилище возвращает значение null, которое затем передается методу View() контроллера, чтобы указать данные модели представления для представления Index.cshtml. Когда выражение Razor в файле Index.cshtml пытается обратиться к свойствам объекта модели представления, возникает исключение NullReferenceException, как было проиллюстрировано на рис. 26.3.

Это означает, что методы действий должны быть написаны так, чтобы справиться со стандартными значениями, предоставляемыми системой привязки моделей, что может быть сделано несколькими способами. Можно добавить стандартные значения в шаблоны URL маршрутов (как было описано в главе 15), присвоить стандартные значения параметрам методов действий или обеспечить, чтобы методы действий не передавали недопустимые значения данных в качестве части своих ответов. Предпочтительный подход будет зависеть от того, что предпринимает метод действия; в листинге 26.9 избран последний подход, который предусматривает изменение метода действия так, чтобы он всегда передавал методу View() объект Person, даже если аргумент id не соответствует какому-то объекту в модели данных.

Листинг 26.9. Защита от стандартных значений привязки модели в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
using System.Linq;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public ViewResult Index(int id) =>
      View(repository[id] ?? repository.People.First());
  }
}
```

В методе действия с применением LINQ и операции объединения с null возвращается первый объект из хранилища, если значение параметра id не привело к извлечению объекта.

Привязка простых типов

Когда доступно подходящее значение, оно преобразуется в значение типа C#, которое может использоваться при вызове метода действия. Простые типы — это значения, происходящие из одного элемента данных в запросе, который может быть разобран из строки. В их состав входят числовые значения, булевские значения, даты и, конечно же, значения `string`.

Аргумент `id` метода действия `Index()` имеет тип `int`, так что процесс привязки моделей снабжает инфраструктуру MVC значением за счет разбора переменной сегмента `id` в значение `int`.

Если значение из запроса не может быть преобразовано (например, для параметра, требующего значение `int`, указано значение `apple`), тогда процесс привязки моделей окажется неспособным предоставить значение приложению, и будет применяться стандартное значение.

В итоге возникает проблема, т.к. метод действия будет получать стандартное значение `0` в двух ситуациях. Первая из них — когда запрос содержит значение, которое не может быть преобразовано в тип аргумента, как в URL вида `/Home/Index/Apple`. Вторая ситуация — когда запрос содержит значение, поддающееся преобразованию, но равное `0`, как в URL вида `/Home/Index/0`.

Большинству приложений нужна возможность различения таких ситуаций, и легче всего это делается с использованием для аргумента метода действия типа, допускающего `null` (листинг 26.10).

Листинг 26.10. Применение типа, допускающего `null`, в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
    public HomeController(IRepository repo) {
      repository = repo;
    }
    public IActionResult Index(int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }
  }
}
```

Стандартным значением для типов, допускающих `null`, является `null`. Оно позволяет проводить различие между запросами, не содержащими значение, которое может быть преобразовано в `int`, и запросами со значением, поддающимся преобразованию в `int`, но равным `0`. Реализация метода `Index()` в этом примере использует метод `NotFound()` для возвращения ошибки 404, если аргумент типа, допускающего

null, не имеет значения или если его значение не соответствует какому-то объекту в модели. Такой подход более надежен, чем просто полагаться на то, что первый объект в модели окажется подходящим, как делалось в предыдущем разделе.

Привязка сложных типов

Когда параметр метода действия имеет сложный тип (другими словами, любой тип, который не может быть разобран из одиночного строкового значения), то процесс привязки моделей применяет рефлексию для получения набора открытых свойств целевого типа и выполняет процесс привязки в отношении каждого из них по очереди. Чтобы посмотреть, как работает процесс, добавим в контроллер Home два метода действия (листинг 26.11).

Листинг 26.11. Добавление методов действий в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public IActionResult Index(int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }

    public ViewResult Create() => View(new Person());

    [HttpPost]
    public ViewResult Create(Person model) => View("Index", model);
  }
}
```

Версия метода Create() без параметров создает новый объект Person и передает его методу View(), который выбирает стандартное представление, ассоциированное с действием. Добавим в папку Views/Home файл представления по имени Create.cshtml и поместим в него разметку, приведенную в листинге 26.12.

Листинг 26.12. Содержимое файла Create.cshtml из папки Views/Home

```
@model Person
@{
  ViewBag.Title = "Create Person";
  Layout = "_Layout";
}
```

```
<form asp-action="Create" method="post">
  <div class="form-group">
    <label asp-for="PersonId"></label>
    <input asp-for="PersonId" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="FirstName"></label>
    <input asp-for="FirstName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="LastName"></label>
    <input asp-for="LastName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Role"></label>
    <select asp-for="Role" class="form-control"
            asp-items="@new SelectList(Enum.GetNames(typeof(Role)))">
    </select>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Представление `Create.cshtml` содержит форму, позволяющую предоставлять значения для ряда свойств объекта `Person`, элемент `form` которой отправляет данные обратно версии метода `Create()` контроллера `Home`, декорированной атрибутом `HttpPost`.

Метод действия, который получает данные формы, использует для их отображения представление `Views/Home/Index.cshtml`. Чтобы посмотреть на все это в работе, запустим приложение, перейдем к URL вида `/Home/Create`, заполним форму и щелкнем на кнопке Submit (Отправить); результат показан на рис. 26.4.

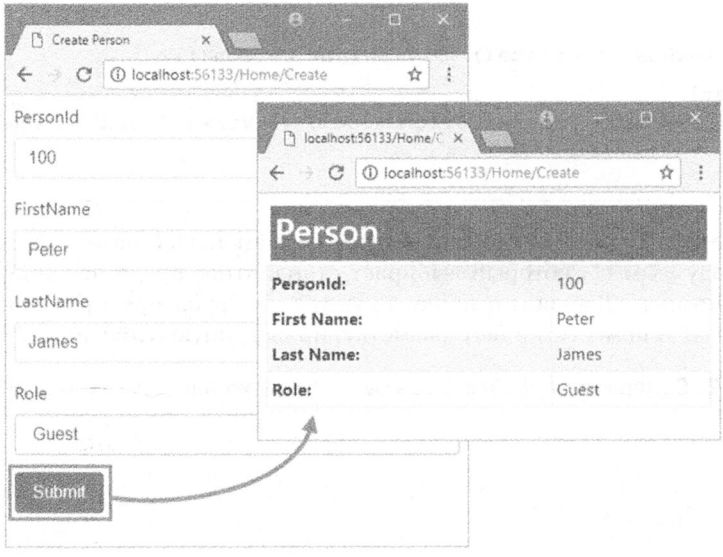


Рис. 26.4. Применение методов действий `Create()`

После отправки данных формы серверу процесс привязки моделей обнаруживает, что метод действия требует сложного типа — объекта Person. Класс Person исследуется на предмет открытых свойств. Для каждого открытого свойства, которое возвращает простой тип, связыватель модели пытается найти значение в запросе, как делалось в предыдущем примере.

Таким образом, например, связыватель модели встречает свойство PersonId и ищет значение PersonId в тех же самых местах, где производился поиск значения id в предыдущем разделе. Поскольку данные формы содержат подходящее значение, настроенное с использованием вспомогательной функции дескриптора asp-for в элементе input, именно это значение и будет применено.

Если свойство требует другого сложного типа, тогда процесс повторяется для нового типа. Сначала получается набор открытых свойств и связыватель пытается найти значения для всех полученных свойств. Разница в том, что имена свойств являются вложенными. Например, свойство HomeAddress класса Person имеет тип Address, как выделено ниже:

```
using System;

namespace MvcModels.Models {

  public class Person {
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
    public Address HomeAddress { get; set; }
    public bool IsApproved { get; set; }
    public Role Role { get; set; }
  }
  public class Address {
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
  }
  public enum Role {
    Admin,
    User,
    Guest
  }
}
```

При поиске значения для свойства Line1 связыватель модели ищет значение для HomeAddress.Line1, т.к. имя свойства в объекте модели комбинируется с именем свойства во вложенном типе модели.

Создание легко привязываемой HTML-разметки

Использование префиксов означает, что представления должны включать информацию, интересующую связыватель модели. Это легко делается с применением вспомогательных функций дескрипторов, которые автоматически добавляют требуемые префиксы к трансформируемым элементам. В листинге 26.13 форма расширена, чтобы принимать данные адреса.

Листинг 26.13. Обновление формы в файле `Create.cshtml` из папки `Views/Home`

```
@model Person
@{
  ViewBag.Title = "Create Person";
  Layout = "_Layout";
}
<form asp-action="Create" method="post">
  <div class="form-group">
    <label asp-for="PersonId"></label>
    <input asp-for="PersonId" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="FirstName"></label>
    <input asp-for="FirstName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="LastName"></label>
    <input asp-for="LastName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Role"></label>
    <select asp-for="Role" class="form-control"
            asp-items="@new SelectList(Enum.GetNames(typeof(Role)))">
    </select>
  </div>
  <div class="form-group">
    <label asp-for="HomeAddress.City"></label>
    <input asp-for="HomeAddress.City" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="HomeAddress.Country"></label>
    <input asp-for="HomeAddress.Country" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Когда используется вспомогательная функция дескриптора, вложенное имя свойства указывается с применением соглашений C#, так что внешние и вложенные имена свойств отделяются точками: `HomeAddress.Country`. Запустив приложение, запросив URL вида `/Home/Create` и просмотрев HTML-разметку, отправленную браузеру, вы заметите, что для некоторых атрибутов используется другое соглашение:

```
<div class="form-group">
  <label for="HomeAddress_City">City</label>
  <input class="form-control" type="text" id="HomeAddress_City"
         name="HomeAddress.City" value="" />
</div>
<div class="form-group">
  <label for="HomeAddress_Country">Country</label>
  <input class="form-control" type="text" id="HomeAddress_Country"
         name="HomeAddress.Country" value="" />
</div>
```

Атрибуты `name` элементов `input` следуют стилю C#, но в атрибутах `for` элементов `label` и атрибутах `id` элементов `input` имена свойств отделяются с помощью символов подчеркивания. Если вы предпочитаете определять HTML-элементы без вспомогательных функций дескрипторов, то должны обеспечить применение одной и той же схемы именования. Как следствие, предпринимать какие-либо специальные действия, чтобы гарантировать создание связывателем модели объекта `Address` для свойства `HomeAddress` не понадобится. Сказанное можно продемонстрировать, изменив представление `Index.cshtml` для отображения свойств `HomeAddress`, когда они отправляются из формы (листинг 26.14).

Листинг 26.14. Отображение свойств `HomeAddress` в файле `Index.cshtml` из папки `Views/Home`

```
@model Person
@{ Layout = "_Layout"; }

<div class="bg-primary m-1 p-1 text-white"><h2>Person</h2></div>

<table class="table table-sm table-bordered table-striped">
  <tr><th>PersonId:</th><td>@Model.PersonId</td></tr>
  <tr><th>First Name:</th><td>@Model.FirstName</td></tr>
  <tr><th>Last Name:</th><td>@Model.LastName</td></tr>
  <tr><th>Role:</th><td>@Model.Role</td></tr>
  <tr><th>City:</th><td>@Model.HomeAddress?.City</td></tr>
  <tr><th>Country:</th><td>@Model.HomeAddress?.Country</td></tr>
</table>
```

Запустив приложение и перейдя на URL вида `/Home/Create`, можно ввести значения для свойств `City` и `Country`, после чего отправить форму и убедиться, что значения были привязаны к объекту модели (рис. 26.5).

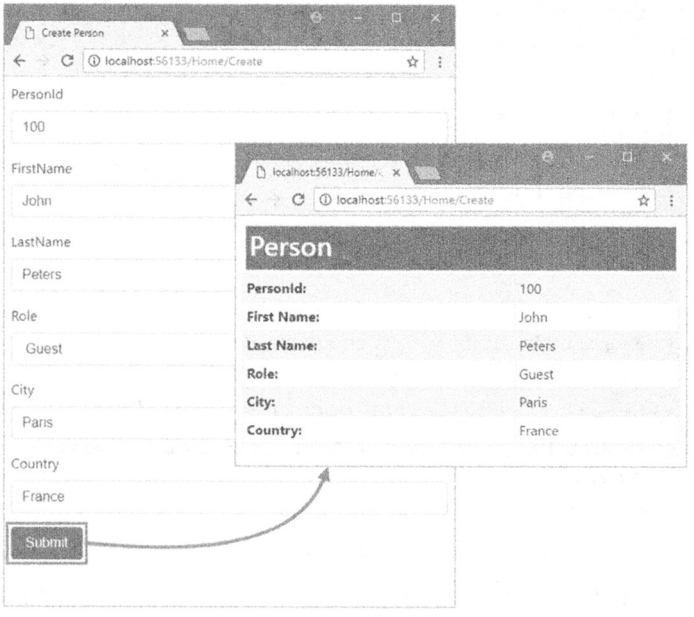


Рис. 26.5. Привязка свойств в сложных объектах

Указание специальных префиксов

Встречаются ситуации, когда генерируемая HTML-разметка относится к одному типу объекта, но ее необходимо привязать к другому типу. Это означает, что префиксы, содержащиеся в представлении, не будут соответствовать структуре, которую ожидает связыватель модели, и данные не смогут быть правильно обработаны. Чтобы взглянуть на проблему, добавим в папку `Models` файл по имени `AddressSummary.cs` с определением класса, показанным в листинге 26.15.

Листинг 26.15. Содержимое файла `AddressSummary.cs` из папки `Models`

```
namespace MvcModels.Models {

  public class AddressSummary {
    public string City { get; set; }
    public string Country { get; set; }
  }
}
```

Добавим в контроллер `Home` новый метод действия, который использует класс `AddressSummary` (листинг 26.16).

Листинг 26.16. Добавление метода действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public IActionResult Index(int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }

    public ViewResult Create() => View(new Person());

    [HttpPost]
    public ViewResult Create(Person model) => View("Index", model);

    public ViewResult DisplaySummary(AddressSummary summary) =>
      View(summary);
  }
}
```

Новый метод действия называется `DisplaySummary()`. Он принимает параметр `AddressSummary`, который передается методу `View()`, так что параметр мо-

жет быть отображен в стандартном представлении. Создадим в папке Views/Home файл DisplaySummary.cshtml и поместим в него разметку, приведенную в листинге 26.17.

Листинг 26.17. Содержимое файла DisplaySummary.cshtml из папки Views/Home

```
@model AddressSummary
@{
  ViewBag.Title = "DisplaySummary";
  Layout = "_Layout";
}

<div class="bg-primary m-1 p-1 text-white"><h2>Address</h2></div>

<table class="table table-sm table-bordered table-striped">
  <tr><th>City:</th><td>@Model.City</td></tr>
  <tr><th>Country:</th><td>@Model.Country</td></tr>
</table>
```

Представление DisplaySummary.cshtml отображает значения двух свойств, определенных в классе AddressSummary. Для иллюстрации проблемы с префиксами во время привязки моделей разных типов модифицируем элемент form в представлении Create.cshtml, чтобы он отправлял данные действию DisplaySummary (листинг 26.18).

Листинг 26.18. Изменение целевого действия формы в файле Create.cshtml из папки Views/Home

```
@model Person
@{
  ViewBag.Title = "Create Person";
  Layout = "_Layout";
}

<form asp-action="DisplaySummary" method="post">
  <!-- Для краткости HTML-элементы не показаны -->
</form>
```

Запустив приложение и перейдя на URL вида /Home/CreatePerson, можно выяснить, что происходит. После отправки формы значения, которые были введены для свойств City и Country, не отображаются в HTML-разметке, сгенерированной представлением DisplaySummary.cshtml.

Проблема в том, что атрибуты name в форме имеют префикс HomeAddress, не являющийся тем, который ищет связыватель модели при попытке привязать тип AddressSummary.

Чтобы исправить проблему, к параметру метода действия можно применить атрибут Bind, в котором указывается префикс, подлежащий использованию во время привязки модели (листинг 26.19).

Листинг 26.19. Изменение префикса привязки модели в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
```

```
namespace MvcModels.Controllers {
  public class HomeController : Controller {
    private IRepository repository;
    public HomeController(IRepository repo) {
      repository = repo;
    }
    public IActionResult Index(int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }
    public ViewResult Create() => View(new Person());
    [HttpPost]
    public ViewResult Create(Person model) => View("Index", model);
    public ViewResult DisplaySummary(
      [Bind(Prefix = nameof(Person.HomeAddress))] AddressSummary summary)
        => View(summary);
  }
}
```

Синтаксис неудобен, но результат приемлем. При заполнении свойств объекта `AddressSummary` связыватель модели будет искать в запросе значения данных `HomeAddress.City` и `HomeAddress.Country`. В результате запуска приложения и отправки формы еще раз обнаружится, что значения, введенные в поля `City` и `Country`, теперь отображаются корректно (рис. 26.6).



Рис. 26.6. Привязка свойств другого объектного типа

Решение может выглядеть слишком длинным для такой простой проблемы, но потребность в привязке разных видов объектов возникает на удивление часто, и о таком приеме полезно знать.

Избирательная привязка свойств

Предположим, что свойство Country класса AddressSummary является критически важным, и пользователь не должен иметь возможность указывать для него значения. Первое, что мы можем предпринять — воспрепятствовать просмотру и редактированию свойства Country пользователем, удостоверившись в отсутствии внутри представлений приложения любых HTML-элементов, которые ссылаются на данное свойство.

Тем не менее, злоумышленник может просто отредактировать данные формы, посылаемые серверу, во время отправки формы и указать нужное ему значение для свойства Country. На самом деле связывателю модели необходимо сообщить о том, чтобы он не привязывал свойство Country к значению из запроса. Для этого можно сконфигурировать атрибут Bind на параметре метода действия, указав имена только тех свойств, которые должны привязываться (листинг 26.20).

Листинг 26.20. Указание свойств, подлежащих привязке, в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public IActionResult Index(int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }

    public ViewResult Create() => View(new Person());

    [HttpPost]
    public ViewResult Create(Person model) => View("Index", model);

    public ViewResult DisplaySummary(
      [Bind(nameof(AddressSummary.City), Prefix = nameof(Person.HomeAddress))]
        AddressSummary summary) => View(summary);
  }
}
```

Первый аргумент атрибута Bind представляет собой разделяемый запятыми список имен свойств, которые должны быть включены в процесс привязки моделей.

В листинге 26.20 указано, что свойство `City` должно быть включено в процесс, а поскольку свойство `Country` в списке отсутствует, то оно будет исключено из процесса привязки моделей.

Запустив приложение, запросив URL вида `/Home/Create`, заполнив форму и отправив ее, легко заметить, что для свойства `Country` никакого значения не отображается, хотя оно было отправлено браузером как часть HTTP-запроса `POST` (рис. 26.7).

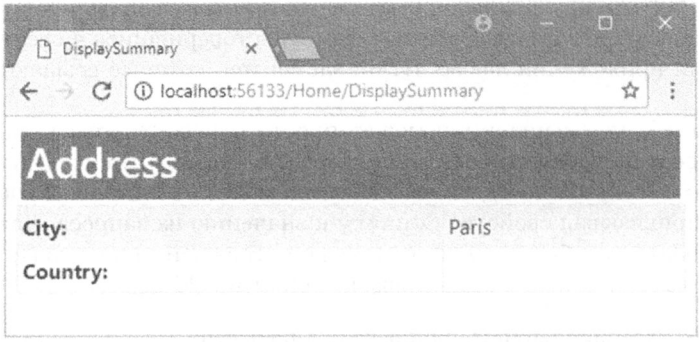


Рис. 26.7. Исключение свойства из процесса привязки моделей

Когда атрибут `Bind` применяется к параметру метода действия, он влияет только на экземпляры данного класса, которые привязываются для этого метода действия; все остальные методы действий продолжат попытки привязать все свойства, определенные типом параметра. Если нужно получить более обширный эффект, тогда атрибут `Bind` можно применить к самому классу модели, как демонстрируется в листинге 26.21.

Листинг 26.21. Применение атрибута `Bind` в файле `AddressSummary.cs` из папки `Models`

```
using Microsoft.AspNetCore.Mvc;

namespace MvcModels.Models {

  [Bind(nameof(City))]
  public class AddressSummary {
    public string City { get; set; }

    public string Country { get; set; }
  }
}
```

Свойства можно также явно исключать, декорируя их атрибутом `BindNever` (листинг 26.22), но тогда новые свойства, добавляемые в класс модели, будут включаться в процесс привязки моделей, только если не забыть о применении к ним упомянутого атрибута.

Листинг 26.22. Применение атрибута `NeverBind` в файле `AddressSummary.cs` из папки `Models`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;
```

```
namespace MvcModels.Models {
  public class AddressSummary {
    public string City { get; set; }
    [BindNever]
    public string Country { get; set; }
  }
}
```

Совет. Имеется также атрибут `BindRequired`, который сообщает процессу привязки моделей о том, что запрос должен включать значение для свойства. Если запрос не содержит требуемого значения, тогда возникает ошибка проверки достоверности модели, как будет объясняться в главе 27.

Привязка массивов и коллекций

Процесс привязки моделей обладает рядом элегантных возможностей для привязки данных запросов к массивам и коллекциям, которые будут описаны в последующих разделах.

Привязка массивов

Одной из элегантных возможностей стандартного связывателя модели является его поддержка для методов действий параметров, представляющих собой массивы. В целях демонстрации добавим в контроллер `Home` новый метод по имени `Names()`, код которого показан в листинге 26.23.

Листинг 26.23. Добавление метода действия в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
  public class HomeController : Controller {
    private IRepository repository;
    public HomeController(IRepository repo) {
      repository = repo;
    }
    public IActionResult Index(int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }
    public ViewResult Create() => View(new Person());
    [HttpPost]
    public ViewResult Create(Person model) => View("Index", model);
```

```
    public ViewResult DisplaySummary(
      [Bind(nameof(AddressSummary.City), Prefix = nameof(Person.HomeAddress))]
        AddressSummary summary) => View(summary);

    public ViewResult Names(string[] names) => View(names ?? new string[0]);
  }
}
```

Метод действия `Names()` имеет параметр типа строкового массива по имени `names`. Связыватель модели будет искать любые элементы данных под названием `names` и создавать массив, содержащий эти значения. Чтобы снабдить метод действия представлением, создадим в папке `Views/Home` файл Razor по имени `Names.cshtml` и поместим в него разметку из листинга 26.24.

Листинг 26.24. Содержимое файла `Names.cshtml` из папки `Views/Home`

```
@model string[]
@{
  ViewBag.Title = "Names";
  Layout = "_Layout";
}

@if (Model.Length == 0) {
  <form asp-action="Names" method="post">
    @for (int i = 0; i < 3; i++) {
      <div class="form-group">
        <label>Name @(i + 1):</label>
        <input id="names" name="names" class="form-control" />
      </div>
    }
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>
} else {
  <table class="table table-sm table-bordered table-striped">
    @foreach (string name in Model) {
      <tr><th>Name:</th><td>@name</td></tr>
    }
  </table>
  <a asp-action="Names" class="btn btn-primary">Back</a>
}
```

Представление `Names.cshtml` отображает разное содержимое на основе количества элементов в модели представления. Если элементы отсутствуют, то представление выводит форму, которая содержит три идентичных элемента `input`:

```
...
<form method="post" action="/Home/Names">
  <div class="form-group">
    <label>Name 1:</label>
    <input id="names" name="names" class="form-control" />
  </div>
  <div class="form-group">
    <label>Name 2:</label>
    <input id="names" name="names" class="form-control" />
  </div>
  <div class="form-group">
```

```
    <label>Name 3:</label>
    <input id="names" name="names" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
...
```

Когда форма отправляется, процесс привязки моделей обнаруживает, что целевой метод действия принимает массив, и производит поиск элементов данных, которые имеют такое же имя, как у параметра метода действия. В настоящем примере это означает, что все значения из элементов `input` с атрибутом `name`, установленным в `names`, собираются вместе для создания массива, который будет использоваться в качестве аргумента при вызове метода действия. Чтобы взглянуть на результат, понадобится запустить приложение, перейти на URL вида `/Home/Names` и заполнить форму. После отправки формы можно заметить, что отображаются все введенные значения (рис. 26.8).

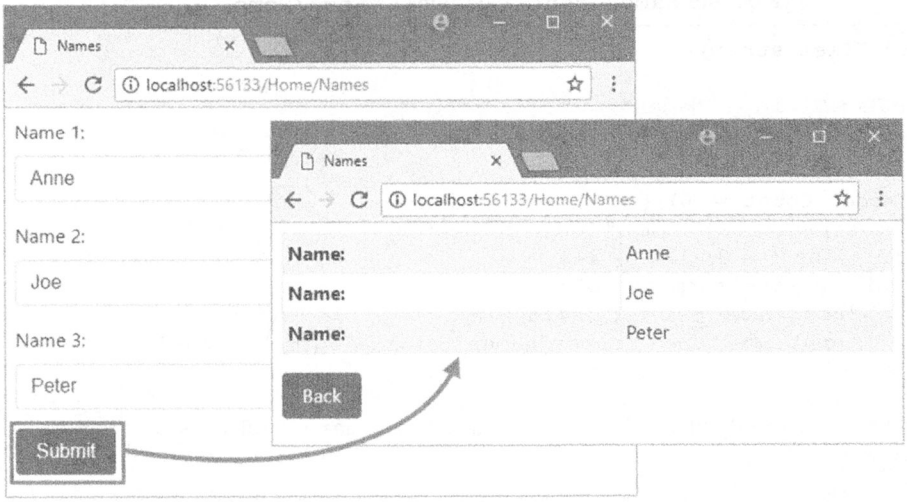


Рис. 26.8. Привязка моделей для массивов

Привязка коллекций

Процесс привязки моделей способен создавать не только массивы. Он также поддерживает классы коллекций. В листинге 26.25 тип параметра метода действия `Names()` изменен на строго типизированный список.

Листинг 26.25. Применение строго типизированной коллекции в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
using System.Collections.Generic;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
```

```
    public HomeController(IRepository repo) {
      repository = repo;
    }
    // ...для краткости другие методы действий не показаны...
    public ViewResult Names(IList<string> names) =>
      View(names ?? new List<string>());
  }
}
```

Здесь используется интерфейс `IList<T>`. Указывать конкретный класс реализации не нужно, хотя при желании его можно было бы указать. В листинге 26.26 приведено модифицированное содержимое файла представления `Names.cshtml`, в котором задействован новый тип модели.

Листинг 26.26. Применение коллекции в качестве типа модели в файле `Names.cshtml` из папки `Views/Home`

```
@model IList<string>
@{
  ViewBag.Title = "Names";
  Layout = "_Layout";
}
@if (Model.Count == 0) {
  <form asp-action="Names" method="post">
    @for (int i = 0; i < 3; i++) {
      <div class="form-group">
        <label>Name @(i + 1):</label>
        <input id="names" name="names" class="form-control" />
      </div>
    }
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>
} else {
  <table class="table table-sm table-bordered table-striped">
    @foreach (string name in Model) {
      <tr><th>Name:</th><td>@name</td></tr>
    }
  </table>
  <a asp-action="Names" class="btn btn-primary">Back</a>
}
```

Функциональность действия `Names` не изменилась, но теперь оно способно работать с классом коллекции вместо массива.

Привязка коллекций сложных типов

Индивидуальные значения данных можно также привязывать к массиву сложных типов, что позволяет собирать из единственного запроса множество объектов (таких как класс модели `AddressSummary` в примере). В листинге 26.27 контроллер `Home` дополняется новым методом действия по имени `Address()`, параметр которого представляет собой список объектов `AddressSummary`.

Листинг 26.27. Определение метода действия в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;
using System.Collections.Generic;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
    public HomeController(IRepository repo) {
      repository = repo;
    }
    public IActionResult Index(int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }
    public ViewResult Create() => View(new Person());

    [HttpPost]
    public ViewResult Create(Person model) => View("Index", model);
    public ViewResult DisplaySummary(
      [Bind(nameof(AddressSummary.City), Prefix = nameof(Person.HomeAddress))]
        AddressSummary summary) => View(summary);
    public ViewResult Names(IList<string> names) =>
      View(names ?? new List<string>());
    public ViewResult Address(IList<AddressSummary> addresses) =>
      View(addresses ?? new List<AddressSummary>());
  }
}
```

Чтобы снабдить новый метод действия представлением, добавим в папку Views/Home файл по имени Address.cshtml с разметкой из листинга 26.28.

Листинг 26.28. Содержимое файла Address.cshtml из папки Views/Home

```
@model IList<AddressSummary>
@{
  ViewBag.Title = "Address";
  Layout = "_Layout";
}

@if (Model.Count() == 0) {
  <form asp-action="Address" method="post">
    @for (int i = 0; i < 3; i++) {
      <fieldset class="form-group">
        <legend>Address @(i + 1)</legend>
        <div class="form-group">
          <label>City:</label>
          <input name="[@i].City" class="form-control" />
        </div>
```

```
      <div class="form-group">
        <label>Country:</label>
        <input name="[@i].Country" class="form-control" />
      </div>
    </fieldset>
  }
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
} else {
  <table class="table table-sm table-bordered table-striped">
    <tr><th>City</th><th>Country</th></tr>
    @foreach (var address in Model) {
      <tr><td>@address.City</td><td>@address.Country</td></tr>
    }
  </table>
  <a asp-action="Address" class="btn btn-primary">Back</a>
}
```

Представление `Address.cshtml` визуализирует элемент `form`, если элементы в коллекции модели отсутствуют. Элемент `form` состоит из пары элементов `input`, атрибуты `name` которых имеют префиксы в виде индексов массива:

```
...
<form method="post" action="/Home/Address">
  <fieldset class="form-group">
    <legend>Address 1</legend>
    <div class="form-group">
      <label>City:</label>
      <input name="[0].City" class="form-control" />
    </div>
    <div class="form-group">
      <label>Country:</label>
      <input name="[0].Country" class="form-control" />
    </div>
  </fieldset>
  <fieldset class="form-group">
    <legend>Address 2</legend>
    <div class="form-group">
      <label>City:</label>
      <input name="[1].City" class="form-control" />
    </div>
    <div class="form-group">
      <label>Country:</label>
      <input name="[1].Country" class="form-control" />
    </div>
  </fieldset>
  <fieldset class="form-group">
    <legend>Address 3</legend>
    <div class="form-group">
      <label>City:</label>
      <input name="[2].City" class="form-control" />
    </div>
    <div class="form-group">
      <label>Country:</label>
```

```
        <input name="[2].Country" class="form-control" />
      </div>
    </fieldset>
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>
  ...
```

Когда форма отправляется, связыватель модели выясняет, что необходимо создать коллекцию объектов `AddressSummary`, и использует префиксы индексов массива в атрибутах `name`, чтобы получить значения для свойств этих объектов. Свойства с префиксом `[0]` применяются для первого объекта `AddressSumary`, свойства с префиксом `[1]` — для второго объекта `AddressSumary` и т.д.

Представление `Address.cshtml` определяет элементы `input` для трех таких индексированных объектов и отображает их, если коллекция модели содержит элементы. Перед тем, как можно будет продемонстрировать это в работе, понадобится удалить атрибут `BindNever` из класса модели `AddressSummary`, иначе связыватель модели проигнорирует свойство `Country` (листинг 26.29).

Листинг 26.29. Удаление атрибута `BindNever` в файле `AddressSummary.cs` из папки `Models`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace MvcModels.Models {

  public class AddressSummary {

    public string City { get; set; }

    // [BindNever]
    public string Country { get; set; }
  }
}
```

Запустив приложение и перейдя на URL вида `/Home/Address`, можно понаблюдать за работой процесса привязки для коллекций специальных объектов. Введем значения в полях для города и страны, после чего щелкнем на кнопке Submit, чтобы отправить форму серверу.

Процесс привязки моделей найдет и обработает индексируемые значения данных и использует их для создания коллекции объектов `AddressSummary`, предоставляемой методу действия, который затем применяет удобный метод `View()`, чтобы передать их обратно представлению с целью отображения (рис. 26.9).

Указание источника данных привязки моделей

Как объяснялось в начале главы, стандартный процесс привязки моделей ищет значения в трех местах: значения данных формы, переменные маршрутизации и строки запросов.

Стандартная последовательность поиска не всегда полезна либо потому, что данные должны всегда поступать из специфической части запроса, либо потому, что нужно использовать источник данных, в котором поиск по умолчанию не производится. Средство привязки моделей включает набор атрибутов, которые применяются для переопределения стандартного поведения поиска и описаны в табл. 26.3.

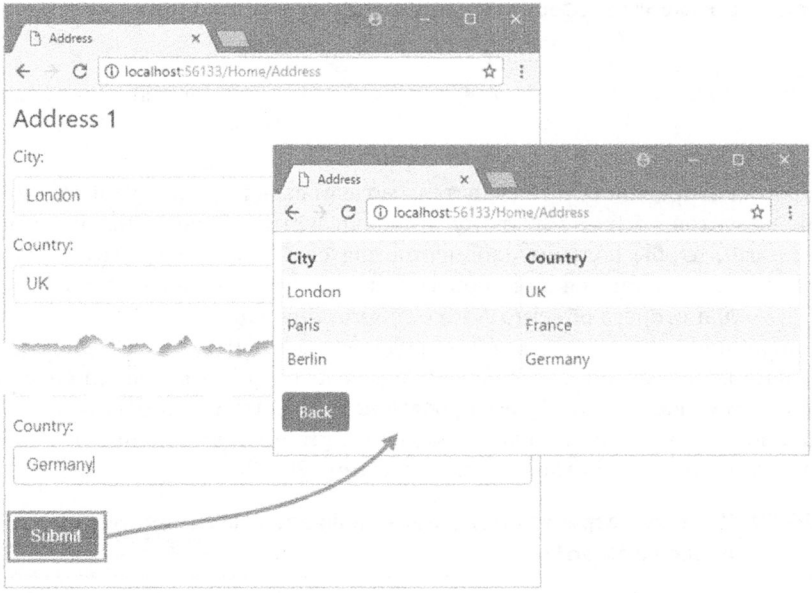


Рис. 26.9. Привязка коллекций специальных объектов

Таблица 26.3. Атрибуты для источников данных привязки

Имя	Описание
`FromForm`	Этот атрибут используется для выбора данных формы в качестве источника данных привязки. По умолчанию для нахождения значения данных формы применяется имя параметра, но такое поведение можно изменить с использованием свойства `Name`, которое позволяет указывать другое имя
`FromRoute`	Этот атрибут применяется для выбора системы маршрутизации в качестве источника данных привязки. По умолчанию для нахождения значения данных маршрута используется имя параметра, но такое поведение можно изменить с применением свойства `Name`, которое позволяет указывать другое имя
`FromQuery`	Этот атрибут используется для выбора строки запроса в качестве источника данных привязки. По умолчанию для нахождения значения строки запроса применяется имя параметра, но такое поведение можно изменить с использованием свойства `Name`, которое позволяет указывать другой ключ строки запроса
`FromHeader`	Этот атрибут применяется для выбора заголовка запроса в качестве источника данных привязки. По умолчанию имя параметра используется как имя заголовка, но поведение можно изменить с применением свойства `Name`, которое позволяет указывать другое имя заголовка
`FromBody`	Этот атрибут используется для указания, что в качестве источника данных привязки должно применяться тело запроса. Требуется в ситуациях, когда из запросов необходимо получать данные, не закодированные как данные формы, что происходит в контроллерах API

Выбор стандартного источника данных привязки

Атрибуты FromForm, FromRoute и FromQuery позволяют указывать, что данные привязки модели будут получаться из стандартного местоположения, но без нормальной последовательности поиска. Ранее в главе использовался следующий URL:

```
/Home/Index/3?id=1
```

Приведенный URL содержит два возможных значения, которые могут применяться для параметра id метода действия Index() в контроллере Home. Система маршрутизации назначит последний сегмент URL переменной по имени id, которая определена в шаблоне URL внутри класса Startup. Строка запроса также содержит значение id. Стандартный шаблон поиска означает, что значения привязки модели будут извлекаться из данных маршрутизации, а строка запроса игнорируется.

Чтобы изменить такое поведение, в листинге 26.30 к методу действия применяется атрибут FromQuery. Ради простоты также удалены все остальные методы действий, которые были определены в предшествующих примерах.

Листинг 26.30. Выбор строки запроса для привязки модели в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public IActionResult Index([FromQuery] int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }
  }
}
```

Здесь к параметру id применяется атрибут FromQuery, т.е. во время поиска процессом привязки моделей значения данных для id будет использоваться только строка запроса.

Совет. При указании источника данных привязки моделей, такого как строка запроса, по-прежнему можно осуществлять привязку сложных типов. Для каждого простого свойства в типе параметра процесс привязки моделей будет искать ключ строки запроса с тем же самым именем.

Использование заголовков в качестве источников данных привязки

Атрибут FromHeader позволяет задействовать заголовки HTTP-запроса как источники данных привязки. В листинге 26.31 в контроллер Home добавлен простой метод действия, который получает привязку параметра с применением данных из стандартного заголовка HTTP-запроса.

Листинг 26.31. Привязка модели из заголовка HTTP-запроса в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public IActionResult Index([FromQuery] int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }

    public string Header([FromHeader]string accept) => $"Header: {accept}";
  }
}
```

Метод действия Header() определяет параметр accept, значение которого будет браться из заголовка Accept в текущем запросе и возвращаться в качестве результата метода. После запуска приложения и запрашивания URL вида /Home/Header будет получен примерно такой результат (точный результат может отличаться в зависимости от используемого браузера):

```
Header: text/html,application/xhtml+xml,application/xml;q=0.9,
 image/webp,*/*;q=0.8
```

Не все имена HTTP-заголовков могут легко выбираться, полагаясь на имя параметра метода действия, потому что система привязки моделей не преобразует соглашения по именованию C# в соглашения, принятые в HTTP-заголовках. В таких ситуациях конфигурировать атрибут FromHeader необходимо с применением свойства Name с целью указания имени заголовка (листинг 26.32).

Листинг 26.32. Указание имени заголовка в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
```

```
  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public IActionResult Index([FromQuery] int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }

    public string Header([FromHeader(Name = "Accept-Language")] string accept)
      => $"Header: {accept}";
  }
}
```

Использовать `Accept-Language` для имени параметра C# нельзя, и связыватель модели не будет автоматически преобразовывать имя вроде `AcceptLanguage` в `Accept-Language`, чтобы оно соответствовало заголовку. Взамен с помощью свойства `Name` атрибут конфигурируется для соответствия правильному заголовку. В результате запуска приложения и запрашивания URL вида `/Home/Header` будет получен ответ, подобный показанному ниже, который будет варьироваться на основе региональных настроек:

```
Header: en-US,en;q=0.8
```

Привязка сложных типов из заголовков

Несмотря на то что подобное требуется редко, привязку сложных типов можно выполнять, используя значения заголовков, за счет применения атрибута `FromHeader` к свойствам класса модели. Добавим в папку `Models` файл по имени `HeaderModel.cs` и определим в нем класс, представленный в листинге 26.33.

Листинг 26.33. Содержимое файла `HeaderModel.cs` из папки `Models`

```
using Microsoft.AspNetCore.Mvc;

namespace MvcModels.Models {

  public class HeaderModel {

    [FromHeader]
    public string Accept { get; set; }

    [FromHeader(Name = "Accept-Language")]
    public string AcceptLanguage { get; set; }

    [FromHeader(Name = "Accept-Encoding")]
    public string AcceptEncoding { get; set; }
  }
}
```

В классе HeaderModel определены три свойства, каждое из которых декорировано атрибутом FromHeader. В двух атрибутах с помощью свойства Name указываются имена заголовков, которые невозможно выразить как имена параметров C#. В листинге 26.34 приведен обновленный метод действия Header() контроллера Home, теперь получающий объект HeaderModel.

Листинг 26.34. Использование класса HeaderModel в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public IActionResult Index([FromQuery] int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }

    public ViewResult Header(HeaderModel model) => View(model);
  }
}
```

Чтобы завершить пример, добавим в папку Views/Home файл представления по имени Header.cshtml с разметкой из листинга 26.35.

Листинг 26.35. Содержимое файла Header.cshtml из папки Views/Home

```
@model HeaderModel
@{
  ViewBag.Title = "Headers";
  Layout = "_Layout";
}

<table class="table table-sm table-bordered table-striped">
  <tr><th>Accept:</th><td>@Model.Accept</td></tr>
  <tr><th>Accept-Encoding:</th><td>@Model.AcceptEncoding</td></tr>
  <tr><th>Accept-Language:</th><td>@Model.AcceptLanguage</td></tr>
</table>
```

Процесс привязки моделей будет исследовать свойства сложных типов в поиске атрибутов, описанных в табл. 26.3. Это дает возможность посредством атрибута FromHeader определить сложный тип, свойства которого являются моделью, привязываемой из заголовков, в чем легко удостовериться, запустив приложение и запросив URL вида /Home/Header (рис. 26.10).

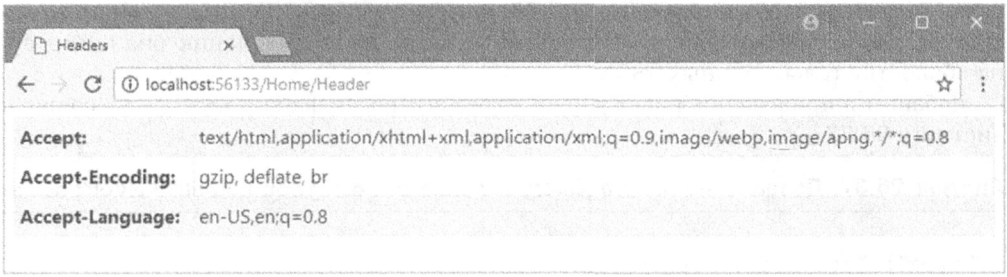


Рис. 26.10. Привязка сложного типа из заголовков запроса

Использование тел запросов в качестве источников данных привязки

Не все данные, отправляемые клиентами, посылаются в виде данных формы, как происходит в ситуации, когда клиент JavaScript отправляет данные JSON контроллеру API. Атрибут `FromBody` указывает, что тело запроса должно быть декодировано и применяться в качестве источника данных привязки модели. В листинге 26.36 показаны новые методы действий `Body()`, которые демонстрируют работу атрибута `FromBody`.

Листинг 26.36. Добавление методов действий в файле `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) {
      repository = repo;
    }

    public IActionResult Index([FromQuery] int? id) {
      Person person;
      if (id.HasValue && (person = repository[id.Value]) != null) {
        return View(person);
      } else {
        return NotFound();
      }
    }
    public ViewResult Header(HeaderModel model) => View(model);
    public ViewResult Body() => View();
    [HttpPost]
    public Person Body([FromBody]Person model) => model;
  }
}
```

Параметр метода `Body()`, принимающего запросы `POST`, декорирован атрибутом `FromBody`, т.е. содержимое тела запроса будет декодировано и использовано для при-

вязки модели. Как объяснялось в главе 20, инфраструктура MVC имеет расширяемую систему для работы с разными форматами данных, но по умолчанию она настроена на обработку только данных JSON.

Добавим в приложение пакет jQuery, отредактировав файл `bower.json` согласно листингу 26.37.

Листинг 26.37. Добавление пакета jQuery в файле `bower.json` из папки `MvcModels`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

Чтобы снабдить метод действия требуемыми данными, добавим в папку `Views/Home` файл по имени `Body.cshtml` и поместим в него содержимое, приведенное в листинге 26.38.

Листинг 26.38. Содержимое файла `Body.cshtml` из папки `Views/Home`

```
@{
  ViewBag.Title = "Address";
  Layout = "_Layout";
}
@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script type="text/javascript">
    $(document).ready(function () {
      $("button").click(function (e) {
        $.ajax("/Home/Body", {
          method: "post",
          contentType: "application/json",
          data: JSON.stringify({
            firstName: "Bob",
            lastName: "Smith"
          }),
          success: function (data) {
            $("#firstName").text(data.firstName);
            $("#lastName").text(data.lastName);
          }
        });
      });
    });
  </script>
}
<table class="table table-sm table-bordered table-striped">
  <tr><th>First Name:</th><td id="firstName"></td></tr>
  <tr><th>Last Name:</th><td id="lastName"></td></tr>
</table>
<button class="btn btn-primary">Submit</button>
```

Для простоты представление содержит встраиваемый код JavaScript, который с помощью jQuery посылает HTTP-запрос `POST`, содержащий данные JSON, на URL вида `/Home/Body`, когда производится щелчок на элементе `button`. Сервер кодирует объект, созданный с применением привязки модели, и отправляет его обратно клиенту как закодированные данные JSON. Запустив приложение, запросив URL вида `/Home/Body` и щелкнув на кнопке Submit, можно просмотреть результат (рис. 26.11).

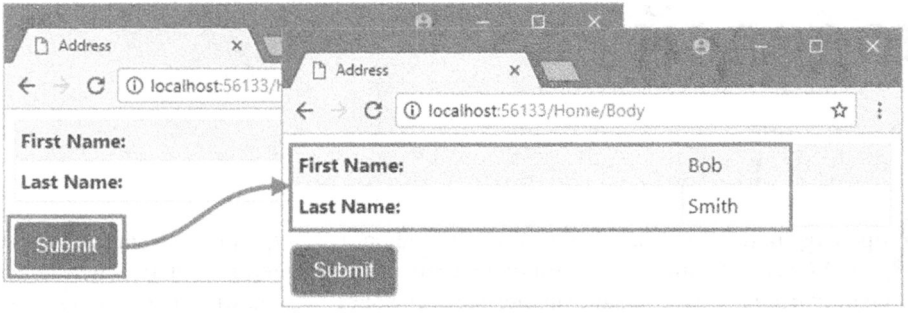


Рис. 26.11. Использование тела запроса для привязки моделей

Совет. Не весь клиентский код JavaScript требует использования атрибута `FromBody`. В рассмотренном примере намеренно не применялся удобный метод jQuery для отправки Ajax-запросов `POST`, потому что он кодирует информацию как данные формы. Взамен использовался другой метод, который позволил послать данные JSON.

Атрибут `FromBody` можно применять для привязки только одного параметра метода действия, и в случае использования этого атрибута более одного раза в отдельно взятом методе генерируется исключение. Если нужно получить несколько объектов модели из тела запроса, тогда придется создать простой класс передачи данных, который имеет все необходимые свойства и применяет содержащиеся в нем данные для создания объектов, требуемых внутри метода действия.

Резюме

В главе рассматривался процесс привязки моделей, который позволяет предоставлять методам действий необходимые аргументы, используя значения данных из обрабатываемого HTTP-запроса. Было показано, как привязывать простые и сложные типы, каким образом иметь дело с массивами и коллекциями, а также как управлять процессом привязки моделей за счет применения атрибутов к параметрам методов действий или свойствам классов моделей. В следующей главе будет описано средство проверки достоверности моделей.

ГЛАВА 27

Проверка достоверности моделей

В предыдущей главе было показано, как инфраструктура MVC создает объекты моделей из HTTP-запросов с помощью процесса привязки моделей. В ней повсеместно предполагалось, что предоставляемые пользователем данные были допустимыми. Реальность же такова, что пользователи часто вводят данные, которые не являются допустимыми и не могут использоваться, а потому темой настоящей главы будет проверка достоверности моделей.

Проверка достоверности моделей представляет собой процесс, который обеспечивает пригодность данных, получаемых приложением, для привязки моделей. Если данные не подходят, то пользователям предоставляется полезная информация, которая помогает устранить проблему.

Первая часть процесса — проверка полученных данных — является одним из основных способов предохранить целостность модели предметной области. Отклонение данных, которые не имеют смысла в контексте предметной области, может предотвратить возникновение странных и нежелательных состояний в приложении. Вторая часть — помощь пользователям в корректировке проблемы — важна в равной степени. Без такой информации и обратной связи, необходимой для взаимодействия с приложением, пользователи будут недовольны и озадачены. В общедоступных приложениях это означает, что пользователи просто прекратят ими пользоваться. В корпоративных приложениях это означает нарушение рабочего потока пользователей. Ни тот, ни другой исход не является желательным, но к счастью инфраструктура MVC предоставляет широкую поддержку проверки достоверности моделей.

В табл. 27.1 приведена сводка, позволяющая поместить проверку достоверности моделей в контекст.

В табл. 27.2 приведена сводка по главе.

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `ModelValidation` с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core. Чтобы включить инфраструктуру MVC и другие компоненты промежуточного ПО, внесем в класс `Startup` изменения, показанные в листинге 27.1.

Таблица 27.1. Помещение проверки достоверности моделей в контекст

Вопрос	Ответ
Что это такое?	Проверка достоверности моделей — это процесс выяснения, что данные, предоставленные в запросе, пригодны для применения в приложении
Чем она полезна?	Пользователи не всегда вводят допустимые данные. Использование таких данных в приложении может приводить к неожиданным и нежелательным последствиям
Как она используются?	Контроллеры исследуют результат процесса проверки достоверности, а посредством вспомогательных функций дескрипторов в представления, отображаемые пользователям, включается отклик проверки. Проверка достоверности автоматически выполняется во время процесса привязки моделей. Она обычно дополняется специальной проверкой в классе контроллера или за счет применения атрибутов проверки достоверности
Существуют ли какие-то скрытые ловушки или ограничения?	Важно протестировать эффективность кода проверки достоверности, убедившись в том, что он способен справляться с полным диапазоном значений, которые может получать приложение
Существуют ли альтернативы?	Нет, проверка достоверности моделей тесно интегрирована в ASP.NET Core MVC

Таблица 27.2. Сводка по главе

Задача	Решение	Листинг
Явная проверка достоверности модели	Используйте объект `ModelState` для регистрации ошибок проверки достоверности	27.9, 27.10
Генерация сводки по ошибкам проверки достоверности	Примените атрибут `asp-validation-summary` к элементу `div`	27.11
Изменение стандартных сообщений привязки моделей	Переопределите функции сообщений в поставщике сообщений привязки моделей	27.12
Генерация ошибок проверки достоверности на уровне свойств	Примените атрибут `asp-validation-for` к элементу `span`	27.13
Генерация ошибок проверки достоверности на уровне модели	Используйте объект `ModelState` для регистрации ошибок проверки достоверности, которые не ассоциированы со специфическими свойствами, и укажите значение `ModelOnly` для атрибута `asp-validation-summary` в элементе `div`	27.14, 27.15
Определение самопроверяемой модели	Примените атрибуты проверки достоверности данных к свойствам модели	27.16, 27.17
Создание специального атрибута проверки достоверности	Реализуйте интерфейс `IModelValidator`	27.18, 27.19
Выполнение проверки достоверности на стороне клиента	Используйте пакеты ненавязчивой проверки достоверности jQuery	27.20, 27.21
Выполнение удаленной проверки достоверности	Определите метод действия для выполнения проверки достоверности и примените атрибут `Remote` к свойству модели	27.22, 27.23

Листинг 27.1. Содержимое файла `Startup.cs` из папки `ModelValidation`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ModelValidation {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание модели

Создадим папку `Models` и добавим в нее файл класса по имени `Appointment.cs` с определением, приведенным в листинге 27.2.

Листинг 27.2. Содержимое файла `Appointment.cs` из папки `Models`

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {
  public class Appointment {

    public string ClientName { get; set; }

    [UIHint("Date")]
    public DateTime Date { get; set; }

    public bool TermsAccepted { get; set; }
  }
}
```

В классе `Appointment` определены три свойства и с помощью атрибута `UIHint` указано, что свойство `Date` должно выражаться как дата без компонента времени.

Создание контроллера

Создадим папку `Controllers`, добавим файл класса по имени `HomeController.cs` и поместим в него определение контроллера из листинга 27.3, который оперирует с классом модели `Appointment`.

Листинг 27.3. Содержимое файла HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;

namespace ModelValidation.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() =>
      View("MakeBooking", new Appointment { Date = DateTime.Now });

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) =>
      View("Completed", appt);
  }
}
```

Действие Index визуализирует представление MakeBooking с новым объектом Appointment в качестве модели представления. Метод действия MakeBooking() более интересен, т.к. в нем будет выполняться проверка достоверности модели.

На заметку! Пример приложения настолько прост, что в нем даже не определено хранилище и не добавлен код для сохранения объектов Appointment, которые выпускаются процессом привязки моделей. Тем не менее, важно иметь в виду, что главной причиной проверки достоверности модели является предотвращение попадания в хранилище неправильных или бессмысленных данных с последующим возникновением проблем (либо при попытке сохранения данных, либо при попытке их обработки в более позднее время).

Создание компоновки и представлений

Для ряда примеров, приводимых в главе, требуется простая компоновка. Создадим папку Views/Shared и добавим в нее файл _Layout.cshtml, содержимое которого показано в листинге 27.4.

Листинг 27.4. Содержимое файла _Layout.cshtml из папки Views/Shared

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>Model Validation</title>
  <link asp-href-include=
    "/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
  @RenderSection("scripts", false)
</head>
<body class="m-1 p-1">
  @RenderBody()
</body>
</html>
```

Чтобы снабдить методы действий представлениями, создадим папку `Views/Home` и добавим в нее файл по имени `MakeBooking.cshtml` с разметкой из листинга 27.5.

Листинг 27.5. Содержимое файла `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Когда форма, содержащаяся в файле `Index.cshtml`, отправляется обратно приложению, метод действия `MakeBooking()` отображает детали созданной пользователем встречи с применением представления `Completed.cshtml` из папки `Views/Home` (листинг 27.6).

Листинг 27.6. Содержимое файла `Completed.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

<div class="bg-success m-1 p-1 text-white"><h2>Your Appointment</h2>
  </div>

<table class="table table-bordered">
  <tr>
    <th>Your name is:</th>
    <td>@Model.ClientName</td>
  </tr>
  <tr>
    <th>Your appontment date is:</th>
    <td>@Model.Date.ToString("d")</td>
  </tr>
</table>
<a class="btn btn-success" asp-action="Index">Make Another Appointment</a>
```

При стилизации HTML-элементов представления полагаются на CSS-пакет Bootstrap. Создадим в корневой папке проекта файл `bower.json` с использованием шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел `dependencies` (листинг 27.7). Добавим также пакет jQuery, который понадобится позже в главе.

Листинг 27.7. Добавление пакета Bootstrap в файле `bower.json` из папки `ModelValidation`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

Последний подготовительный шаг связан с созданием файла `_ViewImports.cshtml` в папке `Views`, в котором настраиваются встроенные вспомогательные функции дескрипторов для применения в представлениях Razor и импортируется пространство имен модели (листинг 27.8).

Листинг 27.8. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@using ModelValidation.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Как вы уже наверняка догадались, пример основан на создании встреч. Взглянуть на него в работе можно, запустив приложение и запросив стандартный URL. Ввод в элементах формы информации о встрече и щелчок на кнопке Make Booking (Создать встречу) приведет к отправке данных серверу, который выполнит процесс привязки моделей для создания объекта `Appointment`, детали которого затем визуализируются с использованием представления `Completed.cshtml` (рис. 27.1).

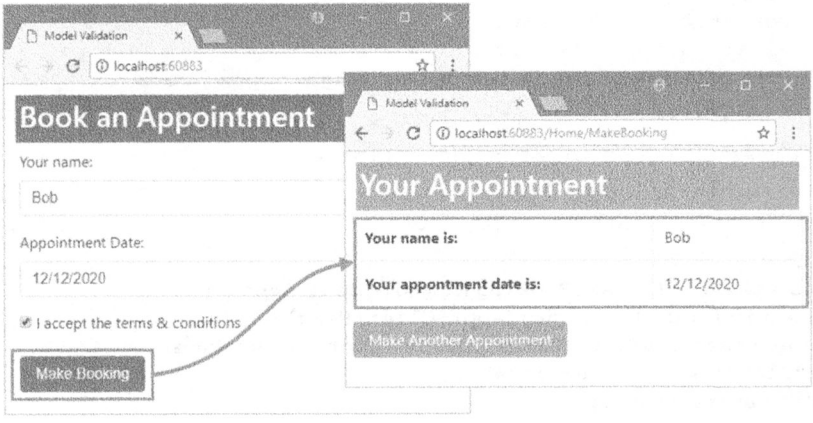


Рис. 27.1. Выполнение примера приложения

Необходимость в проверке достоверности модели

Проверка достоверности модели — это процесс принудительного применения требований, которые приложение налагает на данные, получаемые от клиентов. Без проверки достоверности приложение будет пытаться оперировать с любыми полученными данными, что может привести к генерации исключений и непредвиденному поведению, которое отразит немедленные или долговременные проблемы, постепенно накапливаемые по мере того, как хранилище наполняется неправильными, незавершенными или злонамеренными данными.

В настоящее время пример приложения будет принимать любые данные, которые отправляет пользователь. Чтобы предохранить целостность приложения и модели предметной области, должны быть удовлетворены три перечисленных ниже условия, прежде чем станет известно, что пользователь предоставляет приемлемый объект `Appointment`:

- пользователь должен предоставить имя;
- пользователь должен предоставить дату, относящуюся к будущему;
- пользователь должен отметить флажок для принятия условий.

В последующих разделах демонстрируется использование проверки достоверности моделей для навязывания указанных требований за счет контроля получаемых приложением данных и предоставления пользователям обратной связи, когда приложение не может работать с данными, которые они отправили.

Явная проверка достоверности модели

Самый прямой способ проверки достоверности модели предусматривает ее выполнение в методе действия. В листинге 27.9 показан новый метод действия `MakeBooking()` с добавленными явными проверками для каждого свойства, определяемого классом `Appointment`.

Листинг 27.9. Явная проверка достоверности модели в файле `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() =>
      View("MakeBooking", new Appointment { Date = DateTime.Now });

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {
      if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError(nameof(appt.ClientName),
          "Please enter your name");
        // Введите свое имя
      }
```

```
    if (ModelState.GetValidationState("Date")
        == ModelValidationState.Valid && DateTime.Now > appt.Date) {
      ModelState.AddModelError(nameof(appt.Date),
        "Please enter a date in the future");
       // Введите дату, относящуюся к будущему
    }
    if (!appt.TermsAccepted) {
      ModelState.AddModelError(nameof(appt.TermsAccepted),
        "You must accept the terms");
       // Вы должны принять условия
    }
    if (ModelState.IsValid) {
      return View("Completed", appt);
    } else {
      return View();
    }
  }
}
```

В коде проверяются значения, которые связыватель модели присвоил свойствам объекта параметра. Любые обнаруженные ошибки регистрируются с применением объекта `ModelStateDictionary`, который возвращается свойством `ModelState`, унаследованным из базового класса `Controller`.

Как подсказывает его имя, класс `ModelStateDictionary` — это словарь, который используется для отслеживания деталей состояния объекта модели, с акцентом на ошибках проверки достоверности. В табл. 27.3 описаны наиболее важные члены класса `ModelStateDictionary`.

Таблица 27.3. Избранные члены класса `ModelStateDictionary`

Имя	Описание
`AddModelError(property, message)`	Этот метод применяется для регистрации ошибки проверки достоверности модели, связанной с указанным свойством
`GetValidationState(property)`	Этот метод используется при выяснении, есть ли ошибки проверки достоверности модели для специфического свойства; результат выражается как значение перечисления `ModelValidationState`
`IsValid`	Это свойство возвращает `true`, если все свойства модели допустимы, и `false` в противном случае

В качестве примера применения `ModelStateDictionary` ниже демонстрируется выполнение проверки достоверности свойства `ClientName`:

```
...
if (string.IsNullOrEmpty(appt.ClientName)) {
  ModelState.AddModelError(nameof(appt.ClientName),
  "Please enter your name");
}
...
```

Одна из целей приведенного примера связана с гарантированием того, что пользователь предоставляет значение для данного свойства, поэтому здесь используется статический метод `string.IsNullOrEmpty()` для проверки значения свойства, которое процесс привязки моделей извлек из запроса. Если свойство `ClientName` равно `null` или пустой строке, тогда известно, что цель проверки достоверности не была достигнута. В таком случае посредством метода `ModelState.AddModelError()` регистрируется ошибка проверки достоверности с указанием имени свойства (`ClientName`) и сообщения, которое будет отображаться пользователю для объяснения природы проблемы (`Please enter your name` (Введите свое имя)).

Система привязки моделей также применяет объект `ModelStateDictionary` для регистрации любых проблем с нахождением и присваиванием значений свойствам модели. Метод `GetValidationState()` используется с целью выяснения, зарегистрированы ли для свойства модели какие-то ошибки процессом привязки моделей или вызовом `AddModelError()` во время явной проверки достоверности в методе действия. Метод `GetValidationState()` возвращает значение перечисления `ModelValidationState`, которое описано в табл. 27.4.

Таблица 27.4. Значения перечисления `ModelValidationState`

Имя	Описание
`Unvalidated`	Указывает, что в отношении свойства модели проверка достоверности не выполнялась, обычно из-за отсутствия в запросе значения, которое бы соответствовало имени свойства
`Valid`	Указывает, что значение в запросе, ассоциированное со свойством, является допустимым
`Invalid`	Указывает, что значение в запросе, ассоциированное со свойством, является недопустимым и применяться не должно
`Skipped`	Указывает, что свойство модели не было обработано. Обычно это говорит о наличии настолько большого количества ошибок проверки достоверности, что продолжать проверку не имеет смысла

Для свойства `Date` выполняется проверка, сообщил ли процесс привязки моделей о проблеме во время преобразования отправленного браузером значения в объект `DateTime`:

```
...
if (ModelState.GetValidationState("Date") == ModelValidationState.Valid
    && DateTime.Now > appt.Date) {
  ModelState.AddModelError(nameof(appt.Date),
  "Please enter a date in the future");
}
...
```

Цель проверки достоверности для свойства `Date` — обеспечить предоставление пользователем допустимой даты в будущем. Метод `GetValidationState()` используется для выяснения, смог ли процесс привязки моделей преобразовать значение из запроса в объект `DateTime`, за счет проверки на предмет равенства `ModelValidationState.Valid`. В случае допустимой даты выполняется проверка, относится ли она к будущему, и если нет, то посредством метода `AddModelError()` регистрируется проблема проверки достоверности.

После того, как все свойства объекта модели проверены, с помощью свойства ModelState.IsValid выясняется, возникали ли ошибки. Данное свойство возвращает true, если во время проверок вызывался метод Model.State.AddModelError() или у связывателя модели возникали проблемы с созданием объекта Appointment:

```
...
if (ModelState.IsValid) {
  return View("Completed", appt);
} else {
  return View();
}
...
```

Объект Appointment является допустимым, если свойство IsValid возвращает true, в случае чего метод действия визуализирует представление Completed.cshtml. Если же свойство IsValue возвращает false, тогда есть проблема проверки достоверности, которая обрабатывается путем вызова метода View() для визуализации стандартного представления.

Отображение пользователю ошибок проверки достоверности

Иметь дело с ошибкой проверки достоверности, вызывая метода View(), может показаться странным подходом, но данные контекста, которыми инфраструктура MVC снабжает представление, содержат детали ошибок, возникших при проверке достоверности модели. Такие детали автоматически обнаруживаются и применяются вспомогательной функцией дескриптора, которая используется для трансформации элементов input.

Чтобы посмотреть, как все работает, запустим приложение и щелкнем на кнопке Make Booking, не заполняя форму данными о встрече. Визуальных изменений в окне браузера не будет, но если просмотреть HTML-разметку, которую MVC возвращает из запроса POST, то можно заметить, что изменился атрибут class элемента формы. Вот как выглядел элемент ClientName до отправки формы:

```
<input class="form-control" type="text" id="ClientName"
       name="ClientName" value="">
```

А после отправки пустой формы элемент ClientName принимает следующий вид:

```
<input class="form-control input-validation-error"
       type="text" id="ClientName" name="ClientName" value="">
```

Вспомогательная функция дескриптора добавляет элементы, чьи значения не прошли проверку достоверности, в класс input-validation-error, который затем можно стилизовать с целью выделения проблемы.

Делать это можно за счет определения специальных стилей CSS в таблице стилей, но если нужно задействовать встроенные стили для проверки, предоставляемые CSS-библиотеками вроде Bootstrap, то придется проделать небольшую дополнительную работу. Поскольку имя класса, добавляемое к элементам формы, изменять нельзя, потребуется код JavaScript для сопоставления имени, применяемого инфраструктурой MVC, и классами ошибок CSS, предлагаемыми библиотекой Bootstrap.

Совет. Использовать код JavaScript подобного рода может быть неудобно, из-за чего возникает соблазн применять специальные стили CSS даже при работе с CSS-библиотеками типа Bootstrap. Однако цвета, используемые классами проверки достоверности в Bootstrap, можно переопределять посредством тем или путем настройки пакета и определения собственных стилей, т.е. вам придется обеспечить соответствие между любыми изменениями в теме и изменениями в любых специальных стилях, которые вы определили. В идеальном случае разработчики из Microsoft сделают имена классов проверки достоверности конфигурируемыми в будущем выпуске ASP.NET Core MVC, но до тех пор написание кода JavaScript для применения стилей Bootstrap является более надежным подходом, чем создание специальных таблиц стилей.

В листинге 27.10 к представлению `MakeBooking` добавляется код jQuery для поиска элементов в классе `input-validation-error`, нахождения ближайшего родительского элемента, который был назначен классу `form-group`, и добавления этого элемента в класс `has-danger` (используемый библиотекой Bootstrap для установки цвета ошибки в элементах формы).

Листинг 27.10. Назначение элементов классам проверки достоверности в файле `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script type="text/javascript">
    $(document).ready(function () {
      $("input.input-validation-error")
        .closest(".form-group").addClass("has-danger");
    });
  </script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Добавленный код jQuery выполняется, когда браузер завершает разбор всех элементов в HTML-документе, и результатом будет выделение элементов input, которые назначены классу input-validaton-error. Запустив приложение и отправив форму без заполнения всех полей, можно получить результат, изображенный на рис. 27.2.

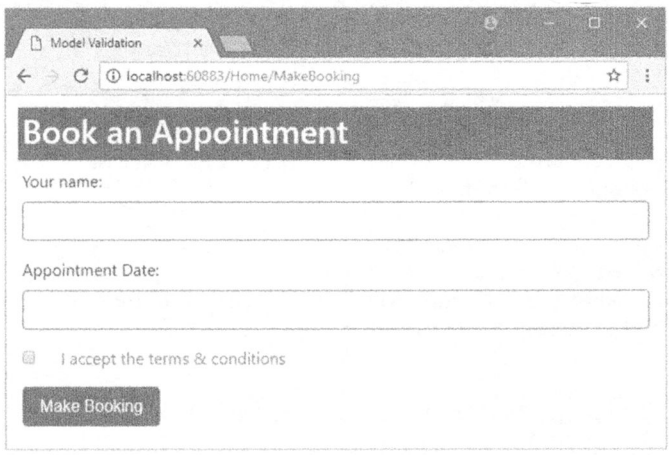


Рис. 27.2. Выделение ошибок проверки достоверности

Если форма отправляется без ввода каких-либо данных, то все три свойства выделяются как содержащие ошибки. Пользователь не увидит представление Completed.cshtml до тех пор, пока форма не будет отправлена с данными, которые могут быть разобраны связывателем модели и успешно проходят явные проверки достоверности в методе MakeBooking(). Пока подобное не произойдет, отправка формы будет приводить к визуализации представления MakeBooking.cshtml с текущими ошибками проверки достоверности.

Отображение сообщений об ошибках проверки достоверности

Классы CSS, которые вспомогательные функции дескрипторов применяют к элементам input, указывают на наличие проблемы с полем формы, но не сообщают пользователю, в чем конкретно состоит проблема. Предоставление пользователю дополнительной информации требует использования другой вспомогательной функции дескриптора, которая добавляет в представление сводку по проблемам (листинг 27.11).

Листинг 27.11. Отображение сводки по проверке достоверности в файле MakeBooking.cshtml из папки Views/Home

```
@model Appointment
@{ Layout = "_Layout"; }
@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script type="text/javascript">
    $(document).ready(function () {
      $("input.input-validation-error")
        .closest(".form-group").addClass("has-danger");
    });
  </script>
}
```

```
<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>
<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div asp-validation-summary="All" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Класс `ValidationSummaryTagHelper` обнаруживает атрибут `asp-validation-summary` в элементах `div` и реагирует добавлением сообщений, которые описывают любые ошибки проверки достоверности, выявленные методом действия. В атрибуте `asp-validation-summary` указывается одно из значений перечисления `ValidationSummary`, которые описаны в табл. 27.5 и вскоре будут продемонстрированы в работе.

Таблица 27.5. Значения перечисления `ValidationSummary`

Имя	Описание
`All`	Применяется для отображения всех ошибок проверки достоверности, которые были зарегистрированы
`ModelOnly`	Используется для отображения только ошибок проверки достоверности, относящихся ко всей модели, за исключением тех, которые были зарегистрированы для индивидуальных свойств, как описано в разделе "Отображение сообщений об ошибках проверки достоверности на уровне модели" далее в главе
`None`	Применяется для отключения вспомогательной функции дескриптора, так что она не будет трансформировать HTML-элемент

Если запустить приложение и отправить форму, не внося какие-либо изменения, то появится сводка, которую сгенерировала вспомогательная функция дескриптора. Цвет текста в рассматриваемом примере определяется классом `text-danger` из Bootstrap, который гарантирует, что цвет текста соответствует цвету, используемому для выделения текстовых полей (рис. 27.3).

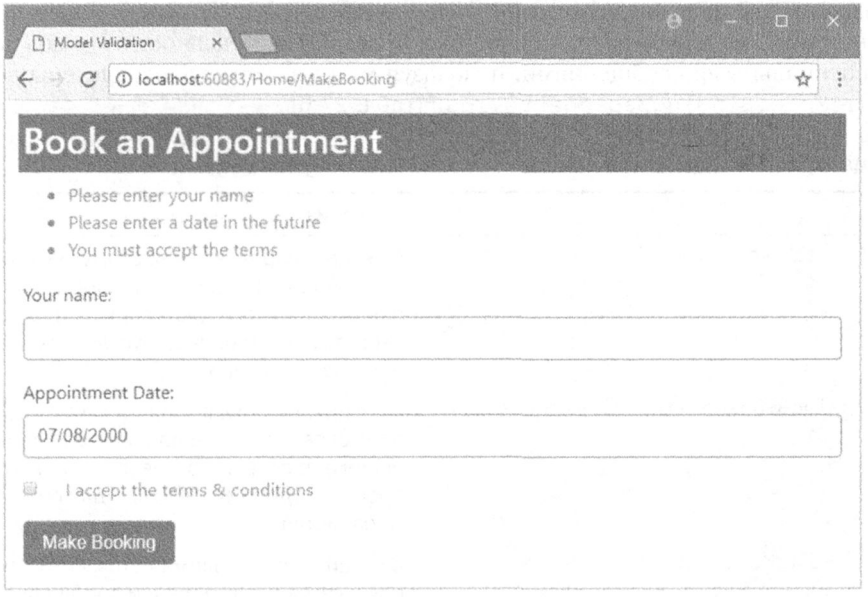


Рис. 27.3. Отображение пользователю сводки по проверке достоверности

Заглянув в HTML-разметку, которая была получена браузером, легко заметить, что сообщения проверки достоверности отправлялись в виде списка:

```
<div class="text-danger validation-summary-errors"
     data-valmsg-summary="true">
  <ul>
    <li>Please enter your name</li>
    <li>Please enter a date in the future</li>
    <li>You must accept the terms</li>
  </ul>
</div>
```

Конфигурирование стандартных сообщений об ошибках проверки достоверности

Процесс привязки моделей, описанный в главе 26, выполняет собственную проверку достоверности, когда он пытается предоставить значения данных, требующиеся для вызова метода действия. Чтобы увидеть, как это работает, запустим приложение, очистим содержимое поля Appointment Date (Дата встречи) и отправим форму. Вы обнаружите, что одно из отображаемых сообщений проверки достоверности изменилось, и нового сообщения нет среди строк, передаваемых методу `AddModelError()` внутри метода действия:

```
The value '' is invalid
```

Показанное сообщение добавляется в `ModelStateDictionary` процессом привязки моделей, когда он не может найти значение для свойства или находит значение, но не в состоянии его преобразовать. В данном случае ошибка возникла из-за отправки в данных формы пустой строки, которая не может быть преобразована в объект `DateTime` для свойства `Date` класса `Appointment`.

Связыватель модели располагает набором предопределенных сообщений, которые он применяет для обозначения ошибок проверки достоверности. Их можно заменить специальными сообщениями, используя методы, которые определены в классе `DefaultModelBindingMessageProvider` (табл. 27.6).

Таблица 27.6. Методы класса `DefaultModelBindingMessageProvider`

Имя	Описание
`SetValueMustNotBeNullAccessor()`	Функция, передаваемая этому методу, используется для генерации сообщения об ошибке проверки достоверности, когда свойству модели, не допускающему `null`, поступает значение `null`
`SetMissingBindRequiredValueAccessor()`	Функция, передаваемая этому методу, применяется для генерации сообщения об ошибке проверки достоверности, когда запрос не содержит значение для обязательного свойства
`SetMissingKeyOrValueAccessor()`	Функция, передаваемая этому методу, используется для генерации сообщения об ошибке проверки достоверности, когда данные, требующиеся объекту словаря, содержат ключи или значения, равные `null`
`SetAttemptedValueIsInvalidAccessor()`	Функция, передаваемая этому методу, применяется для генерации сообщения об ошибке проверки достоверности, когда системе привязки моделей не удается преобразовать значение данных в требуемый тип C#
`SetUnknownValueIsInvalidAccessor()`	Функция, передаваемая этому методу, используется для генерации сообщения об ошибке проверки достоверности, когда системе привязки моделей не удается преобразовать значение данных в требуемый тип C#
`SetValueMustBeANumberAccessor()`	Функция, передаваемая этому методу, применяется для генерации сообщения об ошибке проверки достоверности, когда значение данных не может быть преобразовано в числовой тип C#
`SetValueIsInvalidAccessor()`	Функция, передаваемая этому методу, используется для генерации запасного сообщения об ошибке проверки достоверности, которое применяется в качестве последнего средства

Каждый метод, описанный в табл. 27.6, получает функцию, которая вызывается для получения сообщения об ошибке, подлежащего отображению. Эти методы используются в классе `Startup` для конфигурирования приложения, как показано в листинге 27.12, где производится замена стандартного сообщения, которое отображается в случае значения `null`.

Листинг 27.12. Замена функции генерации сообщений при привязке моделей в файле `Startup.cs` из папки `ModelValidation`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ModelValidation {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc().AddMvcOptions(opts =>
        opts.ModelBindingMessageProvider
            .SetValueMustNotBeNullAccessor(value => "Please enter a value")
      );
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Указанная функция получает значение, предоставленное пользователем, хотя это не особенно удобно, когда приходится иметь дело со значениями `null`. Чтобы увидеть специальное сообщение, понадобится снова запустить приложение и отправить форму, очистив поле Appointment Date (рис. 27.4).



Рис. 27.4. Изменение сообщений об ошибках привязки моделей

Отображение сообщений об ошибках проверки достоверности на уровне свойств

Несмотря на то что специальное сообщение об ошибке более выразительно, чем стандартное сообщение, польза от него по-прежнему невелика, т.к. оно не указывает ясно пользователю на проблему. Для ошибок такого рода отображать сообщения об ошибках проверки достоверности более практично рядом с HTML-элементами, которые содержат проблемные данные, что можно делать с применением класса вспомогательной функции дескриптора `ValidationMessageTag`. Он ищет элементы `span` с атрибутом `asp-validation-for`, используемым для указания свойства модели, к которому должны относиться отображаемые сообщения об ошибках.

В листинге 27.13 для каждого элемента `input` внутри формы добавляются элементы сообщений об ошибках проверки достоверности на уровне свойств. Кроме того, удален раздел `scripts`, поскольку индивидуальные сообщения об ошибках проверки достоверности обеспечат достаточное выделение, чтобы можно было понять, какие элементы содержат ошибки.

Листинг 27.13. Добавление сообщений об ошибках проверки достоверности на уровне свойств в файле `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

...

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div asp-validation-summary="All" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <div><span asp-validation-for="ClientName" class="text-danger"></span>
    </div>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <div><span asp-validation-for="Date" class="text-danger"></span></div>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <span asp-validation-for="TermsAccepted" class="text-danger"></span>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Так как элементы `span` отображаются внутри, нужно позаботиться о том, чтобы было вполне очевидно, к каким элементам относятся сообщения об ошибках проверки достоверности. Запустив приложение и отправив форму без ввода каких-нибудь данных, можно увидеть новые сообщения об ошибках проверки достоверности (рис. 27.5).

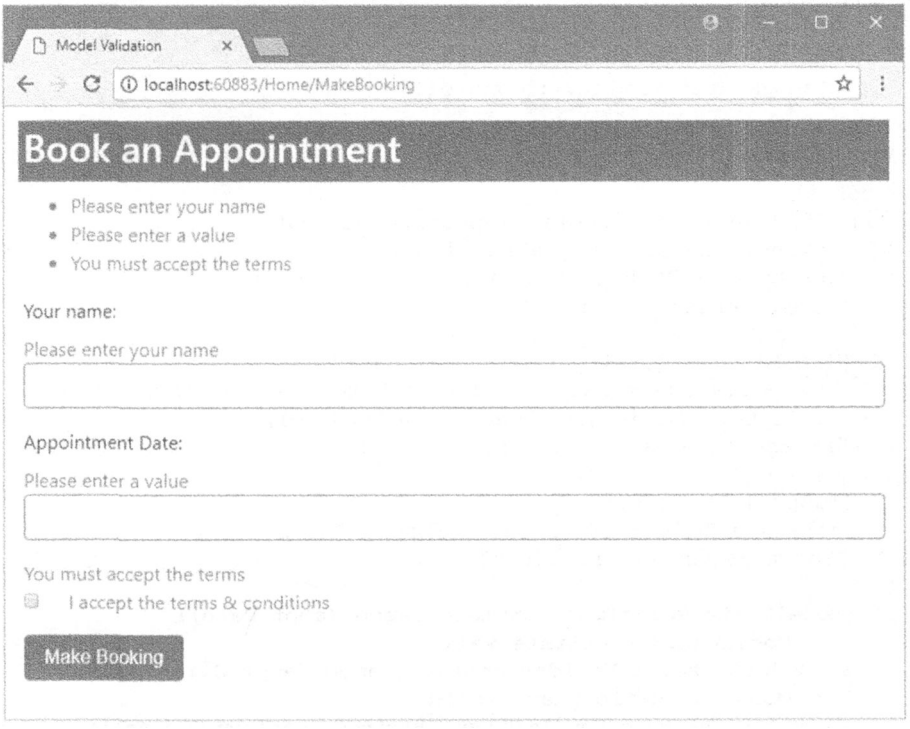


Рис. 27.5. Применение сообщений об ошибках проверки достоверности на уровне свойств

Отображение сообщений об ошибках проверки достоверности на уровне модели

Может показаться, что сводка по проверке достоверности в приложении избыточна, т.к. она просто дублирует сообщения уровня свойств, которые в целом более полезны пользователю, поскольку они находятся рядом с элементами формы, где должны быть устранены проблемы. Но сводка позволяет предпринять удобный трюк, который заключается в возможности отображать сообщения, применимые ко всей модели, а не только к отдельным свойствам. Другими словами, можно сообщать об ошибках, которые возникают в комбинации индивидуальных свойств, например, когда заданная дата допустима только в сочетании со специфическим именем.

В листинге 27.14 добавлена проверка достоверности, которая предотвращает назначение встречи пользователем по имени Джо (`Joe`) по понедельникам (`Monday`).

Листинг 27.14. Выполнение проверки на уровне модели в файле HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() =>
      View("MakeBooking", new Appointment() { Date = DateTime.Now });

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {
      if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError(nameof(appt.ClientName),
          "Please enter your name");
      }
      if (ModelState.GetValidationState("Date")
          == ModelValidationState.Valid && DateTime.Now > appt.Date) {
        ModelState.AddModelError(nameof(appt.Date),
          "Please enter a date in the future");
      }
      if (!appt.TermsAccepted) {
        ModelState.AddModelError(nameof(appt.TermsAccepted),
          "You must accept the terms");
      }
      if (ModelState.GetValidationState(nameof(appt.Date))
            == ModelValidationState.Valid
          && ModelState.GetValidationState(nameof(appt.ClientName))
            == ModelValidationState.Valid
          && appt.ClientName.Equals("Joe", StringComparison.OrdinalIgnoreCase)
          && appt.Date.DayOfWeek == DayOfWeek.Monday) {
        ModelState.AddModelError("",
          "Joe cannot book appointments on Mondays");
      }
      if (ModelState.IsValid) {
        return View("Completed", appt);
      } else {
        return View();
      }
    }
  }
}
```

Код выглядит более запутанным, чем есть на самом деле, что отражает саму природу проверки достоверности данных. Допустимость значений `ClientName` и `Date` проверяется путем инспектирования состояния модели перед выяснением, выпадает ли указанная дата на понедельник и содержит ли свойство `ClientName` строку `Joe`. Если Джо пытается назначить встречу на понедельник, тогда вызывается метод `AddModelError()` с передачей в первом аргументе пустой строки (`""`), которая указывает на то, что ошибка касается всей модели, а не отдельного свойства.

В листинге 27.15 значение атрибута asp-validation-summary установлено в ModelOnly, что приводит к исключению ошибок уровня свойств, поэтому сводка будет отображать только сообщения об ошибках, которые применимы к модели целиком.

Листинг 27.15. Отображение сообщений об ошибках проверки достоверности на уровне модели в файле MakeBooking.cshtml из папки Views/Home

```
@model Appointment
@{ Layout = "_Layout"; }

@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script type="text/javascript">
    $(document).ready(function () {
      $("input.input-validation-error")
        .closest(".form-group").addClass("has-danger");
    });
  </script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</h2>
</div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <div><span asp-validation-for="ClientName" class="text-danger">
        </span>
    </div>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <div><span asp-validation-for="Date" class="text-danger"></span>
    </div>
    <input asp-for="Date" type="text" asp-format="{0:d}"
          class="form-control" />
  </div>
  <span asp-validation-for="TermsAccepted" class="text-danger"></span>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Запустим приложение, введем Joe в поле ClientName и выберем дату, выпадающую на понедельник, такую как 18 января 2027 года (01/18/2027). После отправки формы будет получен ответ, показанный на рис. 27.6.

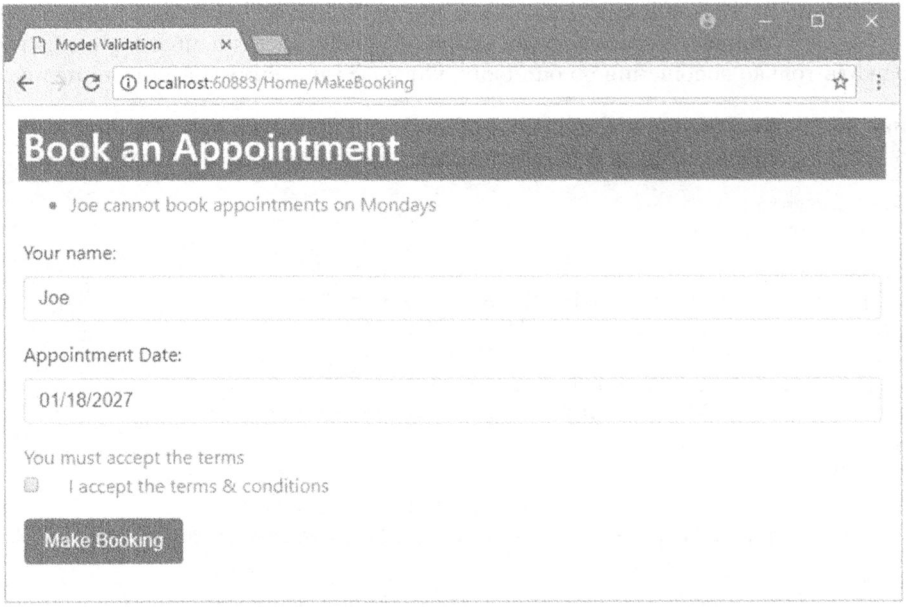


Рис. 27.6. Использование сообщений об ошибках проверки достоверности на уровнях модели и свойств

Указание правил проверки достоверности с помощью метаданных

Одна из проблем с помещением логики проверки достоверности внутрь метода действия связана с тем, что в итоге она дублируется в каждом методе действия, который получает данные от пользователя. Чтобы помочь сократить дублирование, процесс проверки достоверности поддерживает использование атрибутов. Атрибуты позволяют выражать правила проверки достоверности моделей прямо в классе модели, гарантируя применение одного и того же набора правил независимо от метода, который используется для обработки запроса.

В листинге 27.16 к классу `Appointment` применяются атрибуты для навязывания того же набора правил проверки достоверности на уровне свойств, который использовался в предыдущем разделе.

Листинг 27.16. Применение атрибутов проверки достоверности в файле `Appointment.cs` из папки `Models`

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {

  public class Appointment {

    [Required]
    [Display(Name = "name")]
    public string ClientName { get; set; }
```

```
    [UIHint("Date")]
    [Required(ErrorMessage = "Please enter a date")]
    public DateTime Date { get; set; }

    [Range(typeof(bool), "true", "true",
      ErrorMessage = "You must accept the terms")]
    public bool TermsAccepted { get; set; }
  }
}
```

Здесь используются два атрибута проверки достоверности — `Required` и `Range`. Атрибут `Required` указывает, что ошибка проверки достоверности возникает, если пользователь не отправил значение для свойства. Атрибут `Range` задает подмножество приемлемых значений. В табл. 27.7 приведен набор встроенных атрибутов проверки достоверности, доступных в приложении MVC.

Таблица 27.7. Встроенные атрибуты проверки достоверности

Атрибут	Пример	Описание
`Compare`	`[Compare ("`*ДругоеСвойство*`")]`	Данный атрибут гарантирует, что свойства имеют одно и то же значение. Это полезно, когда вы предлагаете пользователю два раза предоставить ту же самую информацию, такую как адрес электронной почты или пароль
`Range`	`[Range(10, 20)]`	Этот атрибут гарантирует, что числовое значение (или значение свойства любого типа, реализующего интерфейс `IComparable`) не находится за рамками заданных минимального и максимального значений. Чтобы указать границу только с одной стороны, применяйте константу `MinValue` или `MaxValue` (например, `[Range(int.MinValue, 50)]`)
`RegularExpression`	`[RegularExpression ("`*шаблон*`")]`	Данный атрибут гарантирует, что строковое значение соответствует указанному шаблону регулярного выражения. Обратите внимание, что шаблон должен соответствовать *всему* предоставленному пользователем значению, а не только подстроке внутри него. По умолчанию при сопоставлении учитывается регистр символов, но с помощью модификатора `(?i)` его можно сделать нечувствительным к регистру, например, `[RegularExpression("(?i)`*шаблон*`")]`

Окончание табл. 27.7

Атрибут	Пример	Описание
Required	[Required]	Этот атрибут гарантирует, что значение не является пустой строкой или строкой, состоящей только из пробелов. Чтобы трактовать пробельные символы как допустимые, необходимо использовать [Required(AllowEmptyStrings=true)]
StringLength	[StringLength(10)]	Данный атрибут гарантирует, что строковое значение не превышает указанную максимальную длину. Можно также задавать минимальную длину: [StringLength(10, MinimumLength=2)]

Все атрибуты проверки достоверности позволяют указывать специальное сообщение об ошибке за счет установки значения для свойства ErrorMessage, например:

```
...
[UIHint("Date")]
[Required(ErrorMessage = "Please enter a date")]
public DateTime Date { get; set; }
...
```

Если специальное сообщение об ошибке не указано, тогда будут применяться стандартные сообщения, но они склонны раскрывать детали класса модели, которые не имеют смысла для пользователя, если только также не используется атрибут Display, как делалось в отношении свойства ClientName:

```
...
[Required]
[Display(Name = "name")]
public string ClientName { get; set; }
...
```

Стандартное сообщение, генерируемое атрибутом Required, отражает имя, указанное с помощью атрибута Display, поэтому оно не раскрывает пользователю имя самого свойства.

Обеспечение согласованной работы такой проверки достоверности требует определенного внимания. В качестве примера вот атрибут проверки достоверности, примененный к свойству TermsAccepted:

```
...
[Range(typeof(bool), "true", "true",
ErrorMessage="You must accept the terms")]
public bool TermsAccepted { get; set; }
...
```

В данном случае необходимо удостовериться в том, что пользователь отметил флажок для принятия условий. Использовать атрибут Required нельзя, т.к. браузер отправит для данного свойства значение false, если пользователь не отметил флажок. Проблема решается за счет возможности атрибута Range предоставлять объект Type

и указывать верхнюю и нижнюю границы в виде строковых значений. Установка обеих границ в `true` создает эквивалент атрибута `Required` для свойств типа `bool`, которые редактируются с применением флажков. Гарантирование того, что атрибуты проверки достоверности и данные, отправляемые браузером, успешно работают вместе, может потребовать некоторого экспериментирования.

Использование атрибутов проверки достоверности в классе модели означает, что метод действия в контроллере можно упростить, как показано в листинге 27.17.

Листинг 27.17. Удаление проверки достоверности на уровне свойств в файле `HomeController.cs` из папки `Controllers`

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {
  public class HomeController : Controller {
    public IActionResult Index() =>
      View("MakeBooking", new Appointment() { Date = DateTime.Now });
    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {
      if (ModelState.GetValidationState(nameof(appt.Date))
            == ModelValidationState.Valid
          && ModelState.GetValidationState(nameof(appt.ClientName))
            == ModelValidationState.Valid
          && appt.ClientName.Equals("Joe", StringComparison.OrdinalIgnoreCase)
          && appt.Date.DayOfWeek == DayOfWeek.Monday) {
        ModelState.AddModelError("",
          "Joe cannot book appointments on Mondays");
      }
      if (ModelState.IsValid) {
        return View("Completed", appt);
      } else {
        return View();
      }
    }
  }
}
```

Атрибуты проверки достоверности применяются перед вызовом метода действия, т.е. при выполнении проверки достоверности на уровне модели для выяснения, допустимы ли индивидуальные свойства, можно по-прежнему полагаться на состояние модели. Чтобы взглянуть на атрибуты проверки достоверности в действии, запустим приложение и отправим форму, не вводя какие-либо данные (рис. 27.7).

Создание специального атрибута проверки достоверности для свойства

Процесс проверки достоверности может быть расширен за счет создания атрибута, который реализует интерфейс `IModelValidator`. Создадим папку `Infrastructure` и добавим в нее файл класса по имени `MustBeTrueAttribute.cs` с определением, приведенным в листинге 27.18.

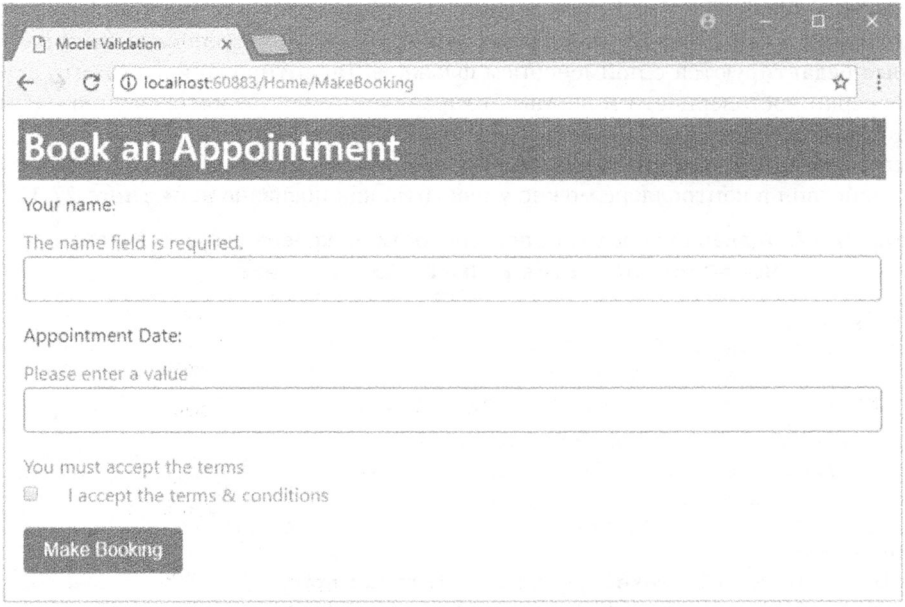


Рис. 27.7. Использование атрибутов проверки достоверности

Листинг 27.18. Содержимое файла `MustBeTrueAttribute.cs` из папки `Infrastructure`

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;

namespace ModelValidation.Infrastructure {
  public class MustBeTrueAttribute : Attribute, IModelValidator {

    public bool IsRequired => true;

    public string ErrorMessage { get; set; } = "This value must be true";

    public IEnumerable<ModelValidationResult> Validate(
            ModelValidationContext context) {
      bool? value = context.Model as bool?;
      if (!value.HasValue || value.Value == false) {
        return new List<ModelValidationResult> {
          new ModelValidationResult("", ErrorMessage)
        };
      } else {
        return Enumerable.Empty<ModelValidationResult>();
      }
    }
  }
}
```

В интерфейсе IModelValidator определено свойство IsRequired, которое применяется для указания, требуется ли проверка достоверности с помощью этого класса (что немного вводит в заблуждение, т.к. значение, возвращаемое данным свойством, просто используется для упорядочения атрибутов проверки достоверности, чтобы обязательные атрибуты выполнялись первыми). Метод Validate() применяется для выполнения проверки достоверности и получает информацию через экземпляр класса ModelValidationContext, полезные свойства которого описаны в табл. 27.8.

Таблица 27.8. Полезные свойства класса ModelValidationContext

Имя	Описание
Model	Возвращает значение свойства, подлежащего проверке достоверности, которым в рассматриваемом примере будет значение TermsAccepted
Container	Возвращает содержащий свойство объект, которым в рассматриваемом примере будет объект Appointment
ActionContext	Возвращает объект ActionContext, предоставляющий данные контекста и описывающий метод действия, который будет обрабатывать запрос
ModelMetadata	Возвращает объект ModelMetadata, который детально описывает класс модели, подвергающийся проверке достоверности

Метод Validate() возвращает последовательность объектов ModelValidationResult, каждый из которых описывает одиночную ошибку проверки достоверности. В примере атрибута объект ModelValidationResult создается, если значение context.Model не равно true.

В первом аргументе конструктору ModelValidationResult передается имя свойства, с которым ассоциирована ошибка, что указывается как пустая строка при проверке достоверности индивидуальных свойств. Во втором аргументе задается сообщении об ошибке, которое будет отображаться пользователю. В листинге 27.19 атрибут Range заменяется специальным атрибутом.

Листинг 27.19. Применение специального атрибута в файле Appointment.cs из папки Models

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;

namespace ModelValidation.Models {

  public class Appointment {

    [Required]
    [Display(Name = "name")]
    public string ClientName { get; set; }

    [UIHint("Date")]
    [Required(ErrorMessage = "Please enter a date")]
    public DateTime Date { get; set; }

    [MustBeTrue(ErrorMessage = "You must accept the terms")]
    public bool TermsAccepted { get; set; }
  }
}
```

Результат использования специального атрибута проверки достоверности будет точно таким же, как и атрибута `Range`, но при чтении кода цель специального атрибута более очевидна.

Выполнение проверки достоверности на стороне клиента

Все продемонстрированные до сих пор приемы проверки были примерами *проверки достоверности на стороне сервера*. Это означает, что пользователь посылает свои данные серверу, сервер проверяет данные и отправляет обратно результаты проверки (либо признак успешности, либо список ошибок, подлежащих исправлению).

В веб-приложениях пользователи обычно ожидают немедленного отклика проверки достоверности без необходимости отправки чего-либо серверу. Такая возможность известна как *проверка достоверности на стороне клиента* и реализуется с применением JavaScript. Вводимые пользователем данные проверяются на предмет достоверности перед их отправкой серверу, снабжая пользователя немедленным откликом и шансом скорректировать любые проблемы.

Инфраструктура MVC поддерживает *ненавязчивую проверку достоверности на стороне клиента*. Термин "ненавязчивая" означает, что правила проверки достоверности выражаются с использованием атрибутов, которые добавляются к HTML-элементам, генерируемым представлениями. Атрибуты интерпретируются библиотекой JavaScript, входящей в состав инфраструктуры MVC, которая в свою очередь конфигурирует библиотеку jQuery Validation, выполняющую действительную работу по проверке достоверности. В последующих разделах будет показано, как работает встроенная поддержка проверки достоверности, и продемонстрированы способы расширения ее функциональности для обеспечения проверки достоверности на стороне клиента.

Совет. Проверка достоверности на стороне клиента сосредоточена на проверке достоверности индивидуальных свойств. В действительности настроить проверку достоверности клиентской стороны на уровне модели с помощью встроенной в MVC поддержки нелегко. С этой целью в большинстве приложений MVC проверка на стороне клиента применяется для свойств, а проверка на стороне сервера — для всей модели.

Прежде всего, понадобится добавить в приложение новые пакеты JavaScript, используя Bower (листинг 27.20).

Листинг 27.20. Добавление пакетов в файле `bower.json` из папки `ModelValidation`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Применение проверки достоверности на стороне клиента означает добавление к представлению трех файлов JavaScript: библиотеки jQuery, библиотеки проверки достоверности jQuery и библиотеки ненавязчивой проверки достоверности от Microsoft; все они задействованы в листинге 27.21.

Совет. Инструмент Bower не всегда корректно выполняет установку пакетов проверки достоверности. Если вы обнаружите, что папка `wwwroot/lib` не содержит необходимых файлов, тогда удалите ее вместе с содержимым. Откройте окно PowerShell, перейдите в папку проекта и запустите по очереди команды `bower cache clean` и `bower install`, чтобы загрузить актуальные копии пакетов проверки достоверности.

Листинг 27.21. Добавление элементов проверки достоверности JavaScript в файле `MakeBooking.cshtml` из папки `Views/Home`

```
@model Appointment
@{ Layout = "_Layout"; }

@section scripts {
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script src="/lib/jquery-validation/dist/jquery.validate.min.js">
  </script>
  <script
    src=
"/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
  </script>
}

<div class="bg-primary m-1 p-1 text-white"><h2>Book an Appointment</
h2></div>

<form class="m-1 p-1" asp-action="MakeBooking" method="post">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="ClientName">Your name:</label>
    <div><span asp-validation-for="ClientName" class="text-danger"></span>
    </div>
    <input asp-for="ClientName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Date">Appointment Date:</label>
    <div><span asp-validation-for="Date" class="text-danger"></span></div>
    <input asp-for="Date" type="text" asp-format="{0:d}"
           class="form-control" />
  </div>
  <span asp-validation-for="TermsAccepted" class="text-danger"></span>
  <div class="radio form-group">
    <input asp-for="TermsAccepted" />
    <label asp-for="TermsAccepted" class="form-check-label">
      I accept the terms & conditions
    </label>
  </div>
  <button type="submit" class="btn btn-primary">Make Booking</button>
</form>
```

Файлы должны добавляться в показанном порядке. Когда вспомогательные функции дескрипторов трансформируют элементы `input`, они инспектируют атрибуты проверки достоверности, примененные к свойствам класса модели, и добавляют атрибуты к выходным элементам. Запустив приложение и просмотрев HTML-разметку, отправленную браузеру, можно заметить элемент следующего вида:

```
<input class="form-control" type="text" data-val="true"
       data-val-required="The name field is required." id="ClientName"
       name="ClientName" value="" />
```

Код JavaScript ищет элементы с атрибутом `data-val` и выполняет локальную проверку достоверности в браузере, когда пользователь отправляет форму, не посылая HTTP-запрос серверу. Чтобы увидеть эффект, запустим приложение и отправим форму, одновременно используя инструменты <F12> для наблюдения за тем, что сообщения об ошибках проверки достоверности отображаются, хотя никакие HTTP-запросы серверу не отправлялись.

Избегание конфликтов с проверкой достоверности, встроенной в браузеры

Ряд браузеров, поддерживающих HTML5, обеспечивают простую проверку достоверности на стороне клиента, которая основана на атрибутах, применяемых к элементам `input`. Общая идея в том, что элемент `input`, к которому применен, скажем, атрибут `required`, вызовет отображение браузером сообщения об ошибке проверки достоверности, если пользователь попытается отправить форму, не предоставив значение для этого элемента.

В случае генерации элементов форм из моделей, как делалось ранее в главе, никаких проблем со встроенной в браузеры проверкой достоверности возникать не будет, поскольку для указания правил проверки инфраструктура MVC генерирует и использует атрибуты данных. (Таким образом, например, элемент `input`, который обязан иметь значение, помечается атрибутом `data-val-required`, не распознаваемым браузерами.)

Тем не менее, вы можете столкнуться с проблемами, если не в состоянии полностью контролировать разметку в приложении, что часто происходит при поступлении содержимого, сгенерированного в другом месте. В результате с формой может работать и проверка достоверности jQuery, и проверка достоверности, встроенная в браузер, что всего лишь сбивает с толку пользователя. Чтобы избежать такой проблемы, к элементу `form` можно добавить атрибут `novalidate`.

Одна из замечательных характеристик проверки достоверности на стороне клиента MVC связана с тем, что те же самые атрибуты, используемые для указания правил проверки достоверности, применяются внутри клиента *и* на сервере. Это означает, что данные, поступающие из браузеров, которые не поддерживают JavaScript, подвергаются такой же проверке достоверности, что и данные из браузеров, поддерживающих JavaScript, безо всяких дополнительных усилий. Однако это также означает, что специальные атрибуты проверки достоверности при проверке на стороне клиента не поддерживаются, потому что код JavaScript не имеет возможности реализовать специальную логику внутри клиента. Иными словами, при желании использовать проверку достоверности на стороне клиента необходимо придерживаться встроенных атрибутов, описанных в табл. 27.7.

Сравнение проверки достоверности на стороне клиента в MVC и проверки достоверности с помощью библиотеки jQuery Validation

Функциональность проверки достоверности на стороне клиента в MVC построена на основе библиотеки jQuery Validation. Если хотите, то можете работать с библиотекой jQuery Validation напрямую, игнорируя средства MVC. Библиотека jQuery Validation обладает высокой гибкостью и большими возможностями. Ее имеет смысл исследовать хотя бы для того, чтобы понять, каким образом настраивать средства MVC с целью извлечения максимальной пользы из доступных вариантов проверки. Библиотека jQuery Validation подробно описана в книге *jQuery 2.0 для профессионалов* (ИД "Вильямс").

Выполнение удаленной проверки достоверности

В завершение главы мы рассмотрим *удаленную (дистанционную) проверку достоверности*. Это прием проверки достоверности на стороне клиента, при котором для выполнения проверки вызывается метод действия на сервере.

Распространенный пример удаленной проверки достоверности предусматривает выяснение доступности имени пользователя в приложениях, где оно должно быть уникальным, когда пользователь посылает данные и производится проверка достоверности на стороне клиента. В качестве части такого процесса серверу отправляется запрос Ajax для проверки имени пользователя. Если имя пользователя уже было выдано, тогда отображается сообщение об ошибке проверки и пользователю предоставляется возможность ввести другое имя.

Процесс может выглядеть похожим на обычную проверку достоверности на стороне сервера, но такой подход обладает рядом преимуществ. Во-первых, удаленно проверяться будут только некоторые свойства; ко всем остальным значениям данных, которые ввел пользователь, по-прежнему будет применяться проверка достоверности на стороне клиента. Во-вторых, запрос является относительно легковесным и сосредоточенным на проверке достоверности, а не на обработке целого объекта модели.

Третье отличие связано с тем, что удаленная проверка достоверности выполняется в фоновом режиме. Пользователю не приходится щелкать на кнопке отправки и ожидать визуализации нового представления. В итоге пользователи получают более отзывчивый интерфейс, что особенно важно в случае медленного сетевого соединения между браузером и сервером.

Тем не менее, удаленная проверка достоверности сопряжена с компромиссом. Она соблюдает баланс между проверками на стороне клиента и на стороне сервера, но требует отправки запросов серверу приложений, поэтому не будет настолько быстрой, как обычная проверка достоверности на стороне клиента.

Первый шаг в сторону использования удаленной проверки достоверности предполагает создание метода действия, который может проверить одно из свойств модели. Мы будем проверять свойство `Date` модели `Appointment`, чтобы обеспечить нахождение запрошенной даты встречи в будущем. (Это одно из исходных правил проверки достоверности, применяемых в начале главы, но его невозможно реализовать с помощью стандартных средств проверки на стороне клиента.)

В листинге 27.22 приведен код метода действия `ValidateDate()`, добавленного в контроллер `Home`.

Листинг 27.22. Добавление метода действия для проверки достоверности в файле HomeController.cs из папки Controllers

```
using System;
using Microsoft.AspNetCore.Mvc;
using ModelValidation.Models;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace ModelValidation.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() =>
      View("MakeBooking", new Appointment() { Date = DateTime.Now });

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {
      if (ModelState.GetValidationState(nameof(appt.Date))
            == ModelValidationState.Valid
          && ModelState.GetValidationState(nameof(appt.ClientName))
            == ModelValidationState.Valid
          && appt.ClientName.Equals("Joe", StringComparison.OrdinalIgnoreCase)
          && appt.Date.DayOfWeek == DayOfWeek.Monday) {
        ModelState.AddModelError("",
          "Joe cannot book appointments on Mondays");
      }

      if (ModelState.IsValid) {
        return View("Completed", appt);
      } else {
        return View();
      }
    }

    public JsonResult ValidateDate(string Date) {
      DateTime parsedDate;

      if (!DateTime.TryParse(Date, out parsedDate)) {
        return Json("Please enter a valid date (mm/dd/yyyy)");
      } else if (DateTime.Now > parsedDate) {
        return Json("Please enter a date in the future");
      } else {
        return Json(true);
      }
    }
  }
}
```

Методы действий, поддерживающие удаленную проверку достоверности, должны возвращать объект типа `JsonResult`, который сообщает инфраструктуре MVC о том, что работа производится с данными JSON, как объяснялось в главе 20. В дополнение к возвращению такого результата методы действий для проверки достоверности обязаны определять параметр, который имеет то же самое имя, что и проверяемое поле данных; в рассматриваемом примере это `Date`. Внутри метода действий проверка достоверности выполняется путем преобразования значения в объект `DateTime` и выяснения, относится ли дата к будущему.

Совет. Можно было бы воспользоваться привязкой моделей, так что параметром метода действия стал бы объект `DateTime`, но тогда метод проверки достоверности не вызывался бы в ситуации, когда пользователь ввел бессмысленное значение вроде `apple`. Причина в том, что связывателю модели не удается создать объект `DateTime` из `apple` и генерируется исключение. Средство удаленной проверки достоверности не способно обработать такое исключение, поэтому исключение молча отбрасывается. В итоге возникает нежелательный эффект: поле данных *не* подсвечивается, создавая впечатление, что введенное пользователем значение является допустимым. Как правило, наилучший подход к удаленной проверке достоверности предусматривает получение методом действия параметра `string` и явное выполнение любого преобразования типа, разбора или привязки модели.

Результаты проверки достоверности выражаются с применением метода `Json()`, создающего результат в формате JSON, который может разобрать и обработать сценарий удаленной проверки достоверности на стороне клиента. Если значение допустимо, тогда в качестве параметра методу `Json()` передается `true`:

```
...
return Json(true);
...
```

При наличии проблемы в параметре передается сообщение об ошибке проверки, которое должен увидеть пользователь:

```
...
return Json("Please enter a date in the future");
...
```

Чтобы использовать метод удаленной проверки достоверности, к нужному свойству класса модели применяется атрибут `Remote` (листинг 27.23).

Листинг 27.23. Использование атрибута `Remote` в файле `Appointment.cs` из папки `Models`

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;
using Microsoft.AspNetCore.Mvc;

namespace ModelValidation.Models {

  public class Appointment {

    [Required]
    [Display(Name = "name")]
    public string ClientName { get; set; }

    [UIHint("Date")]
    [Required(ErrorMessage = "Please enter a date")]
    [Remote("ValidateDate", "Home")]
    public DateTime Date { get; set; }

    [MustBeTrue(ErrorMessage = "You must accept the terms")]
    public bool TermsAccepted { get; set; }
  }
}
```

В качестве аргументов атрибуту `Remote` указываются имя действия и контроллер, предназначенные для генерации URL, по которому библиотека проверки достоверности JavaScript будет обращаться с целью выполнения проверки — в рассматриваемом случае это действие `ValidateDate` из контроллера `Home`.

Чтобы посмотреть на работу удаленной проверки достоверности, запустим приложение, перейдем на URL вида `/Home` и введем дату, относящуюся к прошлому. После того как фокус ввода переместится на другой элемент, появится сообщение об ошибке проверки достоверности (рис. 27.8).

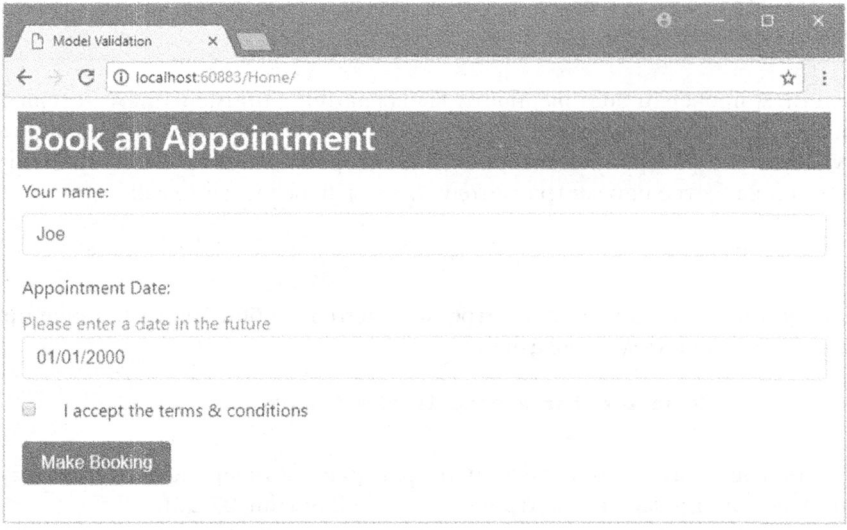


Рис. 27.8. Выполнение удаленной проверки достоверности

Внимание! Метод действия для проверки достоверности будет вызываться, когда пользователь впервые отправляет форму, и затем каждый раз, когда данные редактируются. Для элементов ввода текста любое нажатие клавиши будет иметь следствием обращение к серверу. В некоторых приложениях это может вылиться в значительное количество запросов, что должно быть учтено при описании требований к производительности сервера и ширине полосы пропускания в производственной среде. Кроме того, может быть принято решение *отказаться* от удаленной проверки достоверности для свойств, проверка которых сопряжена с высокими затратами (например, если выяснение уникальности имени пользователя требует обращения к медленному серверу).

Резюме

В настоящей главе был представлен широкий спектр приемов, доступных для выполнения проверки достоверности моделей, которая гарантирует, что предоставляемые пользователем данные удовлетворяют ограничениям, налагаемым на модель данных. Проверка достоверности моделей является важной темой, и наличие в приложении подходящих средств проверки жизненно важно для обеспечения пользователям комфортных условий работы. В следующей главе будет показано, как защитить приложение MVC с применением ASP.NET Core Identity.

ГЛАВА 28

Введение в ASP.NET Core Identity

Система ASP.NET Core Identity представляет собой API-интерфейс от Microsoft, предназначенный для управления пользователями в приложениях ASP.NET. В настоящей главе демонстрируется процесс настройки ASP.NET Core Identity и создания простого инструмента администрирования пользователей, который управляет индивидуальными пользовательскими учетными записями, хранящимися в базе данных.

Система ASP.NET Core Identity поддерживает другие виды пользовательских учетных записей, такие как записи, хранящиеся с использованием Active Directory, но здесь они не рассматриваются, потому что редко применяются вне корпораций (где реализации Active Directory оказываются настолько замысловатыми, что очень трудно отыскать полезные общие примеры).

На заметку! Настоящая глава требует наличия установленного средства SQL Server LocalDB для Visual Studio. Чтобы добавить LocalDB, понадобится запустить программу установки Visual Studio и выбрать вариант SQL Server Express 2016 LocalDB в разделе Individual Components (Индивидуальные компоненты).

В главе 29 будет показано, как выполнять аутентификацию и авторизацию с помощью таких пользовательских учетных записей, а в главе 30 — каким образом выйти за рамки основ и применять ряд более сложных приемов. В табл. 28.1 приведена сводка, позволяющая поместить систему ASP.NET Core Identity в контекст.

Таблица 28.1. Помещение системы ASP.NET Core Identity в контекст

Вопрос	Ответ
Что это такое?	Система ASP.NET Core Identity — это API-интерфейс для управления пользователями и запоминания пользовательских данных в хранилищах, таких как реляционные базы данных, посредством Entity Framework Core
Чем она полезна?	Управление пользователями является важной возможностью для большинства приложений, и ASP.NET Core Identity предлагает готовую хорошо протестированную платформу, которая не требует создания специальных версий распространенных функций

Окончание табл. 28.1

Вопрос	Ответ
Как она используются?	Система Identity используется через службы и промежуточное ПО, добавляемое в классе `Startup`, и посредством классов, которые действуют в качестве шлюзов между приложением и функциональностью Identity
Существуют ли какие-то скрытые ловушки или ограничения?	В Microsoft скомпенсировали жесткость ранних API-интерфейсов управления пользователями ASP.NET, сделав систему Identity настолько гибкой и конфигурируемой, что выяснение того, что возможно и что необходимо, может оказаться проблематичным. В книге мы лишь слегка коснемся поверхности этой глубокой и сложной системы
Существуют ли альтернативы?	Можно было бы реализовать собственные API-интерфейсы, но такая задача обычно требует большого объема работы и вдобавок чревата созданием уязвимостей защиты, если не выполнять ее крайне аккуратно

В табл. 28.2 приведена сводка по главе.

Таблица 28.2. Сводка по главе

Задача	Решение	Листинг
Добавление Identity в проект	Добавьте пакеты и промежуточное ПО для ASP.NET Identity Core и Entity Framework Core, создайте класс пользователя и класс контекста базы данных, а также создайте миграцию базы данных	28.1–28.13
Чтение данных о пользователе	Выполните запрос к базе данных Identity с применением класса контекста	28.14, 28.15
Создание пользовательской учетной записи	Вызовите метод `UserManager.CreateAsync()`	28.18–28.18
Изменение стандартной политики проверки паролей	Установите параметры паролей в классе `Startup`	28.19
Реализация специальной проверки паролей	Реализуйте интерфейс `IPasswordValidator` либо унаследуйте класс от класса `PasswordValidator`	28.20–28.22
Изменение политики проверки учетных записей	Установите параметры пользовательских учетных записей в классе `Startup`	28.23
Реализация специальной проверки учетных записей	Реализуйте интерфейс `IUserValidator` либо унаследуйте класс от класса `UserValidator`	28.24–28.26
Удаление пользовательской учетной записи	Вызовите метод `UserManager.DeleteAsync()`	28.27, 28.28
Редактирование пользовательской учетной записи	Вызовите метод `UserManager.UpdateAsync()`	28.29–28.31

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени Users с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core. Примеру приложения требуются инструменты командной строки Entity Framework Core, которые должны быть добавлены в проект путем ручного редактирования файла .csproj. Щелкнем правой кнопкой мыши на элементе проекта Users в окне Solution Explorer, выберем в контекстном меню пункт Edit Users.csproj file (Редактировать файл Users.csproj) и добавим элемент, показанный в листинге 28.1.

Листинг 28.1. Добавление пакета в файле Users.csproj из папки Users

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <DotNetCliToolReference
        Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
        Version="2.0.0" />
  </ItemGroup>

</Project>
```

В листинге 28.2 приведен код класса Startup, который настраивает MVC Framework и компоненты промежуточного ПО, как было описано в главе 14.

Листинг 28.2. Содержимое файла Startup.cs из папки Users

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace Users {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание контроллера и представления

Создадим папку `Controllers`, добавим файл класса по имени `HomeController.cs` и поместим в него определение контроллера из листинга 28.3. Этот контроллер будет применяться для описания деталей пользовательских учетных записей и данных, а его метод действия `Index()` передает словарь значений стандартному представлению через метод `View()`.

Листинг 28.3. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

namespace Users.Controllers {

  public class HomeController : Controller {

    public ViewResult Index() =>
      View(new Dictionary<string, object>
        {["Placeholder"] = "Placeholder" });
  }
}
```

Чтобы снабдить контроллер представлением, создадим папку `Views/Home` и добавим в нее файл представления по имени `Index.cshtml` с разметкой, приведенной в листинге 28.4.

Листинг 28.4. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model Dictionary<string, object>

<div class="bg-primary m-1 p-1 text-white"><h4>User Details</h4></div>

<table class="table table-sm table-bordered m-1 p-1">
  @foreach (var kvp in Model) {
    <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
  }
</table>
```

Представление отображает содержимое словаря модели в таблице. Для поддержки представления создадим папку `Views/Shared` и добавим в нее файл по имени `_Layout.cshtml` с разметкой, показанной в листинге 28.5.

Листинг 28.5. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <title>Users</title>
  <meta name="viewport" content="width=device-width" />
  <link href="/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
  @RenderBody()
</body>
</html>
```

При стилизации HTML-элементов представление полагается на CSS-пакет Bootstrap. Создадим в корневой папке проекта файл `bower.json` с использованием шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел `dependencies` (листинг 28.6).

Листинг 28.6. Добавление пакета Bootstrap в файле `bower.json` из папки `Users`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Последний подготовительный шаг связан с созданием файла `_ViewImports.cshtml` в папке `Views`, в котором настраиваются встроенные вспомогательные функции дескрипторов для применения в представлениях (листинг 28.7).

Листинг 28.7. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Наконец, создадим в папке `Views` файл запуска представления по имени `_ViewStart.cshtml` с содержимым из листинга 28.8. Он обеспечит использование компоновки, созданной в листинге 28.5, всеми представлениями в приложении.

Листинг 28.8. Содержимое файла `_ViewStart.cshtml` из папки `Views`

```
@{
  Layout = "_Layout";
}
```

В результате запуска приложения появится вывод, приведенный на рис. 28.1.

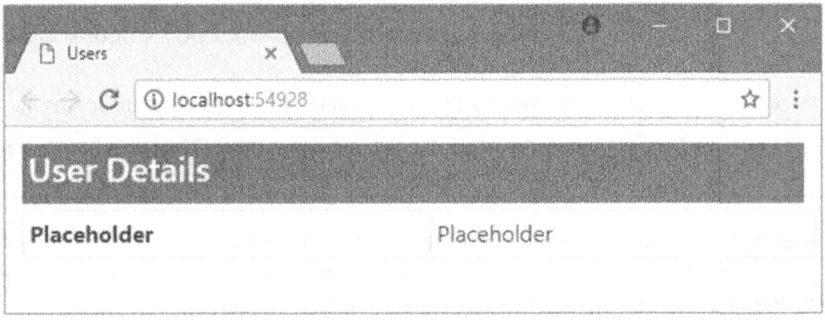


Рис. 28.1. Выполнение примера приложения

Настройка ASP.NET Core Identity

Процесс настройки системы Identity затрагивает почти все части приложения, требуя новых классов моделей, изменений в конфигурации, а также контроллеров и действий для поддержки операций аутентификации и авторизации. В последующих разделах мы пройдем по процессу настройки Identity в базовой конфигурации, чтобы продемонстрировать разнообразные шаги, которые с ним связаны. Задействовать систему Identity в приложении можно многими разными способами, но конфигурация, применяемая в настоящей главе, предусматривает использование самых простых и ходовых параметров.

Создание класса пользователя

Первый шаг заключается в определении класса, предназначенного для представления пользователя в приложении, который называется *классом пользователя*. Класс пользователя является производным от класса `IdentityUser`, определенного в пространстве имен `Microsoft.AspNetCore.Identity.EntityFrameworkCore`. Класс `IdentityUser` обеспечивает базовое представление пользователя, которое можно расширять, добавляя свойства к производному классу, как будет описано в главе 30. В табл. 28.3 перечислены наиболее полезные встроенные свойства, которые определены в `IdentityUser`, включая применяемые в текущей главе.

Таблица 28.3. Свойства класса `IdentityUser`

Имя	Описание
`Id`	Это свойство содержит уникальный идентификатор пользователя
`UserName`	Это свойство возвращает имя пользователя
`Claims`	Это свойство возвращает коллекцию заявок (claim) пользователя, которая будет описана в главе 30
`Email`	Это свойство содержит адрес электронной почты пользователя
`Logins`	Это свойство возвращает коллекцию входов пользователя, используемую для сторонней аутентификации, как объясняется в главе 30
`PasswordHash`	Это свойство возвращает хешированную форму пароля пользователя, которая применяется в разделе "Реализация возможности редактирования" далее в главе
`Roles`	Это свойство возвращает коллекцию ролей, к которым принадлежит пользователь (глава 29)
`PhoneNumber`	Это свойство возвращает телефонный номер пользователя
`SecurityStamp`	Это свойство возвращает значение, которое изменяется, когда меняется удостоверение пользователя, например, из-за смены пароля

Индивидуальные свойства в настоящий момент неважны. Важно то, что класс `IdentityUser` предоставляет доступ к базовой информации о пользователе, включающей имя пользователя, адрес электронной почты, телефонный номер, хеш пароля, членство в ролях и т.д. При желании хранить дополнительные сведения о пользователе придется добавить свойства в класс, производный от `IdentityUser`, который будет использоваться для представления пользователей в приложении.

Чтобы создать класс пользователя для приложения, создадим папку Models и добавим в нее файл класса по имени AppUser.cs с определением класса AppUser, приведенным в листинге 28.9.

Листинг 28.9. Содержимое файла AppUser.cs из папки Models

```
using Microsoft.AspNetCore.Identity;

namespace Users.Models {

  public class AppUser : IdentityUser {
    // Для базовой установки Identity
    // дополнительные члены не требуются
  }
}
```

На данный момент имеется все, что нужно, хотя мы еще вернемся к классу AppUser в главе 30 при рассмотрении способа добавления свойств данных о пользователе, специфичных для приложения.

Конфигурирование импортирования представлений

Несмотря на то что это не относится непосредственно к настройке ASP.NET Core Identity, в следующем разделе мы будем работать с объектами AppUser в представлениях. Чтобы упростить написание представлений, добавим пространство имен Users.Models в файл импортирования представлений (листинг 28.10).

Листинг 28.10. Добавление пространства имен в файле _ViewImports.cshtml из папки Views

```
@using Users.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Создание класса контекста базы данных

Следующий шаг предусматривает создание класса контекста базы данных Entity Framework Core, который оперирует с классом AppUser. Класс контекста унаследован от IdentityDbContext<T>, где T — класс пользователя (AppUser в рассматриваемом примере). Добавим в папку Models файл класса по имени AppIdentityDbContext.cs и определим в нем класс, как показано в листинге 28.11.

Листинг 28.11. Содержимое файла AppIdentityDbContext.cs из папки Models

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Users.Models {

  public class AppIdentityDbContext : IdentityDbContext<AppUser> {

    public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext>
     options)
      : base(options) { }
  }
}
```

Класс контекста базы данных может быть расширен для изменения способа, которым база данных настраивается и используется, но в случае элементарного приложения ASP.NET Core Identity простого определения класса вполне достаточно, чтобы начать и получить заполнитель для любой настройки в будущем.

На заметку! Не переживайте, если роль упомянутых выше классов не особенно ясна. Если ранее вы не имели дела с инфраструктурой Entity Framework Core, тогда можете трактовать ее как "черный ящик". После того, как основные строительные блоки окажутся на месте (и вы можете их копировать в свои проекты, чтобы все заработало), потребность в их редактировании будет возникать редко.

Конфигурирование настройки строки подключения к базе данных

Первый шаг по конфигурированию ASP.NET Core Identity связан с определением строки подключения к базе данных. По соглашению строка подключения помещается в файл `appsettings.json`, который затем загружается при запуске приложения и доступен в классе `Startup`, как объяснялось в главе 14. Создадим в корневой папке проекта файл `appsettings.json` с использованием шаблона элемента ASP.NET Configuration File (Файл конфигурации ASP.NET) и добавим в него конфигурационные настройки из листинга 28.12.

Листинг 28.12. Содержимое файла `appsettings.json` из папки `Users`

```
{
  "Data": {
    "SportStoreIdentity": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=
IdentityUsers;Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}
```

В строке подключения присутствует параметр `localdb`, который обеспечивает удобную поддержку баз данных для разработчиков. В качестве имени базы данных указывается `IdentityUsers`.

На заметку! Ширина печатной страницы не позволяет соблюдать правильный формат строки подключения, которая должна выглядеть как одиночная неразрывная строка. В редакторе Visual Studio это не проблема, но в листинге строку пришлось разбить на части. При добавлении строки подключения в свой проект удостоверьтесь, что вводите ее в одной строке.

Имея строку подключения к базе данных, можно обновить класс `Startup` для получения конфигурационных данных (листинг 28.13).

Листинг 28.13. Получение настроек приложения в файле `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
```

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;

namespace Users {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));

      services.AddIdentity<AppUser, IdentityRole>()
        .AddEntityFrameworkStores<AppIdentityDbContext>()
        .AddDefaultTokenProviders();

      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Для создания базовой установки ASP.NET Core Identity требуются три набора изменений. Сначала настраивается инфраструктура Entity Framework Core, которая предоставляет приложениям MVC службы доступа к данным:

```
...
services.AddDbContext<AppIdentityDbContext>(options =>
  options.UseSqlServer(Configuration[
    "Data:SportStoreIdentity:ConnectionString"]));
...
```

Метод `AddDbContext()` добавляет службы, требующиеся для Entity Framework Core, а метод `UseSqlServer()` настраивает поддержку, необходимую для хранения данных с применением Microsoft SQL Server. Метод `AddDbContext()` позволяет использовать ранее созданный класс контекста базы данных и указать, что он будет копироваться с базой данных SQL Server, строка подключения для которой получается из конфигурации приложения (файл `appsettings.json` в примере приложения).

Понадобится также настроить службы для ASP.NET Core Identity, что делается следующим образом:

```
...
services.AddIdentity<AppUser, IdentityRole>()
  .AddEntityFrameworkStores<AppIdentityDbContext>()
  .AddDefaultTokenProviders();
...
```

Метод `AddIdentity()` имеет параметры типов, которые указывают класс, применяемый для представления пользователей, и класс, используемый для представления ролей. Здесь задается класс `AppUser` для пользователей и класс `IdentityRole` для ролей. Метод `AddEntityFrameworkStores()` указывает, что система Identity должна использовать инфраструктуру Entity Framework Core для сохранения и извлечения своих данных с применением созданного ранее класса контекста базы данных. Метод `AddDefaultTokenProviders()` использует стандартную конфигурацию для поддержки операций, требующих маркера, таких как изменение пароля.

Последнее изменение в классе `Startup` касается добавления системы ASP.NET Core Identity в конвейер обработки запросов. Это позволяет ассоциировать пользовательские учетные данные с запросами на основе cookie-наборов или переписывания URL, т.е. детали пользовательских учетных записей не включаются прямо в HTTP-запросы, отправляемые приложению, или в ответы, которые оно генерирует:

```
...
app.UseAuthentication();
...
```

Создание базы данных Identity

Почти все на месте, и осталось лишь фактически создать базу данных, которая будет использоваться для хранения данных Identity. Откроем окно командной строки или окно PowerShell, перейдем в папку проекта `Users` (ту, что содержит файл `Startup.cs`) и введем следующую команду:

```
dotnet ef migrations add Initial
```

Как объяснялось при настройке базы данных для приложения SportsStore, инфраструктура Entity Framework Core управляет изменениями в схемах баз данных через средство, которое называется *миграции*. В случае модификации классов модели, применяемых для генерации схемы, можно сгенерировать файл миграции, который содержит команды SQL, предназначенные для обновления базы данных. Приведенная выше команда создает файлы миграции, которые будут настраивать базу данных Identity.

Когда команда `dotnet ef` завершится, в окне Solution Explorer появится папка `Migrations`. При просмотре содержимого файлов из папки `Migrations` обнаружатся команды SQL, которые будут использоваться для создания начальной базы данных. Чтобы задействовать файлы миграции для создания базы данных, введем такую команду:

```
dotnet ef database update
```

Выполнение команды может занять некоторое время, а после ее завершения база данных будет создана и готова к применению.

Использование ASP.NET Core Identity

После проведения базовой настройки можно приступать к применению ASP.NET Core Identity, чтобы добавить в пример приложения поддержку управления пользователями. В последующих разделах будет продемонстрировано, как использовать API-интерфейс Identity для создания инструментов администрирования, которые делают возможным централизованное управление пользователями.

Инструменты централизованного администрирования пользователей полезны практически во всех приложениях, даже в тех, которые позволяют пользователям создавать и управлять собственными учетными записями. Всегда найдутся пользователи, которым требуется, скажем, пакетное создание учетных записей и поддержка задач, предполагающих инспектирование и корректировку пользовательских данных. С точки зрения настоящей главы инструменты администрирования удобны из-за того, что они организуют множество основных функций управления пользователями в небольшое число классов, превращая их в полезные примеры для демонстрации фундаментальных возможностей системы ASP.NET Core Identity.

Перечисление пользовательских учетных записей

Отправной точкой текущего раздела является перечисление всех пользовательских учетных записей в базе данных, что позволит увидеть эффект от кода, который будет добавлен в приложение позже. Первым делом добавим в папку `Controllers` файл класса по имени `AdminController.cs` и определим в нем контроллер, как показано в листинге 28.14, который будет применяться для реализации функциональности администрирования пользователей.

Листинг 28.14. Содержимое файла `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;

namespace Users.Controllers {

  public class AdminController : Controller {
    private UserManager<AppUser> userManager;

    public AdminController(UserManager<AppUser> usrMgr) {
      userManager = usrMgr;
    }

    public ViewResult Index() => View(userManager.Users);
  }
}
```

Метод действия `Index()` перечисляет пользователей, управляемых системой Identity; разумеется, в текущий момент пользователи отсутствуют, но вскоре они появятся. Доступ к пользовательским данным осуществляется через объект `UserManager<AppUser>`, который конструктор контроллера получает посредством внедрения зависимостей.

С помощью объекта `UserManager<AppUser>` можно запрашивать хранилище данных. Свойство `Users` возвращает перечисление объектов пользователей (экземпляров класса `AppUser` в рассматриваемом приложении), с которым удобно работать, используя LINQ. Внутри метода действия значение свойства `Users`, которое будет перечислять всех пользователей в базе данных, передается методу `View()`, так что он сможет отобразить детали учетных записей. Чтобы снабдить метод действия представлением, создадим папку `Views/Admin`, добавим файл по имени `Index.cshtml` и поместим в него разметку из листинга 28.15.

Листинг 28.15. Содержимое файла `Index.cshtml` из папки `Views/Admin`

```
@model IEnumerable<AppUser>

<div class="bg-primary m-1 p-1 text-white"><h4>User Accounts</h4></div>

<table class="table table-sm table-bordered">
  <tr><th>ID</th><th>Name</th><th>Email</th></tr>

  @if (Model.Count() == 0) {
    <tr><td colspan="3" class="text-center">No User Accounts</td></tr>
  } else {
    foreach (AppUser user in Model) {
      <tr>
        <td>@user.Id</td>
        <td>@user.UserName</td>
        <td>@user.Email</td>
      </tr>
    }
  }
</table>

<a class="btn btn-primary" asp-action="Create">Create</a>
```

Представление `Index.cshtml` содержит таблицу, в которой для каждого пользователя предусмотрена строка с колонками, отображающими уникальный идентификатор, имя пользователя и адрес электронной почты. Если пользователи в базе данных отсутствуют, тогда выводится соответствующее сообщение, как показано на рис. 28.2, для чего понадобится запустить приложение и запросить URL вида `/Admin`.

В представление включена ссылка Create (Создать), стилизованная под кнопку, которая нацелена на действие `Create` контроллера `Admin`. Это действие будет поддерживать добавление пользователей.

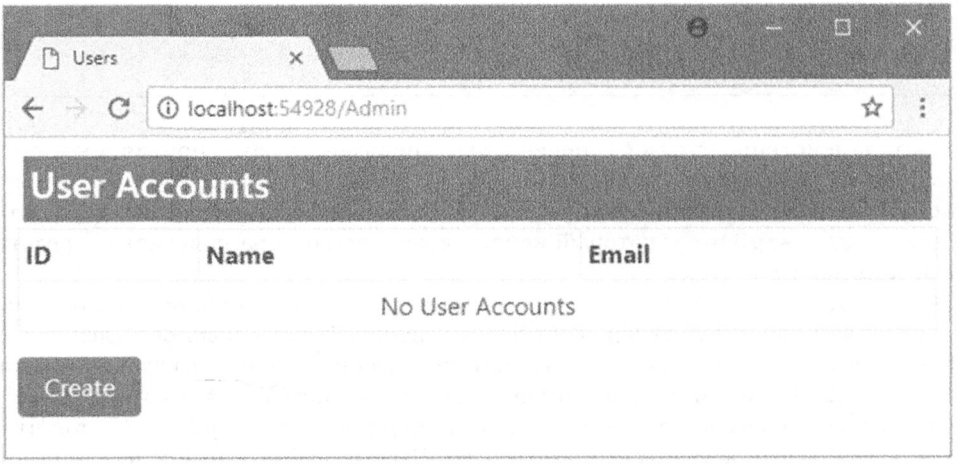


Рис. 28.2. Отображение (пустого) списка пользователей

Переустановка базы данных

Увидеть созданную для системы Identity базу данных можно, открыв окно SQL Server Object Explorer (Проводник объектов SQL Server) в Visual Studio. Если вы впервые используете окно SQL Server Object Explorer, то должны выбрать в меню Tools (Сервис) пункт Connect to Database (Подключиться к базе данных), чтобы сообщить среде Visual Studio о базе данных, с которой нужно работать. В качестве источника данных выберите Microsoft SQL Server, для имени сервера укажите `(localdb)\mssqllocaldb`, оставьте отмеченным флажок Use Windows Authentication (Использовать аутентификацию Windows) и щелкните на стрелке, раскрывающей поле Select Or Enter a Database Name (Выберите или введите имя базы данных). Спустя несколько секунд отобразится список доступных баз данных LocalDB, в котором должна быть возможность выбора базы данных `IdentityUsers`, относящейся к примеру приложения. Щелкните на кнопке OK, после чего в окне SQL Server Object Explorer появится новая запись. Среда Visual Studio запомнит базу данных, так что описанный процесс необходимо выполнить только один раз.

Для просмотра базы данных понадобится раскрыть элемент `(localdb)\mssqllocaldb⇨Databases⇨IdentityUsers` в окне SQL Server Object Explorer. Вы увидите таблицы, которые были созданы файлами миграции, с именами вроде `AspNetUsers` и `AspNetRoles`. После добавления пользователей, как будет показано в следующем разделе, в базу данных можно отправлять запросы для просмотра содержимого таблиц.

Чтобы удалить базу данных, щелкните правой кнопкой мыши на элементе `IdentityUsers` в окне SQL Server Object Explorer и выберите в контекстном меню пункт Delete (Удалить). В диалоговом окне Delete Database (Удаление базы данных) отметьте оба имеющихся флажка и щелкните на кнопке OK для удаления базы данных.

Чтобы заново создать базу данных, откройте окно консоли диспетчера пакетов и введите следующую команду:

```
dotnet ef database update
```

База данных будет воссоздана и готова к применению, когда вы снова запустите приложение.

Создание пользователей

В отношении входных данных, получаемых приложением, будет использоваться проверка достоверности моделей MVC, и легче всего это организовать за счет создания простых моделей представлений для каждой операции, поддерживаемой контроллером. Создадим в папке `Models` файл класса по имени `UserViewModels.cs` с содержимым, приведенным в листинге 28.16.

Листинг 28.16. Содержимое файла `UserViewModels.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;
namespace Users.Models {
  public class CreateModel {
    [Required]
    public string Name { get; set; }
    [Required]
    public string Email { get; set; }
    [Required]
    public string Password { get; set; }
  }
}
```

Начальная модель называется `CreateModel` и определяет основные свойства, которые требуются для создания пользовательской учетной записи: имя пользователя, адрес электронной почты и пароль. Атрибут `Required` из пространства имен `System.ComponentModel.DataAnnotations` применяется для указания на обязательность значений всех трех свойств, определяемых моделью.

В листинге 28.17 к контроллеру `Admin` добавлена пара методов действий `Create()`, на которые нацелена ссылка в представлении `Index` из предыдущего раздела; они используют стандартный прием для отображения пользователю представления в случае запроса `GET` и обработки данных формы при поступлении запроса `POST`.

Листинг 28.17. Определение методов действий `Create()` в файле `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;

namespace Users.Controllers {

  public class AdminController : Controller {
    private UserManager<AppUser> userManager;

    public AdminController(UserManager<AppUser> usrMgr) {
      userManager = usrMgr;
    }

    public ViewResult Index() => View(userManager.Users);

    public ViewResult Create() => View();

    [HttpPost]
    public async Task<IActionResult> Create(CreateModel model) {
      if (ModelState.IsValid) {
        AppUser user = new AppUser {
          UserName = model.Name,
          Email = model.Email
        };
        IdentityResult result
          = await userManager.CreateAsync(user, model.Password);

        if (result.Succeeded) {
          return RedirectToAction("Index");
        } else {
          foreach (IdentityError error in result.Errors) {
            ModelState.AddModelError("", error.Description);
          }
        }
      }
      return View(model);
    }
  }
}
```

Важной частью листинга является метод действия `Create()`, принимающий аргумент `CreateModel`; этот метод будет вызываться, когда администратор отправляет

данные формы. Свойство ModelState.IsValid применяется для проверки того, что данные содержат обязательные значения, и если содержат, тогда создается новый экземпляр класса AppUser, который передается асинхронному методу UserManager.CreateAsync():

```
...
AppUser user = new AppUser { UserName = model.Name, Email = model.Email };
IdentityResult result = await userManager.CreateAsync(user, model.Password);
...
```

Результатом, возвращаемым из метода CreateAsync(), будет объект IdentityResult, который описывает исход операции посредством свойств, перечисленных в табл. 28.4.

Таблица 28.4. Свойства класса IdentityResult

Имя	Описание
Succeeded	Возвращает true, если операция выполнилась успешно
Errors	Возвращает последовательность объектов IdentityError, описывающих ошибки, которые возникли при попытке выполнения операции. Класс IdentityError предлагает свойство Description со сводкой по проблеме

В методе действия Create() с помощью свойства Succeeded выясняется, была ли создана новая запись о пользователе в базе данных. Если свойство Succeeded возвращает true, тогда клиент перенаправляется на действие Index, так что отобразится список пользователей:

```
...
if (result.Succeeded) {
  return RedirectToAction("Index");
} else {
  foreach (IdentityError error in result.Errors) {
    ModelState.AddModelError("", error.Description);
  }
}
...
```

Если свойство Succeeded возвращает false, тогда производится перечисление последовательности объектов IdentityError, которую предоставляет свойство Errors. Свойство Description используется для создания ошибки проверки достоверности на уровне модели с помощью метода ModelState.AddModelError(), как объяснялось в главе 27.

Чтобы снабдить новые методы действий представлением, создадим в папке Views/Admin файл представления по имени Create.cshtml и добавим в него разметку, показанную в листинге 28.18.

Листинг 28.18. Содержимое файла Create.cshtml из папки Views/Admin

```
@model CreateModel

<div class="bg-primary m-1 p-1 text-white"><h4>Create User</h4></div>
<div asp-validation-summary=" All" class="text-danger"></div>
<form asp-action="Create" method="post">
```

```
  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Password"></label>
    <input asp-for="Password" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Create</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

В представлении `Create.cshtml` нет ничего особо примечательного — в нем присутствует простая форма для сбора значений, которые инфраструктура MVC привяжет к свойствам объекта модели, переданного методу действия `Create()`, а также сводка по возможным ошибкам проверки достоверности.

Тестирование функциональности создания

Чтобы протестировать возможность создания новой пользовательской учетной записи, запустим приложение, перейдем на URL вида `/Admin` и щелкнем на кнопке Create (Создать). Заполним форму значениями из табл. 28.5.

Таблица 28.5. Значения для создания примера пользователя

Имя	Значение
Name	Joe
Email	joe@example.com
Password	Secret123$

Совет. Существуют домены, зарезервированные для целей тестирования, в числе которых `example.com`. Полный список таких доменов доступен по адресу `https://tools.ietf.org/html/rfc2606`.

После ввода значений щелкнем на кнопке Create. Система ASP.NET Core Identity создаст пользовательскую учетную запись, которая будет отображаться после перенаправления браузера на метод действия `Index()`, что видно на рис. 28.3. (Вы получите другой идентификатор, т.к. идентификаторы для пользовательских учетных записей генерируются случайным образом.)

Щелкнем на кнопке Create еще раз и введем в элементах формы те же самые значения из табл. 28.5. На этот раз после отправки формы будет получена ошибка, о которой сообщается в сводке по проверке достоверности модели (рис. 28.4).



Рис. 28.3. Добавление новой пользовательской учетной записи

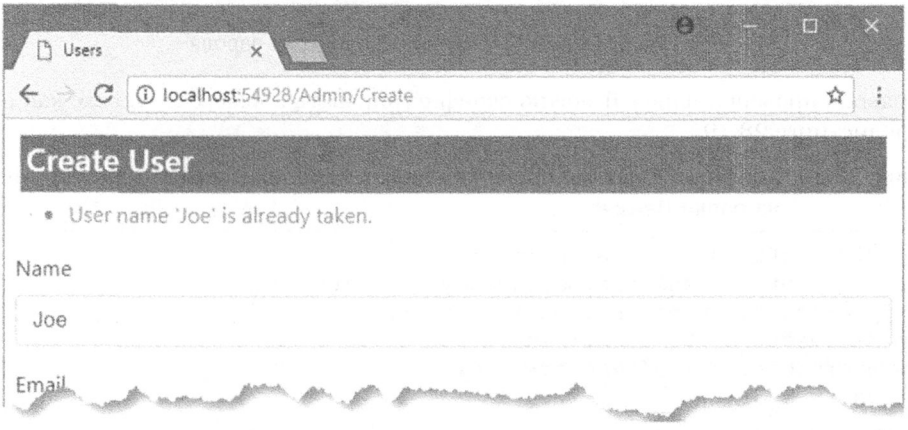


Рис. 28.4. Ошибка при попытке создать нового пользователя

Проверка паролей

Одним из наиболее распространенных требований, особенно в корпоративных приложениях, является принудительное применение политики проверки паролей. Чтобы взглянуть на стандартную политику, запустим приложение, запросим URL вида `/Admin/Create` и заполним форму данными, приведенными в табл. 28.6; их важное отличие от данных из предыдущего раздела связано со значением, вводимым в поле пароля.

Таблица 28.6. Значения для создания примера пользователя

Имя	Значение
Name	Alice
Email	alice@example.com
Password	secret

Когда форма отправляется серверу, система Identity проверяет пароль-кандидат и генерирует ошибки, если он не удовлетворяет требованиям (рис. 28.5).

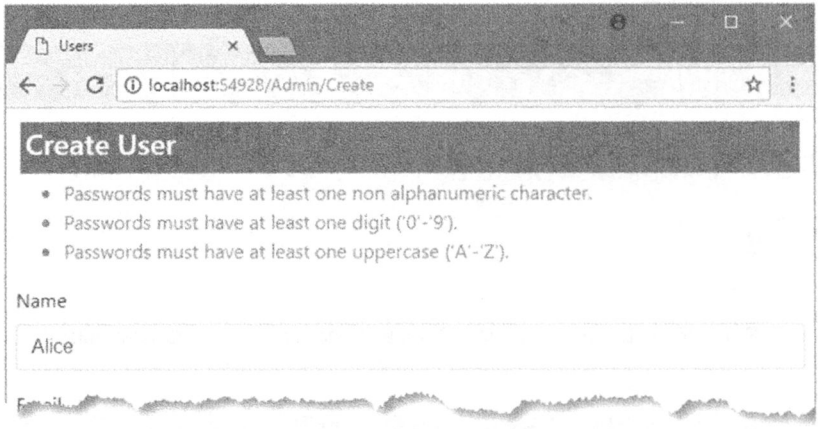


Рис. 28.5. Ошибки в результате проверки пароля

Правила проверки паролей можно сконфигурировать в классе `Startup`, как показано в листинге 28.19.

Листинг 28.19. Конфигурирование правил проверки паролей в файле `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;

namespace Users {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));
      services.AddIdentity<AppUser, IdentityRole>(opts => {
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>()
          .AddDefaultTokenProviders();
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Метод `AddIdentity()` может использоваться с функцией, принимающей объект `IdentityOptions`, свойство `Password` которого возвращает экземпляр класса `PasswordOptions`. Класс `PasswordOptions` предоставляет свойства для управления политикой проверки паролей, описанные в табл. 28.7.

Таблица 28.7. Свойства класса `PasswordOptions`

Имя	Описание
`RequiredLength`	Это свойство типа `int` применяется для указания минимальной длины паролей
`RequireNonAlphanumeric`	Установка этого свойства типа `bool` в `true` требует, чтобы пароли содержали хотя бы один символ, не являющийся буквой или цифрой
`RequireLowercase`	Установка этого свойства типа `bool` в `true` требует, чтобы пароли содержали хотя бы один символ нижнего регистра
`RequireUppercase`	Установка этого свойства типа `bool` в `true` требует, чтобы пароли содержали хотя бы один символ верхнего регистра
`RequireDigit`	Установка этого свойства типа `bool` в `true` требует, чтобы пароли содержали хотя бы один цифровой символ

В листинге 28.19 указано, что пароли должны иметь минимальную длину в шесть символов, а другие ограничения отключены. Это не должно делаться в реальном проекте, но позволяет получить эффективную демонстрацию. Запустив приложение, перейдя на URL вида `/Admin/Create` и повторив отправку формы, можно обнаружить, что пароль `secret` теперь принимается и новая пользовательская учетная запись успешно создается (рис. 28.6).

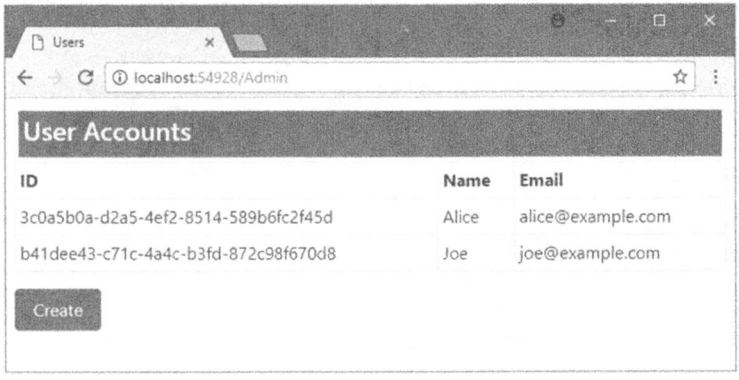


Рис. 28.6. Изменение политики проверки паролей

Реализация специального класса проверки паролей

Для большинства приложений вполне достаточно и встроенной проверки паролей, но может возникнуть необходимость в реализации специальной политики, особенно при разработке корпоративного производственного приложения, в котором сложные политики проверки паролей — обычное явление. Функциональность проверки паролей определяется интерфейсом `IPasswordValidator<T>` из пространства имен `Microsoft.AspNetCore.Identity`, где `T` — класс пользователя, специфичный для приложения (`AppUser` в рассматриваемом примере):

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Identity {
  public interface IPasswordValidator<TUser> where TUser : class {
    Task<IdentityResult> ValidateAsync(UserManager<TUser> manager,
      TUser user, string password);
  }
}
```

Для проверки пароля вызывается метод `ValidateAsync()`, которому передаются данные контекста через объект `UserManager` (позволяющий выполнять запросы к базе данных Identity), представляющий объект пользователя и пароль-кандидат. В результате возвращается объект `IdentityResult`, создаваемый с использованием статического свойства `Success`, если проблемы отсутствуют, или вызывается статический метод `Failed()`, которому передается массив объектов `IdentityError`, описывающих возникшие во время проверки проблемы.

Чтобы продемонстрировать применение специальной политики проверки, создадим папку `Infrastructure` и добавим в нее файл класса по имени `CustomPasswordValidator.cs` с определением из листинга 28.20.

Листинг 28.20. Содержимое файла `CustomPasswordValidator.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
namespace Users.Infrastructure {
  public class CustomPasswordValidator : IPasswordValidator<AppUser> {
    public Task<IdentityResult> ValidateAsync(UserManager<AppUser> manager,
          AppUser user, string password) {
      List<IdentityError> errors = new List<IdentityError>();
      if (password.ToLower().Contains(user.UserName.ToLower())) {
        errors.Add(new IdentityError {
          Code = "PasswordContainsUserName",
          Description = "Password cannot contain username"
        });
      }
      if (password.Contains("12345")) {
        errors.Add(new IdentityError {
          Code = "PasswordContainsSequence",
          Description = "Password cannot contain numeric sequence"
        });
      }
```

```
      return Task.FromResult(errors.Count == 0 ?
        IdentityResult.Success : IdentityResult.Failed(errors.ToArray()));
    }
  }
}
```

Класс `CustomPasswordValidator` проверяет, не содержит ли пароль имя пользователя или последовательность `12345`.

В листинге 28.21 класс `CustomPasswordValidator` регистрируется как средство проверки паролей для объектов `AppUser`.

Листинг 28.21. Регистрация специального класса проверки паролей в файле Startup.cs из папки Users

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;

namespace Users {

  public class Startup {
    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {

      services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();

      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));
      services.AddIdentity<AppUser, IdentityRole>(opts => {
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>()
          .AddDefaultTokenProviders();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Чтобы протестировать специальную политику, запустим приложение, запросим URL вида `/Admin/Create` и заполним форму значениями, приведенными в табл. 28.8.

Таблица 28.8. Значения для создания примера пользователя

Имя	Значение
Name	Bob
Email	bob@example.com
Password	bob12345

Пароль в табл. 28.8 нарушает оба правила проверки, навязываемые специальным классом, и вызывает отображение сообщений об ошибках, как показано на рис. 28.7.



Рис. 28.7. Использование специального класса проверки паролей

Можно также реализовать специальную политику проверки, основанную на встроенном классе, который применяется по умолчанию. Стандартный класс называется `PasswordValidator` и определен в пространстве имен `Microsoft.AspNetCore.Identity`.

В листинге 28.22 специальный класс проверки изменен так, что теперь он унаследован от класса `PasswordValidator` и построен на основе поддерживаемых им базовых проверках.

Листинг 28.22. Наследование от встроенного класса проверки в файле CustomPasswordValidator.cs из папки Infrastructure

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
using System.Linq;

namespace Users.Infrastructure {

  public class CustomPasswordValidator : PasswordValidator<AppUser> {
    public override async Task<IdentityResult> ValidateAsync(
        UserManager<AppUser> manager, AppUser user, string password) {
```

```
        IdentityResult result = await base.ValidateAsync(manager,
          user, password);
        List<IdentityError> errors = result.Succeeded ?
          new List<IdentityError>() : result.Errors.ToList();
        if (password.ToLower().Contains(user.UserName.ToLower())) {
          errors.Add(new IdentityError {
            Code = "PasswordContainsUserName",
            Description = "Password cannot contain username"
          });
        }
        if (password.Contains("12345")) {
          errors.Add(new IdentityError {
            Code = "PasswordContainsSequence",
            Description = "Password cannot contain numeric sequence"
          });
        }
        return errors.Count == 0 ? IdentityResult.Success
          : IdentityResult.Failed(errors.ToArray());
      }
    }
}
```

Для тестирования объединенной проверки запустим приложение и заполним данными из табл. 28.9 форму, возвращаемую для URL вида `/Admin/Create`.

Таблица 28.9. Значения для создания примера пользователя

Имя	Значение
Name	Bob
Email	bob@example.com
Password	12345

После отправки формы отобразится сочетание сообщений об ошибках специальной и встроенной проверки (рис. 28.8).



Рис. 28.8. Объединение специальной и встроенной проверки паролей

Проверка деталей, связанных с пользователем

При создании учетной записи проверка выполняется также в отношении имени пользователя и адреса электронной почты. Чтобы взглянуть на работу встроенной проверки, запустим приложение, запросим URL вида `/Admin/Create` и заполним форму данными из табл. 28.10.

Таблица 28.10. Значения для создания примера пользователя

Имя	Значение
`Name`	`Bob!`
`Email`	`alice@example.com`
`Password`	`secret`

В результате отправки формы будет получено сообщение об ошибке (рис. 28.9).



Рис. 28.9. Ошибка при проверке пользовательской учетной записи

Проверку можно конфигурировать в классе `Startup` с использованием свойства `IdentityOptions.User`, которое возвращает экземпляр класса `UserOptions`. Свойства `UserOptions` описаны в табл. 28.11.

Таблица 28.11. Свойства класса `UserOptions`

Имя	Описание
`AllowedUserNameCharacters`	Это свойство типа `string` содержит все разрешенные символы, которые могут применяться в имени пользователя. В стандартном значении указаны символы `a-z`, `A-Z`, `0-9`, переноса, точки, подчеркивания и `@`. Данное свойство не является регулярным выражением, и каждый разрешенный символ должен задаваться явно в строке
`RequireUniqueEmail`	Установка этого свойства типа `bool` в `true` требует, чтобы для новых учетных записей указывались адреса электронной почты, которые не использовались ранее

В листинге 28.23 конфигурация приложения изменена так, чтобы уникальные адреса электронной почты были обязательными и в именах пользователей допускались только алфавитные символы нижнего регистра.

Листинг 28.23. Изменение настроек проверки пользовательских учетных записей в файле Startup.cs из папки Users

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;

namespace Users {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();

      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));

      services.AddIdentity<AppUser, IdentityRole>(opts => {
        opts.User.RequireUniqueEmail = true;
        opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>()
          .AddDefaultTokenProviders();

      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

После повторной отправки данных из предыдущего теста легко заметить, что адрес электронной почты сейчас приводит к ошибке, а символы в имени пользователя по-прежнему отклоняются (рис. 28.10).

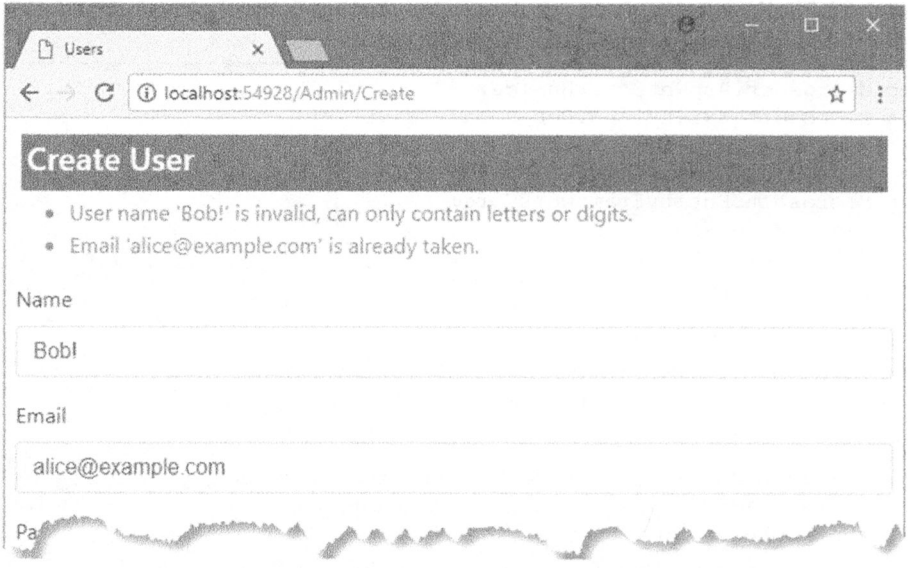


Рис. 28.10. Изменение настроек проверки пользовательских учетных записей

Реализация специальной проверки пользователей

Функциональность проверки задается с помощью интерфейса `IUserValidator<T>`, который определен в пространстве имен `Microsoft.AspNetCore.Identity`:

```
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Identity {

  public interface IUserValidator<TUser> where TUser : class {
    Task<IdentityResult> ValidateAsync(UserManager<TUser> manager,
      TUser user);
  }
}
```

Метод `ValidateAsync()` вызывается для выполнения проверки. В результате возвращается объект того же самого класса `IdentityResult`, который применялся при проверке паролей. Чтобы продемонстрировать работу специального класса проверки, добавим в папку `Infrastructure` файл класса по имени `CustomUserValidator.cs` с определением, представленным в листинге 28.24.

Листинг 28.24. Содержимое файла `CustomUserValidator.cs` из папки `Infrastructure`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;
```

```
namespace Users.Infrastructure {
  public class CustomUserValidator : IUserValidator<AppUser> {
    public Task<IdentityResult> ValidateAsync(UserManager<AppUser> manager,
        AppUser user) {
      if (user.Email.ToLower().EndsWith("@example.com")) {
        return Task.FromResult(IdentityResult.Success);
      } else {
        return Task.FromResult(IdentityResult.Failed(new IdentityError {
          Code = "EmailDomainError",
          Description = "Only example.com email addresses are allowed"
        }));
      }
    }
  }
}
```

Класс `CustomUserValidator` проверяет домен адреса электронной почты, чтобы удостовериться в том, что он является частью домена `example.com`. В листинге 28.25 специальный класс регистрируется как средство проверки для объектов `AppUser`.

Листинг 28.25. Регистрация специального класса проверки пользователей в файле `Startup.cs` из папки `Users`

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddTransient<IPasswordValidator<AppUser>,
    CustomPasswordValidator>();
  services.AddTransient<IUserValidator<AppUser>,
    CustomUserValidator>();
  services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreIdentity:ConnectionString"]));
  services.AddIdentity<AppUser, IdentityRole>(opts => {
    opts.User.RequireUniqueEmail = true;
    opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
    opts.Password.RequiredLength = 6;
    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireDigit = false;
  }).AddEntityFrameworkStores<AppIdentityDbContext>()
      .AddDefaultTokenProviders();
  services.AddMvc();
}
...
```

Для тестирования специального класса проверки запустим приложение и заполним форму, возвращаемую для URL вида `/Admin/Create`, данными из табл. 28.12.

Таблица 28.12. Значения для создания примера пользователя

Имя	Значение
Name	bob
Email	bob@invalid.com
Password	secret

Имя пользователя и пароль успешно проходят проверку, но адрес электронной почты не относится к корректному домену. В результате отправки формы отобразится сообщение об ошибке проверки (рис. 28.11).



Рис. 28.11. Выполнение специальной проверки пользователей

Процесс сочетания встроенной проверки, обеспечиваемой классом `UserValidator<T>`, и специальной проверки аналогичен такому же процессу для проверки паролей (листинг 28.26).

Листинг 28.26. Расширение встроенного класса проверки пользователей в файле `CustomUserValidator.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;

namespace Users.Infrastructure {

  public class CustomUserValidator : UserValidator<AppUser> {

    public override async Task<IdentityResult> ValidateAsync(
          UserManager<AppUser> manager,
          AppUser user) {

      IdentityResult result = await base.ValidateAsync(manager, user);

      List<IdentityError> errors = result.Succeeded ?
        new List<IdentityError>() : result.Errors.ToList();
```

```
        if (!user.Email.ToLower().EndsWith("@example.com")) {
          errors.Add(new IdentityError {
            Code = "EmailDomainError",
            Description = "Only example.com email addresses are allowed"
          });
        }
        return errors.Count == 0 ? IdentityResult.Success
          : IdentityResult.Failed(errors.ToArray());
      }
    }
}
```

Завершение построения средств администрирования

Чтобы завершить построение инструмента администрирования, осталось только реализовать средства для редактирования и удаления пользователей.

В листинге 28.27 показаны изменения, внесенные в файл `Views/Admin/Index.cshtml` для нацеливания на действия `Edit` и `Delete` контроллера `Admin`.

Листинг 28.27. Добавление кнопок редактирования и удаления в файле Index.cshtml из папки Views/Admin

```
@model IEnumerable<AppUser>

<div class="bg-primary m-1 p-1 text-white"><h4>User Accounts</h4></div>

<div class="text-danger" asp-validation-summary="ModelOnly"></div>

<table class="table table-sm table-bordered">
  <tr><th>ID</th><th>Name</th><th>Email</th></tr>
  @if (Model.Count() == 0) {
    <tr><td colspan="3" class="text-center">No User Accounts</td></tr>
  } else {
    foreach (AppUser user in Model) {
      <tr>
        <td>@user.Id</td><td>@user.UserName</td><td>@user.Email</td>
        <td>
          <form asp-action="Delete" asp-route-id="@user.Id" method="post">
            <a class="btn btn-sm btn-primary" asp-action="Edit"
               asp-route-id="@user.Id">Edit</a>
            <button type="submit"
                    class="btn btn-sm btn-danger">Delete</button>
          </form>
        </td>
      </tr>
    }
  }
</table>
<a class="btn btn-primary" asp-action="Create">Create</a>
```

Кнопка Delete (Удалить) отправляет форму действию `Delete` контроллера `Admin`, что важно, поскольку при изменении состояния приложения требуется запрос `POST`. Кнопка Edit (Редактировать) является якорным элементом, который будет отправлять запрос `GET`, т.к. первый шаг в процессе редактирования предусматривает отображение текущих данных. Кнопка Edit содержится в элементе `form`, поэтому CSS-стили Bootstrap не укладывают ее вертикально. Кроме того, в представление добавлена сводка по проверке достоверности модели, так что можно легко отображать сообщения об ошибках, которые поступают из остальных средств администрирования.

Реализация возможности удаления

В классе `UserManager<T>` определен метод `DeleteAsync()`, который принимает экземпляр класса пользователя и удаляет его из базы данных. В листинге 28.28 метод `DeleteAsync()` используется для реализации средства удаления в контроллере `Admin`.

Листинг 28.28. Удаление пользователей в файле `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;

namespace Users.Controllers {
  public class AdminController : Controller {
    private UserManager<AppUser> userManager;
    public AdminController(UserManager<AppUser> usrMgr) {
      userManager = usrMgr;
    }

    // ...для краткости другие действия не показаны...

    [HttpPost]
    public async Task<IActionResult> Delete(string id) {
      AppUser user = await userManager.FindByIdAsync(id);
      if (user != null) {
        IdentityResult result = await userManager.DeleteAsync(user);
        if (result.Succeeded) {
          return RedirectToAction("Index");
        } else {
          AddErrorsFromResult(result);
        }
      } else {
        ModelState.AddModelError("", "User Not Found");
      }
      return View("Index", userManager.Users);
    }
    private void AddErrorsFromResult(IdentityResult result) {
      foreach (IdentityError error in result.Errors) {
        ModelState.AddModelError("", error.Description);
      }
    }
  }
}
```

Метод действия `Delete()` получает в своем аргументе уникальный идентификатор пользователя и применяет метод `FindByIdAsync()` для нахождения соответствующего объекта пользователя, который можно передать методу `DeleteAsync()`. В качестве результата метод `DeleteAsync()` возвращает объект `IdentityResult`, который обрабатывается таким же образом, как в предшествующих примерах, чтобы обеспечить отображение пользователю сообщений о любых ошибках. Для тестирования функциональности удаления создадим нового пользователя и затем щелкнем на кнопке Delete, расположенной рядом с отображаемыми деталями об этом пользователе в представлении `Index` (рис. 28.12).

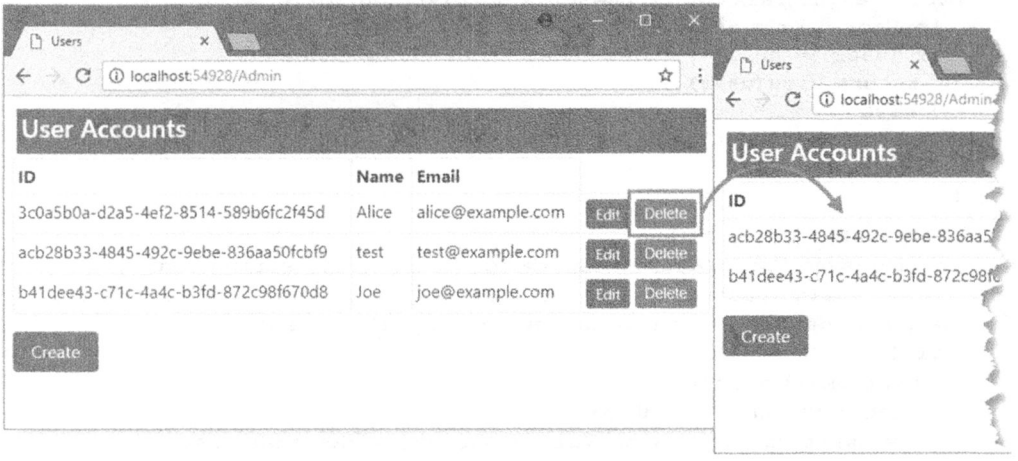


Рис. 28.12. Удаление пользовательской учетной записи

Реализация возможности редактирования

Для завершения инструмента администрирования необходимо добавить поддержку редактирования адреса электронной почты и пароля, которые связаны с пользовательской учетной записью. В настоящий момент имеются только свойства, определяемые пользователями, но в главе 30 будет показано, как расширить схему специальными свойствами. В листинге 28.29 приведен код методов действий `Edit()`, добавленных в контроллер `Admin`.

Листинг 28.29. Добавление действий `Edit` в файле `AdminController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;

namespace Users.Controllers {

  public class AdminController : Controller {
    private UserManager<AppUser> userManager;
    private IUserValidator<AppUser> userValidator;
    private IPasswordValidator<AppUser> passwordValidator;
    private IPasswordHasher<AppUser> passwordHasher;
```

```
public AdminController(UserManager<AppUser> usrMgr,
       IUserValidator<AppUser> userValid,
       IPasswordValidator<AppUser> passValid,
       IPasswordHasher<AppUser> passwordHash) {
  userManager = usrMgr;
  userValidator = userValid;
  passwordValidator = passValid;
  passwordHasher = passwordHash;
}

// ...для краткости другие действия не показаны...

public async Task<IActionResult> Edit(string id) {
  AppUser user = await userManager.FindByIdAsync(id);
  if (user != null) {
    return View(user);
  } else {
    return RedirectToAction("Index");
  }
}

[HttpPost]
public async Task<IActionResult> Edit(string id, string email,
       string password) {
  AppUser user = await userManager.FindByIdAsync(id);
  if (user != null) {
    user.Email = email;
    IdentityResult validEmail
      = await userValidator.ValidateAsync(userManager, user);
    if (!validEmail.Succeeded) {
      AddErrorsFromResult(validEmail);
    }
    IdentityResult validPass = null;
    if (!string.IsNullOrEmpty(password)) {
      validPass = await passwordValidator.ValidateAsync(userManager,
        user, password);
      if (validPass.Succeeded) {
        user.PasswordHash = passwordHasher.HashPassword(user,
          password);
      } else {
        AddErrorsFromResult(validPass);
      }
    }
    if ((validEmail.Succeeded && validPass == null)
        || (validEmail.Succeeded
        && password != string.Empty && validPass.Succeeded)) {
      IdentityResult result = await userManager.UpdateAsync(user);
      if (result.Succeeded) {
        return RedirectToAction("Index");
      } else {
        AddErrorsFromResult(result);
      }
    }
  } else {
```

```
            ModelState.AddModelError("", "User Not Found");
        }
        return View(user);
    }

    private void AddErrorsFromResult(IdentityResult result) {
        foreach (IdentityError error in result.Errors) {
            ModelState.AddModelError("", error.Description);
        }
    }
}
}
```

Действие Edit, на которое нацелены запросы GET, использует строку идентификатора, встроенную в представление Index, чтобы вызвать метод FindByIdAsync() для получения объекта AppUser, представляющего пользователя.

Более сложная реализация действия Edit получает запрос POST и имеет аргументы для идентификатора пользователя, нового адреса электронной почты и пароля. Завершение операции редактирования требует выполнения нескольких задач.

Первая задача связана с проверкой полученных значений. В настоящий момент работа ведется с простым объектом пользователя (хотя в главе 30 объясняется, как настраивать данные, сохраняемые для пользователей), но даже в таком случае нужно проверить пользовательские данные, чтобы обеспечить соблюдение специальных политик, которые были определены в разделах "Проверка паролей" и "Проверка деталей, связанных с пользователем" ранее в главе. Сначала проверяется адрес электронной почты:

```
...
user.Email = email;
IdentityResult validEmail = await userValidator.
ValidateAsync(userManager, user);
if (!validEmail.Succeeded) {
    AddErrorsFromResult(validEmail);
}
...
```

К конструктору контроллера добавлена зависимость от IUserValidator<AppUser>, чтобы можно было проверять новый адрес электронной почты. Обратите внимание, что перед выполнением проверки значение свойства Email должно быть изменено, т.к. метод ValidateAsync() принимает только экземпляры класса пользователя.

Вторая задача связана с изменением пароля, если он был предоставлен. Система ASP.NET Core Identity сохраняет хеши паролей, а не сами пароли. Целью является препятствование похищению паролей. Необходимо получить проверенный пароль и сгенерировать хеш-код, который будет сохранен в базе данных, чтобы пользователь мог быть аутентифицирован, как демонстрируется в главе 29.

Пароли преобразуются в хеши через реализацию интерфейса IPasswordHasher <AppUser>, которая получается за счет объявления зависимости конструктора, распознаваемой посредством внедрения зависимостей. В интерфейсе IPasswordHasher определен метод HashPassword(), который принимает строковый аргумент и возвращает его хешированное значение:

```
...
if (!string.IsNullOrEmpty(password)) {
    validPass = await passwordValidator.ValidateAsync(userManager, user,
        password);
```

```
    if (validPass.Succeeded) {
      user.PasswordHash = passwordHasher.HashPassword(user, password);
    } else {
      AddErrorsFromResult(validPass);
    }
  }
  ...
```

Изменения, внесенные в класс пользователя, не сохраняются в базе данных до тех пор, пока не будет вызван метод `UpdateAsync()`:

```
  ...
  if ((validEmail.Succeeded && validPass == null) || (validEmail.Succeeded
      && password != string.Empty && validPass.Succeeded)) {
    IdentityResult result = await userManager.UpdateAsync(user);
    if (result.Succeeded) {
      return RedirectToAction("Index");
    } else {
      AddErrorsFromResult(result);
    }
  }
  ...
```

Создание представления

Финальным компонентом является представление, которое будет отображать текущие значения для пользователя и позволит отправлять контроллеру новые значения. Добавим в папку `Views/Admin` файл по имени `Edit.cshtml` с содержимым, приведенным в листинге 28.30.

Листинг 28.30. Содержимое файла `Edit.cshtml` из папки `Views/Admin`

```
@model AppUser

<div class="bg-primary m-1 p-1"><h4>Edit User</h4></div>

<div asp-validation-summary="All" class="text-danger"></div>

<form asp-action="Edit" method="post">
  <div class="form-group">
    <label asp-for="Id"></label>
    <input asp-for="Id" class="form-control" disabled />
  </div>
  <div class="form-group">
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input name="password" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Представление отображает идентификатор пользователя, который не может быть изменен, как статический текст и предлагает форму редактирования адреса элект-

ронной почты и пароля (рис. 28.13). Обратите внимание, что для элементов пароля вспомогательная функция дескриптора не применяется, т.к. класс пользователя не содержит информации о пароле, поскольку в базе данных хранятся только хешированные значения.

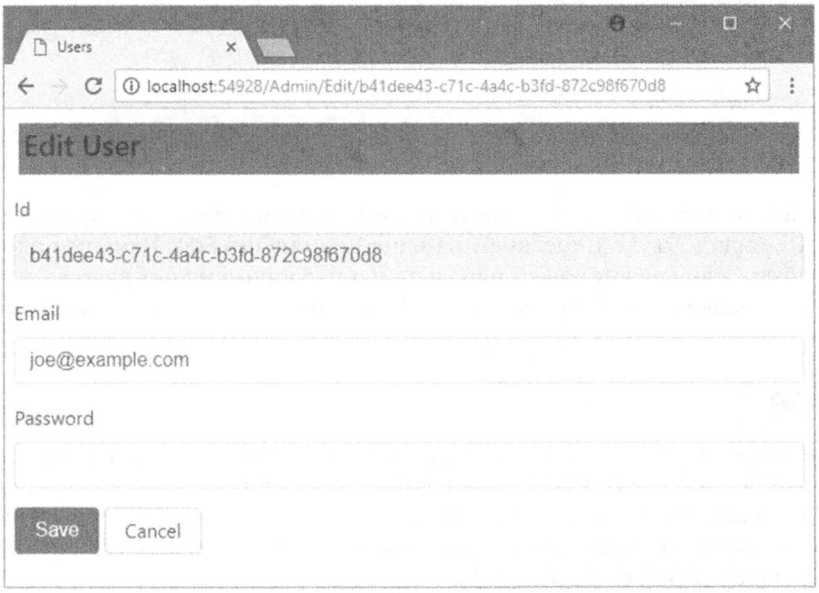


Рис. 28.13. Редактирование пользовательской учетной записи

Осталось закомментировать настройки проверки пользователей в классе `Startup`, чтобы для имен пользователей использовались стандартные символы (листинг 28.31). Ввиду того, что некоторые учетные записи в базе данных были созданы до изменения настроек проверки, отредактировать их не удастся, т.к. имена пользователей не пройдут проверку. А поскольку проверка применяется ко всему объекту пользователя, когда проверяется адрес электронной почты, результатом оказывается пользовательская учетная запись, которую невозможно изменить.

Листинг 28.31. Отключение настроек проверки пользователей в файле `Startup.cs` из папки `Users`

```
...
public void ConfigureServices(IServiceCollection services) {

  services.AddTransient<IPasswordValidator<AppUser>,
    CustomPasswordValidator>();
  services.AddTransient<IUserValidator<AppUser>,
    CustomUserValidator>();

  services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreIdentity:ConnectionString"]));

  services.AddIdentity<AppUser, IdentityRole>(opts => {
    opts.User.RequireUniqueEmail = true;
```

```
    // opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
    opts.Password.RequiredLength = 6;
    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireDigit = false;
  }).AddEntityFrameworkStores<AppIdentityDbContext>()
      .AddDefaultTokenProviders();

  services.AddMvc();
}
...
```

Чтобы протестировать возможности редактирования, запустим приложение, запросим URL вида `/Admin` и щелкнем на одной из кнопок Edit. Изменим адрес электронной почты или введем новый пароль (либо предпримем то и другое), после чего щелкнем на кнопке Save (Сохранить) для обновления базы данных и возвращения к URL вида `/Admin`.

Резюме

В главе было показано, как создавать конфигурацию и классы, требующиеся для использования системы ASP.NET Core Identity, а также продемонстрировано, каким образом их можно применять для создания инструмента администрирования пользователей. В следующей главе вы узнаете, как выполнять аутентификацию и авторизацию с помощью ASP.NET Core Identity.

ГЛАВА 29

Применение ASP.NET Core Identity

В настоящей главе будет показано, как применять систему ASP.NET Core Identity для аутентификации и авторизации пользовательских учетных записей, созданных в предыдущей главе. В табл. 29.1 приведена сводка по главе.

Таблица 29.1. Сводка по главе

Задача	Решение	Листинг
Ограничение доступа к методу действия	Применяйте атрибут `Authorize`	29.1
Аутентификация пользователей	Создайте контроллер `Account`, который получает пользовательские учетные данные и проверяет их с использованием класса `UserManager`	29.2–29.5
Создание и управление ролями	Используйте класс `RoleManager`	29.6–29.10
Авторизация доступа к действию	Добавьте пользовательские учетные записи к ролям и применяйте атрибут `Authorize`, чтобы указать, какие роли могут иметь доступ к методам действий	29.11–29.18
Обеспечение наличия учетной записи администратора	Поместите в базу данных начальные данные для автоматического создания учетной записи	29.19–29.24

Подготовка проекта для примера

В главе будет продолжена работа с проектом `Users`, созданным в главе 28. В качестве подготовительных шагов запустим приложение, перейдем на URL вида `/Admin` и, щелкая на кнопке Create (Создать), добавим в базу данных пользовательские учетные записи из табл. 29.2.

Таблица 29.2. Пользовательские учетные записи, требующиеся для настоящей главы

Имя пользователя	Адрес электронной почты	Пароль
`Joe`	`joe@example.com`	`secret123`
`Alice`	`alice@example.com`	`secret123`
`Bob`	`bob@example.com`	`secret123`

По завершении запрос URL вида `/Admin` должен привести к отображению списка пользователей, включая описанные в табл. 29.2 (не имеет значения, если вы создадите дополнительных пользователей; важно, чтобы присутствовали пользователи, перечисленные в таблице), как показано на рис. 29.1.

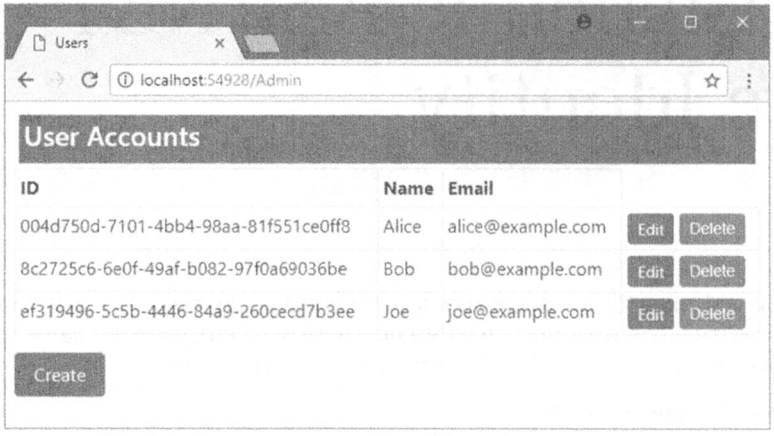


Рис. 29.1. Выполнение примера приложения

Аутентификация пользователей

Наиболее фундаментальной работой для системы ASP.NET Core Identity является аутентификация пользователей. Основной инструмент для ограничения доступа к методам действий — атрибут `Authorize`, который сообщает инфраструктуре MVC о том, что обрабатываться должны только запросы от аутентифицированных пользователей. В листинге 29.1 атрибут `Authorize` применяется к действию `Index` контроллера `Home`.

Листинг 29.1. Ограничение доступа в файле `HomeController.cs` из папки `Controllers`

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

  public class HomeController : Controller {

    [Authorize]
    public ViewResult Index() =>
      View(new Dictionary<string, object> { ["Placeholder"] = "Placeholder" });
  }
}
```

После запуска приложения браузер отправит запрос на стандартный URL, который будет нацелен на метод действия, декорированный атрибутом `Authorize`. Пока что у пользователей нет никакой возможности аутентифицировать себя, поэтому в результате возникает ошибка (рис. 29.2).

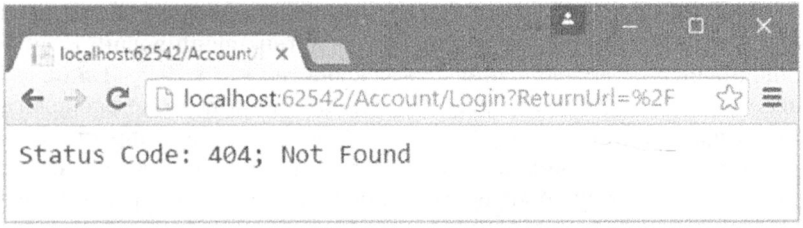


Рис. 29.2. Нацеливание на защищенный метод действия

Атрибут `Authorize` не указывает, как пользователь должен быть аутентифицирован, и не имеет прямой ссылки на ASP.NET Core Identity. Службы Identity и промежуточное ПО охватывают всю платформу ASP.NET Core, что делает их интеграцию в приложения MVC простой и бесшовной. Задача решается модификацией объектов контекста, которые описывают HTTP-запросы, и снабжением инфраструктуры MVC исходом процесса аутентификации без необходимости в предоставлении любых других деталей.

Платформа ASP.NET предоставляет информацию о пользователе через объект `HttpContext`, который используется атрибутом `Authorize` для проверки состояния текущего запроса и выяснения, был ли пользователь аутентифицирован. Свойство `HttpContext.User` возвращает реализацию интерфейса `IPrincipal`, который определен в пространстве имен `System.Security.Principal`. Интерфейс `IPrincipal` определяет свойство и метод, показанные в табл. 29.3.

Таблица 29.3. Избранные члены, определяемые интерфейсом `IPrincipal`

Имя	Описание
`Identity`	Возвращает реализацию интерфейса `IIdentity`, который описывает пользователя, ассоциированного с запросом
`IsInRole(role)`	Возвращает `true`, если пользователь является членом указанной роли. Управление авторизацией с помощью ролей объясняется в разделе "Авторизация пользователей с помощью ролей" далее в главе

Реализация интерфейса `IIdentity`, возвращаемая свойством `IPrincipal.Identity`, предлагает базовую, но полезную информацию о текущем пользователе через свойства, которые описаны в табл. 29.4.

Таблица 29.4. Избранные свойства, определяемые интерфейсом `IIdentity`

Имя	Описание
`AuthenticationType`	Возвращает строку, которая описывает механизм, используемый для аутентификации пользователя
`IsAuthenticated`	Возвращает `true`, если пользователь был аутентифицирован
`Name`	Возвращает имя текущего пользователя

Совет. В главе 30 рассматривается класс реализации, который ASP.NET Core Identity применяет для интерфейса `IIdentity`.

Промежуточное ПО ASP.NET Core Identity применяет cookie-наборы, посылаемые браузером, для выяснения, был ли пользователь аутентифицирован. Если пользователь успешно прошел аутентификацию, тогда свойство `IIdentity.IsAuthenticated` устанавливается в `true`. Поскольку пример приложения пока не располагает механизмом аутентификации, свойство `IsAuthenticated` всегда возвращает `false`, что приводит к ошибке аутентификации. В результате клиент перенаправляется на URL вида `/Account/Login`, который является стандартным URL для предоставления учетных данных аутентификации.

Браузер запрашивает URL вида `/Account/Login`, но из-за того, что в проекте он не соответствует какому-либо контроллеру или действию, сервер возвращает ответ `404 - Not Found` (404 — не найдено), давая в итоге сообщение об ошибке, показанное на рис. 29.2.

Изменение URL для входа

Хотя `/Account/Login` — стандартный URL, на который клиенты перенаправляются, когда требуется авторизация, в методе `ConfigureServices()` класса `Startup` можно указать собственный URL, изменив параметр конфигурации при настройке служб ASP.NET Core Identity:

```
...
services.ConfigureApplicationCookie(opts =>
  opts.LoginPath = "/Users/Login");
...
```

При генерации своих URL система Identity не может полагаться на систему маршрутизации, так что цель перенаправления должна указываться буквально. В случае изменения схемы маршрутизации, используемой приложением, потребуется также обеспечить изменение настройки Identity, чтобы URL по-прежнему достигал целевого контроллера.

Подготовка к реализации аутентификации

Несмотря на то что запрос заканчивается выводом сообщения об ошибке, он иллюстрирует, каким образом система ASP.NET Core Identity вписывается в стандартный жизненный цикл запросов ASP.NET. Следующий шаг заключается в реализации контроллера, который будет получать запросы для URL вида `/Account/Login` и аутентифицировать пользователя. Добавим новый класс модели в файл `UserViewModels.cs` (листинг 29.2).

Листинг 29.2. Добавление нового класса модели в файле `UserViewModels.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;

namespace Users.Models {

  public class CreateModel {
    [Required]
    public string Name { get; set; }
    [Required]
    public string Email { get; set; }
    [Required]
    public string Password { get; set; }
  }
```

```
  public class LoginModel {
    [Required]
    [UIHint("email")]
    public string Email { get; set; }
    [Required]
    [UIHint("password")]
    public string Password { get; set; }
  }
}
```

Новый класс модели имеет свойства `Email` и `Password`, декорированные атрибутом `Required`, что позволяет применять проверку достоверности моделей для контроля, предоставил ли пользователь значения. Свойства также декорированы атрибутом `UIHint`, который гарантирует, что элементы `input`, визуализируемые вспомогательной функцией дескриптора в представлении, будут иметь соответствующим образом установленные атрибуты `type`.

Совет. В реальном проекте для выяснения, предоставил ли пользователь значения для имени и пароля, перед отправкой формы серверу можно использовать проверку достоверности на стороне клиента, которая была описана в главе 27.

Добавим в папку `Controllers` файл класса по имени `AccountController.cs` и поместим в него определение контроллера, приведенное в листинге 29.3.

Листинг 29.3. Содержимое файла `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;

namespace Users.Controllers {
  [Authorize]
  public class AccountController : Controller {
    [AllowAnonymous]
    public IActionResult Login(string returnUrl) {
      ViewBag.returnUrl = returnUrl;
      return View();
    }
    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginModel details,
          string returnUrl) {
      return View(details);
    }
  }
}
```

В листинге 29.3 логика аутентификации не была реализована, потому что мы собираемся определить представление и затем пройти через процесс проверки пользовательских учетных данных и входа пользователей в приложение.

Хотя контроллер Account пока еще не аутентифицирует пользователей, он содержит удобную инфраструктуру, заслуживающую объяснения отдельно от кода ASP.NET Core Identity, который вскоре будет добавлен в метод действия Login().

Первым делом обратите внимание, что обе версии метода действия Login() принимают аргумент по имени returnUrl. Когда пользователь запрашивает ограниченный URL, он перенаправляется на URL вида /Account/Login со строкой запроса, в которой указан URL, куда пользователь должен быть направлен после того, как он успешно пройдет аутентификацию. Удостовериться в этом можно, запустив приложение и запросив URL вида /Home/Index. Браузер будет перенаправлен примерно так:

```
/Account/Login?ReturnUrl=%2FHome%2FIndex
```

Значение параметра ReturnUrl строки запроса делает возможным такое перенаправление пользователя, что навигация между открытыми и защищенными частями приложения превращается в простой и гладкий процесс.

Далее обратите внимание на атрибуты, которые были применены в контроллере Account. Контроллеры, управляющие пользовательскими учетными записями, содержат функциональность, которая должна быть доступна только аутентифицированным пользователям, подобную сбросу пароля. С этой целью к классу контроллера был применен атрибут Authorize, а к индивидуальным методам действий — атрибут AllowAnonymous. В итоге доступ к методам действий по умолчанию ограничивается аутентифицированными пользователями, но пользователям, не прошедшим аутентификацию, разрешено входить в приложение. Кроме того, применяется описанный в главе 24 атрибут ValidateAntiForgeryToken, который работает в сочетании со вспомогательной функцией дескриптора для элемента form в целях противодействия подделке межсайтовых запросов.

Последний подготовительный шаг связан с созданием представления, которое будет визуализироваться для сбора учетных данных от пользователя. Создадим папку Views/Account и добавим в нее файл представления по имени Login.cshtml с разметкой из листинга 29.4.

Листинг 29.4. Содержимое файла Login.cshtml из папки Views/Account

```
@model LoginModel

<div class="bg-primary m-1 p-1 text-white"><h4>Log In</h4></div>

<div class="text-danger" asp-validation-summary="All"></div>

<form asp-action="Login" method="post">
  <input type="hidden" name="returnUrl" value="@ViewBag.returnUrl" />
  <div class="form-group">
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Password"></label>
    <input asp-for="Password" class="form-control" />
  </div>
  <button class="btn btn-primary" type="submit">Log In</button>
</form>
```

Единственный примечательный аспект данного представления — скрытый элемент input, который сохраняет аргумент returnUrl. Во всех остальных отношениях

это стандартное представление Razor, но оно завершает подготовку к аутентификации и демонстрирует способ перехвата и перенаправления запросов, не прошедших аутентификацию. Запустим приложение, чтобы протестировать новый контроллер. Когда браузер запрашивает стандартный URL приложения, он перенаправляется на URL вида `/Account/Login`, что выдает содержимое, показанное на рис. 29.3.

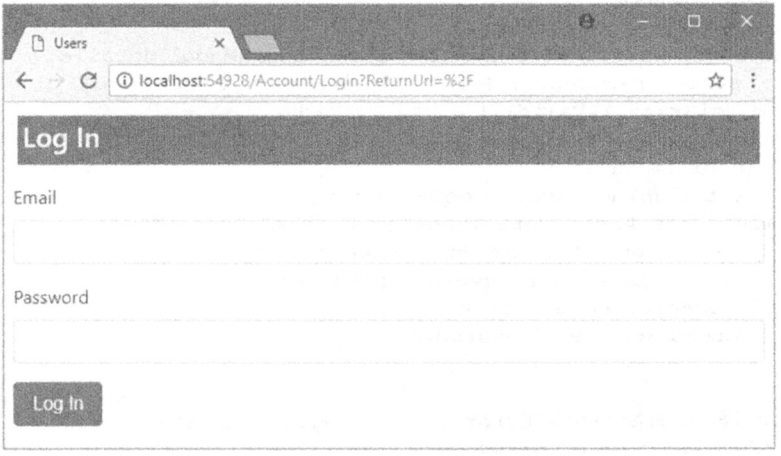


Рис. 29.3. Вывод пользователю приглашения предоставить свои учетные данные

Добавление аутентификации пользователей

Запросы к защищенным методам действий корректно перенаправляются контроллеру `Account`, но учетные данные, предоставляемые пользователем, пока еще не используются для аутентификации. В листинге 29.5 завершается реализация действия `Login` за счет применения служб ASP.NET Core Identity для аутентификации пользователя с участием деталей, хранящихся в базе данных.

Листинг 29.5. Добавление аутентификации в файле `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;

    public AccountController(UserManager<AppUser> userMgr,
            SignInManager<AppUser> signinMgr) {
      userManager = userMgr;
      signInManager = signinMgr;
    }
```

```
    [AllowAnonymous]
    public IActionResult Login(string returnUrl) {
      ViewBag.returnUrl = returnUrl;
      return View();
    }
    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginModel details,
            string returnUrl) {
      if (ModelState.IsValid) {
        AppUser user = await userManager.FindByEmailAsync(details.Email);
        if (user != null) {
          await signInManager.SignOutAsync();
          Microsoft.AspNetCore.Identity.SignInResult result =
            await signInManager.PasswordSignInAsync(
              user, details.Password, false, false);
          if (result.Succeeded) {
            return Redirect(returnUrl ?? "/");
          }
        }
        ModelState.AddModelError(nameof(LoginModel.Email),
          "Invalid user or password");
      }
      return View(details);
    }
  }
}
```

Простейшей частью является получение объекта AppUser, который представляет пользователя, что делается посредством метода FindByEmailAsync() класса UserManager<AppUser>:

```
...
AppUser user = await userManager.FindByEmailAsync(details.Email);
...
```

Метод FindByEmailAsync() находит пользовательскую учетную запись, используя адрес электронной почты, который применялся при ее создании. Есть также альтернативные методы поиска по идентификатору, по имени и по входу. Адрес электронной почты используется для входа из-за того, что такой подход принят в большинстве веб-приложений, доступных через Интернет, и он набирает популярность также в корпоративных приложениях.

Если учетная запись с указанным пользователем адресом электронной почты существует, тогда производится аутентификация с применением класса SignInManager<AppUser>, для которого добавляется аргумент конструктора, распознаваемый с помощью внедрения зависимостей. Класс SignInManager используется для выполнения двух шагов аутентификации:

```
...
await signInManager.SignOutAsync();
Microsoft.AspNetCore.Identity.SignInResult result =
  await signInManager.PasswordSignInAsync(user, details.Password,
    false, false);
...
```

Метод `SignOutAsync()` аннулирует любой имеющийся у пользователя сеанс, а метод `PasswordSignIn()` проводит саму аутентификацию. В качестве аргументов метод `PasswordSignInAsync()` получает объект пользователя, предоставленный пользователем пароль, булевское значение, управляющее постоянством cookie-набора аутентификации (отключено), и признак, должна ли учетная запись блокироваться в случае некорректного пароля (отключено). Результатом метода `PasswordSignInAsync()` будет объект `SignInResult`, в котором определено булевское свойство `Succeeded`, указывающее на успешность аутентификации.

В рассматриваемом примере проверяется свойство `Succeeded`; если оно равно `true`, то пользователь перенаправляется на местоположение `returnUrl`, а если `false`, тогда добавляется ошибка проверки достоверности и затем представление `Login` отображается заново, чтобы пользователь смог повторить попытку.

Как часть процесса аутентификации система Identity добавляет к ответу cookie-набор, который браузер затем включает в любые последующие запросы, чтобы идентифицировать сеанс пользователя и ассоциированную с ним учетную запись. Вы не обязаны создавать или управлять этим cookie-набором напрямую, т.к. он поддерживается автоматически промежуточным ПО Identity.

Учет двухфакторной аутентификации

В настоящей главе выполнялась однофакторная аутентификация, при которой пользователь может быть аутентифицирован с применением одиночной порции информации, известной ему заранее: пароля.

Система ASP.NET Core Identity поддерживает также двухфакторную аутентификацию, при которой пользователю нужно кое-что дополнительное, обычно получаемое в момент, когда он желает пройти аутентификацию. Наиболее распространенным примером может служить значение из маркера SecureID или код аутентификации, который отправляется в виде сообщения электронной почты либо текстового сообщения. (Строго говоря, в качестве второго фактора может выступать что угодно, включая снятие отпечатков пальцев, сканирование радужной оболочки глаза и распознавание голоса, хотя в большинстве веб-приложений такие варианты востребованы редко.)

Защита в итоге усиливается, потому что злоумышленнику необходимо знать пароль пользователя *и* иметь доступ к тому, что предоставляется как второй фактор, например, к учетной записи электронной почты или сотовому телефону.

Двухфакторная аутентификация в книге не рассматривается по двум причинам. Во-первых, она требует большой подготовительной работы, связанной с настройкой инфраструктуры, которая распространяет сообщения электронной почты и текстовые сообщения второго фактора, а также реализации логики проверки, что выходит за рамки тематики настоящей книги.

Во-вторых, двухфакторная аутентификация вынуждает пользователя помнить о необходимости прохождения второго шага аутентификации, для чего держать поблизости, например, сотовый телефон или маркер безопасности, что в случае веб-приложений не всегда оказывается подходящим. Я более десяти лет на различных работах проносил с собой маркер SecureID того или иного вида и потерял счет, сколько раз не мог войти в систему работодателя из-за того, что забывал свой маркер дома.

Если вы заинтересованы в двухфакторной аутентификации, тогда рекомендуется опираться на стороннего поставщика вроде Google, который позволяет пользователю самостоятельно выбрать, желает ли он иметь дополнительную защиту (и смириться с неудобствами), обеспечиваемую двухфакторной аутентификацией. Использование сторонней аутентификации демонстрируется в главе 30.

Тестирование аутентификации

Чтобы протестировать аутентификацию пользователей, запустим приложение и запросим URL вида `/Home/Index`. После перенаправления на URL вида `/Account/Login` введем учетные данные одного из пользователей, перечисленных в начале главы (например, адрес электронной почты `joe@example.com` и пароль `secret123`). Щелкнем на кнопке Log In (Вход) и браузер будет перенаправлен обратно на `/Home/Index`, но на этот раз он отправит cookie-набор аутентификации, который предоставит доступ к методу действия (рис. 29.4).

Совет. Для просмотра cookie-наборов, применяемых при идентификации аутентифицированных запросов, можно использовать инструменты разработчика, встроенные в браузер.

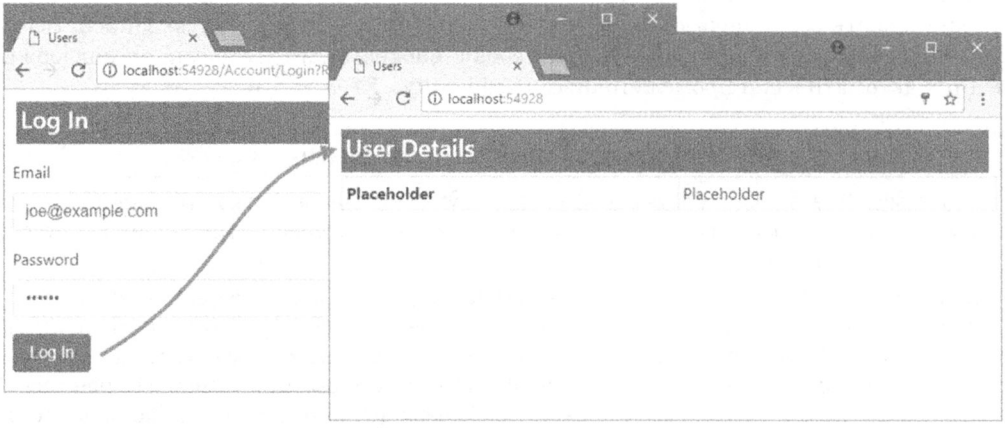


Рис. 29.4. Аутентификация пользователя

Авторизация пользователей с помощью ролей

В предыдущем разделе атрибут `Authorize` применялся в самой базовой форме, которая позволяет любому аутентифицированному пользователю выполнять метод действия. Его также можно использовать для уточнения авторизации, чтобы получить более детальный контроль над тем, какие пользователи могут выполнять те или иные действия, основываясь на принадлежности пользователя к роли.

Роль — всего лишь произвольная метка, которая определяется с целью представления разрешения на выполнение набора действий внутри приложения. Практически каждое приложение проводит различие между пользователями, которые могут выполнять административные функции, и пользователями, которые не могут. В мире ролей цель достигается за счет создания и назначения пользователям роли `Administrators`. Пользователи могут принадлежать ко многим ролям, а связанные с ролями разрешения могут быть настолько крупнозернистыми или мелкозернистыми, насколько это желательно. Таким образом, с применением разных ролей можно проводить различие между администраторами, которым позволено выполнять базовые задачи, подобные созданию новых учетных записей, и администраторами, которым разрешено выполнять более критичные операции вроде доступа к данным о платежах.

Система ASP.NET Core Identity берет на себя ответственность за управление набором ролей, определенных в приложении, и отслеживание членства пользователей в них. Но ей ничего не известно о том, что означает каждая роль; такая информация содержится внутри части MVC приложения, где доступ к методам действий ограничивается на основе членства в ролях.

Для доступа и управления ролями в ASP.NET Core Identity предусмотрен строго типизированный базовый класс по имени `RoleManager<T>`, где `T` — класс, который представляет роли в механизме хранения. Инфраструктура Entity Framework Core использует для представления ролей класс `IdentityRole`, в котором определены свойства, перечисленные в табл. 29.5.

Таблица 29.5. Избранные свойства класса `IdentityRole`

Имя	Описание
`Id`	Определяет уникальный идентификатор для роли
`Name`	Определяет имя роли
`Users`	Возвращает коллекцию объектов `IdentityUserRole`, которые представляют члены роли

При желании расширить встроенную функциональность, которая описана в главе 30 для объектов пользователей, можно создать класс роли, специфичный для приложения, но здесь будет применяться класс `IdentityRole`, т.к. он делает все, в чем нуждается большинство приложений. Когда конфигурировалось приложение в главе 28, системе ASP.NET Core Identity уже было указано на необходимость использования класса `IdentityRole` для представления ролей, что демонстрирует следующий оператор в методе `ConfigureServices()` класса `Startup`:

```
...
services.AddIdentity<AppUser, IdentityRole>(opts => {
  opts.User.RequireUniqueEmail = true;
  // opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
  opts.Password.RequiredLength = 6;
  opts.Password.RequireNonAlphanumeric = false;
  opts.Password.RequireLowercase = false;
  opts.Password.RequireUppercase = false;
  opts.Password.RequireDigit = false;
}).AddEntityFrameworkStores<AppIdentityDbContext>()
    .AddDefaultTokenProviders();
...
```

Параметры типов в методе `AddIdentity()` указывают классы, которые будут применяться для представления пользователей и ролей. В примере приложения для представления пользователей используется класс `AppUser`, а для представления ролей — встроенный класс `IdentityRole`.

Создание и удаление ролей

Чтобы продемонстрировать применение ролей, мы создадим инструмент администрирования для управления ими, начав с методов действий, которые могут создавать и удалять роли. Добавим в папку `Controllers` файл класса по имени `RoleAdminController.cs` и определим в нем контроллер, как показано в листинге 29.6.

Листинг 29.6. Содержимое файла `RoleAdminController.cs` из папки `Controllers`

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc;

namespace Users.Controllers {
  public class RoleAdminController : Controller {
    private RoleManager<IdentityRole> roleManager;
    public RoleAdminController(RoleManager<IdentityRole> roleMgr) {
      roleManager = roleMgr;
    }
    public ViewResult Index() => View(roleManager.Roles);
    public IActionResult Create() => View();
    [HttpPost]
    public async Task<IActionResult> Create([Required]string name) {
      if (ModelState.IsValid) {
        IdentityResult result
          = await roleManager.CreateAsync(new IdentityRole(name));
        if (result.Succeeded) {
          return RedirectToAction("Index");
        } else {
          AddErrorsFromResult(result);
        }
      }
      return View(name);
    }
    [HttpPost]
    public async Task<IActionResult> Delete(string id) {
      IdentityRole role = await roleManager.FindByIdAsync(id);
      if (role != null) {
        IdentityResult result = await roleManager.DeleteAsync(role);
        if (result.Succeeded) {
          return RedirectToAction("Index");
        } else {
          AddErrorsFromResult(result);
        }
      } else {
        ModelState.AddModelError("", "No role found");
      }
      return View("Index", roleManager.Roles);
    }
    private void AddErrorsFromResult(IdentityResult result) {
      foreach (IdentityError error in result.Errors) {
        ModelState.AddModelError("", error.Description);
      }
    }
  }
}
```

Управление ролями производится с использованием класса `RoleManager<T>`, где `T` — тип, предназначенный для представления ролей (встроенный класс `IdentityRole` в текущем приложении). Конструктор `RoleAdminController` объявляет зависимость

от RoleManager<IdentityRole>, распознаваемую посредством внедрения зависимостей при создании объекта контроллера.

В классе RoleManager<T> определены методы и свойства, перечисленные в табл. 29.6, которые позволяют создавать и управлять ролями.

Таблица 29.6. Члены, определяемые классом RoleManager<T>

Имя	Описание
CreateAsync(role)	Создает новую роль
DeleteAsync(role)	Удаляет указанную роль
FindByIdAsync(id)	Находит роль по ее идентификатору
FindByNameAsync(name)	Находит роль по ее имени
RoleExistsAsync(name)	Возвращает true, если роль с указанным именем существует
UpdateAsync(role)	Сохраняет изменения в указанной роли
Roles	Возвращает перечисление ролей, которые были определены

Метод действия Index() нового контроллера отображает все роли в приложении. Метод действия Create() применяется для отображения формы и получения ее данных, которые используются при создании новой роли с помощью метода CreateAsync(). Метод действия Delete() получает запрос POST и принимает уникальный идентификатор роли, который применяется для ее удаления из приложения через метод DeleteAsync(), находя объект роли с применением метода FindByIdAsync().

Создание представлений

Чтобы отобразить детали ролей в приложении, создадим папку Views/RoleAdmin и добавим в нее файл Index.cshtml с разметкой из листинга 29.7.

Листинг 29.7. Содержимое файла Index.cshtml из папки Views/RoleAdmin

```
@model IEnumerable<IdentityRole>

<div class="bg-primary m-1 p-1"><h4>Roles</h4></div>

<div class="text-danger" asp-validation-summary="ModelOnly"></div>

<table class="table table-sm table-bordered table-bordered">
  <tr><th>ID</th><th>Name</th><th>Users</th><th></th></tr>
  @if (Model.Count() == 0) {
    <tr><td colspan="4" class="text-center">No Roles</td></tr>
  } else {
    foreach (var role in Model) {
      <tr>
        <td>@role.Id</td>
        <td>@role.Name</td>
        <td identity-role="@role.Id"></td>
        <td>
          <form asp-action="Delete" asp-route-id="@role.Id" method="post">
            <a class="btn btn-sm btn-primary" asp-action="Edit"
               asp-route-id="@role.Id">Edit</a>
```

```
            <button type="submit" class="btn btn-sm btn-danger">
              Delete
            </button>
          </form>
        </td>
      </tr>
    }
  }
</table>
<a class="btn btn-primary" asp-action="Create">Create</a>
```

Для отображения деталей о ролях в приложении представление использует таблицу. В третьей колонке применяется специальный атрибут элемента:

```
...
<td identity-role="@role.Id"></td>
...
```

Нужно отобразить список пользователей, которые являются членами каждой роли, что требует включения в представление очень большого объема кода. Чтобы сохранить представление простым, добавим в папку `Infrastructure` файл класса по имени `RoleUsersTagHelper.cs` и поместим в него определение класса вспомогательной функции дескриптора, приведенное в листинге 29.8.

Листинг 29.8. Содержимое файла `RoleUsersTagHelper.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Users.Models;

namespace Users.Infrastructure {
  [HtmlTargetElement("td", Attributes = "identity-role")]
  public class RoleUsersTagHelper : TagHelper {
    private UserManager<AppUser> userManager;
    private RoleManager<IdentityRole> roleManager;
    public RoleUsersTagHelper(UserManager<AppUser> usermgr,
          RoleManager<IdentityRole> rolemgr) {
      userManager = usermgr;
      roleManager = rolemgr;
    }
    [HtmlAttributeName("identity-role")]
    public string Role { get; set; }
    public override async Task ProcessAsync(TagHelperContext context,
          TagHelperOutput output) {
      List<string> names = new List<string>();
      IdentityRole role = await roleManager.FindByIdAsync(Role);
      if (role != null) {
        foreach (var user in userManager.Users) {
          if (user != null
              && await userManager.IsInRoleAsync(user, role.Name)) {
            names.Add(user.UserName);
          }
```

```
            }
        }
        output.Content.SetContent(names.Count == 0 ?
            "No Users" : string.Join(", ", names));
    }
}
}
```

Класс `RoleUsersTagHelper` оперирует на элементах `td` посредством атрибута `identity-role`, который используется для получения имени обрабатываемой роли. Объекты `RoleManager<IdentityRole>` и `UserManager<AppUser>` позволяют отправлять запросы базе данных Identity для построения списка имен пользователей в роли. В листинге 29.9 к файлу импортирования представлений добавляется класс вспомогательной функции дескриптора `RoleUsersTagHelper` и выражение `@using`, чтобы на типы EF Core можно было ссылаться внутри представлений, не указывая пространство имен.

Листинг 29.9. Добавление класса вспомогательной функции дескриптора в файле `_ViewImports.cshtml` из папки

```
@using Users.Models
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper Users.Infrastructure.*, Users
```

Добавим в папку `Views/RoleAdmin` файл представления по имени `Create.cshtml` с разметкой из листинга 29.10 для поддержки добавления новых ролей.

Листинг 29.10. Содержимое файла `Create.cshtml` из папки `Views/RoleAdmin`

```
@model string

<div class="bg-primary m-1 p-1"><h4>Create Role</h4></div>

<div asp-validation-summary="All" class="text-danger"></div>

<form asp-action="Create" method="post">
  <div class="form-group">
    <label for="name"></label>
    <input name="name" class="form-control" />
  </div>
  <button type="submit" class="btn btn-primary">Create</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Для создания роли из данных формы требуется только имя — вот почему в `Create.cshtml` есть возможность применения `string` в качестве класса модели представления. Мы хотим воспользоваться преимуществами проверки достоверности модели, чтобы гарантировать предоставление пользователем значения, когда форма отправлена, но для такой простой задачи не стоит создавать отдельный класс модели. Взглянув на метод `Create()`, принимающий запросы `POST` в листинге 29.6, вы заметите, что атрибут проверки достоверности `Required` применяется прямо к параметру. Результат будет таким же, как в случае применения этого атрибута в классе модели, и появляется возможность задействовать встроенный процесс проверки достоверности моделей.

Тестирование создания и удаления ролей

Чтобы протестировать новый контроллер, запустим приложение и перейдем на URL вида `/RoleAdmin`. Щелкнем на кнопке Create (Создать), введем имя в элементе `input` и щелкнем на второй кнопке Create. Новая роль будет сохранена в базе данных и отображена после перенаправления браузера на действие `Index` (рис. 29.5). Щелкнув на кнопке Delete (Удалить), роль можно удалить из приложения.

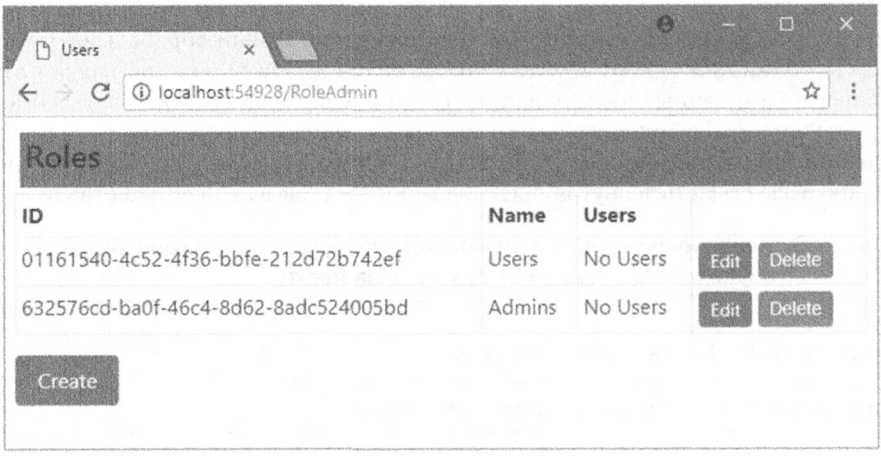


Рис. 29.5. Создание новой роли

Управление членством в ролях

Следующий шаг — обеспечение возможности добавления и удаления пользователей из ролей. Сам процесс несложен; он предусматривает взятие данных роли из класса `RoleManager` и их ассоциирование с деталями индивидуальных пользователей.

Для начала понадобится определить несколько классов моделей представлений, которые будут представлять членство в роли и получать новый набор инструкций относительно членства от пользователя. В листинге 29.11 показаны добавления, внесенные в файл `UserViewModels.cs` из папки `Models`.

Листинг 29.11. Добавление моделей представлений в файле `UserViewModels.cs` из папки `Models`

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using Microsoft.AspNetCore.Identity;

namespace Users.Models {

  public class CreateModel {
    [Required]
    public string Name { get; set; }
    [Required]
    public string Email { get; set; }
    [Required]
    public string Password { get; set; }
  }
```

```
  public class LoginModel {
    [Required]
    [UIHint("email")]
    public string Email { get; set; }
    [Required]
    [UIHint("password")]
    public string Password { get; set; }
  }
  public class RoleEditModel {
    public IdentityRole Role { get; set; }
    public IEnumerable<AppUser> Members { get; set; }
    public IEnumerable<AppUser> NonMembers { get; set; }
  }
  public class RoleModificationModel {
    [Required]
    public string RoleName { get; set; }
    public string RoleId { get; set; }
    public string[] IdsToAdd { get; set; }
    public string[] IdsToDelete { get; set; }
  }
}
```

Класс RoleEditModel представляет роль и детали о пользователях в системе, категоризированные по членству в роли. Класс RoleModificationModel представляет набор изменений роли.

В листинге 29.12 к контроллеру RoleAdmin добавляются новые методы действий, которые используют модели представлений из листинга 29.11 для управления членством в ролях.

Листинг 29.12. Добавление методов действий в файле RoleAdminController.cs из папки Controllers

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Collections.Generic;

namespace Users.Controllers {
  public class RoleAdminController : Controller {
    private RoleManager<IdentityRole> roleManager;
    private UserManager<AppUser> userManager;

    public RoleAdminController(RoleManager<IdentityRole> roleMgr,
        UserManager<AppUser> userMrg) {
      roleManager = roleMgr;
      userManager = userMrg;
    }

    // ...для краткости другие методы действий не показаны...

    public async Task<IActionResult> Edit(string id) {
      IdentityRole role = await roleManager.FindByIdAsync(id);
      List<AppUser> members = new List<AppUser>();
```

```
    List<AppUser> nonMembers = new List<AppUser>();
    foreach (AppUser user in userManager.Users) {
      var list = await userManager.IsInRoleAsync(user, role.Name)
        ? members : nonMembers;
      list.Add(user);
    }
    return View(new RoleEditModel {
      Role = role,
      Members = members,
      NonMembers = nonMembers
    });
  }
  [HttpPost]
  public async Task<IActionResult> Edit(RoleModificationModel model) {
    IdentityResult result;
    if (ModelState.IsValid) {
      foreach (string userId in model.IdsToAdd ?? new string[] { }) {
        AppUser user = await userManager.FindByIdAsync(userId);
        if (user != null) {
          result = await userManager.AddToRoleAsync(user,
            model.RoleName);
          if (!result.Succeeded) {
            AddErrorsFromResult(result);
          }
        }
      }
      foreach (string userId in model.IdsToDelete ?? new string[] { }) {
        AppUser user = await userManager.FindByIdAsync(userId);
        if (user != null) {
          result = await userManager.RemoveFromRoleAsync(user,
            model.RoleName);
          if (!result.Succeeded) {
            AddErrorsFromResult(result);
          }
        }
      }
    }
    if (ModelState.IsValid) {
      return RedirectToAction(nameof(Index));
    } else {
      return await Edit(model.RoleId);
    }
  }
  private void AddErrorsFromResult(IdentityResult result) {
    foreach (IdentityError error in result.Errors) {
      ModelState.AddModelError("", error.Description);
    }
  }
}
```

Большая часть кода в версии GET метода действия Edit() отвечает за генерацию наборов членов и не членов выбранной роли. После того как все пользователи катего-

ризированы, новый экземпляр класса RoleEditModel передается методу View(), так что данные могут быть отображены с применением стандартного представления.

Версия POST метода действия Edit() отвечает за добавление и удаление пользователей в и из ролей. Класс UserManager<T> предлагает методы для работы с ролями, которые описаны в табл. 29.7.

Таблица 29.7. Методы, связанные с ролями, которые определяет класс UserManager<T>

Имя	Описание
AddToRoleAsync(user, name)	Добавляет идентификатор пользователя к роли с указанным именем
GetRolesAsync(user)	Возвращает список имен ролей, членом которых является пользователь
IsInRoleAsync(user, name)	Возвращает true, если пользователь имеет членство в роли с указанным именем
RemoveFromRoleAsync(user, name)	Удаляет пользователя как члена из роли с указанным именем

Причудливость методов, относящихся к ролям, связана с тем, что они оперируют с именами ролей, хотя роли имеют также и уникальные идентификаторы. По указанной причине класс модели представления RoleModificationModel имеет свойство RoleName.

В листинге 29.13 приведено содержимое файла представления Edit.cshtml, добавленного в папку Views/RoleAdmin, которое позволяет пользователю редактировать членство в роли.

Листинг 29.13. Содержимое файла Edit.cshtml из папки Views/RoleAdmin

```
@model RoleEditModel

<div class="bg-primary m-1 p-1 text-white"><h4>Edit Role</h4></div>

<div asp-validation-summary="All" class="text-danger"></div>
<form asp-action="Edit" method="post">
  <input type="hidden" name="roleName" value="@Model.Role.Name" />
  <input type="hidden" name="roleId" value="@Model.Role.Id" />
  <h6 class="bg-info p-1 text-white">Add To @Model.Role.Name</h6>
  <table class="table table-bordered table-sm">
    @if (Model.NonMembers.Count() == 0) {
      <tr><td colspan="2">All Users Are Members</td></tr>
    } else {
      @foreach (AppUser user in Model.NonMembers) {
        <tr>
          <td>@user.UserName</td>
          <td>
            <input type="checkbox" name="IdsToAdd" value="@user.Id">
          </td>
        </tr>
      }
    }
  </table>
```

```
  <h6 class="bg-info p-1 text-white">Remove From @Model.Role.Name</h6>
  <table class="table table-bordered table-sm">
    @if (Model.Members.Count() == 0) {
      <tr><td colspan="2">No Users Are Members</td></tr>
    } else {
      @foreach (AppUser user in Model.Members) {
        <tr>
          <td>@user.UserName</td>
          <td>
            <input type="checkbox" name="IdsToDelete" value="@user.Id">
          </td>
        </tr>
      }
    }
  </table>
  <button type="submit" class="btn btn-primary">Save</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Представление `Edit.cshtml` содержит две таблицы: одну для пользователей, не являющихся членами выбранной роли, и еще одну для пользователей, принадлежащих роли. Рядом с именем каждого пользователя отображается флажок, который позволяет изменять членство. Таблицы находятся внутри формы, которая отправляется методу действия `Edit()` и привязана к классу модели `RoleModificationModel`, обеспечивая легкий доступ к списку изменений членства в роли.

Тестирование редактирования членства в роли

Чтобы протестировать поддержку членства в ролях, запустим приложение, перейдем на URL вида `/RoleAdmin` и создадим новую роль по имени `Users`. Щелкнем на кнопке Edit (Редактировать); все пользователи в приложении отобразятся внутри списка не членов (рис. 29.6).

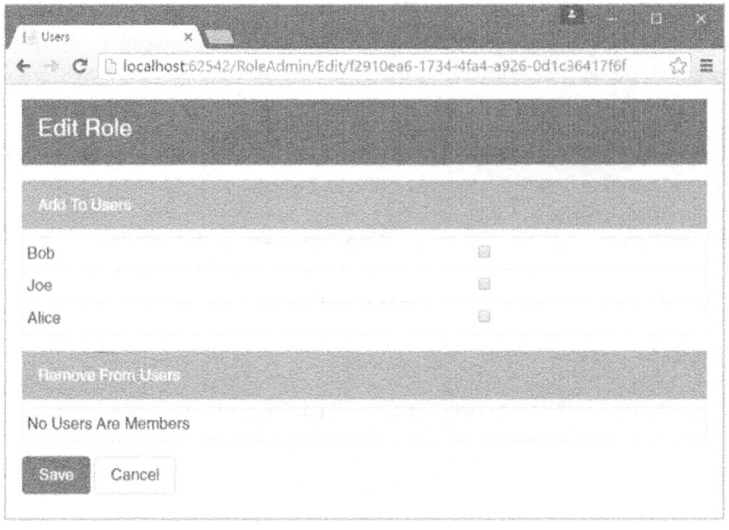


Рис. 29.6. Просмотр и редактирование членства в ролях

Отметим флажки для пользователей Alice и Joe (две учетные записи из числа добавленных в систему Identity в начале главы) и щелкнем на кнопке Save (Сохранить). В списке членов роли Users появятся пользователи Alice и Joe, как показано на рис. 29.7.

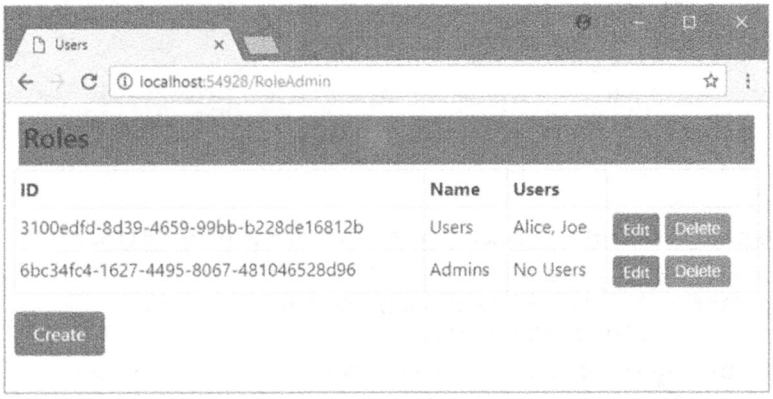


Рис. 29.7. Управление членством в ролях

Использование ролей для авторизации

Теперь, когда в приложении присутствуют роли, их можно применять в качестве основы для авторизации посредством атрибута Authorize. Чтобы облегчить тестирование авторизации на основе ролей, добавим в контроллер Account метод Logout(), который сделает возможным выход и последующий вход от имени другого пользователя для демонстрации работы членства в ролях (листинг 29.14).

Листинг 29.14. Добавление метода Logout() в файле AccountController.cs из папки Controllers

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;

    // ...для краткости другие методы действий не показаны...

    [Authorize]
    public async Task<IActionResult> Logout() {
      await signInManager.SignOutAsync();
      return RedirectToAction("Index", "Home");
    }
  }
}
```

Обновим контроллер Home, добавив новый метод действия и передав представлению информацию об аутентифицированном пользователе (листинг 29.15).

Листинг 29.15. Добавление метода действия и информации об учетной записи в файле HomeController.cs из папки Controllers

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

  public class HomeController : Controller {

    [Authorize]
    public IActionResult Index() => View(GetData(nameof(Index)));

    [Authorize(Roles = "Users")]
    public IActionResult OtherAction() => View("Index",
      GetData(nameof(OtherAction)));

    private Dictionary<string, object> GetData(string actionName) =>
      new Dictionary<string, object> {
        ["Action"] = actionName,
        ["User"] = HttpContext.User.Identity.Name,
        ["Authenticated"] = HttpContext.User.Identity.IsAuthenticated,
        ["Auth Type"] = HttpContext.User.Identity.AuthenticationType,
        ["In Users Role"] = HttpContext.User.IsInRole("Users")
      };
  }
}
```

Атрибут Authorize для метода действия Index() не изменялся, но при его применении к методу OtherAction() было установлено свойство Roles с целью указания на то, что доступ к этому методу должен быть разрешен только членам роли Users. Кроме того, определен метод GetData(), который добавляет базовые сведения об удостоверении пользователя с использованием свойств, доступных через объект HttpContext.

Совет. Атрибут Authorize можно также применять для авторизации доступа на основе списка индивидуальных пользовательских имен. Такая возможность привлекательна в небольших проектах, но требует изменения кода в контроллерах всякий раз, когда изменяется набор авторизуемых пользователей, и обычно означает необходимость в повторном проходе через цикл тестирования и развертывания. Использование ролей для авторизации изолирует приложение от изменений в отдельных пользовательских учетных записях и позволяет управлять доступом к приложению посредством информации о членстве в ролях, хранящейся в системе ASP.NET Core Identity.

Последнее изменение касается файла Index.cshtml из папки Views/Home, который применяется обоими действиями в контроллере Home, и связано с добавлением ссылки, нацеленной на метод Logout() контроллера Account (листинг 29.16).

Листинг 29.16. Добавление ссылки на метод `Logout()` в файле `Index.cshtml` из папки `Views/Home`

```
@model Dictionary<string, object>

<div class="bg-primary m-1 p-1 text-white"><h4>User Details</h4></div>

<table class="table table-sm table-bordered m-1 p-1">
  @foreach (var kvp in Model) {
    <tr><th>@kvp.Key</th><td>@kvp.Value</td></tr>
  }
</table>

@if (User?.Identity?.IsAuthenticated ?? false) {
  <a asp-controller="Account" asp-action="Logout"
    class="btn btn-danger">Logout</a>
}
```

Чтобы протестировать аутентификацию, запустим приложение и перейдем на URL вида `/Home/Index`. Браузер будет перенаправлен так, что можно ввести пользовательские учетные данные. Совершенно не имеет значения, какой пользователь из табл. 29.2 будет выбран, поскольку атрибут `Authorize`, примененный к действию `Index`, разрешает доступ любому аутентифицированному пользователю.

Однако если запросить URL вида `/Home/OtherAction`, то выбор пользователя из табл. 29.2 будет иметь значение, т.к. членами роли `Users`, требующейся для доступа к методу `OtherAction()`, являются только пользователи `Alice` и `Joe`. В случае входа от имени пользователя `Bob` браузер будет перенаправлен на URL вида `/Account/AccessDenied`, который используется, когда пользователь не имеет возможности получить доступ к методу действия. Для обработки такой ситуации в контроллер `Account` добавлен метод `AccessDenied()`, поэтому теперь есть действие, обрабатывающее запрос (листинг 29.17).

Совет. Устанавливая конфигурационное свойство `AccessDeniedPath`, можно изменять URL вида `/Account/AccessDenied`. Во врезке "Изменение URL для входа" ранее в главе был приведен похожий пример.

Листинг 29.17. Добавление метода действия в файле `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;

namespace Users.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;

    public AccountController(UserManager<AppUser> userMgr,
          SignInManager<AppUser> signinMgr) {
```

```
      userManager = userMgr;
      signInManager = signinMgr;
    }

    // ...для краткости другие методы действий не показаны...

    [AllowAnonymous]
    public IActionResult AccessDenied() {
      return View();
    }
  }
}
```

Чтобы снабдить действие `AccessDenied` представлением для отображения, создадим в папке `Views/Account` файл по имени `AccessDenied.cshtml` с содержимым из листинга 29.18.

Листинг 29.18. Содержимое файла `AccessDenied.cshtml` из папки `Views/Account`

```
<div class="bg-danger mb-1 p-2 text-white"><h4>Access Denied</h4></div>
<a asp-action="Index" asp-controller="Home" class="btn btn-primary">OK</a>
```

Запустим приложение, запросим URL вида `/Account/Login` и войдем от имени `bob@example.com`. Когда процесс аутентификации завершится, браузер будет перенаправлен на URL вида `/Home/Index`, который отобразит детали учетной записи, как показано слева на рис. 29.8, проясняя, что пользователь `Bob` не является членом роли `Users`. Затем запросим URL вида `/Home/OtherAction`, который нацелен на действие, защищенное с помощью доступа на основе ролей. Пользователь `Bob` не имеет требуемого членства в роли, поэтому браузер будет перенаправлен на URL вида `/Account/AccessDenied`, как демонстрируется справа на рис. 29.8.

Совет. Роли загружаются во время входа пользователя, т.е. изменение ролей для пользователя, который в текущий момент аутентифицирован, вступит в силу только после того, как пользователь выйдет и затем аутентифицируется заново.

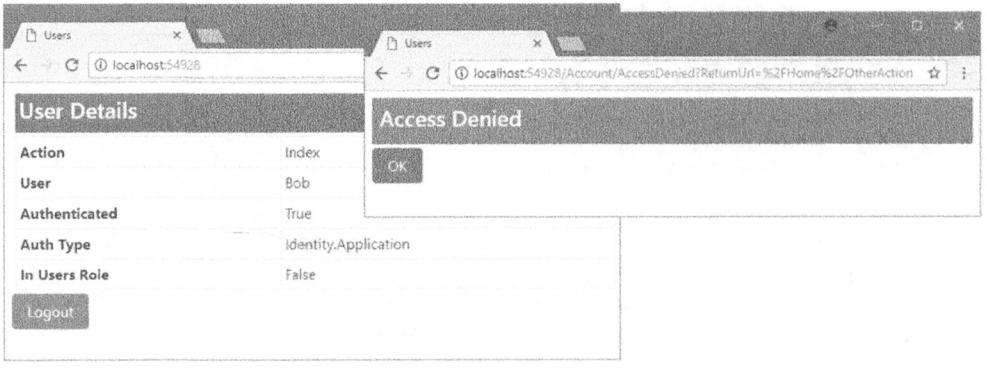


Рис. 29.8. Применение авторизации на основе ролей

Помещение в базу данных начальной информации

В рассматриваемом проекте осталась одна проблема — доступ к контроллерам Admin и RoleAdmin не ограничен. Это классическая проблема курицы и яйца. Дело в том, что для ограничения доступа необходимо создать пользователей и роли, но контроллеры Admin и RoleAdmin являются инструментами управления пользователями, и если защитить их с помощью атрибута Authorize, тогда не будет никаких учетных данных, которые предоставят к ним доступ, особенно при первом развертывании приложения.

Проблема решается за счет помещения в базу данных начальной информации при запуске приложения. В листинге 29.19 приведено содержимое файла appsettings.json с добавленными конфигурационными данными, указывающими детали для учетной записи, которая будет создана.

Листинг 29.19. Добавление конфигурационных данных в файле appsettings.json из папки Users

```
{
  "Data": {
    "AdminUser": {
      "Name": "Admin",
      "Email": "admin@example.com",
      "Password": "secret",
      "Role": "Admins"
    },
    "SportStoreIdentity": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;
        Database=IdentityUsers;
        Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}
```

В категории Data:AdminUser предоставлены четыре значения, требующиеся для создания учетной записи и ее назначения роли, которая обеспечит возможность использования административных инструментов.

Внимание! Помещение паролей в текстовые конфигурационные файлы означает необходимость добавления в процесс развертывания приложения возможности изменять пароль стандартной учетной записи и инициализировать новую базу данных при начальном развертывании.

Добавим в класс AppIdentityDbContext статический метод, как показано в листинге 29.20. Код для создания стандартной учетной записи вовсе не обязан находиться в данном классе, но для меня это место выглядит вполне естественным, и я поступаю так в своих проектах. Вы можете применять отдельный класс, как делается в приложении SportsStore.

Листинг 29.20. Добавление метода в файле AppIdentityDbContext.cs из папки Models

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;

namespace Users.Models {

  public class AppIdentityDbContext : IdentityDbContext<AppUser> {

    public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext>
      options)
        : base(options) { }

    public static async Task CreateAdminAccount(
        IServiceProvider serviceProvider,
        IConfiguration configuration) {

      UserManager<AppUser> userManager =
        serviceProvider.GetRequiredService<UserManager<AppUser>>();
      RoleManager<IdentityRole> roleManager =
        serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();

      string username = configuration["Data:AdminUser:Name"];
      string email = configuration["Data:AdminUser:Email"];
      string password = configuration["Data:AdminUser:Password"];
      string role = configuration["Data:AdminUser:Role"];

      if (await userManager.FindByNameAsync(username) == null) {
        if (await roleManager.FindByNameAsync(role) == null) {
          await roleManager.CreateAsync(new IdentityRole(role));
        }

        AppUser user = new AppUser {
          UserName = username,
          Email = email
        };

        IdentityResult result = await userManager
          .CreateAsync(user, password);
        if (result.Succeeded) {
          await userManager.AddToRoleAsync(user, role);
        }
      }
    }
  }
}
```

Метод CreateAdminAccount() принимает объект реализации IServiceProvider, который применяется для получения объектов UserManager и RoleManager, а также объект реализации IConfiguration, используемый для извлечения данных из файла appsetting.json. Код в методе CreateAdminAccount() проверяет, существует ли пользователь, и если нет, то создает его и назначает указанной роли, которая также

при необходимости создается. В листинге 29.21 в класс Startup добавлен оператор, вызывающий метод CreateAdminAccount() после того, как остальная часть приложения настроена и сконфигурирована.

Листинг 29.21. Вызов метода базы данных в файле Startup.cs из папки Users

```
...
public void Configure(IApplicationBuilder app) {
  app.UseStatusCodePages();
  app.UseDeveloperExceptionPage();
  app.UseStaticFiles();
  app.UseAuthentication();
  app.UseMvcWithDefaultRoute();
  AppIdentityDbContext.CreateAdminAccount(app.ApplicationServices,
    Configuration).Wait();
}
...
```

Поскольку здесь производится доступ к службе с ограниченной областью действия через поставщик IApplicationBuilder.ApplicationServices, потребуется также отключить проверку области действия в классе Program (листинг 29.22).

Листинг 29.22. Отключение проверки области действия в файле Program.cs из папки Users

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace Users {
  public class Program {
    public static void Main(string[] args) {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseDefaultServiceProvider(options =>
          options.ValidateScopes = false)
        .Build();
  }
}
```

Располагая надежной стандартной учетной записью в базе данных Identity, можно применять атрибут Authorize для защиты контроллеров Admin и RoleAdmin. В листинге 29.23 приведены изменения, внесенные в контроллер Admin.

Листинг 29.23. Ограничение доступа в файле AdminController.cs из папки Controllers

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {

  [Authorize(Roles = "Admins")]
  public class AdminController : Controller {
    // ...для краткости операторы не показаны...
  }
}
```

В листинге 29.24 показано соответствующее изменение, внесенное в контроллер RoleAdmin.

Листинг 29.24. Ограничение доступа в файле RoleAdminController.cs из папки Controllers

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using System.Collections.Generic;
using Microsoft.AspNetCore.Authorization;

namespace Users.Controllers {
  [Authorize(Roles = "Admins")]
  public class RoleAdminController : Controller {
    // ...для краткости операторы не показаны...
  }
}
```

Теперь следует запустить приложение и запросить URL вида /Admin или /RoleAdmin. Если ранее уже был осуществлен вход от имени какого-то другого пользователя, тогда понадобится выйти. В противном случае будет предложено предоставить учетные данные, так что можно ввести admin@example.com и пароль secret и получить доступ к административным функциям.

Резюме

В главе объяснялось, как использовать систему ASP.NET Core Identity для аутентификации и авторизации пользователей. Вы узнали, каким образом собирать и проверять пользовательские учетные данные и ограничивать доступ к методам действий на основе ролей, членом которых является пользователь. В следующей главе будут продемонстрированы некоторые расширенные средства, предлагаемые системой ASP.NET Core Identity.

ГЛАВА 30

Расширенные средства ASP.NET Core Identity

В настоящей главе описание системы ASP.NET Core Identity завершается рассмотрением ряда более сложных средств, которые она предлагает. Будет продемонстрировано, как расширять схему базы данных за счет определения специальных свойств в классе пользователя, и каким образом использовать миграции базы данных для применения этих свойств, не удаляя записи из базы данных ASP.NET Core Identity. Кроме того, вы узнаете, как система ASP.NET Core Identity поддерживает концепцию заявок (claim), и ознакомитесь с их использованием для гибкой авторизации доступа к методам действий через политики. Ближе к концу главы будет показано, каким образом система ASP.NET Core Identity облегчает аутентификацию пользователей с участием третьих сторон. Вы увидите способ аутентификации с помощью учетных записей Google, но имейте в виду, что в ASP.NET Core Identity есть встроенная поддержка также для учетных записей Microsoft, Facebook и Twitter. В табл. 30.1 приведена сводка по главе.

Таблица 30.1. Сводка по главе

Задача	Решение	Листинг
Сохранение специальных данных для пользователей	Добавьте свойства в класс пользователя и обновите базу данных Identity	30.1–30.3
Выполнение детализированной авторизации	Используйте заявки	30.4–30.7
Создание специальных заявок	Применяйте трансформацию заявок	30.8, 30.9
Использование данных заявок для оценки доступа пользователей	Создайте политики	30.10–30.14
Применение политик для доступа к ресурсам	Оценивайте политики внутри методов действий	30.15–30.20
Разрешение выполнять аутентификацию третьим сторонам	Принимайте заявки от поставщиков аутентификации, таких как Microsoft, Google и Facebook	30.21–30.24

Подготовка проекта для примера

В главе будет продолжена работа с проектом `Users`, созданным в главе 28 и усовершенствованным в главе 29. Запустим приложение и удостоверимся в наличии пользователей в базе данных. На рис. 30.1 показано состояние базы данных, которая содержит пользователей `Admin`, `Alice`, `Bob` и `Joe` из предыдущей главы. Чтобы проверить пользователей, запустим приложение, запросим URL вида `/Admin` и аутентифицируемся как пользователь `Admin`, используя адрес `admin@example.com` и пароль `secret`.

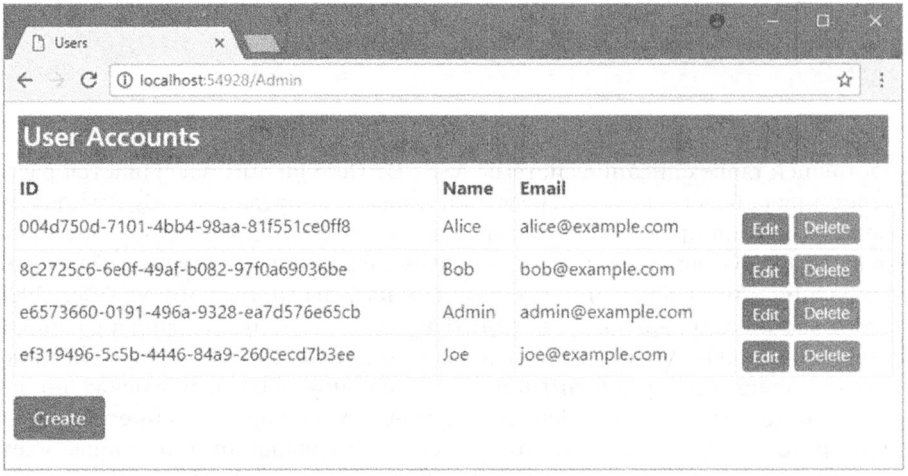


Рис. 30.1. Начальные пользователи в базе данных Identity

В текущей главе также понадобится несколько ролей. Перейдем на URL вида `/RoleAdmin`, создадим роли с именами `Users` и `Employees` и назначим им пользователей согласно табл. 30.2.

Таблица 30.2. Роли и их члены, требующиеся для примера приложения

Роль	Члены
`Users`	`Alice, Joe`
`Employees`	`Alice, Bob`

На рис. 30.2 показана нужная конфигурация ролей, отображаемая контроллером `RoleAdmin`.

Добавление специальных свойств в класс пользователя

При создании класса `AppUser` для представления пользователей в главе 28 упоминалось, что базовый класс определяет основной набор свойств, описывающих пользователя, таких как адрес электронной почты и телефонный номер.

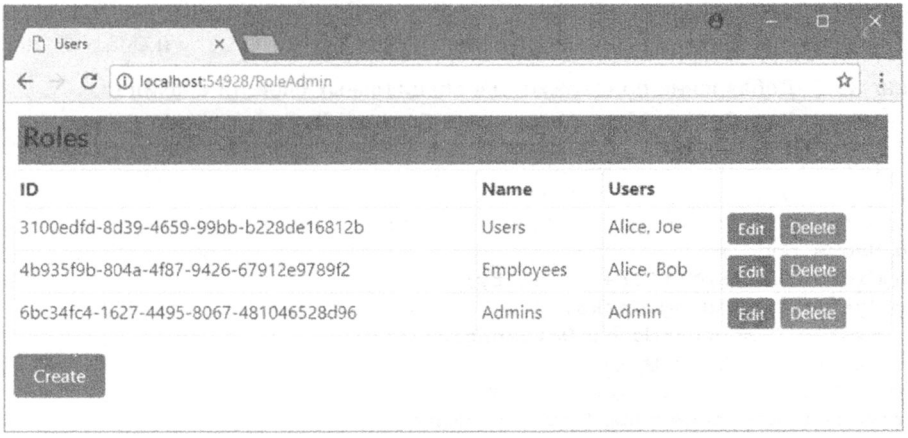


Рис. 30.2. Конфигурирование ролей для целей главы

В большинстве приложений необходимо хранить дополнительную информацию о пользователях, включая постоянные предпочтения в приложении и сведения вроде адресов — короче говоря, любые данные, которые полезны при выполнении приложения и должны предохраняться между сеансами. Поскольку по умолчанию для хранения своих данных система ASP.NET Core Identity применяет инфраструктуру Entity Framework Core, определение дополнительной информации о пользователе означает добавление свойств в класс пользователя и предоставление Entity Framework Core возможности создать схему базы данных для их сохранения.

В листинге 30.1 приведен код класса `AppUser` с добавленными двумя простыми свойствами, предназначенными для представления города, в котором живет пользователь, и уровня его квалификации.

Листинг 30.1. Добавление свойств в файле `AppUser.cs` из папки `Models`

```
using Microsoft.AspNetCore.Identity;

namespace Users.Models {

  public enum Cities {
    None, London, Paris, Chicago
  }

  public enum QualificationLevels {
    None, Basic, Advanced
  }

  public class AppUser : IdentityUser {
    public Cities City { get; set; }
    public QualificationLevels Qualifications { get; set; }
  }
}
```

Перечисления `Cities` и `QualificationLevels` определяют значения для нескольких городов и уровней квалификации. Указанные перечисления используются свойствами `City` и `Qualification`, добавленными в класс `AppUser`.

Действия, которые добавлены к контроллеру Home в листинге 30.2, позволяют пользователю просматривать и редактировать свои свойства City и Qualification.

Листинг 30.2. Добавление поддержки для специальных свойств в файле HomeController.cs из папки Controllers

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Users.Models;
using Microsoft.AspNetCore.Identity;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;
namespace Users.Controllers {
  public class HomeController : Controller {
    private UserManager<AppUser> userManager;
    public HomeController(UserManager<AppUser> userMgr) {
      userManager = userMgr;
    }
    [Authorize]
    public IActionResult Index() => View(GetData(nameof(Index)));
    [Authorize(Roles = "Users")]
    public IActionResult OtherAction() => View("Index",
      GetData(nameof(OtherAction)));
    private Dictionary<string, object> GetData(string actionName) =>
      new Dictionary<string, object> {
        ["Action"] = actionName,
        ["User"] = HttpContext.User.Identity.Name,
        ["Authenticated"] = HttpContext.User.Identity.IsAuthenticated,
        ["Auth Type"] = HttpContext.User.Identity.AuthenticationType,
        ["In Users Role"] = HttpContext.User.IsInRole("Users"),
        ["City"] = CurrentUser.Result.City,
        ["Qualification"] = CurrentUser.Result.Qualifications
      };
    [Authorize]
    public async Task<IActionResult> UserProps() {
      return View(await CurrentUser);
    }
    [Authorize]
    [HttpPost]
    public async Task<IActionResult> UserProps(
          [Required]Cities city,
          [Required]QualificationLevels qualifications) {
      if (ModelState.IsValid) {
        AppUser user = await CurrentUser;
        user.City = city;
        user.Qualifications = qualifications;
        await userManager.UpdateAsync(user);
        return RedirectToAction("Index");
      }
      return View(await CurrentUser);
    }
    private Task<AppUser> CurrentUser =>
      userManager.FindByNameAsync(HttpContext.User.Identity.Name);
  }
}
```

Новое свойство CurrentUser с помощью класса UserManager<AppUser> извлекает экземпляр AppUser для представления текущего пользователя. Объект AppUser применяется в качестве объекта модели представления в версии GET метода действия UserProps(), а версия POST этого метода использует его для обновления значений новых свойств City и QualificationLevel.

Метод GetData() был модифицирован так, чтобы возвращаемый им словарь содержал значения специальных свойств для текущего пользователя, т.е. значения специальных свойств будут видны в представлениях, которые отображаются методами действий Index() и OtherAction().

Чтобы снабдить методы действий UserProps() представлением, добавим в папку Views/Home файл по имени UserProps.cshtml и поместим в него разметку из листинга 30.3.

Листинг 30.3. Содержимое файла UserProps.cshtml из папки Views/Home

```
@model AppUser

<div class="bg-primary m-1 p-1 text-white"><h4>@Model.UserName</h4>
</div>

<div asp-validation-summary="All" class="text-danger"></div>

<form asp-action="UserProps" method="post">
  <div class="form-group">
    <label asp-for="City"></label>
    <select asp-for="City" class="form-control"
          asp-items="@new SelectList(Enum.GetNames(typeof(Cities)))">
      <option disabled selected value="">Select a City</option>
    </select>
  </div>
  <div class="form-group">
    <label asp-for="Qualifications"></label>
    <select asp-for="Qualifications" class="form-control"
 asp-items="@new SelectList(Enum.GetNames(typeof(QualificationLevels)))">
      <option disabled selected value="">Select a City</option>
    </select>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</form>
```

Представление содержит форму с элементами select, которые заполняются значениями из перечислений, определенных в листинге 30.1. Когда форма отправляется, из базы данных Identity извлекается объект AppUser, представляющий текущего пользователя, а значения специальных свойств обновляются с применением значений, выбранных пользователем:

```
...
AppUser user = await CurrentUser;
user.City = city;
user.Qualifications = qualifications;
await userManager.UpdateAsync(user);
return RedirectToAction("Index");
...
```

Обратите внимание, что для отражения изменений диспетчеру пользователей потребуется явно сообщить о необходимости обновления записи базы данных, относящейся к пользователю, путем вызова метода `UpdateAsync()`. Ранее поступать так было не обязательно, потому что внутри методов, которые использовались для внесения изменений в Identity, метод `UpdateAsync()` вызывался автоматически, но при изменении свойств напрямую ответственность за указание диспетчеру пользователей о том, что нужно выполнить обновление, возлагается на нас.

Подготовка миграции базы данных

Все связующие механизмы приложения, предназначенные для поддержки новых свойств, теперь на месте, и осталось лишь обновить базу данных, чтобы ее таблицы сохраняли значения специальных свойств.

Инфраструктура Entity Framework Core не обладает встроенной поддержкой для работы с начальными данными, и при создании миграций необходимо позаботиться об отключении начального заполнения (листинг 30.4), иначе новые свойства, добавленные к классу модели в листинге 30.1, вызовут ошибку. После создания и применения миграции базы данных оператор начального заполнения можно снова активизировать.

Листинг 30.4. Отключение начального заполнения базы данных в файле `Startup.cs` из папки `Users`

```
...
public void Configure(IApplicationBuilder app) {
  app.UseStatusCodePages();
  app.UseDeveloperExceptionPage();
  app.UseStaticFiles();
  app.UseAuthentication();
  app.UseMvcWithDefaultRoute();
  // AppIdentityDbContext.CreateAdminAccount(app.ApplicationServices,
  //   Configuration).Wait();
}
...
```

После отключения начального заполнения понадобится создать новую миграцию базы данных, которая будет содержать команды SQL, требующиеся для обновления схемы базы данных. Откроем окно командной строки или PowerShell и введем следующую команду в папке проекта `Users`:

```
dotnet ef migrations add CustomProperties
```

В результате выполнения команды внутри папки `Migrations` появится новый файл, имя которого состоит из строки `CustomProperties` и числового идентификатора. Открыв файл, в нем можно обнаружить класс C#, содержащий метод по имени `Up()`, который выполняет команды SQL, необходимые для добавления в базу данных поддержки специальных свойств. Имеется также метод `Down()`, который выполняет команды, восстанавливающие предыдущую схему базы данных.

Затем выполним миграцию базы данных в новую схему с помощью показанной ниже команды:

```
dotnet ef database update
```

Когда команда завершится, таблица базы данных, которая хранит данные о пользователях, будет содержать новые столбцы, представляющие специальные свойства.

Внимание! Будьте внимательны при выполнении миграций производственных баз данных, содержащих реальные данные о пользователях. Довольно легко создать миграцию, которая отбросит столбцы или целые таблицы и может привести к разрушительным последствиям. Удостоверьтесь в том, что миграции баз данных тщательно протестированы, и позаботьтесь о создании резервной копии важных данных на случай, если что-то пойдет не так, как ожидалось.

Тестирование специальных свойств

Чтобы протестировать результат миграции, запустим приложение и войдем как один из пользователей, хранящихся в Identity (с применением, например, адреса `alice@example.com` и пароля `secret123`). После аутентификации отобразятся стандартные значения для свойств `City` и `QualificationLevel`. Изменить упомянутые свойства можно, запросив URL вида `/Home/UserProps`, выбрав новые значения и щелкнув на кнопке Submit (Отправить), что приведет к обновлению базы данных и перенаправлению опять на URL вида `/Home` с отображением новых значений (рис. 30.3).



Рис. 30.3. Использование специальных свойств в классе пользователя

Работа с заявками и политиками

В более старых системах управления пользователями, таких как ASP.NET Membership, которая предшествовала ASP.NET Core Identity, приложение считалось полномочным источником всей информации о пользователе; по существу приложение трактовалось как замкнутый мир с полным доверием к содержащимся внутри него данным.

Такой подход к разработке ПО укоренился настолько прочно, что можно даже не осознавать сам факт его применения. Пример приема с замкнутым миром приводился в главе 29, когда пользователи аутентифицировались с помощью учетных сведений, хранящихся в базе данных, а доступ предоставлялся на основе ролей, ассоциированных с учетными сведениями. То же самое делалось еще раз в настоящей главе, когда в класс пользователя были добавлены специальные свойства.

Каждый фрагмент информации, необходимой для управления аутентификацией и авторизацией пользователей, поступал изнутри приложения; такой подход великолепно подходит для большинства веб-приложений и потому он демонстрировался здесь во всех подробностях.

Система ASP.NET Core Identity также поддерживает альтернативный подход к работе с пользователями, который приемлем в ситуации, когда приложение MVC не является единственным источником информации о пользователях, и может применяться для авторизации пользователей более гибкими способами, чем позволяют традиционные роли. Альтернативный подход предусматривает использование заявок, и в настоящем разделе будет описано, как система ASP.NET Core Identity поддерживает авторизацию на основе заявок.

Совет. Вы вовсе не обязаны применять заявки в своих приложениях и, как было показано в главе 29, система ASP.NET Core Identity прекрасно обеспечивает приложения службами аутентификации и авторизации безо всякой потребности в понимании заявок.

Понятие заявок

Заявка — это порция информации о пользователе наряду со сведениями о том, откуда информация поступила. Изучать заявки легче всего на практических демонстрациях, в отсутствие которых любое обсуждение становится слишком абстрактным, чтобы быть по-настоящему полезным. Прежде всего, добавим в папку `Controllers` файл класса по имени `ClaimsController.cs` и определим в нем контроллер, как показано в листинге 30.4.

Совет. Вы можете пребывать в некоторой растерянности, следя за определением кода и знакомясь с описаниями классов для рассматриваемого примера. В текущий момент переживать по поводу деталей не стоит — просто придерживайтесь инструкций до тех пор, пока не увидите вывод из метода действия и представление, которое будет определено. Это лучше всего другого содействует пониманию заявок.

Листинг 30.5. Содержимое файла `ClaimsController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace Users.Controllers {

  public class ClaimsController : Controller {

    [Authorize]
    public ViewResult Index() => View(User?.Claims);
  }
}
```

Получать заявки, ассоциированные с пользователем, можно разными путями. Свойство `User` (также доступное как свойство `HttpContext.User`) возвращает объект `ClaimsPrincipal`, который реализует принцип, задействованный в данном примере. Набор заявок, связанных с пользователем, доступен через методы и свойства класса `ClaimsPrincipal`, перечисленные в табл. 30.3.

Таблица 30.3. Избранные члены класса `ClaimsPrincipal`

Имя	Описание
`Identity`	Получает объект реализации `IIdentity`, который ассоциирован с текущим пользователем, как рассказывается в последующих разделах
`FindAll(type)` `FindAll(predicate)`	Возвращают все заявки специфического типа или те, которые удовлетворяют предикату (`predicate`)
`FindFirst(type)` `FindFirst(predicate)`	Возвращают первую заявку специфического типа или ту, которая удовлетворяет предикату
`HasClaim(type, value)` `HasClaim(predicate)`	Возвращают `true`, если пользователь имеет заявку указанного типа с заданным значением или если есть заявка, удовлетворяющая предикату
`IsInRole(name)`	Возвращает `true`, если пользователь является членом роли с указанным именем

Как объяснялось в главе 28, свойство `HttpContext.User.Identity` возвращает реализацию интерфейса `IIdentity`, которой в случае использования системы ASP.NET Core Identity является объект `ClaimsIdentity`. В табл. 30.4 описаны члены класса `ClaimsIdentity`, имеющие отношение к главе.

Таблица 30.4. Избранные члены класса `ClaimsIdentity`

Имя	Описание
`Claims`	Возвращает перечисление объектов `Claim`, представляющих заявки для пользователя
`AddClaim(claim)`	Добавляет заявку к удостоверению пользователя
`AddClaims(claims)`	Добавляет перечисление объектов `Claim` к удостоверению пользователя
`HasClaim(predicate)`	Возвращает `true`, если удостоверение пользователя содержит заявку, которая совпадает с указанной
`RemoveClaim(claim)`	Удаляет заявку из удостоверения пользователя

Доступны также другие методы и свойства, но перечисленные в табл. 30.4 члены применяются в веб-приложениях наиболее часто по причинам, которые станут очевидными по мере демонстрации места заявок в рамках более широкой платформы ASP.NET Core.

В листинге 30.5 свойство `Controller.User` использовалось для получения объекта `ClaimsPrincipal` и передачи значения свойства `Claims` как модели представления стандартному представлению. Объект `Claim` представляет одиночную порцию данных о пользователе, а в классе `Claim` определены свойства, описанные в табл. 30.5.

Чтобы отобразить детали заявок, ассоциированных с пользователем, создадим папку `Views/Claims` и добавим в нее файл по имени `Index.cshtml` с содержимым из листинга 30.6.

Таблица 30.5. Свойства класса `Claim`

Имя	Описание
`Issuer`	Возвращает имя системы, которая предоставила заявку
`Subject`	Возвращает объект `ClaimsIdentity` для пользователя, к которому относится заявка
`Type`	Возвращает тип информации, которую представляет заявка
`Value`	Возвращает порцию информации, которую представляет заявка

Листинг 30.6. Содержимое файла `Index.cshtml` из папки `Views/Claims`

```
@model IEnumerable<System.Security.Claims.Claim>

<div class="bg-primary m-1 p-1 text-white"><h4>Claims</h4></div>
<table class="table table-sm table-bordered">
  <tr>
    <th>Subject</th><th>Issuer</th><th>Type</th><th>Value</th>
  </tr>
  @if (Model == null || Model.Count() == 0) {
    <tr><td colspan="4" class="text-center">No Claims</td></tr>
  } else {
    @foreach (var claim in Model.OrderBy(x => x.Type)) {
      <tr>
        <td>@claim.Subject.Name</td>
        <td>@claim.Issuer</td>
        <td identity-claim-type="@claim.Type"></td>
        <td>@claim.Value</td>
      </tr>
    }
  }
</table>
```

В представлении `Index.cshtml` применяется таблица для отображения заявок, переданных в модели представления. Значением свойства `Claim.Type` является URI для схемы Microsoft, которая не особенно полезна. В качестве значений для полей класса `System.Security.Claims.ClaimTypes` используются популярные схемы, поэтому чтобы облегчить чтение вывода из представления `Index.cshtml`, к элементу `td`, отображающему свойство `Type`, был добавлен специальный атрибут:

```
...
<td identity-claim-type="@claim.Type"></td>
...
```

Поместим в папку `Infrastructure` файл класса по имени `ClaimTypeTagHelper.cs` и определим в нем класс вспомогательной функции дескриптора, который транслирует значение атрибута в более читабельную строку (листинг 30.7).

Листинг 30.7. Содержимое файла `ClaimTypeTagHelper.cs` из папки `Infrastructure`

```
using System.Linq;
using System.Reflection;
using System.Security.Claims;
using Microsoft.AspNetCore.Razor.TagHelpers;
```

```
namespace Users.Infrastructure {
  [HtmlTargetElement("td", Attributes = "identity-claim-type")]
  public class ClaimTypeTagHelper : TagHelper {
    [HtmlAttributeName("identity-claim-type")]
    public string ClaimType { get; set; }
    public override void Process(TagHelperContext context,
          TagHelperOutput output) {
      bool foundType = false;
      FieldInfo[] fields = typeof(ClaimTypes).GetFields();
      foreach (FieldInfo field in fields) {
        if (field.GetValue(null).ToString() == ClaimType) {
          output.Content.SetContent(field.Name);
          foundType = true;
        }
      }
      if (!foundType) {
        output.Content.SetContent(ClaimType.Split('/', '.').Last());
      }
    }
  }
}
```

Чтобы выяснить, почему контроллер, в котором применяются заявки, был создан без подробных объяснений, запустим приложение и аутентифицируемся как пользователь `Alice` (указав адрес `alice@example.com` и пароль `secret123`). После прохождения аутентификации запросим URL вида `/Claims` для просмотра заявок, ассоциированных с пользователем (рис. 30.4).

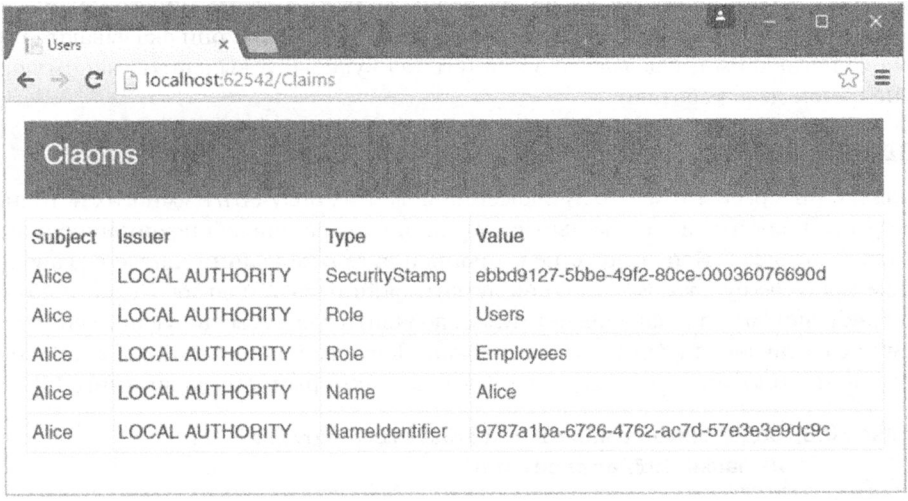


Рис. 30.4. Вывод из действия `Index` контроллера `Claims`

Для удобства детали заявок дополнительно воспроизведены в табл. 30.6.

Таблица 30.6. Данные, показанные на рис. 30.4

Субъект	Выдавшая система	Тип	Значение
Alice	LOCAL AUTHORITY	SecurityStamp	Уникальный идентификатор
Alice	LOCAL AUTHORITY	Role	Users
Alice	LOCAL AUTHORITY	Role	Employees
Alice	LOCAL AUTHORITY	Name	Alice
Alice	LOCAL AUTHORITY	NameIdentifier	Идентификатор пользователя Alice

В табл. 30.6 отражен самый важный аспект заявок, который заключается в том, что они уже применялись при реализации стандартных средств аутентификации и авторизации в главе 29. Здесь видно, что некоторые из заявок относятся к удостоверению пользователя (заявка `Name` — это `Alice`, а заявка `NameIdentifier` — уникальный идентификатор пользователя `Alice` в базе данных ASP.NET Core Identity). Другие заявки показывают членство в ролях — есть две заявки `Role`, свидетельствующие о том факте, что пользователю `Alice` назначены роли `Users` и `Employees`.

Когда информация подобного рода выражается как набор заявок, отличие состоит в том, что появляется возможность определить, откуда поступили данные. Для всех заявок, приведенных в табл. 30.6, свойство `Issuer` установлено в `LOCAL AUTHORITY`, указывая на то, что удостоверение пользователя было установлено приложением.

Благодаря ознакомлению с примерами заявок гораздо легче определить, что собой представляет заявка: заявка — это любая порция информации о пользователе, которая доступна приложению, включая удостоверение пользователя и его членство в ролях. Кроме того, система ASP.NET Core Identity автоматически делает доступной в виде заявок информацию о пользователях, которая была определена в предшествующих главах. Хотя поначалу заявки способны сбивать с толку, ничего магического в них нет, и подобно любому другому аспекту приложений MVC они оказываются гораздо менее пугающими, как только заглянуть "за кулисы" и выяснить, каким образом они на самом деле работают.

Создание заявок

Заявки интересны тем, что приложение может их получать из многих источников, а не просто полагаться на локальную базу данных, хранящую информацию о пользователях. В разделе "Использование сторонней аутентификации" далее в главе на реальном примере будет показано, как аутентифицировать пользователей с помощью сторонней системы, а пока в проект нужно добавить класс, эмулирующий систему, которая предоставляет информацию о заявках. Добавим в папку `Infrastructure` файл класса по имени `LocationClaimsProvider.cs` с содержимым из листинга 30.8.

Листинг 30.8. Содержимое файла `LocationClaimsProvider.cs` из папки `Infrastructure`

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication;

namespace Users.Infrastructure {
```

```
public class LocationClaimsProvider : IClaimsTransformation {
  public Task<ClaimsPrincipal> TransformAsync(ClaimsPrincipal principal) {
    if (principal != null && !principal.HasClaim(c =>
        c.Type == ClaimTypes.PostalCode)) {
      ClaimsIdentity identity = principal.Identity as ClaimsIdentity;
      if (identity != null && identity.IsAuthenticated
          && identity.Name != null) {
        if (identity.Name.ToLower() == "alice") {
          identity.AddClaims(new Claim[] {
            CreateClaim(ClaimTypes.PostalCode, "DC 20500"),
            CreateClaim(ClaimTypes.StateOrProvince, "DC")
          });
        } else {
          identity.AddClaims(new Claim[] {
            CreateClaim(ClaimTypes.PostalCode, "NY 10036"),
            CreateClaim(ClaimTypes.StateOrProvince, "NY")
          });
        }
      }
    }
    return Task.FromResult(principal);
  }
  private static Claim CreateClaim(string type, string value) =>
    new Claim(type, value, ClaimValueTypes.String, "RemoteClaims");
}
}
```

Метод `TransformAsync()`, который определен интерфейсом `IClaimsTransformation`, получает объект `ClaimsPrincipal` и инспектирует его, приводя значение свойства `Identity` к типу `ClaimsIdentity`. Затем значение свойства `Name` применяется для создания заявок о почтовом коде и штате, в котором проживает пользователь.

Класс `LocationClaimsProvider` эмулирует систему вроде центральной базы данных человеческих ресурсов, которая могла бы служить полномочным источником информации, скажем, о месте жительства персонала. Чтобы зарегистрировать источник заявок, определим службу в методе `ConfigureServices()` класса `Startup` (листинг 30.9).

Листинг 30.9. Включение трансформации заявок в файле `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;
using Microsoft.AspNetCore.Authentication;

namespace Users {

  public class Startup {

    public Startup(IConfiguration configuration) =>
      Configuration = configuration;
```

```
    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();
      services.AddTransient<IUserValidator<AppUser>,
        CustomUserValidator>();
      services.AddSingleton<IClaimsTransformation,
        LocationClaimsProvider>();

      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));

      services.AddIdentity<AppUser, IdentityRole>(opts => {
        opts.User.RequireUniqueEmail = true;
        // opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>()
          .AddDefaultTokenProviders();

      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Каждый раз, когда поступает запрос, промежуточное ПО трансформации заявок вызывает метод `LocationClaimsProvider.TransformAsync()`, который эмулирует источник данных человеческих ресурсов и создает специальные заявки. Для просмотра эффекта от специальных заявок запустим приложение, войдем от имени разрешенного пользователя и запросим URL вида `/Claims`.

На рис. 30.5 показаны заявки для пользователя `Alice`. Можно выйти и снова зайти, наблюдая за изменениями.

Получение заявок из множества местоположений означает, что приложение не обязано дублировать данные, которые хранятся где-то в другом месте, и делает возможной интеграцию данных от внешних участников. Свойство `Claim.Issuer` сообщает, откуда происходит заявка, что помогает судить о том, насколько точными вероятно будут данные, и какой вес они должны иметь в приложении. Например, сведения о месте жительства, полученные из центральной базы данных человеческих ресурсов, по всей видимости, будут более точными и надежными, нежели сведения, которые получены от внешнего поставщика списков рассылки.

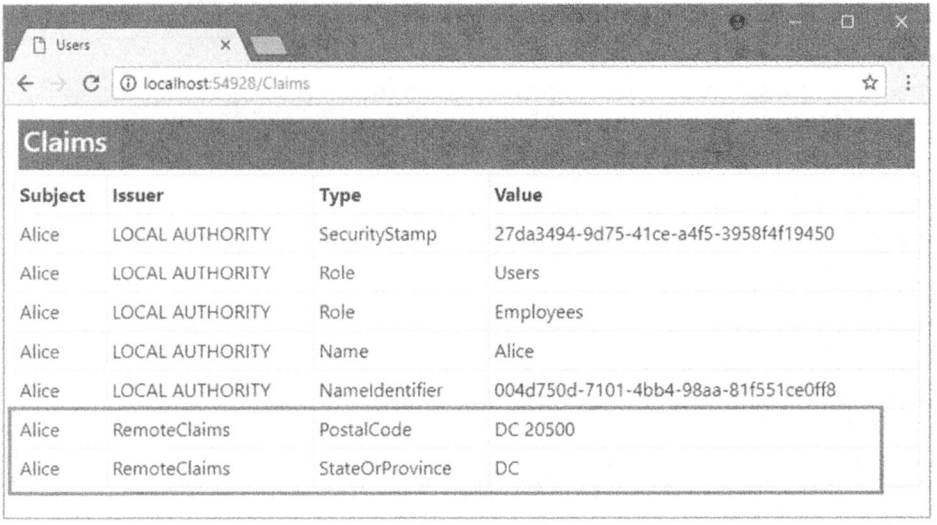


Рис. 30.5. Определение дополнительных заявок для пользователей

Создание специальных заявок удостоверений

Если в приложение необходимо добавить специальные локальные заявки, то это можно делать во время создания новых пользователей. Класс `UserManager<T>` предлагает методы `AddClaimAsync()` и `AddClaimsAsync()`, предназначенные для определения локальных заявок, которые затем хранятся в базе данных и автоматически извлекаются, когда пользователь аутентифицирован (т.е. нет нужды полагаться на средство трансформации заявок). Однако перед применением упомянутых методов следует обдумать, каким образом хранящиеся данные будут поддерживаться в актуальном состоянии, и не лучше ли, чтобы приложение обслуживалось путем динамического извлечения данных из их источника. Как объясняется в следующем разделе, заявки используются при проверках авторизации, а устаревшие данные заявок могут предоставлять пользователям доступ к частям приложения, к которым он должен быть запрещен, и предотвращать доступ к областям, куда он был разрешен.

Использование политик

Имеющиеся рабочие заявки можно применять для управления доступом пользователей к приложению более гибким образом, чем с помощью стандартных ролей. Проблема ролей в том, что они статичны, и после того, как пользователю была назначена роль, он остается ее членом до тех пор, пока не будет явно удален. Вот почему длительно работающие сотрудники крупных корпораций в итоге обладают буквально немыслимым доступом к внутренним системам: им назначаются роли, требуемые для выполнения каждого нового задания, но старые роли удаляются редко.

Заявки используются для построения политик авторизации, которые являются частью конфигурации приложения и применяются к методам действий или контроллерам посредством атрибута `Authorize`. В листинге 30.10 представлена простая политика, которая разрешает доступ только пользователям со специфическим типом и значением заявки.

Листинг 30.10. Создание политики авторизации в файле `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;
using Microsoft.AspNetCore.Authentication;
using System.Security.Claims;

namespace Users {

  public class Startup {
    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services) {

     services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();
     services.AddTransient<IUserValidator<AppUser>, CustomUserValidator>();
     services.AddSingleton<IClaimsTransformation, LocationClaimsProvider>();

      services.AddAuthorization(opts => {
        opts.AddPolicy("DCUsers", policy => {
          policy.RequireRole("Users");
          policy.RequireClaim(ClaimTypes.StateOrProvince, "DC");
        });
      });

      services.AddDbContext<AppIdentityDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreIdentity:ConnectionString"]));

      services.AddIdentity<AppUser, IdentityRole>(opts => {
        opts.User.RequireUniqueEmail = true;
        // opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
      }).AddEntityFrameworkStores<AppIdentityDbContext>()
          .AddDefaultTokenProviders();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseAuthentication();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Метод AddAuthorization() настраивает политику авторизации и предоставляет экземпляр класса AuthorizationOptions, в котором определены члены, описанные в табл. 30.7.

Таблица 30.7. Члены класса AuthorizationOptions

Имя	Описание
DefaultPolicy	Это свойство возвращает стандартную политику авторизации, которая используется, когда атрибут Authorize был применен без каких-либо аргументов. По умолчанию политика проверяет, что пользователи прошли аутентификацию
AddPolicy(name, expression)	Этот метод используется для определения новой политики, как объясняется ниже

Политики определяются с применением метода AddPolicy(), работающим с лямбда-выражением, которое имеет дело с экземпляром класса AuthorizationPolicyBuilder, предназначенным для построения политики с помощью своих методов (табл. 30.8).

Таблица 30.8. Избранные методы класса AuthorizationPolicyBuilder

Имя	Описание
RequireAuthenticatedUser()	Этот метод требует, чтобы запрос был ассоциирован с каким-нибудь аутентифицированным пользователем
RequireUserName(name)	Этот метод требует, чтобы запрос был ассоциирован с указанным пользователем
RequireClaim(type)	Этот метод требует, чтобы пользователь имел заявку указанного типа. Проверяется только присутствие заявки и принимается любое значение
RequireClaim(type, values)	Этот метод требует, чтобы пользователь имел заявку указанного типа с одним из диапазона значений. Значения могут быть выражены как аргументы, разделяемые запятыми, или в виде экземпляра реализации IEnumerable<string>
RequireRole(roles)	Этот метод требует, чтобы пользователь имел членство в роли. Множество ролей может быть указано как аргументы, разделяемые запятыми, или в виде экземпляра реализации IEnumerable<string>, и членство в любой роли обеспечит удовлетворение требования
AddRequirements(requirement)	Этот метод добавляет к политике специальное требование, как описано в разделе "Создание специальных требований политики" далее в главе

Политика в листинге 30.9 требует, чтобы пользователь имел членство в роли Users и заявку StateOrProvince со значением DC. Когда требований несколько, тогда все они должны быть удовлетворены, чтобы авторизация была одобрена.

Первый аргумент метода AddPolicy() — имя, посредством которого можно ссылаться на политику при ее применении. Именем политики из листинга 30.10 является DCUsers, и оно используется в атрибуте Authorize для применения политики к контроллеру Home в листинге 30.11.

Листинг 30.11. Применение политики авторизации в файле HomeController.cs из папки Controllers

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Users.Models;
using Microsoft.AspNetCore.Identity;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;

namespace Users.Controllers {

  public class HomeController : Controller {
    private UserManager<AppUser> userManager;

    public HomeController(UserManager<AppUser> userMgr) {
      userManager = userMgr;
    }

    [Authorize]
    public IActionResult Index() => View(GetData(nameof(Index)));

    // [Authorize(Roles = "Users")]
    [Authorize(Policy = "DCUsers")]
    public IActionResult OtherAction() => View("Index",
      GetData(nameof(OtherAction)));

    // ...для краткости другие методы не показаны...

    private Task<AppUser> CurrentUser =>
      userManager.FindByNameAsync(HttpContext.User.Identity.Name);
  }
}
```

Свойство `Policy` используется для указания имени политики, которая будет применяться, чтобы защитить метод действия. Результатом окажется комбинированная проверка имеющихся у пользователя ролей и заявок, которая выполняется, когда запрос нацелен на метод `OtherAction()`. Правильным сочетанием членства в ролях и заявок располагает только учетная запись `Alice`, в чем можно удостовериться, запустив приложение, войдя от имени разных пользователей и запрашивая URL вида `/Home/OtherAction`.

Создание специальных требований политики

Встроенные требования проверяют специфические значения и являются хорошей отправной точкой, но не позволяют обрабатывать каждый конкретный сценарий авторизации. Например, если доступ должен быть запрещен для определенного значения заявки, тогда использование встроенных требований сопряжено со сложностями из-за того, что они попросту не созданы для проверки такого рода.

К счастью, систему политик можно расширять специальными требованиями, которые представляют собой классы, реализующие интерфейс `IAuthorizationRequirement`, и специальными обработчиками авторизации, которые являются подклассами класса `AuthorizationHandler` и занимаются оценкой требования для заданного запроса. Добавим в папку `Infrastructure` файл класса по имени `BlockUsersRequirement.cs`, после чего определим в нем специальное требование и обработчик (листинг 30.12).

Листинг 30.12. Содержимое файла BlockUsersRequirement.cs из папки Infrastructure

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;

namespace Users.Infrastructure {

  public class BlockUsersRequirement : IAuthorizationRequirement {

    public BlockUsersRequirement(params string[] users) {
      BlockedUsers = users;
    }

    public string[] BlockedUsers { get; set; }
  }

  public class BlockUsersHandler
      : AuthorizationHandler<BlockUsersRequirement> {

    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        BlockUsersRequirement requirement) {
      if (context.User.Identity != null
          && context.User.Identity.Name != null
          && !requirement.BlockedUsers
            .Any(user => user.Equals(context.User.Identity.Name,
              StringComparison.OrdinalIgnoreCase))) {
        context.Succeed(requirement);
      } else {
        context.Fail();
      }
      return Task.CompletedTask;
    }
  }
}
```

Класс `BlockUserRequirement` представляет требование и применяется для указания данных, используемых при создании политики; в рассматриваемом случае это список пользователей, которые не будут авторизованы. Класс `BlockUsersHandler` ответственен за оценку запроса на авторизацию с применением данных требования и унаследован от класса `AuthorizationHandler<T>`, где `T` — тип класса требования.

Метод `Handle()` вызывается для класса обработчика, когда система авторизации нуждается в проверке доступа к ресурсу. В качестве аргументов методу передаются объект `AuthorizationHandlerContext`, который определяет члены, описанные в табл. 30.9, и объект требования, который предоставляет доступ к данным, необходимым для выполнения проверки.

Обработчик требования в листинге 30.12 проверяет имя пользователя, чтобы выяснить, находится ли оно в списке запрещенных пользователей, который предоставляет объект `BlockUsersRequirement`, и вызывает метод `Succeed()` или `Fail()` соответственно. Применение специального требования сопряжено с двумя изменениями конфигурации (листинг 30.13).

Таблица 30.9. Избранные члены класса `AuthorizationHandlerContext`

Имя	Описание
`User`	Это свойство возвращает объект `ClaimsPrincipal`, ассоциированный с запросом
`Succeed(requirement)`	Этот метод вызывается, если запрос удовлетворяет требованию. Аргументом является объект реализации `IAuthorizationRequirement`, получаемый методом `Handle()`
`Fail()`	Этот метод вызывается, если запрос не удовлетворяет требованию
`Resource`	Это свойство возвращает объект, который используется для авторизации доступа к одиночному ресурсу приложения, как объясняется в разделе "Использование политик для авторизации доступа к ресурсам" далее в главе

Листинг 30.13. Применение специального требования авторизации в файле `Startup.cs` из папки `Users`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Users.Models;
using Users.Infrastructure;
using Microsoft.AspNetCore.Authentication;
using System.Security.Claims;
using Microsoft.AspNetCore.Authorization;

namespace Users {
  public class Startup {
    public Startup(IConfiguration configuration) =>
      Configuration = configuration;
    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IPasswordValidator<AppUser>,
        CustomPasswordValidator>();
      services.AddTransient<IUserValidator<AppUser>,
        CustomUserValidator>();
      services.AddSingleton<IClaimsTransformation, LocationClaimsProvider>();
      services.AddTransient<IAuthorizationHandler, BlockUsersHandler>();

      services.AddAuthorization(opts => {
        opts.AddPolicy("DCUsers", policy => {
          policy.RequireRole("Users");
          policy.RequireClaim(ClaimTypes.StateOrProvince, "DC");
        });
        opts.AddPolicy("NotBob", policy => {
          policy.RequireAuthenticatedUser();
          policy.AddRequirements(new BlockUsersRequirement("Bob"));
        });
      });
```

```
        services.AddDbContext<AppIdentityDbContext>(options =>
          options.UseSqlServer(
            Configuration["Data:SportStoreIdentity:ConnectionString"]));

        services.AddIdentity<AppUser, IdentityRole>(opts => {
          opts.User.RequireUniqueEmail = true;
          // opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
          opts.Password.RequiredLength = 6;
          opts.Password.RequireNonAlphanumeric = false;
          opts.Password.RequireLowercase = false;
          opts.Password.RequireUppercase = false;
          opts.Password.RequireDigit = false;
        }).AddEntityFrameworkStores<AppIdentityDbContext>()
            .AddDefaultTokenProviders();

        services.AddMvc();
      }

      public void Configure(IApplicationBuilder app) {
        app.UseStatusCodePages();
        app.UseDeveloperExceptionPage();
        app.UseStaticFiles();
        app.UseAuthentication();
        app.UseMvcWithDefaultRoute();
      }
    }
}
```

Первый шаг предусматривает регистрацию класса обработчика как реализации интерфейса `IAuthorizationHandler` с помощью поставщика служб. Второй шаг предполагает добавление к политике специального требования, что делается с использованием метода `AddRequirements()`:

```
...
opts.AddPolicy("NotBob", policy => {
  policy.RequireAuthenticatedUser();
  policy.AddRequirements(new BlockUsersRequirement("Bob"));
});
...
```

Результатом будет политика, которая требует аутентифицированных пользователей, отличных от Bob, и может применяться посредством атрибута `Authorize` с указанием имени политики (листинг 30.14).

Листинг 30.14. Применение специальной политики в файле `HomeController.cs` из папки `Controllers`

```
...
// [Authorize(Roles = "Users")]
[Authorize(Policy = "DCUsers")]
public IActionResult OtherAction() => View("Index",
GetData(nameof(OtherAction)));

[Authorize(Policy = "NotBob")]
public IActionResult NotBob() => View("Index", GetData(nameof(NotBob)));
...
```

В случае аутентификации от имени пользователя Bob получить доступ к URL вида /Home/NotBob не удастся, но всем остальным учетным записям доступ будет предоставлен.

Использование политик для авторизации доступа к ресурсам

Политики могут также применяться для контроля доступа к индивидуальным ресурсам. Ресурсы — это общий термин, обозначающий любой элемент данных, который используется в приложении и требует более детализированного управления, чем возможно на уровне методов действий. Добавим в папку Models файл по имени ProtectedDocument.cs и определим в нем класс, который представляет документ с рядом свойств, касающихся владения (листинг 30.15).

Листинг 30.15. Содержимое файла ProtectedDocument.cs из папки Models

```
namespace Users.Models {

  public class ProtectedDocument {
    public string Title { get; set; }
    public string Author { get; set; }
    public string Editor { get; set; }
  }
}
```

Класс ProtectedDocument — всего лишь заполнитель для реального документа; его ключевой аспект в том, что править каждый документ разрешено только двум лицам: автору и редактору. Реальный документ потребовал бы содержимого, хронологии изменений и многих других характеристик, но для примера вполне достаточно и такого простого класса. Добавим в папку Controllers файл класса по имени DocumentController.cs и создадим в нем контроллер, как показано в листинге 30.16.

Листинг 30.16. Содержимое файла DocumentController.cs из папки Controllers

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using Users.Models;

namespace Users.Controllers {

  [Authorize]
  public class DocumentController : Controller {
    private ProtectedDocument[] docs = new ProtectedDocument[] {
      new ProtectedDocument { Title = "Q3 Budget", Author = "Alice",
        Editor = "Joe" },
      new ProtectedDocument { Title = "Project Plan", Author = "Bob",
        Editor = "Alice" }
    };

    public ViewResult Index() => View(docs);

    public ViewResult Edit(string title) {
      return View("Index", docs.FirstOrDefault(d => d.Title == title));
    }
  }
}
```

Контроллер Document поддерживает фиксированный набор объектов ProtectedDocument. Объекты ProtectedDocument применяются в действии Index, передающем все документы методу View(), и в действии Edit, которое выбирает один документ на основе аргумента title. Оба метода действий используют представление по имени Index.chstml, поэтому создадим новую папку Views/Document и поместим в нее файл Index.chstml с разметкой из листинга 30.17.

Листинг 30.17. Содержимое файла Index.cshtml из папки Views/Document

```
@if (Model is IEnumerable<ProtectedDocument>) {
  <div class="bg-primary m-1 p-1 text-white">
    <h4>Documents (@User?.Identity?.Name)</h4>
  </div>
  <table class="table table-sm table-bordered">
    <tr><th>Title</th><th>Author</th><th>Editor</th><th></th></tr>
    @foreach (var doc in Model) {
      <tr>
        <td>@doc.Title</td>
        <td>@doc.Author</td>
        <td>@doc.Editor</td>
        <td>
          <a class="btn btn-sm btn-primary" asp-action="Edit"
             asp-route-title="@doc.Title">
            Edit
          </a>
        </td>
      </tr>
    }
  </table>
} else {
  <div class="bg-primary m-1 p-1">
    <h4>Editing @Model.Title (@User?.Identity?.Name)</h4>
  </div>
  <div class="m-1 p-1">
    Document editing feature would go here...
  </div>
  <a asp-action="Index" class="btn btn-primary">Done</a>
}
<a asp-action="Logout" asp-controller="Account"
   class="btn btn-danger">Logout</a>
```

Если модель представления оказывается последовательностью объектов ProtectedDocument, то представление отображает таблицу, в которой для каждого документа предусмотрена строка с именами автора и редактора, а также со ссылкой на действие Edit. Если модель представления — одиночный объект ProtectedDocument, тогда представление отобразит заполнитель, на месте которого реальное приложение предложило бы средства редактирования.

В данный момент единственным ограничением авторизации является атрибут Authorize, примененный к классу DocumentController, который означает, что любой документ может редактировать любой пользователь, а не только автор или редактор. В этом легко убедиться, для чего запустить приложение, запросить URL вида /Document, войти от имени любого пользователя приложения и щелкнуть на кнопках

Edit (Редактировать), связанных с документами. На рис. 30.6 иллюстрируется редактирование документа `Project Plan` пользователем `Joe`.

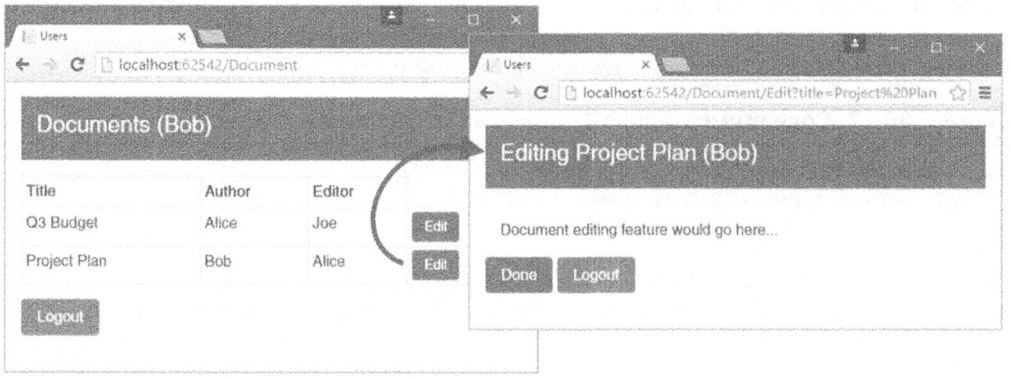


Рис. 30.6. Редактирование документов

Создание политики и обработчика авторизации доступа к ресурсам

Ограничение доступа к индивидуальным документам на уровне методов действий обеспечить трудно, потому что атрибут `Authorize` оценивается перед вызовом метода действия. Другими словами, решение относительно авторизации принимается до того, как объект `ProtectedDocument` извлечен и может быть проинспектирован с целью выяснения, каким пользователям должен быть предоставлен доступ к документу.

Чтобы решить проблему, необходимо создать политику авторизации и обработчик, которым известно, как иметь дело с объектами `ProtectedDocument`, и использовать их внутри метода действия после того, как были выявлены детали о пользователе. Добавим в папку `Infrastructure` файл по имени `DocumentAuthorization.cs` и определим в нем классы, представленные в листинге 30.18.

Листинг 30.18. Содержимое файла `DocumentAuthorization.cs` из папки `Infrastructure`

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Users.Models;

namespace Users.Infrastructure {

  public class DocumentAuthorizationRequirement
  : IAuthorizationRequirement {
     public bool AllowAuthors { get; set; }
     public bool AllowEditors { get; set; }
  }
  public class DocumentAuthorizationHandler
      : AuthorizationHandler<DocumentAuthorizationRequirement> {
    protected override Task HandleRequirementAsync(
              AuthorizationHandlerContext context,
              DocumentAuthorizationRequirement requirement) {
      ProtectedDocument doc = context.Resource as ProtectedDocument;
```

```
      string user = context.User.Identity.Name;
      StringComparison compare = StringComparison.OrdinalIgnoreCase;
      if (doc != null && user != null &&
        (requirement.AllowAuthors && doc.Author.Equals(user, compare))
      || (requirement.AllowEditors && doc.Editor.Equals(user, compare))) {
        context.Succeed(requirement);
      } else {
        context.Fail();
      }
      return Task.CompletedTask;
    }
  }
}
```

Объект AuthorizationHandlerContext предоставляет свойство Resource, обеспечивающее доступ к объекту, который может быть проинспектирован для авторизации. Класс DocumentAuthorizationHandler проверяет, содержит ли свойство Resource объект ProtectedDocument, и если содержит, то выясняет, является ли текущий пользователь автором или редактором, а также разрешает ли объект Document AuthorizationRequirement авторам или редакторам получить доступ к документу.

В листинге 30.19 класс DocumentAuthorizationHandler регистрируется в качестве обработчика для требований DocumentAuthorizationRequirement и определяется политика, которая имеет такое требование.

Листинг 30.19. Регистрация обработчика и определение политики в файле Startup.cs из папки Users

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddTransient<IPasswordValidator<AppUser>,
    CustomPasswordValidator>();
  services.AddTransient<IUserValidator<AppUser>,
    CustomUserValidator>();
  services.AddSingleton<IClaimsTransformation, LocationClaimsProvider>();
  services.AddTransient<IAuthorizationHandler, BlockUsersHandler>();
  services.AddTransient<IAuthorizationHandler,
    DocumentAuthorizationHandler>();
  services.AddAuthorization(opts => {
    opts.AddPolicy("DCUsers", policy => {
      policy.RequireRole("Users");
      policy.RequireClaim(ClaimTypes.StateOrProvince, "DC");
    });
    opts.AddPolicy("NotBob", policy => {
      policy.RequireAuthenticatedUser();
      policy.AddRequirements(new BlockUsersRequirement("Bob"));
    });
    opts.AddPolicy("AuthorsAndEditors", policy => {
      policy.AddRequirements(new DocumentAuthorizationRequirement {
        AllowAuthors = true,
        AllowEditors = true
      });
    });
  });
```

```
  services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreIdentity:ConnectionString"]));
  services.AddIdentity<AppUser, IdentityRole>(opts => {
    opts.User.RequireUniqueEmail = true;
    // opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
    opts.Password.RequiredLength = 6;
    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireDigit = false;
  }).AddEntityFrameworkStores<AppIdentityDbContext>()
      .AddDefaultTokenProviders();
  services.AddMvc();
}
...
```

Финальный шаг связан с применением политики авторизации в методе действия (листинг 30.20).

Листинг 30.20. Применение политики авторизации в файле DocumentController.cs из папки Controllers

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using Users.Models;
using System.Threading.Tasks;

namespace Users.Controllers {
  [Authorize]
  public class DocumentController : Controller {
    private ProtectedDocument[] docs = new ProtectedDocument[] {
      new ProtectedDocument { Title = "Q3 Budget", Author = "Alice",
        Editor = "Joe"},
      new ProtectedDocument { Title = "Project Plan", Author = "Bob",
        Editor = "Alice"}
    };
    private IAuthorizationService authService;
    public DocumentController(IAuthorizationService auth) {
      authService = auth;
    }
    public ViewResult Index() => View(docs);
    public async Task<IActionResult> Edit(string title) {
      ProtectedDocument doc = docs.FirstOrDefault(d => d.Title == title);
      AuthorizationResult authorized = await authService.AuthorizeAsync(User,
        doc, "AuthorsAndEditors");
      if (authorized.Succeeded) {
        return View("Index", doc);
      } else {
        return new ChallengeResult();
      }
    }
  }
}
```

Конструктор контроллера определяет аргумент IAuthorizationService, который предоставляет методы, предназначенные для оценки политик авторизации, и распознается с использованием внедрения зависимостей. В методе Edit() вызывается метод AuthorizeAsync() с передачей ему текущего пользователя, объекта ProtectedDocument и имени политики, подлежащей применению. Если метод AuthorizeAsync() в результате возвращает true, то авторизация одобрена и вызывается метод View(). Если результатом оказывается false, тогда есть проблема с авторизацией, и возвращается объект ChallengeResult, описанный в главе 17, который сообщает инфраструктуре MVC о том, что произошел отказ в авторизации.

Чтобы посмотреть на эффект, понадобится запустить приложение и запросить URL вида /Document, аутентифицируясь от имени разных пользователей. Например, в случае аутентификации как пользователя Joe будет возможность редактировать документ Q3 Budget, но не документ Project Plan.

Использование сторонней аутентификации

Одно из преимуществ системы на основе заявок, подобной ASP.NET Core Identity, связано с тем, что заявки могут поступать из внешней системы, причем даже те, которые идентифицируют пользователя для приложения. Это означает возможность аутентификации пользователя от имени приложения другими системами, и система ASP.NET Core Identity построена с учетом такой идеи, чтобы облегчить добавление поддержки для аутентификации пользователей через третьи стороны, такие как Microsoft, Google, Facebook и Twitter.

Применение сторонней аутентификации дает важные преимущества: многие пользователи уже имеют учетные записи, пользователи получают возможность выбирать использование двухфакторной аутентификации и отсутствует потребность в управлении пользовательскими учетными данными внутри приложения. В последующих разделах будет показано, как настроить и задействовать стороннюю аутентификацию для пользователей Google.

Регистрация приложения в Google

Службы сторонней аутентификации обычно требуют регистрации приложений до того, как они смогут аутентифицировать пользователей. Результатом процесса регистрации будут учетные данные, которые включаются в запрос аутентификации к сторонней службе.

Процесс регистрации Google инициируется по адресу http://console.developers.google.com, и выполняется согласно инструкциям, доступным по ссылке http://developers.google.com/identity/sign-in/web/devconsole-project. Понадобится указать URL обратного обращения, который для стандартной конфигурации выглядит как /signin-google. В среде разработки URL обратного обращения должен быть установлен в http://localhost:порт/signin-google. Для производственного приложения понадобится создать URL, который включает общедоступное имя хоста и порт.

После прохождения процесса регистрации будет получен идентификатор клиента, идентифицирующий приложение в Google, и секретный код клиента, который применяется в качестве меры безопасности, не позволяя другим приложениям выдавать себя за ваше приложение.

На заметку! Вы должны зарегистрировать собственное приложение и использовать идентификатор и секретный код клиента, сгенерированные процессом регистрации. Код в текущем разделе не будет работать до тех пор, пока вы не измените учетные данные на значения, уникальные для вашего приложения.

Включение аутентификации Google

Система ASP.NET Core Identity располагает встроенной поддержкой для аутентификации пользователей с помощью их учетных записей Microsoft, Google, Facebook и Twitter, а также более общей поддержкой для любой службы аутентификации, которая работает по протоколу OAuth. Каждая служба имеет собственный расширяющий метод, применяемый для ее регистрации в приложении внутри класса `Startup`, и в листинге 30.21 демонстрируется настройка службы Google. (Для краткости конфигурационные операторы, относящиеся к более ранним примерам, в листинге не показаны.)

Листинг 30.21. Включение аутентификации Google в файле `Startup.cs` из папки `Users`

```
...
public void ConfigureServices(IServiceCollection services) {

  services.AddTransient<IPasswordValidator<AppUser>,
    CustomPasswordValidator>();
  services.AddTransient<IUserValidator<AppUser>,
    CustomUserValidator>();
  services.AddSingleton<IClaimsTransformation, LocationClaimsProvider>();
  services.AddTransient<IAuthorizationHandler, BlockUsersHandler>();
  services.AddTransient<IAuthorizationHandler,
      DocumentAuthorizationHandler>();

  services.AddAuthorization(opts => {
    opts.AddPolicy("DCUsers", policy => {
      policy.RequireRole("Users");
      policy.RequireClaim(ClaimTypes.StateOrProvince, "DC");
    });
    opts.AddPolicy("NotBob", policy => {
      policy.RequireAuthenticatedUser();
      policy.AddRequirements(new BlockUsersRequirement("Bob"));
    });
    opts.AddPolicy("AuthorsAndEditors", policy => {
      policy.AddRequirements(new DocumentAuthorizationRequirement {
        AllowAuthors = true,
        AllowEditors = true
      });
    });
  });

  services.AddAuthentication().AddGoogle(opts => {
    opts.ClientId = "<идентификатор клиента>";
    opts.ClientSecret = "<секретный код клиента>";
  });

  services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
      Configuration["Data:SportStoreIdentity:ConnectionString"]));
```

```
  services.AddIdentity<AppUser, IdentityRole>(opts => {
    opts.User.RequireUniqueEmail = true;
    opts.Password.RequiredLength = 6;
    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireDigit = false;
  }).AddEntityFrameworkStores<AppIdentityDbContext>()
      .AddDefaultTokenProviders();

  services.AddMvc();
}
...
```

Метод `AddAuthentication().AddGoogle()` настраивает службы, требующиеся для аутентификации пользователей с помощью Google; в его вызове указываются идентификатор и секретный код клиента, которые были созданы в процессе регистрации.

При аутентификации пользователя с участием третьей стороны можно выбрать создание пользователя в базе данных Identity, которая впоследствии будет применяться для управления ролями и заявками как в случае обычных пользователей. В главе 28 был добавлен класс проверки пользователей, предотвращающий создание пользователя, если его адрес электронной почты не относится к домену `example.com`. Поскольку планируется иметь дело с пользователями из любого домена, проверку адреса электронной почты для этого примера необходимо отключить (листинг 30.22).

Листинг 30.22. Отключение проверки домена в файле `CustomUserValidator.cs` из папки `Infrastructure`

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Users.Models;

namespace Users.Infrastructure {

  public class CustomUserValidator : UserValidator<AppUser> {

    public override async Task<IdentityResult> ValidateAsync(
          UserManager<AppUser> manager, AppUser user) {

      IdentityResult result = await base.ValidateAsync(manager, user);

      List<IdentityError> errors = result.Succeeded ?
        new List<IdentityError>() : result.Errors.ToList();

      // if (!user.Email.ToLower().EndsWith("@example.com")) {
      //   errors.Add(new IdentityError {
      //     Code = "EmailDomainError",
      //     Description = "Only example.com email addresses are allowed"
      //   });
      // }
      return errors.Count == 0 ? IdentityResult.Success
        : IdentityResult.Failed(errors.ToArray());
    }
  }
}
```

Далее нужно добавить в файл представления Views/Account/Login.cshtml кнопку, которая позволит пользователям входить через Google (листинг 30.23). Для кнопок в Google предусмотрены изображения, чтобы обеспечить согласованный внешний вид с остальными приложениями, поддерживающими учетные записи Google, но ради простоты мы создадим стандартную кнопку.

Листинг 30.23. Добавление кнопки в файле Login.cshtml из папки Views/Account

```
@model LoginModel

<div class="bg-primary m-1 p-1 text-white"><h4>Log In</h4></div>

<div class="text-danger" asp-validation-summary="All"></div>

<form asp-action="Login" method="post">
  <input type="hidden" name="returnUrl" value="@ViewBag.returnUrl" />
  <div class="form-group">
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Password"></label>
    <input asp-for="Password" class="form-control" />
  </div>
  <button class="btn btn-primary" type="submit">Log In</button>
  <a class="btn btn-info" asp-action="GoogleLogin"
     asp-route-returnUrl="@ViewBag.returnUrl">
    Log In With Google
  </a>
</form>
```

Новая кнопка нацелена на действие GoogleLogin контроллера Account. В листинге 30.24 приведен код метода действия GoogleLogin() и другие изменения, внесенные в контроллер.

Листинг 30.24. Добавление поддержки аутентификации Google в файле AccountController.cs из папки Controllers

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Users.Models;
using Microsoft.AspNetCore.Identity;
using System.Security.Claims;
using Microsoft.AspNetCore.Http.Authentication;

namespace Users.Controllers {

  [Authorize]
  public class AccountController : Controller {
    private UserManager<AppUser> userManager;
    private SignInManager<AppUser> signInManager;

    // ...для краткости другие методы не показаны...

    [AllowAnonymous]
    public IActionResult GoogleLogin(string returnUrl) {
```

```
    string redirectUrl = Url.Action("GoogleResponse", "Account",
      new { ReturnUrl = returnUrl });
    var properties = signInManager
      .ConfigureExternalAuthenticationProperties("Google", redirectUrl);
    return new ChallengeResult("Google", properties);
  }

  [AllowAnonymous]
  public async Task<IActionResult> GoogleResponse(string returnUrl = "/") {
    ExternalLoginInfo info = await signInManager.GetExternalLoginInfoAsync();
    if (info == null) {
      return RedirectToAction(nameof(Login));
    }
    var result = await signInManager.ExternalLoginSignInAsync(
      info.LoginProvider, info.ProviderKey, false);
    if (result.Succeeded) {
      return Redirect(returnUrl);
    } else {
      AppUser user = new AppUser {
        Email = info.Principal.FindFirst(ClaimTypes.Email).Value,
        UserName =
          info.Principal.FindFirst(ClaimTypes.Email).Value
      };
      IdentityResult identResult = await userManager.CreateAsync(user);
      if (identResult.Succeeded) {
        identResult = await userManager.AddLoginAsync(user, info);
        if (identResult.Succeeded) {
          await signInManager.SignInAsync(user, false);
          return Redirect(returnUrl);
        }
      }
      return AccessDenied();
    }
  }
}
```

Метод `GoogleLogin()` создает экземпляр класса `AuthenticationProperties` и устанавливает свойство `RedirectUri` в URL, который нацелен на действие `GoogleResponse` того же контроллера. Следующая часть кода заставляет систему ASP.NET Core Identity реагировать на отсутствие авторизации перенаправлением пользователя на страницу аутентификации Google вместо страницы, определенной в приложении:

```
...
return new ChallengeResult("Google", properties);
...
```

Код означает, что когда пользователь щелкает на кнопке Log In With Google (Войти с помощью Google), браузер перенаправляется на страницу аутентификации Google и затем после успешного прохождения аутентификации направляется обратно на метод действия `GoogleResponse()`. Внутри метода `GoogleResponse()` детали внешнего входа получаются вызовом метода `GetExternalLoginInfoAsync()` объекта `SigninManager`:

```
...
ExternalLoginInfo info = await signInManager.
GetExternalLoginInfoAsync();
...
```

Класс `ExternalLoginInfo` определяет свойство `ExternalPrincipal`, возвращающее объект `ClaimsPrincipal`, который содержит заявки, предоставленные пользователю со стороны Google. Для входа пользователя в приложение используется метод `ExternalLoginSignInAsync()`:

```
...
var result = await signInManager.ExternalLoginSignInAsync(
  info.LoginProvider, info.ProviderKey, false);
...
```

Если вход произвести не удалось, то причина в том, что в базе данных отсутствует запись, которая представляла бы пользователя Google. Проблема решается с применением следующих двух операторов, которые создают нового пользователя и ассоциируют его с учетными данными Google:

```
...
IdentityResult identResult = await userManager.CreateAsync(user);
...
identResult = await userManager.AddLoginAsync(user, info);
...
```

На заметку! При создании пользователя системы Identity для обоих свойств `Email` и `UserName` объекта `AppUser` используется заявка адреса электронной почты, предоставленная Google, поэтому никаких конфликтов имен с пользователями, существующими в базе данных, не возникает.

Чтобы протестировать аутентификацию, запустим приложение, щелкнем на кнопке Log In With Google и введем учетные данные для допустимой учетной записи Google. Когда процесс аутентификации завершится, браузер будет перенаправлен обратно на приложение.

Резюме

В главе были описаны некоторые более сложные средства, поддерживаемые системой ASP.NET Core Identity. Здесь демонстрировалось применение специальных свойств пользователя и обеспечение их поддержки за счет обновления схемы базы данных с использованием миграций базы данных. Кроме того, объяснялась работа заявок и их применение для создания более гибких способов авторизации пользователей посредством политик. Также было показано, как использовать политики для контроля доступа к индивидуальным ресурсам, управляемым приложением. В завершение главы рассматривалась задача аутентификации пользователей через Google, которая построена на идее, лежащей в основе применения заявок. В следующей главе вы ознакомитесь с тем, как в действительности реализованы самые важные соглашения, используемые в приложениях MVC, и каким образом их можно настраивать в собственных приложениях.

ГЛАВА 31

Соглашения по модели и ограничения действий

На протяжении всей книги неоднократно подчеркивалось, что никакой магии с разработкой приложений MVC не связано, и беглый взгляд "за кулисы" позволяет легко выявить, каким образом все части подгоняются друг к другу для доставки функциональности, которая была описана в предшествующих главах.

В последней главе книги рассматриваются два полезных средства, позволяющие настраивать способ работы приложения MVC. Соглашения по модели дают возможность заменять соглашения, используемые при создании контроллеров и действий, с переопределением тех, которые применяются по умолчанию. Ограничения действий позволяют указывать, для какого вида запросов может использоваться то или иное действие. Они предоставляют инфраструктуре MVC инструкции относительно выбора действия, которое будет обрабатывать запрос.

Вы вполне можете не читать настоящую главу (из-за того, что местами материал характеризуется довольно высокой сложностью), тем не менее, имейте ее в виду, когда приложение поведет себя не так, как было задумано. Описанные в данной главе средства не придется задействовать особенно часто (или даже вообще никогда), но чем больше вы узнаете о работе инфраструктуры MVC, тем лучше будете подготовлены к решению возникающих проблем. В табл. 31.1 приведена сводка по главе.

Таблица 31.1. Сводка по главе

Задача	Решение	Листинг
Настройка модели приложения	Используйте один из встроенных атрибутов или создайте специальное соглашение по модели	31.1–31.14
Применение настройки повсюду в приложении	Определите глобальное соглашение по модели	31.15, 31.16
Проведение различия между двумя методами действий, которые могли бы обработать запрос	Используйте ограничения действий	31.17–31.25

Подготовка проекта для примера

Для целей главы мы создадим новый проект типа Empty (Пустой) по имени `ConventionsAndConstraints` с применением шаблона ASP.NET Core Web Application (Веб-приложение ASP.NET Core) из группы .NET Core.

В листинге 31.1 приведен код класса `Startup`, который настраивает MVC Framework и компоненты промежуточного ПО, необходимые при разработке.

Листинг 31.1. Содержимое файла `Startup.cs` из папки `ConventionsAndConstraints`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace ConventionsAndConstraints {

  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание модели представления, контроллера и представления

Во многих примерах главы полезно знать, какой метод использовался для реагирования на запрос. С этой целью создадим папку `Models` и добавим в нее файл класса по имени `Result.cs` с определением, показанным в листинге 31.2. Данный класс позволит контролерам в данной главе передавать представлению информацию о том, как был обработан запрос.

Листинг 31.2. Содержимое файла `Result.cs` из папки `Models`

```
using System.Collections.Generic;

namespace ConventionsAndConstraints.Models {

  public class Result {
    public string Controller { get; set; }
    public string Action { get; set; }
  }
}
```

В главе требуется единственный контроллер и представление. Создадим папку `Controllers` и добавим в нее файл класса по имени `HomeController.cs` с определением из листинга 31.3.

Листинг 31.3. Содержимое файла HomeController.cs из папки Controllers

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
namespace ConventionsAndConstraints.Controllers {
  public class HomeController : Controller {
    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

Оба метода действий в контроллере Home визуализируют представление Result. Создадим папку Views/Home и поместим в нее файл Result.cshtml с разметкой, приведенной в листинге 31.4.

Листинг 31.4. Содержимое файла Result.cshtml из папки Views/Home

```
@model Result
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link href="/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
  <title>Result</title>
</head>
<body class="m-1 p-1">
  <table class="table table-sm table-bordered">
    <tr><th>Controller:</th><td>@Model.Controller</td></tr>
    <tr><th>Action:</th><td>@Model.Action</td></tr>
  </table>
</body>
</html>
```

При стилизации HTML-элементов представление полагается на CSS-пакет Bootstrap. Создадим в корневой папке проекта файл bower.json с использованием шаблона элемента Bower Configuration File (Файл конфигурации Bower) и добавим пакет Bootstrap в раздел dependencies (листинг 31.5).

Листинг 31.5. Добавление пакета Bootstrap в файле bower.json из папки ConventionsAndConstraints

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Финальный подготовительный шаг связан с созданием внутри папки `Views` файла `_ViewImports.cshtml`, в котором настраиваются встроенные вспомогательные функции дескрипторов для применения в представлениях, а также импортируется пространство имен модели (листинг 31.6).

Листинг 31.6. Содержимое файла `_ViewImports.cshtml` из папки `Views`

```
@using ConventionsAndConstraints.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

В результате запуска приложения появится вывод, показанный на рис. 31.1.

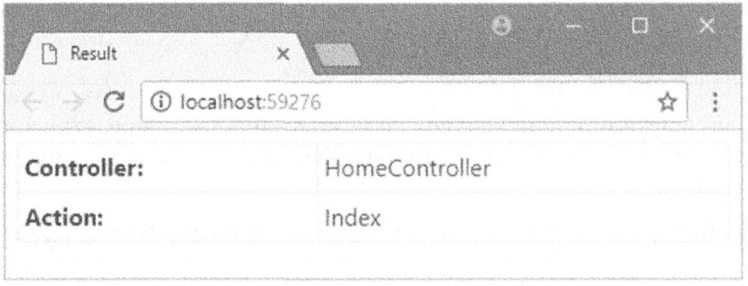


Рис. 31.1. Выполнение примера приложения

Использование модели приложения и соглашений по модели

В MVC поддерживается соглашение по конфигурации, благодаря чему можно просто создать класс с именем, заканчивающимся на `Controller`, и приступить к определению методов действий. Во время выполнения инфраструктура MVC применяет процесс обнаружения для нахождения всех контроллеров и действий в приложении и инспектирует их, чтобы выяснить, используют ли они средства, подобные фильтрам.

Итоговым результатом процесса обнаружения является модель приложения, состоящая из объектов, которые описывают каждый найденный класс контроллера, метод действия и параметр. Соглашения, на которые опирается MVC, применяются к модели приложения по мере ее построения. Например, когда обнаруживается класс контроллера, имя этого класса используется в качестве основы для контроллера, который представляет его в модели; другими словами, класс `HomeController` применяется для создания контроллера `Home`. Когда система маршрутизации идентифицирует запрос, подлежащий обработке контроллером `Home`, именно модель приложения предоставляет отображение на класс `HomeController`.

Модель приложения можно настраивать с использованием соглашений по модели. Соглашения по модели являются классами, которые инспектируют содержимое модели приложения и вносят корректировки, такие как синтез новых действий либо изменение способа применения классов для создания контроллеров. В последующих разделах объясняется, как структурирована модель приложения, рассматриваются различные типы соглашений по модели и демонстрируются приемы использования этих соглашений. В табл. 31.2 приведена сводка, позволяющая поместить модель приложения и соглашения по модели в контекст.

Таблица 31.2. Помещение модели приложения и соглашений по модели в контекст

Вопрос	Ответ
Что это такое?	Модель приложения представляет собой полное описание контроллеров и действий, которые были обнаружены в приложении. Соглашения по модели позволяют применять к модели приложения специальные изменения
Чем они полезны?	Соглашения по модели удобны, потому что они делают возможным внесение изменений в способ отображения классов и методов на контроллеры и действия. Могут проводиться и другие настройки, такие как ограничение HTTP-методов, которые принимаются действиями, или применение ограничений действий (рассматриваются далее в главе)
Как они используются?	Соглашения по модели определяются с использованием набора интерфейсов, описанных в последующих разделах, и применяются через атрибуты или конфигурируются в классе `Startup`
Существуют ли какие-то скрытые ловушки или ограничения?	Со способом применения соглашений по модели связаны определенные странности, которые обсуждаются в последующих разделах
Существуют ли альтернативы?	Нет, хотя можно ввести собственные компоненты для создания специальной модели приложения, если стандартная модель не соответствует имеющимся потребностям

Модель приложения

Во время процесса обнаружения инфраструктура MVC создает экземпляр класса `ApplicationModel` и наполняет его сведениями о найденных контроллерах и действиях. После завершения процесса обнаружения применяются соглашения по модели, чтобы внести все специальные изменения, которые были указаны. Отправной точкой для понимания модели приложения является исследование свойств, определяемых классом `Microsoft.AspNetCore.Mvc.ApplicationModels.ApplicationModel`, которые описаны в табл. 31.3.

На заметку! Может показаться, что мы начинаем с очень простого места, особенно если вы стремитесь немедленно погрузиться в детали, но полезно оценить, насколько полно классы, рассматриваемые в текущем разделе, описывают основные части приложения MVC. Понимание работы модели приложения поможет понять внутреннее функционирование более сложных средств, что способствует лучшей подготовке к диагностированию проблем, когда проекты выдают непредвиденные результаты.

Таблица 31.3. Избранные свойства класса `ApplicationModel`

Имя	Описание
`Controllers`	Это свойство возвращает объект реализации `IList<ControllerModel>`, который содержит все контроллеры в приложении
`Filters`	Это свойство возвращает объект реализации `IList<IFilterMetadata>`, который содержит глобальные фильтры в приложении

Важным свойством в настоящей главе является Controllers, возвращающее список, который содержит объекты ControllerModel для каждого контроллера, обнаруженного в приложении. Наиболее важные свойства класса ControllerModel описаны в табл. 31.4.

Таблица 31.4. Избранные свойства класса ControllerModel

Имя	Описание
ControllerName	Это свойство типа string определяет имя контроллера. Данное имя будет использоваться при сопоставлении с сегментом маршрутизации controller
ControllerType	Это свойство типа TypeInfo определяет тип класса контроллера
ControllerProperties	Это свойство возвращает объект реализации IList<PropertyModel>, который описывает все свойства, определяемые контроллером (табл. 31.5)
Actions	Это свойство возвращает объект реализации IList<ActionModel>, который описывает все действия, определяемые контроллером (табл. 31.6)
Filters	Это свойство возвращает объект реализации IList<IFilterMetadata>, который содержит фильтры, применяемые ко всем действиям в контроллере
RouteConstraints	Это свойство возвращает объект реализации IList<IRouteConstraintProvider>, который используется для ограничения нацеливания маршрутов на действия, определяемые контроллером
Selectors	Это свойство возвращает объект реализации IList<SelectorModel>, который содержит детали ограничений действий (описанные в разделе "Использование ограничений действий" далее в главе) и информацию маршрутизации, применяемую к контроллеру посредством атрибутов, как объяснялось в главе 15

Легко заметить, что классы модели приложения охватывают часть основной функциональности MVC. Например, свойство ControllerName используется для установки имени, которое будет задействовано системой маршрутизации при сопоставлении URL, а свойство ControllerType применяется для установки класса контроллера, к которому это имя относится.

Свойство ControllerProperties возвращает список объектов PropertyModel, каждый из которых описывает свойство, определяемое контроллером. Самые важные свойства класса PropertyModel перечислены в табл. 31.5.

Таблица 31.5. Избранные свойства класса PropertyModel

Имя	Описание
PropertyName	Это свойство типа string возвращает имя свойства
Attributes	Это свойство возвращает список атрибутов, которые были применены к свойству

Свойство Actions возвращает список объектов ActionModel, каждое из которых описывает метод действия, определяемый одиночным классом контроллера. Наиболее важные свойства класса ActionModel перечислены в табл. 31.6.

Таблица 31.6. Избранные свойства класса ActionModel

Имя	Описание
ActionName	Это свойство типа string определяет имя действия, которое будет использоваться при сопоставлении с сегментом маршрутизации action
ActionMethod	Это свойство типа MethodInfo применяется для указания метода, который реализует действие
Controller	Это свойство возвращает объект ControllerModel, описывающий контроллер, которому принадлежит действие
Filters	Это свойство возвращает объект реализации IList<IFilterMetadata>, содержащий фильтры, которые применяются к действию
Parameters	Это свойство возвращает объект реализации IList<PropertyModel>, который содержит описания параметров, требуемых методом действия
RouteConstraints	Это свойство возвращает объект реализации IList<IRouteConstraintProvider>, который используется для ограничения нацеливания маршрутов на действие
Selectors	Это свойство возвращает объект реализации IList<SelectorModel>, который содержит детали ограничений действий (описанные в разделе "Использование ограничений действий" далее в главе) и информацию маршрутизации, применяемую к контроллеру посредством атрибутов, как объяснялось в главе 15

Последний уровень детализации доступен через свойство Parameters, возвращающее список объектов ParameterModel, которые описывают каждый параметр, определяемый методом действия. Наиболее важные свойства класса ParameterModel перечислены в табл. 31.7.

Таблица 31.7. Избранные свойства класса ParameterModel

Имя	Описание
ParameterName	Это свойство типа string используется для указания имени параметра
ParameterInfo	Это свойство типа PropertyInfo применяется для указания параметра
BindingInfo	Это свойство типа BindingInfo используется для конфигурирования процесса привязки моделей, как было показано в главе 27

Типы ApplicationModel, ControllerModel, PropertyModel, ActionModel и ParameterModel применяются для описания любого аспекта классов контроллеров в приложении, а также их методов, свойств и параметров, определяемых каждым методом.

Настройка модели приложения

Инфраструктура MVC имеет несколько встроенных соглашений, которые она применяет в ходе заполнения `ApplicationModel` объектами `ControllerModel`, `PropertyModel`, `ActionModel` и `ParameterModel` с целью описания обнаруживаемых контроллеров.

Одни соглашения являются явными, скажем, удаление строки `Controller` из имени класса контроллера и затем использование результата для установки свойства `ControllerName` объекта `ControllerModel`. Это соглашение позволяет определять класс вроде `HomeController`, но нацеливаться на него с помощью сегмента URL, который содержит `Home`. Другие соглашения являются неявными, например, применение каждого класса для создания одного контроллера и каждого метода для создания одного действия. Большинство разработчиков приложений MVC воспринимают такие соглашения как нечто само собой разумеющееся и совершенно не задумываются о них, однако каждый аспект модели приложения может быть изменен.

В предшествующих главах были описаны атрибуты, которые изменяют способ работы инфраструктуры MVC, и в действительности они представляют собой соглашения по модели. Упомянутые атрибуты перечислены в табл. 31.8.

Таблица 31.8. Базовые атрибуты, которые изменяют стандартные соглашения приложения

Имя	Описание
`ActionName`	Этот атрибут позволяет явно указывать значение для свойства `ActionName` объекта `ActionModel` вместо того, чтобы выводить его из имени метода
`NonController`	Этот атрибут предотвращает применение класса для создания объекта `ControllerModel`
`NonAction`	Этот атрибут предотвращает использование метода для создания объекта `ActionModel`

В листинге 31.7 атрибут `ActionName` применяется для изменения имени действия, которое создано для представления метода `List()` в классе `HomeController`.

Листинг 31.7. Настройка модели приложения в файле `HomeController.cs` из папки `Controllers`

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
namespace ConventionsAndConstraints.Controllers {
  public class HomeController : Controller {
    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });
    [ActionName("Details")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

Здесь указано, что для создания действия должно использоваться имя `Details`, заменяя собой стандартное имя `List`. Чтобы взглянуть на результат, необходимо запустить приложение и запросить URL вида `/Home/Details`. Как видно на рис. 31.2, запрос обрабатывается методом `List()`.

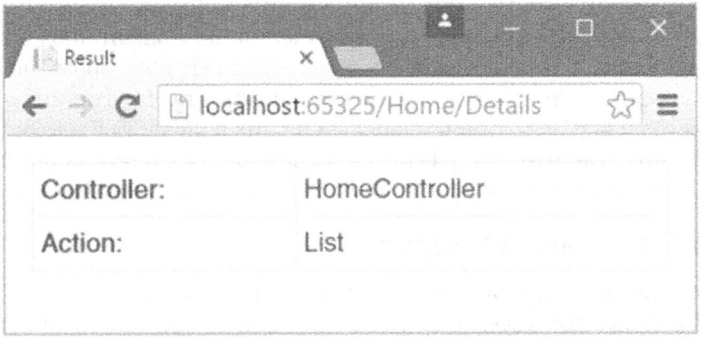


Рис. 31.2. Настройка модели приложения

Роль соглашений по модели

Атрибуты, описанные в табл. 31.8, дают возможность вносить базовые изменения в модель приложения, но ограничены в своей области действия. Для более существенных настроек требуются соглашения по модели (также известные как просто соглашения).

Атрибуты из табл. 31.8 разрешают указывать изменения в объектах модели приложения до их создания, например, переопределение имени, назначаемого действию. По контрасту с этим создание соглашения по модели позволяет изменять модель приложения, модифицируя объекты модели после их создания, что открывает возможности для применения более обширных изменений. Доступны четыре вида соглашений по модели, каждый из которых определяется своим интерфейсом (табл. 31.9).

Таблица 31.9. Интерфейсы соглашений по модели приложения

Имя	Описание
`IApplicationModelConvention`	Этот интерфейс используется для применения соглашения к объекту `ApplicationModel`
`IControllerModelConvention`	Этот интерфейс используется для применения соглашения к объектам `ControllerModel` в модели приложения
`IActionModelConvention`	Этот интерфейс используется для применения соглашения к объектам `ActionModel` в модели приложения
`IParameterModelConvention`	Этот интерфейс используется для применения соглашения к объектам `ParameterModel` в модели приложения

Все четыре интерфейса работают аналогично; меняется только уровень, на котором они функционируют внутри модели представления.

Например, вот определение интерфейса IControllerModelConvention:

```
namespace Microsoft.AspNetCore.Mvc.ApplicationModels {
  public interface IControllerModelConvention {
    void Apply(ControllerModel controller);
  }
}
```

С помощью вызова метода Apply() соглашению по модели предоставляется возможность внести изменения в объект ControllerModel, к которому оно было применено и который передается методу в качестве аргумента. Другие интерфейсы также определяют методы Apply(), причем каждый из них получает объект модели модифицируемого им типа, так что интерфейс IActionModelConvention принимает объект ActionModel, а интерфейс IParameterModelConvention — объект ParameterModel.

Создание соглашения по модели

Соглашения по модели для контроллеров, действий и параметров могут применяться как атрибуты, что облегчает установку области действия вносимых ими изменений. Создадим папку Infrastructure и добавим в нее файл класса по имени ActionNamePrefixAttribute.cs, содержимое которого приведено в листинге 31.8.

Листинг 31.8. Содержимое файла ActionNamePrefixAttribute.cs из папки Infrastructure

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
namespace ConventionsAndConstraints.Infrastructure {
  [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
  public class ActionNamePrefixAttribute
    : Attribute, IActionModelConvention {
    private string namePrefix;
    public ActionNamePrefixAttribute(string prefix) {
      namePrefix = prefix;
    }
    public void Apply(ActionModel action) {
      action.ActionName = namePrefix + action.ActionName;
    }
  }
}
```

Класс ActionNamePrefixAttribute унаследован от класса Attribute и реализует интерфейс IActionModelConvention. Его конструктор принимает строку, используемую в качестве префикса, который применяется путем модификации свойства ActionName объекта ActionModel, полученного методом Apply().

Совет. Обратите внимание, что использование атрибута ActionNamePrefix ограничивается так, что он может быть применен только к методам. Когда соглашения по модели применяются как атрибуты, соглашения по контроллерам оказывают воздействие только в случае их применения к классам, соглашения по действиям — только в случае применения к методам, а соглашения по параметрам — только в случае применения к параметрам. Соглашение, примененное на неправильном уровне, будет просто проигнорировано без возникновения ошибки. Во избежание путаницы используйте AttributeUsage, чтобы ограничить область действия создаваемых атрибутов.

В листинге 31.9 атрибут соглашения по модели применяется к одному из методов действий контроллера Home.

Листинг 31.9. Применение соглашения по модели в файле HomeController.cs из папки Controllers

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });

    [ActionNamePrefix("Do")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

Во время прохождения процесса обнаружения инфраструктура MVC создаст объект ActionModel, описывающий метод List(), выявит ActionNamePrefix и вызовет его метод Apply(). Чтобы посмотреть на эффект, понадобится запустить приложение и запросить URL вида /Home/DoList, заменяющий URL, который был бы нацелен на метод List() при стандартных соглашениях (рис. 31.3).

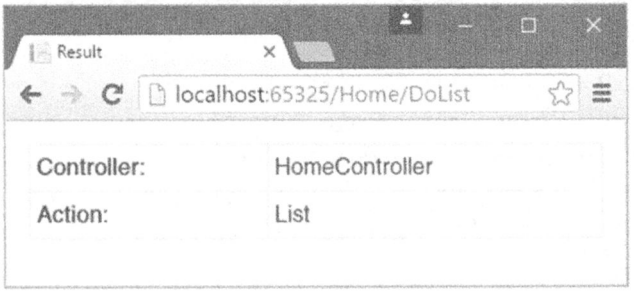


Рис. 31.3. Применение соглашения по модели

Использование соглашений, добавляющих или удаляющих объекты модели

Со способом применения соглашений по модели связана одна индивидуальная особенность, которая предотвращает добавление или удаление ими объектов модели. Например, предположим, что нужно создать соглашение, которое определенные методы смогли бы достигать через два разных действия. Добавим в папку Infrastructure файл класса по имени AddActionAttribute.cs с содержимым из листинга 31.10.

Листинг 31.10. Содержимое файла AddActionAttribute.cs из папки Infrastructure

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace ConventionsAndConstraints.Infrastructure {

  [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
  public class AddActionAttribute : Attribute, IActionModelConvention {
    private string additionalName;

    public AddActionAttribute(string name) {
      additionalName = name;
    }

    public void Apply(ActionModel action) {
      action.Controller.Actions.Add(new ActionModel(action) {
        ActionName = additionalName
      });
    }
  }
}
```

Это соглашение по действиям использует конструктор ActionModel, который дублирует настройки существующего объекта и затем изменяет свойство ActionName нового экземпляра. Новый объект ActionModel добавляется в коллекцию действий контроллера за счет навигации с помощью свойства ActionModel.Controller. В листинге 31.11 демонстрируется применение данного соглашения по модели к контроллеру Home.

Листинг 31.11. Применение соглашения по модели в файле HomeController.cs из папки Controllers

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });

    [AddAction("Details")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

После запуска приложения инфраструктура MVC начнет процесс обнаружения и сообщит о следующей ошибке:

```
InvalidOperationException: Collection was modified;
enumeration operation may not execute.
InvalidOperationException: коллекция была модифицирована;
операция перечисления может не выполниться.
```

Соглашение по модели пытается изменить набор объектов действий модели по мере его перечисления процессом обнаружения, что приводит к исключению. Во избежание ошибки потребуется избрать другой подход (листинг 31.12).

Листинг 31.12. Создание безопасного соглашения по модели в файле AddActionAttribute.cs из папки Infrastructure

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;

namespace ConventionsAndConstraints.Infrastructure {

  [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
  public class AddActionAttribute : Attribute {

    public string AdditionalName { get; }

    public AddActionAttribute(string name) {
      AdditionalName = name;
    }
  }

  [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
  public class AdditionalActionsAttribute : Attribute,
        IControllerModelConvention {

    public void Apply(ControllerModel controller) {
      var actions = controller.Actions
        .Select(a => new {
          Action = a,
          Names = a.Attributes.Select(attr =>
            (attr as AddActionAttribute)?.AdditionalName)
        });

      foreach (var item in actions.ToList()) {
        foreach (string name in item.Names) {
          controller.Actions.Add(new ActionModel(item.Action) {
            ActionName = name
          });
        }
      }
    }
  }
}
```

Модифицировать набор действий, ассоциированных с контроллером, внутри соглашения по действиям невозможно, но все равно необходим какой-нибудь способ обозначения требующихся изменений. По этой причине класс `AddActionAttribute` был сделан просто атрибутом, а не соглашением по модели.

Изменять набор действий внутри соглашения по контроллерам допускается, поэтому был создан класс `AdditionalActionsAttribute`. Метод `Apply()` использует

LINQ для нахождения методов, к которым был применен класс AddActionAttribute, и создает новые объекты ActionModel с указанными именами.

Самой важной частью данного класса является вызов метода ToList(), примененный к результатам LINQ:

```
...
foreach (var item in actions.ToList()) {
...
```

Метод ToList() инициирует перечисление запроса LINQ и помещает результат в новую коллекцию, т.е. цикл foreach проходит по набору объектов, который отличается от набора, перечисляемого инфраструктурой MVC во время применения соглашений по модели. Без вызова метода ToList() было бы получено то же самое сообщение об ошибке, которое сгенерировало соглашение по модели из листинга 31.12; благодаря вызову ToList() появляется возможность создавать новые объекты действий в модели. В листинге 31.13 иллюстрируется применение переделанных атрибутов к контроллеру Home.

Листинг 31.13. Применение переделанного соглашения по модели в файле HomeController.cs из папки Controllers

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {
  [AdditionalActions]
  public class HomeController : Controller {
    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });
    [AddAction("Details")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

Увидеть эффект от переделанного соглашения по модели можно, запустив приложение и запросив URL вида /Home/Details и /Home/List. Соглашение по модели добавляет новое действие, которое обрабатывается методом List(), дополняя созданную по умолчанию модель (рис. 31.4).

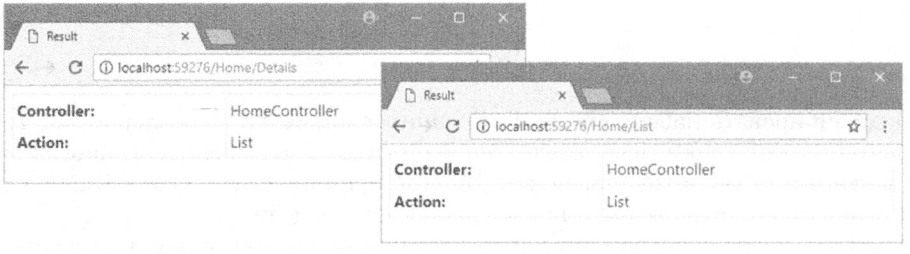


Рис. 31.4. Результат создания действия в модели

Порядок применения соглашений по модели

Соглашения по модели применяются в специфическом порядке, начиная с соглашений, имеющих наиболее широкую область действия: первыми применяются соглашения по контроллерам, затем соглашения по действиям и, наконец, соглашения по параметрам. Для демонстрации такого порядка в листинге 31.14 к методу `List()` класса `HomeController` применяются оба специальных соглашения, созданные в предшествующих примерах.

Листинг 31.14. Применение нескольких соглашений по модели в файле `HomeController.cs` из папки `Controllers`

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

  [AdditionalActions]
  public class HomeController : Controller {

    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });

    [ActionNamePrefix("Do")]
    [AddAction("Details")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

Атрибут `AdditionalActions`, который является соглашением по контроллерам, применяется первым и создает новое действие по имени `Details`. Затем применяется атрибут `ActionNamePrefix`, представляющий собой соглашение по действиям, который применяет префикс `Do` ко всем действиям, ассоциированным с методом. В результате метод `List()` реализует два действия, `DoList` и `DoDetails`, которых можно достичь через URL вида `/Home/DoList` и `/Home/DoDetails` (рис. 31.5).

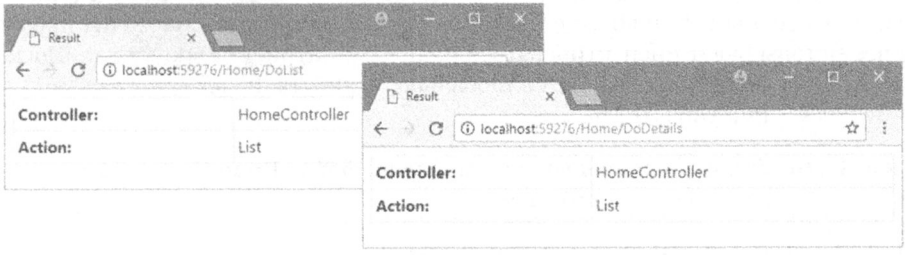


Рис. 31.5. Иллюстрация порядка применения соглашений по модели

Создание глобальных соглашений по модели

Может возникнуть необходимость изменить стандартные соглашения по модели для каждого контроллера, действия или параметра в приложении. В таком случае вместо того, чтобы помнить о необходимости последовательно применять атрибуты к каждому классу контроллера, можно создать глобальное соглашение по модели. Глобальные соглашения по модели конфигурируются в классе Startup, как показано в листинге 31.15.

Листинг 31.15. Создание глобальных соглашений по модели в файле Startup.cs из папки ConventionsAndConstraints

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc().AddMvcOptions(options => {
        options.Conventions.Add(new ActionNamePrefixAttribute("Do"));
        options.Conventions.Add(new AdditionalActionsAttribute());
      });
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Объект MvcOptions, получаемый расширяющим методом AddMvcOptions(), определяет свойство Conventions, которое возвращает коллекцию, куда можно добавлять объекты соглашений по модели. В листинге 31.15 оба специальных соглашения по модели применены глобально, т.е. имена всех действий будут снабжены префиксом Do, а все методы будут инспектироваться на предмет атрибута AddAction. Поскольку соглашения по модели применяются глобально, атрибуты из класса HomeController можно удалить (листинг 31.16).

Листинг 31.16. Удаление соглашений по модели в файле HomeController.cs из папки Controllers

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;
```

```
namespace ConventionsAndConstraints.Controllers {
  // [AdditionalActions]
  public class HomeController : Controller {
    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });
    // [ActionNamePrefix("Do")]
    [AddAction("Details")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

Глобальные соглашения по модели применяются перед соглашениями, примененными напрямую к классам. Если глобальных соглашений несколько, то они применяются в порядке, в котором регистрировались, безотносительно их типов. Соглашение по действиям было зарегистрировано перед соглашением по контроллерам и потому действие `Details`, указанное посредством атрибута `AddAction`, создается после того, как соглашение `ActionNamePrefixAttribute` будет применено ко всем именам действий. В результате метод `List()` реализует два действия, `DoList` и `Details`, которые достижимы через URL вида `/Home/DoList` и `/Home/Details` (рис. 31.6).

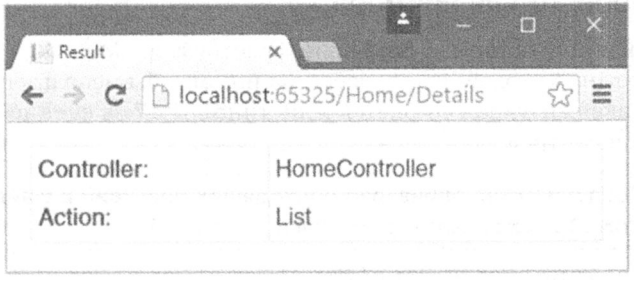


Рис. 31.6. Иллюстрация порядка выполнения глобальных соглашений по модели

Использование ограничений действий

Ограничения действий решают, подходит ли метод действия для обработки специфического запроса, и это может навести на мысль, что ограничения действий подобны фильтрам авторизации, которые рассматривались в главе 19.

На самом деле применение ограничений действий гораздо уже. Когда инфраструктура MVC получает HTTP-запрос, она проходит через процесс выбора с целью идентификации метода действия, который будет использоваться для обработки запроса. При наличии нескольких методов действий, которые могли бы обработать запрос, инфраструктуре MVC нужен способ решения, какой из них применить, и именно здесь используются ограничения действий. В табл. 31.10 приведена сводка, позволяющая поместить ограничения действий в контекст.

Таблица 31.10. Помещение ограничений действий в контекст

Вопрос	Ответ
Что это такое?	Ограничения действий — это классы, которые инфраструктура MVC применяет для выяснения, может ли запрос быть обработан специфическим действием
Чем они полезны?	Если обработать запрос способны два или большее число действий, то MVC необходимы средства для решения, какое действие подходит лучше других. Ограничения действий используются для предоставления информации такого рода
Как они используются?	Ограничения действий применяются как атрибуты, что позволяет их многократно использовать повсюду в приложении, а также означает, что логика определения, должно ли действие обработать запрос, не обязана находиться внутри самого метода действия
Существуют ли какие-то скрытые ловушки или ограничения?	Ограничения действий могут быть применены слишком широко и препятствовать обработке запроса любым подходящим методом действия, приводя к тому, что клиенту отправляется ответ `404 - Not Found` (404 — не найдено)
Существуют ли альтернативы?	Когда нужно ограничить доступ к действиям в специфических обстоятельствах, фильтры оказываются более удобными, т.к. клиента можно перенаправить для отображения полезной страницы ошибки

Подготовка проекта для примера

Цель ограничений действий — помочь инфраструктуре MVC сделать выбор среди двух и более похожих методов, когда любой из них мог бы использоваться для обработки запроса. Такая ситуация создается в листинге 31.17 за счет добавления нового метода действия в контроллер `Home`.

Листинг 31.17. Создание двух одинаково подходящих действий в файле `HomeController.cs` из папки `Controllers`

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

  // [AdditionalActions]
  public class HomeController : Controller {

    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });

    [ActionName("Index")]
    public IActionResult Other() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Other)
    });
```

```
    // [ActionNamePrefix("Do")]
    [AddAction("Details")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

В класс HomeController добавлен новый метод по имени Other(), к которому применен атрибут ActionName, так что он выпускает действие под названием Index. Кроме того, из класса Startup удалены глобальные соглашения по модели, примененные ранее в главе (листинг 31.18).

Листинг 31.18. Удаление глобальных соглашений по модели в файле Startup.cs из папки ConventionsAndConstraints

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints {

  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc().AddMvcOptions(options => {
        // options.Conventions.Add(new ActionNamePrefixAttribute("Do"));
        // options.Conventions.Add(new AdditionalActionsAttribute());
      });
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Таким образом, в контроллере Home есть два действия по имени Index. В результате запуска приложения появится сообщение об ошибке, показанное на рис. 31.7, которое свидетельствует о том, что инфраструктура MVC не знает, какое действие должно использоваться.

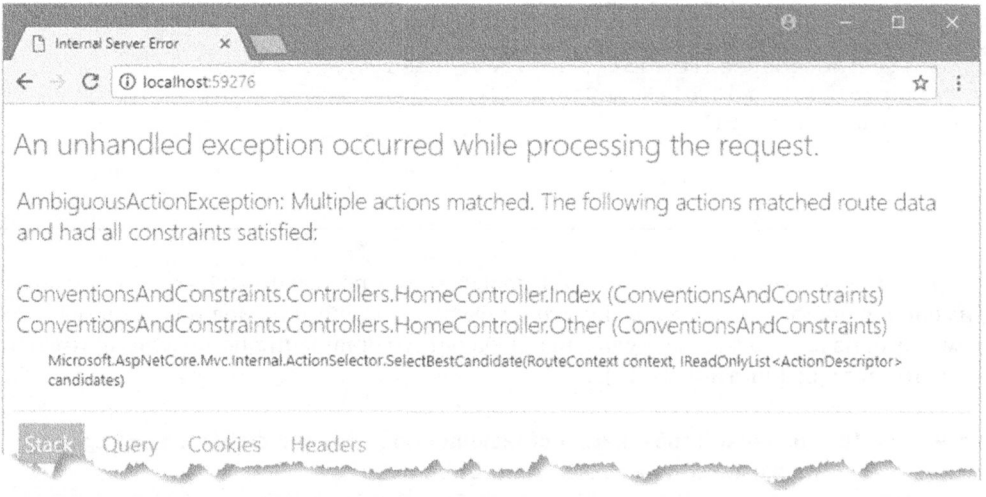


Рис. 31.7. Результат создания двух одинаково подходящих методов действий

Для удобства ниже приведена важная часть сообщения:

```
AmbiguousActionException: Multiple actions matched. The following
actions matched route data and had all constraints satisfied:
ConventionsAndConstraints.Controllers.HomeController.Index
ConventionsAndConstraints.Controllers.HomeController.Other
```

```
AmbiguousActionException: соответствие нескольким действиям. Следующие
действия соответствуют данным маршрута и все их ограничения удовлетворены:
ConventionsAndConstraints.Controllers.HomeController.Index
ConventionsAndConstraints.Controllers.HomeController.Other
```

Ограничения действий

Ограничения действий применяются для указания инфраструктуре MVC о том, может ли метод действия использоваться для обработки запроса, и для реализации интерфейса `IActionConstraint`, который определен следующим образом:

```
namespace Microsoft.AspNetCore.Mvc.ActionConstraints {
  public interface IActionConstraint : IActionConstraintMetadata {
    int Order { get; }
    bool Accept(ActionConstraintContext context);
  }
}
```

Когда инфраструктура MVC проходит через процесс выбора метода действия для обработки запроса, она выясняет, связаны ли с ним ограничения. При наличии ограничений они упорядочиваются в последовательность на основе значения свойства `Order`, и для каждого ограничения по очереди вызывается метод `Accept()`. Если для любого ограничения метод `Accept()` возвращает `false`, тогда инфраструктура MVC считает, что метод действия не может применяться для обработки текущего запроса.

Совет. Интерфейс `IActionConstraint` является производным от интерфейса `IActionConstraintMetadata`, в котором не определены какие-либо члены и который не используется напрямую. При определении специальных ограничений всегда следует применять интерфейс `IActionConstaint`, а при создании ограничения, которое имеет зависимости, подлежащие распознаванию, использовать интерфейс `IActionConstraintFactory`, как описано в разделе "Распознавание зависимостей в ограничениях действий" далее в главе.

Чтобы помочь ограничениям действий принять решение, инфраструктура MVC снабжает их экземпляром класса `ActionConstraintContext` для данных контекста, который определяет свойства, описанные в табл. 31.11.

Таблица 31.11. Свойства класса `ActionConstraintContext`

Имя	Описание
`Candidates`	Это свойство возвращает список объектов `ActionSelectorCandidate`, описывающий набор методов действий, которые инфраструктура MVC включила в окончательный список для обработки текущего запроса
`CurrentCandidate`	Это свойство возвращает объект `ActionSelectorCandidate`, описывающий метод действия, оценка которого затребована у ограничения
`RouteContext`	Это свойство возвращает объект `RouteContext`, который предоставляет информацию о данных маршрутизации (через свойство `RouteData`) и HTTP-запросе (через свойство `HttpContext`)

Создание ограничения действия

Самый распространенный тип ограничения исследует запрос для выяснения, удовлетворена ли некоторая политика, такая как наличие определенного значения в заголовке HTTP. Чтобы посмотреть, каким образом создается данный вид ограничения действия, добавим в папку `Infrastructure` файл класса по имени `UserAgentAttribute.cs` и поместим в него содержимое из листинга 31.19.

Листинг 31.19. Содержимое файла `UserAgentAttribute.cs` из папки `Infrastructure`

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ActionConstraints;

namespace ConventionsAndConstraints.Infrastructure {

  public class UserAgentAttribute : Attribute, IActionConstraint {
    private string substring;

    public UserAgentAttribute(string sub) {
      substring = sub.ToLower();
    }

    public int Order { get; set; } = 0;
```

```
    public bool Accept(ActionConstraintContext context) {
      return context.RouteContext.HttpContext
        .Request.Headers["User-Agent"]
        .Any(h => h.ToLower().Contains(substring));
    }
  }
}
```

Это атрибут ограничения действия, который предотвращает соответствие запроса действиям, когда заголовок `User-Agent` не содержит указанную строку. Внутри метода `Accept()` из объекта `HttpContext` извлекаются заголовки HTTP и посредством LINQ выясняется, содержит ли какой-нибудь из них подстроку, которая получена через конструктор.

На заметку! При идентификации браузеров в реальных приложениях не полагайтесь на заголовок `User-Agent`, потому что его значения часто обманчивы. Например, версия браузера Microsoft Edge, текущая на момент написания главы, отправляет заголовок `User-Agent`, содержащий `Android`, `Apple`, `Chrome` и `Safari`, из-за чего Microsoft Edge легко по ошибке принять за другой браузер. Более надежный подход предусматривает применение библиотеки JavaScript, такой как Modernizr (`https://modernizr.com`), для выявления функциональных возможностей, от которых зависит приложение.

В листинге 31.20 новое ограничение применяется к одному из методов класса `HomeController`.

Листинг 31.20. Применение ограничения действия в файле `HomeController.cs` из папки `Controllers`

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

  // [AdditionalActions]
  public class HomeController : Controller {

    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });

    [ActionName("Index")]
    [UserAgent("Edge")]
    public IActionResult Other() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Other)
    });

    // [ActionNamePrefix("Do")]
    [AddAction("Details")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

К методу `Other()` применяется атрибут `UserAgent`, который указывает, что действию не разрешено получать запросы с заголовком `User-Agent`, не содержащим элемент `Edge`. Запустив приложение и запросив URL вида `/Home/Index` в браузерах Google Chrome и Microsoft Edge, легко заметить, что запросы обрабатываются разными методами (рис. 31.8).

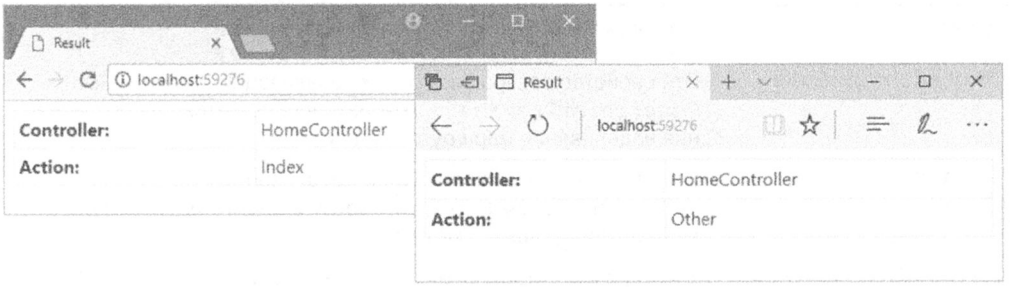


Рис. 31.8. Эффект от ограничения действия

Влияние ограничения на выбор действия

Предыдущий пример раскрыл один аспект, касающийся использования ограничений, который может стать очевидным не сразу: действию с ограничением, чей метод `Accept()` возвращает значение `true` для запроса, отдается предпочтение перед действием, к которому ограничения вообще не применялись.

Контроллер `Home` определяет два действия `Index`, созданные из методов `Index()` и `Other()`. Они оба могут применяться для обработки запросов с заголовком `User-Agent`, содержащим элемент `Edge`. Причина использования метода `Other()` для обработки запроса, поступившего от браузера Edge, связана с тем, что к нему было применено ограничение, метод `Accept()` которого возвращает `true`. Идея в том, что действие, которое имеет ограничение, принимающее запрос, является лучшим кандидатом, чем действие, вообще не имеющее ограничений.

Создание сравнивающего ограничения действия

Посредством свойств `Candidates` и `CurrentCandidate` объекта `ActionConstraintContext` ограничения снабжаются деталями о других действиях, которые являются кандидатами на обработку запроса. Каждое потенциальное соответствие описывается с использованием экземпляра класса `ActionSelectorCandidate`, который определяет свойства, приведенные в табл. 31.12.

Таблица 31.12. Свойства класса `ActionSelectorCandidate`

Имя	Описание
`Action`	Это свойство возвращает объект `ActionDescriptor`, который описывает действие-кандидат
`Constraints`	Это свойство возвращает список объектов реализации `IActionConstraint`, содержащий набор ограничений, которые были применены к действию-кандидату

Класс ActionDescriptor применяется для описания действия через свойства, перечисленные в табл. 31.13, многие из которых аналогичны свойствам, предоставляемым другими объектами контекста.

Таблица 31.13. Избранные свойства класса ActionDescriptor

Имя	Описание
Name	Это свойство возвращает имя действия
RouteConstraints	Это свойство возвращает объект реализации IList<IRouteConstraintProvider>, который используется для ограничения нацеливания маршрутов на действие
Parameters	Это свойство возвращает объект реализации IList<PropertyModel>, который содержит описания параметров, требующихся методу действия
ActionConstraints	Это свойство возвращает объект реализации IList<IActionConstraintMetadata>, содержащий ограничения для данного действия

Ограничения могут инспектировать действия-кандидаты и располагать сведениями о том, как и где они были применены, что можно использовать для точной настройки их работы. В качестве примера рассмотрим способ, которым применяются ограничения к контроллеру Home в листинге 31.21.

Листинг 31.21. Применение ограничения к контроллеру в файле HomeController.cs из папки Controllers

```
using ConventionsAndConstraints.Models;
using Microsoft.AspNetCore.Mvc;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Index)
    });

    [ActionName("Index")]
    [UserAgent("Edge")]
    public IActionResult Other() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(Other)
    });

    [UserAgent("Edge")]
    public IActionResult List() => View("Result", new Result {
      Controller = nameof(HomeController),
      Action = nameof(List)
    });
  }
}
```

В приложении есть только одно действие List и применение к нему ограничения означает, что его могут использовать лишь запросы, заголовок User-Agent которых содержит элемент Edge. Например, в случае отправки запроса из браузера Chrome будет получен ответ 404 - Not Found.

Смысла в этом мало, т.к. пользователи не поймут, по какой причине они получили ошибку, из-за отсутствия поясняющего текста, который бы предложил применить взамен другой браузер.

Ограничения удобны, когда необходимо управлять выбором метода действия для обработки запроса, но не когда нужно полностью предотвратить использование специфического действия. Если целью является как раз последнее, тогда применение фильтра позволит перенаправить клиента на описательную страницу ошибки, что будет существенно более полезным ответом.

Решение проблемы предусматривает обновление класса UserAgentAttribute, чтобы ограничение не отклоняло запросы, когда для обработки запроса доступно только одно действие-кандидат (листинг 31.22).

Листинг 31.22. Проверка наличия других действий-кандидатов в файле UserAgentAttribute.cs из папки Infrastructure

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ActionConstraints;

namespace ConventionsAndConstraints.Infrastructure {

  public class UserAgentAttribute : Attribute, IActionConstraint {
    private string substring;

    public UserAgentAttribute(string sub) {
      substring = sub.ToLower();
    }

    public int Order { get; set; } = 0;

    public bool Accept(ActionConstraintContext context) {
      return context.RouteContext.HttpContext
          .Request.Headers["User-Agent"]
          .Any(h => h.ToLower().Contains(substring))
        || context.Candidates.Count() == 1;
    }
  }
}
```

Дополнительный запрос LINQ проверяет, что действие-кандидат, возвращенное свойством CurrentCandidate, является единственным в коллекции, которую возвратило свойство Candidates. Если это так, тогда ограничение знает, что инфраструктура MVC не располагает альтернативным действием, и разрешает обработку запроса. Результат можно просмотреть, запустив приложение и запросив URL вида /Home/List в браузере Google Chrome. Хотя заголовок User-Agent, отправляемый Chrome, не содержит элемент Edge, который указан с помощью атрибута для метода List(), класс ограничения выясняет, что другие действия-кандидаты отсутствуют, и позволяет обработать запрос.

Распознавание зависимостей в ограничениях действий

Интерфейс IActionConstraintFactory используется, когда необходимо распознавать зависимости в ограничении действия посредством поставщика служб, который был описан в главе 18. Вот определение этого интерфейса:

```
using System;
namespace Microsoft.AspNetCore.Mvc.ActionConstraints {
  public interface IActionConstraintFactory : IActionConstraintMetadata {
    IActionConstraint CreateInstance(IServiceProvider services);
    bool IsReusable { get; }
  }
}
```

Метод CreateInstance() вызывается для создания новых экземпляров класса ограничения действия, а свойство IsReusable применяется для указания, могут ли объекты, возвращаемые методом CreateInstance(), использоваться для множества запросов.

Для демонстрации применения интерфейса IActionConstraintFactory нужна зависимость, которая потребует распознавания. Добавим в папку Infrastructure файл класса по имени UserAgentComparer.cs, содержимое которого приведено в листинге 31.23.

Листинг 31.23. Содержимое файла UserAgentComparer.cs из папки Infrastructure

```
using System.Linq;
using Microsoft.AspNetCore.Http;

namespace ConventionsAndConstraints.Infrastructure {

  public class UserAgentComparer {
    public bool ContainsString(HttpRequest request, string agent) {
      string searchTerm = agent.ToLower();
      return request.Headers["User-Agent"]
        .Any(h => h.ToLower().Contains(searchTerm));
    }
  }
}
```

В классе UserAgentComparer определен единственный метод, который ищет строку внутри заголовка User-Agent в HTTP-запросе. Это та же самая функциональность, которая использовалась ранее, но упакованная в отдельный класс, так что его жизненным циклом можно управлять с применением поставщика служб, сконфигурированного в листинге 31.24.

Листинг 31.24. Регистрация типа для поставщика служб в файле Startup.cs из папки ConventionsAndConstraints

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```
using Microsoft.Extensions.DependencyInjection;
using ConventionsAndConstraints.Infrastructure;

namespace ConventionsAndConstraints {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddSingleton<UserAgentComparer>();
      services.AddMvc().AddMvcOptions(options => {
        // options.Conventions.Add(new ActionNamePrefixAttribute("Do"));
        // options.Conventions.Add(new AdditionalActionsAttribute());
      });
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseStatusCodePages();
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Здесь выбран жизненный цикл одиночки, т.е. будет использоваться единственный экземпляр `UserAgentComparer`. В листинге 31.25 показано обновленное ограничение `UserAgent`, которое теперь делегирует работу по инспектированию заголовка объекту `UserAgentComparer`, получаемому с помощью поставщика служб.

Листинг 31.25. Распознавание зависимостей в файле `UserAgentAttribute.cs` из папки `Infrastructure`

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ActionConstraints;
using Microsoft.Extensions.DependencyInjection;

namespace ConventionsAndConstraints.Infrastructure {

  public class UserAgentAttribute : Attribute, IActionConstraintFactory {
    private string substring;
    public UserAgentAttribute(string sub) {
      substring = sub;
    }
    public IActionConstraint CreateInstance(IServiceProvider services) {
     return new UserAgentConstraint(services.GetService<UserAgentComparer>(),
       substring);
    }
    public bool IsReusable => false;
    private class UserAgentConstraint : IActionConstraint {
      private UserAgentComparer comparer;
      private string substring;
      public UserAgentConstraint(UserAgentComparer comp, string sub) {
        comparer = comp;
        substring = sub.ToLower();
      }
      public int Order { get; set; } = 0;
      public bool Accept(ActionConstraintContext context) {
```

```
            return comparer.ContainsString(context.RouteContext
                .HttpContext.Request, substring)
                || context.Candidates.Count() == 1;
        }
    }
  }
}
```

В приведенной модели атрибут, примененный к методам действий, отвечает за создание экземпляров класса ограничения, когда вызывается его метод `CreateInstance()`. Аргументом метода `CreateInstance()` является объект реализации `IServiceProvider`, используемый в примере для получения объекта `UserAgentComparer`, поэтому можно создавать экземпляр закрытого класса ограничения, который затем будет задействован в процессе выбора.

Избегание ловушки, связанной с областью действия

Подобно другим средствам, основанным на атрибутах, применение атрибута ограничения к классу контроллера эквивалентно применению данного атрибута к каждому отдельному методу. Тем не менее, обычно это приводит к нежелательным результатам, поскольку целью ограничений является помощь инфраструктуре MVC в выборе метода действия, а применение одного и того же ограничения ко всем действиям контроллера в целом не способствует достижению указанной цели.

Например, если применить атрибут `UserAgent` к классу `HomeController`, то действия `Index` перестанут быть достижимыми из любого браузера. Оба действия `Index` будут одинаково подходящими для браузеров, которые включают в заголовок `User-Agent` элемент `Edge`, что в итоге даст исключение. Всем остальным браузерам не подойдет ни одно из действий `Index`, что в результате приведет к ответу `404 - Not Found`.

Внутри ограничения можно было бы воспользоваться объектом контекста, чтобы найти другие ограничения и посмотреть, отклонят ли они запрос, но тогда метод `Accept()` каждого ограничения будет многократно вызываться для каждого запроса, что окажется затратным процессом, которого лучше избегать.

Ограничения хорошо работают в ситуации, когда есть множество методов действий, которые могут обработать один и тот же запрос, из-за чего к ним и применяется ограничение.

Резюме

В главе были описаны два средства, которые используются для настройки работы MVC. Вы узнали, как применять соглашения по модели для изменения способа отображения классов и методов на контроллеры и действия. Кроме того, было показано, каким образом использовать ограничения действий для сужения диапазона запросов, которые действие может обрабатывать, и как их применять для выбора действия из списка кандидатов, идентифицированных при поступлении запроса.

Итак, изучение инфраструктуры ASP.NET Core MVC 2 завершено. Оно начиналось с создания простого приложения, после чего вы совершили комплексный тур по разнообразным компонентам в инфраструктуре, научившись их конфигурировать, настраивать или полностью заменять.

Желаю вам успехов в разработке собственных приложений MVC и надеюсь на то, что вы получили не меньшее удовольствие от чтения этой книги, чем я во время ее написания.

Предметный указатель

A

A/A/A (Arrange/Act/Assert), 184
API (Application Programming Interface), 25
ASP.NET Core, 25
ASP.NET Core Identity, 877; 913; 941
ASP.NET Core MVC, 22; 26; 29; 80; 176
ASP.NET MVC Framework, 24
ASP.NET Web Forms, 23

B

Bootstrap, 152; 243; 561; 728; 793
Bower, 149
  установка, 359
    в Linux и macOS, 359
    в Windows, 359
Browser Link (Ссылка на браузер), 162; 414

C

CDN (Content Delivery Network), 790
CRUD (Create, Read, Update, Delete), 205; 308
CSRF (Cross-Site Request Forgery), 760
CSS (Cascading Style Sheets), 24; 782

D

DAL (Data Access Layer), 78
DI (Dependency Injection), 556

E

Entity Framework Core (EF Core), 221

F

Font Awesome, 286

G

Git
  установка, 358
    в Linux, 358
    в Windows и macOS, 358

H

HTML, 548; 650; 707; 731

J

JavaScript, 782; 871
JQuery, 646; 784
JSON (JavaScript Object Notation), 548; 642; 650; 683

K

Kestrel, 391

L

LINQ (Language Integrated Query), 29
LTS (Long Term Support), 357

M

MaxCDN, 794
Model-view-controller (MVC), 22; 72
Model-view-presenter (MVP), 79
Model-view-view model (MVVM), 79
MVC Framework, 24

N

.NET Core
  установка, 359
    в Linux, 359
    в Windows и macOS, 359
.NET Core 2.0 SDK, 32
Node, 356
  установка
    в Linux, 358
    в Windows и macOS, 357
NPM (Node Package Manager), 359
NuGet, 147; 148; 372
  диспетчер пакетов NuGet, 385

O

ORM (Object-Relational Mapping), 221

P

POCO, 519

R

Razor, 122; 135; 153; 517; 593; 667; 685; 705
Razor Pages, 26; 672
REST (Representational State Transfer), 635

S

SportsStore, 205
SQLite, 371

U

URL (Uniform Resource Locator), 28; 241; 253; 805
  входящий, 435; 494
  генерирование
    без ссылок, 490
    в методах действий, 491
    из специфического маршрута, 488
    исходящих URL, 479; 502
    полностью заданных URL, 487
  исходящий, 435; 477
  маршрутизация, 435
  сегмент URL
    необязательный, 455
    статический, 447
  полезные советы относительно схемы URL, 510
  соответствие URL, 446
  сопоставление URL, 441
  шаблоны URL, 440
  явно определенный, 806

V

Visual Studio, 98; 143; 151; 158; 172
Visual Studio 2017 Community, 31
Visual Studio Code, 356; 360; 363
  установка, 360
    в Linux, 361
    в macOS, 360
    в Windows, 360

X

XHTML, 650
XML, 650
XUnit.net, 196

A

Авторизация, 330; 341
  на основе ролей, 936
  политика авторизации, 335
  пользователей с помощью ролей, 922
Администрирование, 303
Антипаттерн, См. Интерфейс, Интеллектуальный пользовательский, 77
Архитектура
  MVC, 26
  "модель-представление", 77
  трехуровневая, 78
Атрибут
  внедрения в свойства, 587
  встроенной вспомогательной функции дескриптора для элементов script, 783
  проверки достоверности, 865; 867
  установка значений атрибутов, 137
Аутентификация, 330; 341
  включение аутентификации Google, 968
  пользователей, 914
  сторонняя, 967

Б

База данных
  SQLite, 371
  миграции баз данных, 227; 372; 946
  переустановка базы данных, 294
  подготовка базы данных, 221
  расширение базы данных, 294
  создание, 341; 370
Безопасность
  тестирование политики безопасности, 341
Библиотека
  jQuery, 646; 784; 871
  jQuery Validation, 873
Брандмауэр, 342
Брокер типов, 565

B

Виджет
  с итоговой информацией по корзине, 286
Визуализация веб-страниц, 39
Внедрение
  в действия, 586
  в свойства, 587
  зависимостей (DI), 395; 526; 556; 562; 569; 575
    для фильтров, 614
    конфигурирование, 426
Вывод
  генерирование вывода, 732
  динамический, 43
Выражение
  Razor, 135
  лямбда-, 108; 112

Г

Генерирование
  HTML-ответа, 532
  URL
    без ссылок, 490
    в методах действий, 491
    из специфического маршрута, 488
    исходящих, 479; 502
    полностью заданных, 487
  вывода, 732
  исходящих ссылок, 480
  ответа, 528
    с использованием объекта контекста, 529
  ссылок на действия, 509

Д

Данные
  вставка значений данных, 136
  кеш данных, 798
  контекста, 517; 522
  получение данных контекста, 522
  уровень доступа к данным (DAL), 78
  форма для ввода данных, 178
  форматирование данных, 135
Действие, 184; 513; 517
  внедрение в действия, 586
  ловушки, связанные с областью действия, 1000
  ограничение действий, 989; 992
    распознавание зависимостей в ограничениях действий, 998
Диспетчер пакетов NuGet, 385
"Дыры", 98

Ж

Жизненный цикл
  ограниченный областью действия, 583
  одиночки, 586
  фильтров, 618

З

Зависимости
  распознавание зависимостей в ограничениях действий, 998
Заказ
  обработка заказа, 298
  отображение итоговой страницы заказа, 301
  отправка заказов, 289
  управление заказами, 303; 307

Запрос
GET, 512; 546
POST, 512; 546
конвейер запросов, 392
подделка межсайтовых запросов (CSRF), 760
Защита, 330
от стандартных значений привязки модели, 817
средств администрирования, 330
Заявка, 948
создание заявок, 952

И

Имя
получение имен, 120
Инициализатор
коллекции, 99
объекта, 99
Инструмент
Bower, 149
NuGet, 147
Интерполяция строк, 98
Интерфейс
IActionResult, 530
IApplicationBuilder, 229; 407
IAsyncActionFilter, 604
IAsyncExceptionFilter, 611
IConfiguration, 421
IEnumerable<Product>, 105
IEnumerable<T>, 213
IFilterDiagnostics, 615
IFilterMetadata, 596
IHostingEnvironment, 408; 582
IIdentity, 915
ILoggingBuilder, 423
IMvcBuilder, 427
IOrderedFilter, 626
IOrderRepository, 295
IProductRepository, 213
IQueryable<T>, 213
IRepository, 190; 569; 639
IRouter, 493
IUrlHelper, 255
IView, 662
IViewEngine, 671
интеллектуальный пользовательский (антипаттерн), 77
Инфраструктура
xUnit.net, 196
имитации, 201

К

Категория
генерация списка категорий, 260
генерация ссылок на категории с помощью компонента представления, 261
подсветка текущей категории, 262
счетчик товаров определенной категории, 267
Кеш, 798; 803
аннулирование кеша, 789
Класс
ActionConstraintContext, 993
ActionContext, 530
ActionDescriptor, 996
ActionExecutedContext, 602
ActionExecutingContext, 602
ActionModel, 979
AnchorTagHelper, 795
AppUser, 883
Assert, 185
Asset, 184
AuthorizationFilterContext, 598
AuthorizationOptions, 957
AuthorizationPolicyBuilder, 957
ButtonTagHelper, 737
CacheTagHelper, 803
Cart, 269; 281
Claim, 950
ClaimsIdentity, 949
ClaimsPrincipal, 949
Controller, 521; 533; 548; 553
ControllerModel, 978
CustomHtmlResult, 531
DbContext, 223
DefaultModelBindingMessageProvider, 858
DiagnosticsFilter, 617
DictionaryStorage, 573
EFOrderRepository, 295
EnvironmentTagHelper, 781
ErrorMiddleware, 406
ExceptionContext, 612
ExceptionFilterAttribute, 612
FakeProductRepository, 214
FakeRepository, 216
FilterContext, 597
FormTagHelper, 759
HomeController, 195; 561
IdentityResult, 891
IdentityRole, 923
IdentityUser, 882
LecagyRoute, 495
LinkTag, 794
LinkTagHelper, 792
MemoryRepository, 561
Mock, 204
ModelStateDictionary, 851
ModelValidationContext, 869
MvcRouteHandler, 500
OrderController, 297
ParameterModel, 979
PasswordOptions, 895
PocoController, 519
Product, 187
Program, 380; 387
PropertyModel, 978
ProtectedDocument, 962
RazorViewEngineOptions, 686
ReservationController, 639
ResultExecutedContext, 606
ResultExecutingContext, 606
ResultFilterAttribute, 606
RouteContext, 495

ScriptTagHelper, 782; 784
Startup, 157; 208; 219; 380; 391; 393; 894
TagHelperContent, 743
TagHelperContext, 731
TagHelperOutput, 743
TextAreaTagHelper, 777
TimeFilter, 616
TypeBroker, 565
UrlResolutionTagHelper, 805; 807
UserOptions, 900
ViewComponent, 262; 703; 711
ViewComponentContext, 710
ViewContext, 662
ViewDataDictionary, 662
ViewEngineResult, 661
ViewLocationExpanderContext, 689
ViewResult, 532
ViewResultDetailsAttribute, 607
VirtualPathContext, 503
WebBostBuilderContext, 417
вспомогательной функции дескриптора, 729
гибридный, 720
компиляция классов
автоматическая, 155
контекста базы данных, 222
контроллера
гибридный, 718
модели
создание, 289
хранилища
создание, 223
Классы C#, 154
Ключевое слово
async, 118
await, 118
this, 104
Коды HTTP, 552
Коллекция
инициализатор коллекции, 99
Компиляция классов
автоматическая, 155
Компонент
вспомогательной функции дескриптора, 723
изолирование компонента, 189
представления, 693; 700
POCO, 701
асинхронный, 715
модульное тестирование компонентов представлений, 715
навигационный, 257
помещение в контекст, 693
применение данных контекста в компоненте представления, 712
простой, 702
сильно связанный, 562
слабо связанный, 561
Компоновка, 129; 131
создание, 130; 725
Конвейер
запросов, 392
промежуточного ПО, 399
Контекст
получение данных контекста, 522
помещение внедрения зависимостей в контекст, 556
помещение компонентов представлений в контекст, 693
помещение контроллеров API в контекст, 627
применение данных контекста в компоненте представления, 712
Контроллер, 37; 74; 274; 283; 337; 372; 513; 517; 630
API, 627; 635
помещение контроллеров API в контекст, 627
тестирование, 641
CRUD, 308
Document, 963
MVC, 497
Order, 297
POCO, 519
создание, 518
REST, 633
добавление контроллера, 216
создание, 90; 124; 559; 696; 725
гибридных классов контроллеров, 718
примеров контроллеров, 438
тестирование контроллера, 564
Конфигурация маршрутизации
стандартная, 457
Конфигурирование
внедрения зависимостей, 426
механизма Razor, 685
поставщика служб, 570
представления, 217
приложения, 208; 224; 345; 366; 380; 416; 699; 728; 813
ролей, 943
служб MVC, 427
элементов input, 763
Корзина для покупок, 267
виджет с итоговой информацией по корзине, 286
контроллер для корзины, 274
отображение содержимого корзины, 277
службы корзины, 283
удаление элементов из корзины, 284

Л

Лямбда-выражение, 108; 112

М

Маршрут, 39; 638
именованный, 490
навигация с использованием простого маршрута, 443
ограничение маршрута, 460
на основе типов и значений, 465
на уровне сегментов, 462
с использованием регулярного выражения, 464
переменной длины, 458
создание специального класса маршрута, 493
упорядочение маршрутов, 449

Маршрутизация, 28; 39; 407
URL, 28; 435
входящих URL, 494
на контроллеры MVC, 497
на основе соглашений, 471
настройка системы маршрутизации, 492
с помощью атрибутов, 471
стандартная конфигурация маршрутизации, 457
Метод, 517
асинхронный, 116
вспомогательный
HTML, 731
действия, 37; 517
расширяющий, 103
Механизм
Razor, 705
Минификация, 168; 172
Модель, 46; 73; 90
предметной области, 46
привязка модели, 53; 522; 809
приложения, 976
проверка достоверности модели, 320; 844
расширение модели, 303
соглашения по модели, 976; 982
глобальные, 988
создание, 144; 558; 846
класса модели, 289; 437
Модель-представление-контроллер (MVC), 72
Модульное тестирование
в Visual Studio Code, 376
приложений MVC, 176

Н

Навигация, 249
меню навигации по категориям, 257
с использованием простого маршрута, 443

О

Область
генерирование ссылок на действия в областях, 509
заполнение области, 506
создание, 505
маршрута для области, 505
Оператор
case, 102
if, 139
switch, 102; 139
условный, 138
Операция
CRUD, 308
Delete, 645
GET, 641
null-условная, 91
Patch, 644
POST, 643
PUT, 643
Организация, 184
Ошибка
возвращение ошибок, 552
в результате проверки пароля, 894
обработки свойства модели, 816
при попытке создать нового пользователя, 893
проверки достоверности, 300; 857

П

Пакет
Bootstrap, 152; 561; 728
установка, 243
Entity Framework Core
установка, 221
Font Awesome, 286
Microsoft.AspNetCore.All, 147
Moq, 201
NuGet
диспетчер пакетов NuGet, 385
установка, 372
Пакетирование, 168; 172
Пароль
проверка паролей, 893
Паттерн
MVC, 72
в ASP.NET Core, 74
Post/Redirect/Get, 546
REST, 638
"Интеллектуальный пользовательский интерфейс" (Smart UI), 76
"Локатор служб" (Service Locator), 588
"модель-представление-модель представления" (model-view-view model, MVVM), 79
"модель-представление-презентатор" (model-view-presenter, MVP), 79
Перенаправление, 541
временное, 541
на URL системы маршрутизации, 543
на буквальный URL, 542
на метод действия, 544
постоянное, 541
Политики, 955
Пользователь
авторизация пользователей с помощью ролей, 922
аутентификация пользователей, 914
Порт
HTTP, 632
Представление, 74; 90; 337; 372; 559; 630; 658
Razor, 153; 670
визуализация представлений, 673
гибридное, 719
добавление, 217
содержимого JSON в представления, 683
запуск представления, 133
импортирование представлений, 128
конфигурирование, 217
создание, 125; 260; 439; 725; 910
строго типизированное, 46; 537
частичное, 705
создание, 246
Привязка
выбор стандартного источника данных привязки, 837
коллекций, 831
массивов, 829
моделей, 53; 522; 809; 814

защита от стандартных значений привязки модели, 817
свойств, 827
типов
простых, 818
сложных, 819; 839
Приложение
MVC
модульное тестирование, 180
SportsStore, 205
запуск приложения, 211; 220; 366
использование URL, относительных к приложению, 805
конфигурирование, 208; 224; 332; 345; 366; 380; 416; 699; 728; 813
одностраничное, 75
развертывание приложения, 341; 351
регистрация приложения в Google, 967
Проверка достоверности
JavaScript, 871
атрибуты проверки достоверности, 865
модели, 850
явная, 850
на стороне клиента, 870
ненавязчивая, 870
на стороне сервера, 870
правила проверки достоверности с помощью метаданных, 864
удаленная (дистанционная), 873
Программное обеспечение (ПО)
промежуточное, 219; 392; 397
конвейер промежуточного ПО, 399
создание, 400; 402
Пространство имен
Xunit, 184
Протокол
SSL, 591

P

Развертывание, 330
Разметка
HTML, 709
Разработка через тестирование, 194
Редактор
Visual Studio Code, 361; 362; 368
кода, 160
Роль
конфигурирование ролей, 943
соглашений по модели, 981

C

Свойства
создание специального атрибута проверки достоверности для свойства, 867
Сервер
Kestrel, 391
Сеть
доставки содержимого (CDN), 790; 794
"Синтаксический сахар", 99; 101
Система
ASP.NET Core Identity, 877
маршрутизации ASP.NET, 39
Службы
ASP.NET, 394
MVC, 396
конфигурирование, 427
жизненный цикл службы, 577
конфигурирование поставщика служб, 570
корзины, 283
Соглашения
по конфигурации, 84
Содержимое
JSON, 683
динамическое, 674
сеть доставки содержимого, 794
статическое, 168; 674
стилизация содержимого, 65; 242
Среда
Visual Studio, 151; 158
подготовительная (Staging), 409
производственная (Production), 409
разработки (Development), 409
Средство Browser Link, 162
Ссылка
генерирование исходящих ссылок, 480
на проект
создание, 182
на страницы, 239
Стек технологий ASP.NET Web Forms, 22
Стилизация
представления, 66
Страница
добавление ссылок на страницы, 239
отображение ссылок на страницы, 239
Строка
интерполяция строк, 98
Схема URL, 510
Счетчик
страниц, 265
товаров определенной категории, 267

T

Таблица стилей CSS, 63; 169; 792
создание, 63
Тест
модульный, 195; 251
параметризованный, 196
прогон тестов, 378
Тестирование
контроллера, 564
API, 641
механизма визуализации, 665
модульное, 205; 561
в Visual Studio Code, 376
операции
Delete, 645
GET, 641
Patch, 644
POST, 643
PUT, 643
представления, 42
приложений MVC
модульное, 176; 180

Технология
ASP.NET Web Forms, 22
Web Forms, 24
Тип
анонимный, 114
брокер типов, 565
привязка простых типов, 818
привязка сложных типов, 819
Типизация
неявная, 114
Точка останова, 159

У

Уровень доступа к данным (data access layer, DAL), 78
Установка
.NET Core 2.0 SDK, 32
Visual Studio, 31
Утверждение, 184
Учетная запись
добавление новой пользовательской учетной записи, 893

Ф

Файл
bower.json, 150; 365
Comparer.cs, 187
CSS, 167
Default.cshtml, 264
fourth.js, 170
HomeController.cs, 145
Index.cshtml, 145
IProductRepository.cs, 213
JavaScript, 167; 783
JSON, 418
Order.cs, 289; 303
Product.cs, 212
ProductTests.cs, 186
SessionCart.cs, 282
SimpleRepository.cs, 144
Startup.cs, 458
third.js, 170
дистрибутива Bootstrap, 793
запуска представления, 133
импортирования представлений, 128
исключение файлов, 787
минификация, 172
пакетирование, 172
представления, 533
Фильтрация
по категории, 253
Фильтры, 589; 593
глобальные, 596; 621
действий, 601
асинхронные, 604
гибридные, 609
исключений, 611
использование внедрения зависимостей для фильтров, 614
порядок применения фильтров, 624
результатов, 605
асинхронные, 608
гибридные, 609
управление жизненными циклами фильтров, 618
Форма
проверка достоверности форм, 778
Формат
HTML, 650
JSON, 419; 548; 642; 650
XHTML, 650
XML, 650
Функция
дескриптора
вспомогательная, 723; 729; 733; 740; 778; 779

Х

Хранилище, 558; 628
службы хранилища
создание, 214
создание, 212; 725
фиктивное
создание, 213

Ш

Шаблоны URL, 440

Э

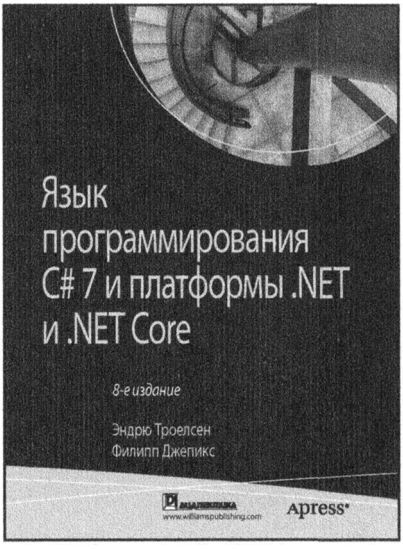
Элемент
cache, 798
img, 797
input
конфигурирование, 763
label, 768
option, 772; 774
script, 783
атрибуты встроенной вспомогательной функции дескриптора, 783
select, 771
textarea, 776
сокращающий, 741
управления
навигационный, 249
якорный, 479; 795

Я

Язык
C#, 87; 117
Якорный элемент, 479

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 7 И ПЛАТФОРМЫ .NET И .NET CORE
8-Е ИЗДАНИЕ

Эндрю Троелсен
Филипп Джепикс



www.williamspublishing.com

Эта классическая книга представляет собой всеобъемлющий источник сведений о языке программирования C# и о связанной с ним инфраструктуре. В 8-м издании книги вы найдете описание функциональных возможностей самых последних версий C# 7.0 и 7.1 и .NET 4.7, а также совершенно новые главы о легковесной межплатформенной инфраструктуре Microsoft .NET Core, включая версию .NET Core 2.0. Книга охватывает ASP.NET Core, Entity Framework (EF) Core и т.д. наряду с последними обновлениями платформы .NET, в том числе внесенными в Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF) и ASP.NET MVC. Книга предназначена для опытных разработчиков ПО, заинтересованных в освоении новых средств .NET 4.7, .NET Core и языка C#. Она будет служить всеобъемлющим руководством и настольным справочником как для тех, кто впервые переходит на платформу .NET, так и для тех, кто ранее писал приложения для предшествующих версий .NET.

ISBN 978-5-6040723-1-8

в продаже

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

ASP.NET Core MVC 2 с примерами на C# для профессионалов

В этом, 7-м, издании книга-бестселлер по MVC обновлена с учетом версии ASP.NET Core MVC 2. Она содержит подробные объяснения функциональности Core MVC, которая позволяет разработчикам выпускать более экономные, оптимизированные под облако и готовые к функционированию на мобильных устройствах приложения для платформы .NET. В книге показано, как поместить ASP.NET Core MVC в контекст разработки, а также глубоко исследованы инструменты и приемы, требуемые для построения современных, оптимизированных под облако, расширяемых веб-приложений. Детально описаны все новые средства MVC и приведены пояснения, как наиболее эффективно применять их к новым и существующим проектам.

Инфраструктура ASP.NET Core MVC Framework — это последнее развитие веб-платформы ASP.NET от Microsoft, построенное на совершенно новом фундаменте. Она представляет основополагающее изменение в том, каким образом Microsoft конструирует и развертывает веб-инфраструктуры, и свободна от наследия более ранних технологий, подобных Web Forms. Инфраструктура ASP. NET Core MVC предлагает "независимую от хоста" структуру и высокопродуктивную модель программирования, которая способствует построению более чистой кодовой архитектуры, разработке через тестирование и значительной расширяемости.

Автор многочисленных бестселлеров Адам Фримен тщательно пересмотрел эту лидирующую на рынке книгу и объяснил, как извлечь максимум из ASP.NET Core MVC. Он начинает с элементарных вещей и рассказывает обо всем вплоть до расширенных средств, чтобы дать вам исчерпывающие знания, в которых вы нуждаетесь. Книга включает полностью работающий учебный пример функционирующего приложения, который вы сможете использовать в качестве шаблона для собственных проектов.

Что нового в этом издании

- Полное обновление с учетом возможностей Visual Studio 2017, C# 7 и .NET Core 2
- Раскрытие новых средств, таких как фильтры представлений
- Более широкий, нежели ранее, охват платформы и инструментария, с большим объемом сведений по Visual Studio Code и работе с .NET Core на платформах, отличных от Windows
- Развертывание приложений на основе контейнера Docker

Благодаря этой книге вы освоите следующие темы

- Обретете глубокое понимание архитектуры ASP.NET Core MVC
- Исследуете инфраструктуру ASP.NET MVC Framework как единое целое
- Увидите в действии инфраструктуру MVC и разработку через тестирование
- Узнаете о новых возможностях ASP.NET Core MVC 2 и научитесь их эффективно применять в своей работе
- Выясните, как создавать веб-службы REST и одностраничные приложения (SPA)
- Сможете воспользоваться имеющимися у вас знаниями предшествующих выпусков MVC для быстрого и эффективного освоения новой модели программирования

НА ВЕБ-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с веб-сайта издательства по адресу: http://www.williamspublishing.com/Books/978-5-6041394-3-1.html.

Категория: программирование
Предмет рассмотрения: ASP.NET Core MVC 2
Уровень: для пользователей средней и высокой квалификации

ISBN 978-5-6041394-3-1



ДИАЛЕКТИКА
www.williamspublishing.com
Apress®