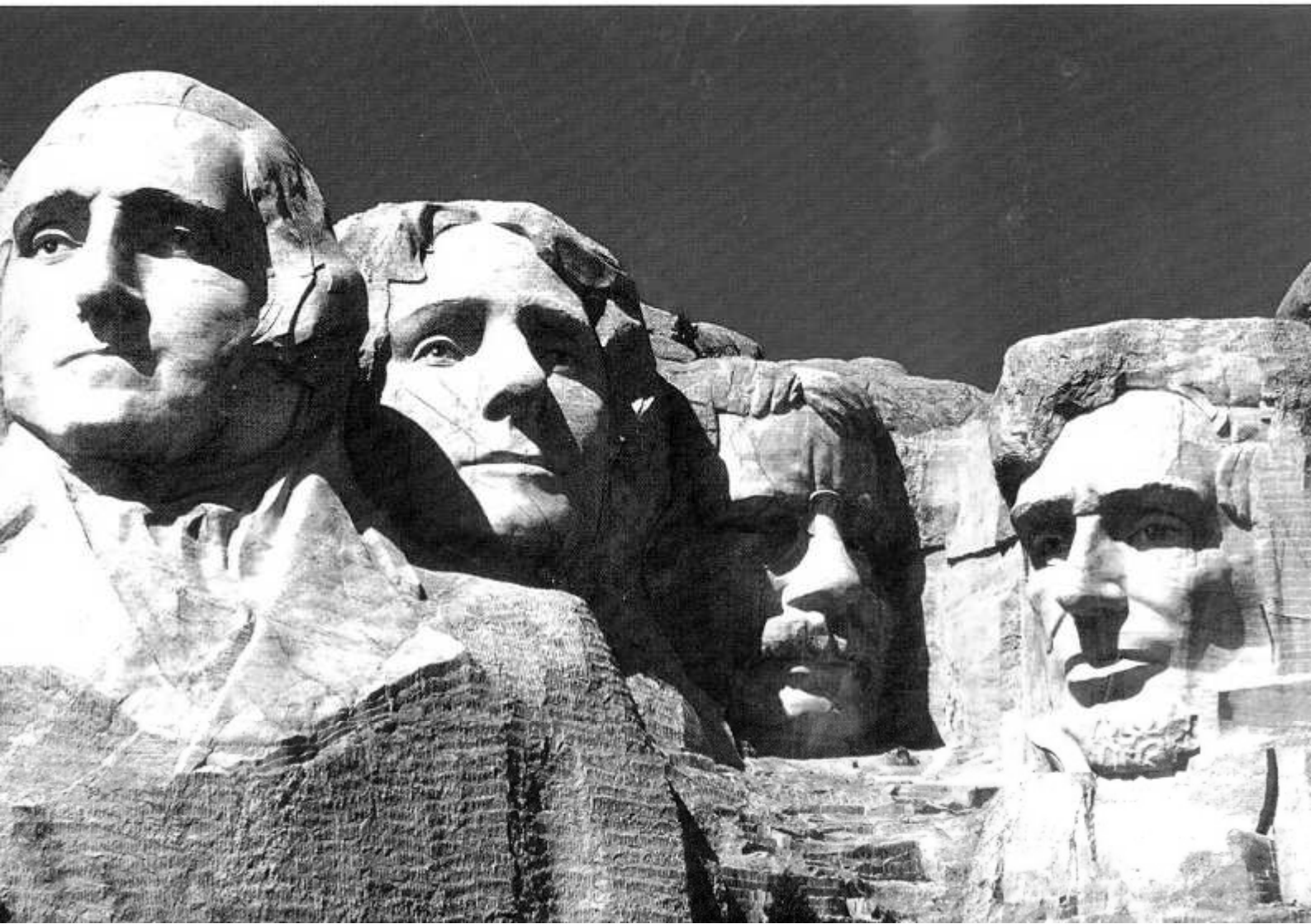
Библиотека профессионала

PRENTICE HALL

JavaServer™ Faces

3-Е ИЗДАНИЕ

ДЭВИД ГЕРИ • КЕЙ ХОРСТМАНН



Core JavaServer™ Faces

Third Edition

David Geary
Cay Horstmann

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Библиотека профессионала

JavaServer™ Faces

3-е издание

Дэвид Гери
Кей Хорстманн

ВНИМАНИЕ!!!
Эта книга досталась тебе БЕЗПЛАТНО.
Но кто-то совершил определённый труд по её написанию, переводу, изданию и сканированию. Если книга оказалась в какой-то мере полезной для тебя, не будь жмотом, перечисли на счёт издательства сумму, в которую ты оцениваешь труд тех, кто сделал эту книгу для тебя. Пусть издатели знают, что в рунете есть платёжеспособный спрос на электронные книги. Это может в будущем избавить от необходимости выполнять монотонную работу по сканированию книг, ускорить выход электронных книг!!!

Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26-018.2.75
Г37
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *К.А. Птицына*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Гери, Дэвид М., **Хорстманн**, Кей С.

Г37 JavaServer Faces. Библиотека профессионала, 3-е изд. : Пер. с англ. – М. : ООО "И.Д. Вильямс", 2011. – 544 с. : ил. – Парал. тит. англ.

ISBN 978-5-8459-1706-5 (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 Prentice Hall, Inc.

Authorized translation from the English language edition published by Prentice Hall, Ptr., Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2011

Научно-популярное издание

Дэвид М. Гери, Кей С. Хорстманн

JavaServer Faces. Библиотека профессионала

3-е издание

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 20.01.2011. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 43,86. Уч.-изд. л. 30,5.
Тираж 1500 экз. Заказ № 25075.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1706-5 (рус.)
ISBN 978-0-13-701289-3 (англ.)

© Издательский дом "Вильямс", 2011
© Oracle and/or its affiliates, 2010

Оглавление

Глава 1. Первое знакомство 19

Глава 2. Управляемые бины 45

Глава 3. Навигация 79

Глава 4. Стандартные теги JSF 103

Глава 5. Технология Facelets 165

Глава 6. Таблицы данных 187

Глава 7. Преобразование и проверка правильности 221

Глава 8. Обработка событий 269

Глава 9. Составные компоненты JSF 2.0 305

Глава 10. Технология Ajax 337

Глава 11. Пользовательские компоненты, преобразователи и средства проверки 365

Глава 12. Внешние службы 421

Глава 13. Дополнительные рекомендации 471

Предметный указатель 523

Содержание

Введение 13
Об этой книге 14
Необходимое программное обеспечение 15
Поддержка в Интернете 15
Благодарности 15

Глава 1. Первое знакомство 19

Преимущества JavaServer Faces 19
Простой пример 20
Составные части 22
Структура каталогов 23
Создание приложения JSF 24
Развертывание приложения JSF 25
Среды разработки для JSF 27
Анализ примера приложения 28
Бины 30
Страницы JSF 31
Конфигурация сервлета 33
Первое знакомство с Ajax JSF 2.0 35
Службы платформы JSF 37
Внутренние процессы 39
Визуализация страниц 39
Декодирование запросов 40
Жизненный цикл 42
Резюме 43

Глава 2. Управляемые бины 45

Определение бина 45
Свойства бинов 47
Выражения значений 49
Вспомогательные бины 49
Бины CDI CDI 50
Связки сообщений 51
Сообщения с переменными частями 52
Установка локали в приложении 53
Пример приложения 54
Области действия бинов 60
Область действия сеанса 60
Область действия запроса 62
Область действия приложения 62

Область действия диалога CDI 63
Область действия просмотра JSF 2.0 64
Пользовательские области действия JSF 2.0 64
Настройка конфигурации бинов 64
Встраивание бинов CDI CDI 64
Встраивание управляемых бинов JSF 2.0 65
Аннотации для жизненного цикла бина 66
Настройка конфигурации управляемых бинов с помощью XML 66
Синтаксис языка выражений 70
Режимы лево- и правосторонних выражений 70
Использование квадратных скобок 71
Выражения карт и списков 72
Вызов методов и функций JSF 2.0 72
Разрешение первого члена 73
Сложные выражения 75
Выражения методов 76
Параметры выражения метода JSF 2.0 76
Резюме 77

Глава 3. Навигация 79

Статическая навигация 79
Динамическая навигация 80
Отображение результатов в идентификаторы представлений 81
Приложение JavaQuiz 82
Перенаправление 90
Перенаправление и флеш-память JSF 2.0 91
Навигация с поддержкой метода REST и применение URL, обеспечивающих 92
Параметры просмотра 93
Ссылки запросов GET 93
Определение параметров запроса 94
Добавление ссылок, обеспечивающих формирование закладок, в приложение для викторины 95
Расширенные правила навигации 98
Подстановочные знаки 99
Использование элемента from-action 100
Возможности условной навигации JSF 2.0 JSF 2.0 100
Динамические идентификаторы целевого представления JSF 2.0 100
Резюме 101

Глава 4. Стандартные теги JSF 103

Общие сведения об основных тегах, применяемых в технологии JSF 103
Атрибуты, параметры и аспекты 105
Общие сведения о тегах HTML, применяемых в технологии JSF 106
Общие атрибуты 108
Панели 114
Теги head, body и form 116
Элементы формы и код JavaScript 118
Текстовые поля и области 121

Скрытые поля 123
Использование текстовых полей и областей 123
Вывод на экран текста и изображений 126
Кнопки и ссылки 129
Использование кнопок 130
Использование командных ссылок 134
Теги выбора 137
Флажки и переключатели 139
Меню и окна списка 142
Элементы 144
Сообщения 158
Резюме 163

Глава 5. Технология Facelets 165

Теги Facelets 165
Применение шаблонов на основе платформы Facelets 166
Формирование страниц исходя из общих шаблонов 167
Организация конкретных представлений 171
Декораторы 176
Параметры 178
Специализированные теги 178
Компоненты и фрагменты 180
Нерассмотренные вопросы 181
Тег <ui:debug> 181
Тег <ui:remove> 183
Обработка пробельных символов 184
Резюме 184

Глава 6. Таблицы данных 187

Тег таблицы данных — h:dataTable 187
Простая таблица 188
Атрибуты тега h:dataTable 190
Атрибуты тега h:column 192
Заголовки, нижние колонтитулы и надписи 192
Стили 194
Применение стилей к отдельным столбцам 195
Применение стилей к отдельным строкам 196
Тег ui:repeat JSF 2.0 196
Компоненты JSF в таблицах 197
Редактирование таблиц 200
Редактирование ячеек таблицы 200
Удаление строк JSF 2.0 203
Таблицы базы данных 205
Модели таблиц 209
Подготовка к отображению номеров строк 209
Поиск выбранной строки 210
Сортировка и фильтрация 210
Способы прокрутки 216

Прокрутка с помощью полосы прокрутки 216
Прокрутка с помощью мини-приложений постраничного просмотра 217
Резюме 218

Глава 7. Преобразование и проверка правильности 221

Общие сведения о процессе преобразования и проверке правильности 221
Использование стандартных преобразователей 222
Преобразование чисел и дат 223
Ошибки преобразования 226
Полный пример преобразователя 231
Использование стандартных средств проверки 233
Проверка длин строк и числовых диапазонов 233
Проверка обязательных значений 235
Отображение ошибок проверки правильности 235
Исключение процедуры проверки правильности 237
Полный пример проверки правильности 237
Проверка правильности бина JSF 2.0 239
Программирование с применением пользовательских преобразователей и средств проверки 244
Реализация пользовательских классов преобразователей 244
Определение преобразователей JSF 2.0 247
Сообщения об ошибках преобразования 248
Получение сообщений об ошибках из связок ресурсов 249
Пример приложения с пользовательским преобразователем 253
Предоставление атрибутов для преобразователей 255
Реализация классов пользовательского средства проверки 256
Регистрация пользовательских средств проверки 256
Проверка с помощью методов бина 259
Проверка связей между несколькими компонентами 260
Реализация пользовательских тегов преобразователей и средств проверки 261
Резюме 267

Глава 8. Обработка событий 269

События и жизненный цикл JSF 270
События изменения значения 271
События действия 275
Теги прослушивателей событий 280
Теги `f:actionListener` и `f:valueChangeListener` 280
Немедленно активизируемые компоненты 281
Использование немедленно активизируемых компонентов ввода 282
Использование немедленно активизируемых командных компонентов 284
Передача данных из пользовательского интерфейса на сервер 285
Параметры выражения метода JSF 2.0 286
Тег `f:param` 286
Тег `f:attribute` 287
Тег `f:setPropertyActionListener` 287
События фазы 288

Системные события JSF 1.2 289
Многокомпонентная проверка правильности 291
Принятие решений перед подготовкой представления к отображению 292
Объединение рассматриваемых средств в одном приложении 296
Резюме 302

Глава 9. Составные компоненты JSF 2.0 305

Библиотека составных тегов 306
Использование составных компонентов 307
Реализация составных компонентов 309
Настройка конфигурации составных компонентов 310
Типы атрибутов 311
Атрибуты `required` и значения атрибутов по умолчанию 311
Манипулирование серверными данными 313
Локализация составных компонентов 315
Обеспечение доступа к отдельным компонентам составных компонентов 316
Предоставление доступа к источникам действий 318
Аспекты 320
Дочерние теги 321
Применение кода JavaScript 322
Вспомогательные компоненты 326
Упаковка составных компонентов в файлах JAR 333
Резюме 334

Глава 10. Технология Ajax 337

Совместное применение Ajax и JSF 337
Жизненный цикл JSF и технология Ajax 338
Рекомендации по применению Ajax JSF 340
Тег `f:ajax` 341
Группы Ajax 343
Проверка правильности полей в Ajax 345
Мониторинг запросов Ajax 346
Пространства имен JavaScript 348
Обработка ошибок в Ajax 349
Ответы Ajax 350
Библиотека JavaScript в версии JSF 2.0 352
Передача дополнительных параметров запроса Ajax 354
Организация очередей событий 355
Объединение событий 356
Перехват значений функции `jsf.ajax.request()` 357
Использование Ajax в составных компонентах 357
Резюме 363

Глава 11. Пользовательские компоненты, преобразователи и средства проверки 365

Реализация классов компонентов 366
Кодирование: формирование разметки 369

Декодирование: обработка значений запроса 372
Файл описания библиотеки тегов JSF 1.2 376
Использование внешнего средства подготовки к отображению 380
Обработка атрибутов тегов JSF 2.0 383
Поддержка прослушивателей изменений значений 384
Поддержка выражений метода 385
Постановка событий в очередь 386
Пример приложения 387
Разработка кода JavaScript 393
Использование дочерних компонентов и аспектов 395
Обработка дочерних тегов SelectItem 398
Обработка аспектов 399
Использование скрытых полей 400
Сохранение и восстановление состояния 405
Частичное сохранение состояния JSF 2.0 406
Создание компонентов Ajax JSF 2.0 409
Реализация самодостаточного кода Ajax в пользовательских компонентах 410
Поддержка тега f:ajax в пользовательских компонентах 414
Резюме 418

Глава 12. Внешние службы 421

Доступ к базе данных с помощью JDBC 421
Передача инструкций SQL на выполнение 421
Управление соединениями 422
Устранение утечек ресурсов соединения 423
Использование подготовленных инструкций 425
Транзакции 426
Использование базы данных Derby 427
Настройка источника данных 428
Доступ к ресурсу, управляемому контейнером 428
Настройка ресурса базы данных в технологии GlassFish 428
Настройка ресурса базы данных на платформе Tomcat 429
Полный пример применения базы данных 431
Использование спецификации JPA 437
Краткое описание архитектуры JPA 437
Использование архитектуры JPA в веб-приложении 439
Использование управляемых бинов и бинов сеанса, не поддерживающих состояние 442
Бины сеанса, поддерживающие состояние CDI 445
Аутентификация и авторизация, управляемые контейнером 448
Отправка почты 458
Использование веб-служб 462
Резюме 468

Глава 13. Дополнительные рекомендации 471

Поиск дополнительных компонентов 471
Поддержка выгрузки файлов 472

Отображение гиперкарты 479
Формирование двоичных данных на странице JSF 481
Одновременное отображение крупного набора данных по одной странице 488
Формирование всплывающего окна 492
Выборочное отображение и сокрытие частей страницы 498
Настройка страниц с сообщениями об ошибках 499
Написание собственного клиентского тега проверки правильности 504
Настройка собственного приложения 510
Расширение языка выражений JSF 511
Добавление функций к языку выражений JSF JSF 2.0 514
Мониторинг трафика между браузером и сервером 515
Отладка застывшей страницы 516
Использование инструментальных средств тестирования при разработке 517
Использование технологии Scala с JSF 519
Использование технологии Groovy с JSF 520
Резюме 522

Предметный указатель **523**

Введение

Впервые услышав о платформе JavaServer Faces на конференции по JavaOne в 2002 году, мы сразу оценили открывшиеся перспективы. Мы оба имели большой опыт в области клиентского программирования на языке Java (причем Дэвид участвовал в написании книги *Graphic Java*™, а Кей – книги *Core Java*™[1]; обе они опубликованы издательством Sun Microsystems Press) и пришли к выводу, что веб-программирование с помощью сервлетов и технологии JavaServer Pages (JSP) является довольно сложным для понимания и трудоемким. Платформа JSF оказалась способной предоставить для разработки веб-приложений удобный инструмент, позволяющий программистам заниматься только созданием текстовых полей и меню, а не заботиться о смене страниц и применении параметров запроса. Каждый из нас предложил представителю издательства Sun Microsystems Press написать книгу на эту тему, а тот, в свою очередь, порекомендовал нам подготовить книгу по JSF совместно.

Спецификация JSF 1.0 и справочная документация были впервые выпущены группой JSF Expert Group (членом которой был и Дэвид) в 2004 г. Почти сразу же после этого вышел исправленный выпуск 1.1, а в 2006 году появилась уже новая, откорректированная и дополненная новыми удобными возможностями версия 1.2.

Первоначальная спецификация JSF была далеко не идеальной. Она страдала чрезмерной общностью и содержала примеры использования, которые, как оказалось, не представляли интереса на практике. Проекту API не было уделено достаточное внимание, что вынуждало программистов писать сложный и трудоемкий код. Поддержка для запросов GET была громоздкой. Обработка ошибок оказалась явно неудовлетворительной, поэтому разработчики называли трассировку стека "дорогой в ад".

Тем не менее платформа JSF имела одну привлекательную особенность: предоставляла широкие возможности дополнения и расширения, поэтому была весьма интересна для разработчиков платформ. Благодаря этому разработчики платформ сумели создать передовое программное обеспечение с открытым исходным кодом, подключаемое к JSF, такое как Facelets, Ajax4jsf, Seam, JSF Templates, Pretty Faces, RichFaces, ICEFaces и т.д.

В 2009 году была выпущена версия JSF 2.0, основанная на опыте разработки подобных платформ с открытым исходным кодом. Почти все авторы первоначальных версий вышеупомянутых платформ приняли участие в работе JSF 2 Expert Group, поэтому версия JSF 2.0, в отличие от JSF 1.0, опиралась на прошедшие суровое испытание на практике проекты с открытым исходным кодом, для полного становления которых было достаточно времени.

Платформа JSF 2.0 гораздо проще в использовании и лучше встраивается в стек технологий Java EE, чем JSF 1.0. В ходе доработки почти каждый фрагмент JSF 1.0 не просто подвергся преобразованию в JSF 2.0, а приобрел значительные усовершенствования. Кроме того, спецификация современной платформы теперь поддерживает новые веб-технологии, такие как Ajax и REST.

---

[1] Кей Хорстманн, Гари Корнелл. *Java 2. Библиотека профессионала*, тома 1 и 2, 8-е изд. (ИД "Вильямс", 2008).

Сегодня JSF представляет собой превосходную платформу для разработки серверных веб-приложений на Java и почти полностью оправдала надежды, которые на нее возлагались. Она реально позволяет разрабатывать пользовательские интерфейсы для веб-приложений путем размещения компонентов в форме и связывания их с Java-объектами, исключая необходимость смешивать код и разметку. Одним из главных преимуществ JSF является применяемая в ней расширяемая модель компонентов, для которой сторонние разработчики уже создали множество новых компонентов. Эта платформа имеет гибкий проект, что позволяет ей совершенствоваться и приспосабливаться к новым технологиям.

Кроме того, JSF – это прежде всего спецификация, а не готовый продукт, поэтому разработчики не находятся во власти единственного поставщика. Реализации, компоненты и инструментальные средства JSF можно получить из многочисленных источников. Мы по-прежнему восхищаемся версией JSF 2.0 и надеемся, что читателя охватит то же чувство, когда он узнает, что эта технология позволяет выполнять работу разработчика веб-приложений намного более эффективно.

Об этой книге

Книга рассчитана на разработчиков веб-приложений, основные усилия которых сосредоточены на реализации пользовательских интерфейсов и бизнес-логики. Она резко отличается от выпущенной официально спецификации JSF, представляющей собой краткий документ со сложными для восприятия формулировками, который в основном рассчитан на разработчиков платформ, а также технических писателей, которые вынуждены глубоко разбираться во всем этом. Платформа JSF создана на основе сервлетов, но с точки зрения разработчика для JSF технология сервлетов представляет собой просто среду низкого уровня. Безусловно, знание других веб-технологий, таких как сервлеты, JSP или Struts, совсем не помешает, но это не является обязательным для прочтения данной книги.

Первая часть книги, до главы 7 включительно, посвящена тегам JSF, которые напоминают теги форм HTML. Они являются основными строительными блоками пользовательских интерфейсов JSF. Использовать теги JSF для создания веб-приложений может любой программист, имеющий основные навыки работы с языком HTML (для проектирования веб-страниц) и стандартного программирования на Java (для разработки логики приложений).

В первой части этой книги рассматриваются следующие темы.

- Подготовка среды программирования (глава 1).
- Соединение JSF-тегов с логикой приложения (глава 2).
- Навигация по страницам (глава 3).
- Использование стандартных тегов JSF (глава 4).
- Применение тегов фейслетов для создания шаблонов (глава 5).
- Таблицы данных (глава 6).
- Преобразование и проверка достоверности входных данных (глава 7).

Начиная с главы 8 мы приступим к глубокому изучению процесса программирования на JSF. Читатель узнает, как решать более сложные задачи и расширять возможности платформы JSF. Основные темы, рассматриваемые во второй части книги, перечислены ниже.

- Обработка событий (глава 8).
- Создание составных компонентов – повторно используемых компонентов со сложным поведением, в состав которых входят более простые компоненты (глава 9).
- Ajax (глава 10).
- Реализация пользовательских компонентов (глава 11).
- Соединение с базами данных и другими внешними службами (глава 12).

Завершает эту книгу глава, в которой мы постарались ответить на вопросы типа "Как сделать то-то и то-то" (глава 13). Мы рекомендуем ознакомиться с этой главой сразу после изучения основ JSF. В ней содержатся полезные рекомендации по отладке и ведению журналов, а также приведены подробные сведения и рабочий код для создания отсутствующих в JSF функций, таких как средства загрузки файлов, всплывающие меню и компоненты, позволяющие выполнять разбивку на страницы в длинных таблицах.

В этом издании все главы подверглись существенной переработке в целях концентрации внимания на новых и усовершенствованных функциях JSF 2.0. В настоящем издании главы 5, 9 и 10 являются новыми.

Необходимое программное обеспечение

Все программное обеспечение, необходимое для работы с этой книгой, доступно бесплатно. Вместе с реализацией JSF можно использовать сервер приложений, который поддерживает Java EE 6 (такой как GlassFish версии 3), или исполнитель сервлетов (наподобие Tomcat 6). Это программное обеспечение может эксплуатироваться в средах Linux, Mac OS X, Solaris и Windows. Широкую поддержку для разработки на JSF с помощью GlassFish или Tomcat предоставляют и Eclipse, и NetBeans.

Поддержка в Интернете

Веб-сайт, сопровождающий эту книгу, находится по адресу `http://corejsf.com`. На этом сайте можно найти следующее.

- Исходный код всех примеров данной книги.
- Полезный справочный материал, который нам показалось более удобным предоставить в электронном, а не в печатном виде.
- Список обнаруженных ошибок в книге и коде.
- Форма для отправки исправлений и предложений.

Благодарности

Прежде всего мы бы хотели поблагодарить Грега Доэнча (Greg Doench), нашего редактора из издательства Prentice Hall, который помогал в нашей работе на протяжении всего проекта, никогда не теряя самообладания, несмотря на многочисленные задержки и трудности. Выражаем признательность Ванессе Мур (Vanessa Moore) за то, что превратила нашу неупорядоченную рукопись в привлекательную книгу, а также за ее терпение и удивительное внимание к деталям.

Мы очень благодарны рецензентам этого и двух предыдущих изданий нашей книги, которые потрудились на славу, отыскивая ошибки и предлагая уточнения для различных вариантов рукописи. Их имена перечислены ниже.

- Гэйл Андерсон (Gail Anderson) из компании Anderson Software Group, Inc.
- Лэрри Браун (Larry Brown) из LMBrown.com, Inc.
- Дамадар Четти (Damodar Chetty) из Software Engineering Solutions, Inc.
- Фрэнк Коэн (Frank Cohen) из PushToTest.
- Брайан Гоэтц (Brian Goetz) из Sun Microsystems, Inc.
- Роб Гордон (Rob Gordon) из Crooked Furrow Farm.
- Марти Холл (Marty Hall), автор книги *Core Servlets and JavaServer Pages™, Second Edition*, (Prentice Hall, 2008).
- Стивен Хэйнес, главный администратор и основатель компании GeekCap, Inc.
- Чарли Хант (Charlie Hunt) из Sun Microsystems, Inc.
- Джефф Лангр (Jeff Langr) из Langr Software Solutions.
- Джейсон Ли (Jason Lee), старший разработчик на Java, компания Sun Microsystems, Inc.
- Билл Льюис (Bill Lewis) из университета Тафтса (Tufts University).
- Кито Манн (Kito Mann), автор книги *JavaServer Faces in Action* (Manning, 2005) и основатель компании JSFCentral.com.
- Джефф Маркхэм (Jeff Markham) из компании Markham Software Company.
- Ангус Макинтайр (Angus McIntyre) из корпорации IBM.
- Джон Мачоу (John Muchow), автор книги *Core J2ME™* (Prentice Hall, 2001).
- Дэн Шеллман (Dan Shellman) из BearingPoint.
- Сергей Смирнов, главный разработчик в Exadel JSF Studio.
- Роман Смолговски (Roman Smolgovsky) из Flytecomm.
- Стефан Стелтинг (Stephen Stelting) из Sun Microsystems, Inc.
- Кристофер Тэйлор (Christopher Taylor) из Nanshu Densetsu.
- Ким Топли (Kim Topley) из Keyboard Edge Limited
- Майкл Юан, один из авторов книги *JBoss® Seam: Simplicity and Power Beyond Java™ EE* (Prentice Hall, 2007).

И, наконец, мы, конечно же, хотели бы выразить благодарность нашим семьям и друзьям, которые поддерживали нас на протяжении всей работы над этим проектом и вместе с нами облегченно вздохнули, когда он наконец-то был завершен.

Первое знакомство

В этой главе...

- Преимущества JavaServer Faces
- Простой пример
- Среды разработки для JSF
- Анализ примера приложения
- Первое знакомство с Ajax JSF 2.0
- Службы платформы JSF
- Внутренние процессы

Глава 1

Преимущества JavaServer Faces

В настоящее время предоставляется широкий выбор платформ для разработки пользовательского интерфейса веб-приложений. Одной из них является JavaServer Faces (JSF) – *компонентно-ориентированная* платформа. Это означает, что для отображения таблицы со строками и столбцами не нужно формировать теги HTML для строк и ячеек в цикле, поскольку достаточно лишь разместить компонент таблицы на странице. (Те, кто знаком с разработкой клиентских приложений на Java, могут считать JSF аналогом Swing для серверных приложений.) Использование компонентов позволяет разрабатывать пользовательский интерфейс на более высоком уровне по сравнению с исходным кодом HTML. Предусмотрена возможность повторно обращаться к собственным компонентам и применять наборы компонентов сторонних разработчиков. Кроме того, существует возможность использовать визуальную среду разработки, которая позволяет перетаскивать компоненты на форму.

Платформа JSF состоит из следующих частей:

- набор заранее подготовленных компонентов для пользовательского интерфейса;
- модель программирования, управляемая событиями;
- модель компонентов, позволяющая сторонним разработчикам предоставлять дополнительные компоненты.

Некоторые компоненты JSF, такие как поля ввода и кнопки, являются простыми, а другие, наподобие таблиц данных и деревьев, намного сложнее.

Платформа JSF предоставляет весь необходимый код для обработки событий и организации компонентов. Разработчики приложений могут смело игнорировать лишние подробности и сосредоточивать свои усилия на реализации логики самих приложений.

JSF – не единственная компонентно-ориентированная платформа для веб-приложений, но рассматривается в стандарте Java EE как относящаяся к уровню представления. Поддержка JSF предусмотрена в каждом сервере приложений Java EE и может быть легко добавлена к автономному веб-контейнеру, такому как Tomcat.

В отличие от большинства других веб-платформ, JSF определена стандартом и имеет многочисленные реализации. Благодаря этому разработчик может выбирать среди продуктов разных поставщиков. Еще одно преимущество состоит в том, что комитет по стандартизации провел большую работу по созданию проекта этой платформы, а сама JSF непрерывно совершенствуется и обновляется.

Настоящая книга в основном посвящена описанию версии JSF 2.0, которая представляет собой большой шаг вперед по сравнению с предыдущими версиями. Версия JSF 2.0 намного проще в использовании, чем JSF 1.x, и предоставляет новые и мощные возможности, такие как простое интегрирование с Ajax и разработка составных компонентов.

Простой пример

Рассмотрим простой пример приложения JSF. Он начинается с экрана входа в систему, показанного на рис. 1.1.

Рис. 1.1. Экран входа в систему

Файл, описывающий этот экран входа в систему, по сути представляет собой HTML-файл с несколькими дополнительными тегами (листинг 1.1). Экран выглядит упрощенно, но может быть легко улучшен дизайнером графики, для чего ему не потребуются даже элементарные навыки программирования.

Листинг 1.1. Файл `login\web\index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:h="http://java.sun.com/jsf/html">
6.    <h:head>
7.       <title>Welcome</title>
8.    </h:head>
9.    <h:body>
10.      <h:form>
11.         <h3>Please enter your name and password.</h3>
12.         <table>
13.            <tr>
14.               <td>Name:</td>
15.               <td><h:inputText value="#{user.name}"/></td>
16.            </tr>
```

```
17.             <tr>
18.                 <td>Password:</td>
19.                 <td><h:inputSecret value="#{user.password}"/></td>
20.             </tr>
21.         </table>
22.         <p><h:commandButton value="Login" action="welcome"/></p>
23.     </h:form>
24. </h:body>
25. </html>
```

Содержимое этого файла более подробно будет рассматриваться чуть позже, в разделе "Страницы JSF" на стр. 31. На данный момент следует обратить внимание только на несколько особенностей.

- Некоторые теги являются стандартными тегами HTML: `p`, `table` и т.д.
- Некоторые теги имеют префиксы, такие как `h:head` и `h:inputText`. Это — теги JSF. Атрибут `xmlns` объявляет пространство имен JSF.
- Теги `h:inputText`, `h:inputSecret` и `h:commandButton` соответствуют текстовой области, полю ввода пароля и кнопке передачи формы (см. рис. 1.1).
- Поля ввода связаны со свойствами объектов. Например, атрибут `value="#{user.name}"` сообщает реализации JSF, что текстовая область должна быть связана со свойством `name` объекта `user`. Об этом связывании более подробно рассказано ниже, в разделе "Бины" на стр. 30.

После того как пользователь вводит имя и пароль, а также щелкает на кнопке Login, отображается файл `welcome.xhtml`, как указано в атрибуте `action` тега `h:commandButton` (рис. 1.2; листинг 1.2).

JSF 2.0

На заметку! До выхода версии JSF 2.0 приходилось добавлять в файл `WEB-INF/faces-config.xml` так называемое *правило навигации*, чтобы указать страницу, которая должна быть отображена после щелчка на кнопке. А в JSF 2.0 можно задать имя страницы непосредственно в атрибуте `action` кнопки. (Возможность использовать правила навигации все еще предусмотрена; эта тема рассматривается в главе 3.)

Рис. 1.2. Начальная страница

Вторая JSF-страница нашего приложения выглядит даже еще проще, чем первая (листинг 1.2). Мы используем выражение `#{user.name}`, чтобы вывести на экран значение свойства `name` объекта `user`, которое было задано на первой странице. Пароль пока игнорируется.

Листинг 1.2. Файл `login\web\welcome.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>Welcome</title>
   </h:head>
   <h:body>
      <h3>Welcome to JavaServer Faces, #{user.name}!</h3>
   </h:body>
</html>
```

Безусловно, главной задачей этого приложения является не создание неизгладимого впечатления на кого-то, а демонстрация различных фрагментов, необходимых для создания приложения JSF.

Составные части

Наше демонстрационное приложение состоит из следующих частей.

- Страницы, которые определяют экран входа в систему и экран приветствия. Они носят имена `index.xhtml` и `welcome.xhtml`.
- Бин, отвечающий за управление данными пользователя (в данном случае таковыми являются имя пользователя и пароль). *Бин* (bean) — это Java-класс, который предоставляет доступ к свойствам, обычно с использованием какого-то простого соглашения об именовании для методов задания и получения свойства. Код этого бина находится в файле `UserBean.java` (листинг 1.3). Обратите внимание на аннотацию `@Named` или `@ManagedBean`, определяющую имя, по которому на JSF-страницах формируются ссылки на объект этого класса. (В целях обеспечения совместимости предусмотрены два альтернативных варианта аннотаций, предназначенных для именования бинов. `@Named` — это лучший выбор, если применяется сервер приложений, совместимый с Java EE 6. Аннотация `@ManagedBean` предназначена для использования с относящимися к предыдущим версиям серверами приложений и автономными исполнителями сервлетов.)
- Кроме того, для нормальной работы сервера приложений необходимо предусмотреть файлы конфигурации `web.xml` и `beans.xml`.

На заметку! До выхода версии JSF 2.0 приходилось объявлять бины в файле `WEB-INF/faces-config.xml`. Это больше не требуется, и наше приложение не нуждается в файле `faces-config.xml`.

Более совершенные приложения JSF имеют такую же структуру, но могут содержать дополнительные Java-классы, такие как обработчики событий, средства проверки достоверности и пользовательские компоненты. Дополнительные параметры конфигурации могут быть помещены в файл `WEB-INF/faces-config.xml`, который будет описан в следующей главе. Для простого приложения этот файл не требуется.

Листинг 1.3. Файл `login\src\java\com\corejsf\UserBean.java`

```
package com.corejsf;

import java.io.Serializable;
```

```
4. import javax.inject.Named;
5.    // или import javax.faces.bean.ManagedBean;
6. import javax.enterprise.context.SessionScoped;
7.    // или import javax.faces.bean.SessionScoped;
8.
9. @Named("user") // или @ManagedBean(name="user")
10. @SessionScoped
11. public class UserBean implements Serializable {
12.    private String name;
13.     private String password;
14.
15.     public String getName() { return name; }
16.     public void setName(String newValue) { name = newValue; }
17.
18.     public String getPassword() { return password; }
19.     public void setPassword(String newValue) { password = newValue; }
20. }
```

Структура каталогов

Приложение JSF развертывается в виде WAR-файла, т.е. архивированного файла с расширением `.war` и структурой каталогов, которая имеет стандартизированную компоновку, как показано на рис. 1.3.

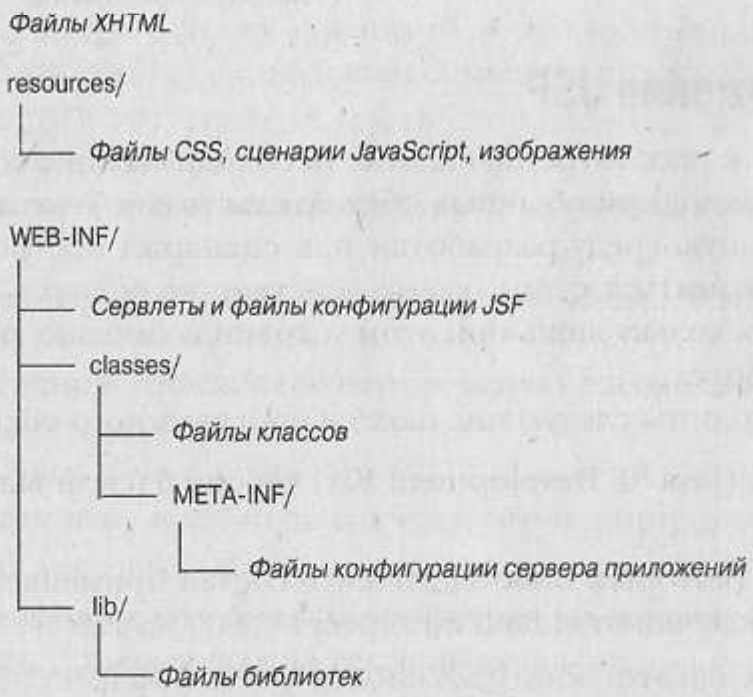


Рис. 1.3. Структура каталогов

Например, в WAR-файле для нашего примера приложения показана структура каталогов, которая приведена на рис. 1.4. Следует отметить, что класс `UserBean` находится в пакете `com.corejsf`.

На заметку! Если используется Tomcat или другой исполнитель сервлетов, то каталог `lib` содержит JAR-файлы реализации JSF. Это не требуется при использовании GlassFish и других серверов приложений Java EE, поскольку в них уже встроена платформа JSF.

Обычно принято упаковывать исходный код приложения с помощью другой структуры каталогов. В настоящей книге применяется соглашение Java Blueprints (`http://java.sun.com/blueprints/code/projectconventions.html`). При такой компоновке

упрощается импорт проектов в интегрированные среды разработки, такие как Eclipse или NetBeans. Исходный код содержится в каталоге `src/java`, а JSF-страницы и файлы конфигурации находятся в каталоге `web` (рис. 1.5).

Рис. 1.4. Структура каталогов типичного WAR-файла

Рис. 1.5. Структура каталогов исходного кода для рассматриваемого примера приложения

Создание приложения JSF

Теперь перейдем к рассмотрению шагов, необходимых для создания приложений JSF вручную. Безусловно, при обычных обстоятельствах в этих целях следует использовать интегрированную среду разработки или сценарий построения. Тем не менее рекомендуется ознакомиться с тем, какие действия выполняет эта среда незаметно для пользователя, поскольку лишь при этом условии возможно эффективное устранение нарушений в работе.

Для начала необходимы следующие пакеты программного обеспечения.

- Комплект JDK (Java SE Development Kit) версии 5.0 или выше (`http://java.sun.com/j2se`).
- Пакет JSF 2.0 (который либо включен в состав применяемого сервера приложений, либо получен отдельно по адресу `http://javaserverfaces.dev.java.net`).
- Примеры кода для этой книги можно получить по адресу `http://corejsf.com`.

В этой книге предполагается, что у читателя уже установлен пакет JDK и что он знаком с инструментальными средствами JDK. Для получения дополнительной информации по JDK см. книгу Кея Хорстманна (Cay Horstmann) и Гэри Корнелла (Gary Cornell) *Core Java™*, 8-е издание, Santa Clara, CA: Sun Microsystems Press/Prentice Hall, 2008.

Поскольку JSF 2.0 является частью спецификации Java EE, проверить JSF в работе легче всего путем использования сервера приложений, совместимого с Java EE 6, такого как GlassFish версии 3 (`http://glassfish.dev.java.net`).

Те, кто не желает устанавливать сервер приложений полностью, могут использовать исполнитель сервлетов, такой как Tomcat (`http://tomcat.apache.org`), вместе со справочной реализацией JSF (доступной по адресу `http://javaserverfaces.dev.java.net`).

В случае применения другого сервера приложений или исполнителя сервлетов необходимо откорректировать приведенные ниже команды.

Инструкции по созданию типового приложения JSF приведены ниже.

1. Запустите командную оболочку.
2. Перейдите к каталогу `corejsf-examples`, т.е. к каталогу, в котором находятся примеры кода для этой книги.
3. Если используется GlassFish или другой сервер приложений, совместимый с Java EE 6, перейдите к подкаталогу `javaee`. Если используется Tomcat, перейдите к подкаталогу `tomcat`.
4. Перейдите к каталогу исходного кода и создайте в нем каталог для хранения файлов классов:

   ```
   cd ch01/login/src/java
   mkdir ../../web/WEB-INF/classes
   ```

   В Windows используйте в качестве разделителей компонентов имен файлов знаки наклонной черты влево.
5. Если используется GlassFish, выполните следующее:

   ```
   javac -d ../../web/WEB-INF/classes -classpath .:glassfish/modules/\*
     com/corejsf/UserBean.java
   ```

   В Windows используйте точку с запятой в обозначении пути к библиотекам классов и не обозначайте специальным символом подстановочный знак *:

   ```
   javac -d ..\..\web\WEB-INF\classes -classpath .;glassfish\modules\*
     com\corejsf\UserBean.java
   ```

   Если применяется Tomcat, воспользуйтесь следующей командой для компиляции кода:

   ```
   javac -d ../../web/WEB-INF/classes -classpath .:jsf-ref-impl/lib/jsf-api.jar
     com/corejsf/UserBean.java
   ```
6. В случае применения Tomcat необходимо включить библиотеки JSF:

   ```
   mkdir ../../web/WEB-INF/lib
   cp jsf-ref-impl/lib/*.jar ../../web/WEB-INF/lib
   ```

   Пропустите этот шаг, если используется сервер приложений, совместимый с Java EE 6.
7. Выполните следующие команды (и проследите за тем, чтобы в конце команды `jar` стояла точка, указывающая на текущий каталог):

   ```
   cd ../..
   jar cvf login.war .
   ```

Развертывание приложения JSF

Установите применяемый сервер и запустите его. Например, чтобы запустить GlassFish в Unix/Linux, используется следующая команда (рис. 1.6):

```
glassfish/bin/asadmin start-domain
```

Для запуска Tomcat в Unix/Linux выполните следующее:

```
tomcat/bin/startup.sh
```

Чтобы проверить, работает ли сервер должным образом, введите в браузере предусмотренный по умолчанию URL. Для GlassFish и Tomcat таковым является `http://localhost:8080`. Должна отобразиться страница приветствия (рис. 1.7).

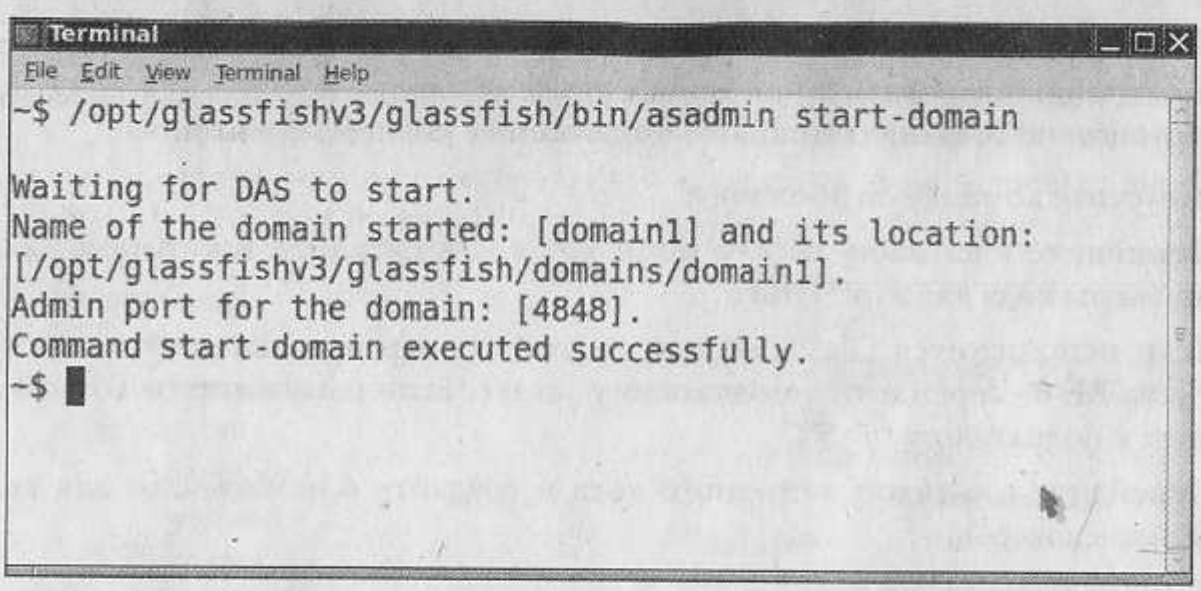


Рис. 1.6. Запуск сервера приложений GlassFish

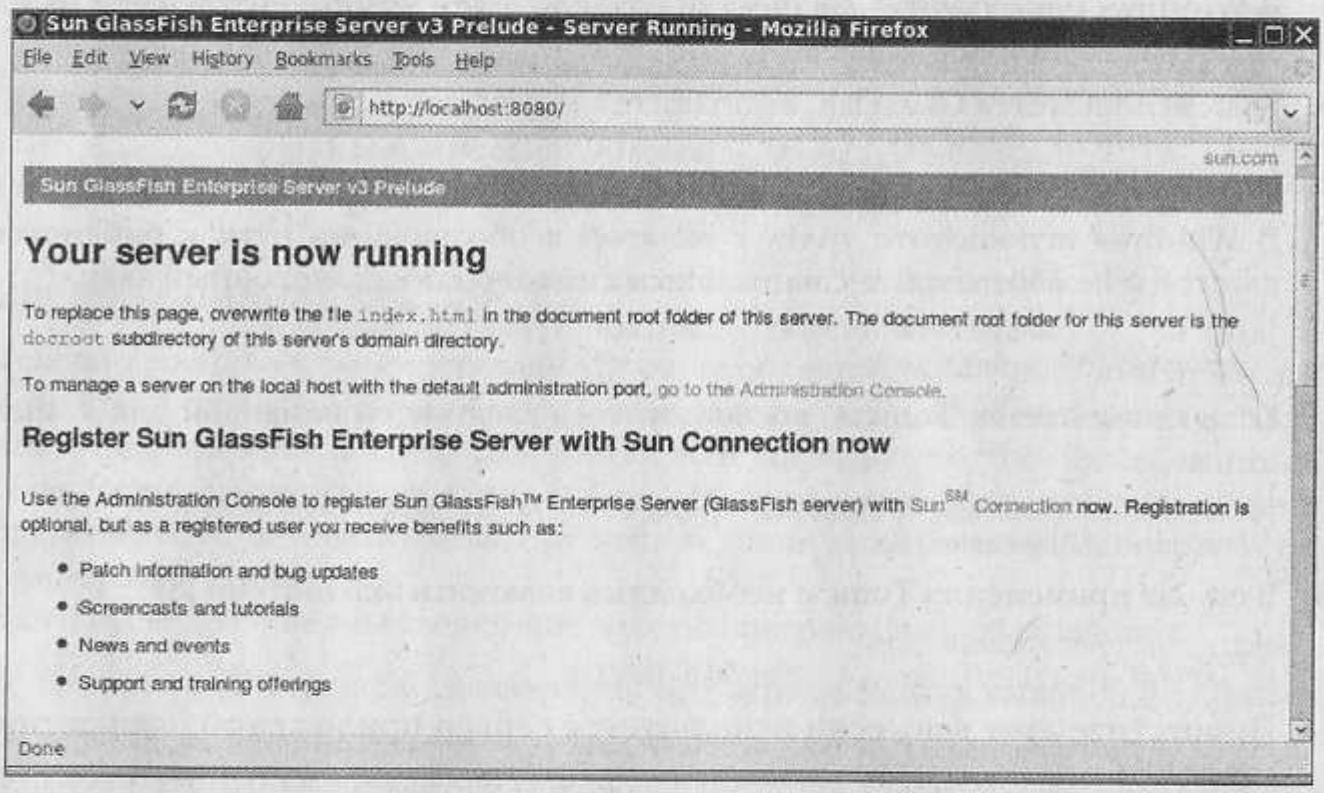


Рис. 1.7. Начальная страница GlassFish

Большинство серверов позволяет развертывать приложения путем копирования WAR-файлов в каталог развертывания. Для GlassFish этим каталогом служит `glassfish/domains/domain1/autodeploy`. При использовании Tomcat WAR-файлы развертываются в каталоге `tomcat/webapps`.

Скопируйте полученный файл `login.war` в каталог развертывания, убедитесь в том, что запуск сервера произошел успешно, и укажите в браузере адрес:

```
http://localhost:8080/login
```

После этого приложение должно запуститься.

На заметку! Если при развертывании этого примера программы будут возникать какие-либо нарушения в работе, необходимо просмотреть журнал в поиске необходимых подсказок. Сервер GlassFish регистрирует все сообщения в файле `glassfish/domains/domain1/logs/server.log`. Программа Tomcat ведет журналы в файле `tomcat/logs/catalina.out`.

Те, кто занимается развертыванием своих собственных приложений, должны знать, что файлы журналов содержат весь вывод, который был отправлен в файлы `System.out` и `System.err`. В конфигурации по умолчанию к этому относятся все журнальные сообщения с уровнем `INFO` и выше.

Иными словами, в целях отладки можно просто включить в свою программу вызовы функций `System.out.println` или `Logger.getGlobal().info`, и сформированный ими вывод появится в файлах журналов.

Среды разработки для JSF

Для простого JSF-приложения страницы и файлы конфигурации можно создавать с помощью текстового редактора. Однако, по мере того, как приложения становятся все сложнее, возникает необходимость в использовании более развитых инструментальных средств.

Заслуженной популярностью у программистов пользуются интегрированные среды разработки (Integrated Development Environment – IDE) наподобие Eclipse и NetBeans. Предусмотренная в них поддержка автозавершения, рефакторизации, отладки и так далее способствует значительному повышению продуктивности труда программиста, особенно в крупных проектах.

Ко времени написания этой книги и в Eclipse, и в Netbeans уже была предусмотрена качественная поддержка JSF. Что касается Netbeans, то его можно рассматривать как наиболее универсальный инструмент, который уже полностью интегрирован с GlassFish и Tomcat. А при использовании Eclipse установите GlassFish или Tomcat отдельно и работайте с дополнением к программе сервера. (Дополнение к программе GlassFish можно получить по адресу `https://glassfishplugins.dev.java.net`.) В Eclipse имеется лишь базисная поддержка JSF. Развитые средства работы с JSF имеются в коммерческих аналогах Eclipse (таких как MyEclipse, JBoss Developer Studio и Rational Application Developer), а также в Oracle JDeveloper. Для всех этих программ предусмотрены доступные для загрузки пробные версии.

Загружая один из примеров проектов из сопровождающего кода к данной книге в применяемую среду IDE, выберите вариант, предусматривающий импорт веб-проекта из существующих источников. Выберите исходный каталог, такой как `corejsf-examples/javaee/ch01/login`. Программа Netbeans автоматически настроит правильные каталоги `source` и `web`. Что же касается Eclipse, то необходимо будет изменить заданные по умолчанию каталоги `src` и `WebContent` на `src/java` и `web` (рис. 1.8).

Если используется интегрированная среда разработки, то задача выполнения и отладки приложений значительно упрощается. На рис. 1.9 показан отладчик Eclipse, остановленный на точке останова в классе `UserBean`. Оба отладчика, Eclipse и Netbeans, поддерживают активные исправления и обладают способностью вносить изменения в JSF-страницы и код Java, которые немедленно отражаются в работающей программе. (*Честное предупреждение.* Применение активных исправлений не всегда возможно, поэтому время от времени возникает необходимость повторно развертывать приложение и даже перезапускать сервер приложений.)

В некоторых интегрированных средах разработки имеется визуальное инструментальное средство построителя программы, которое позволяет разработчику перетаскивать компоненты из палитр на JSF-страницу. На рис. 1.10 показано применение визуального построителя программы в среде JDeveloper.

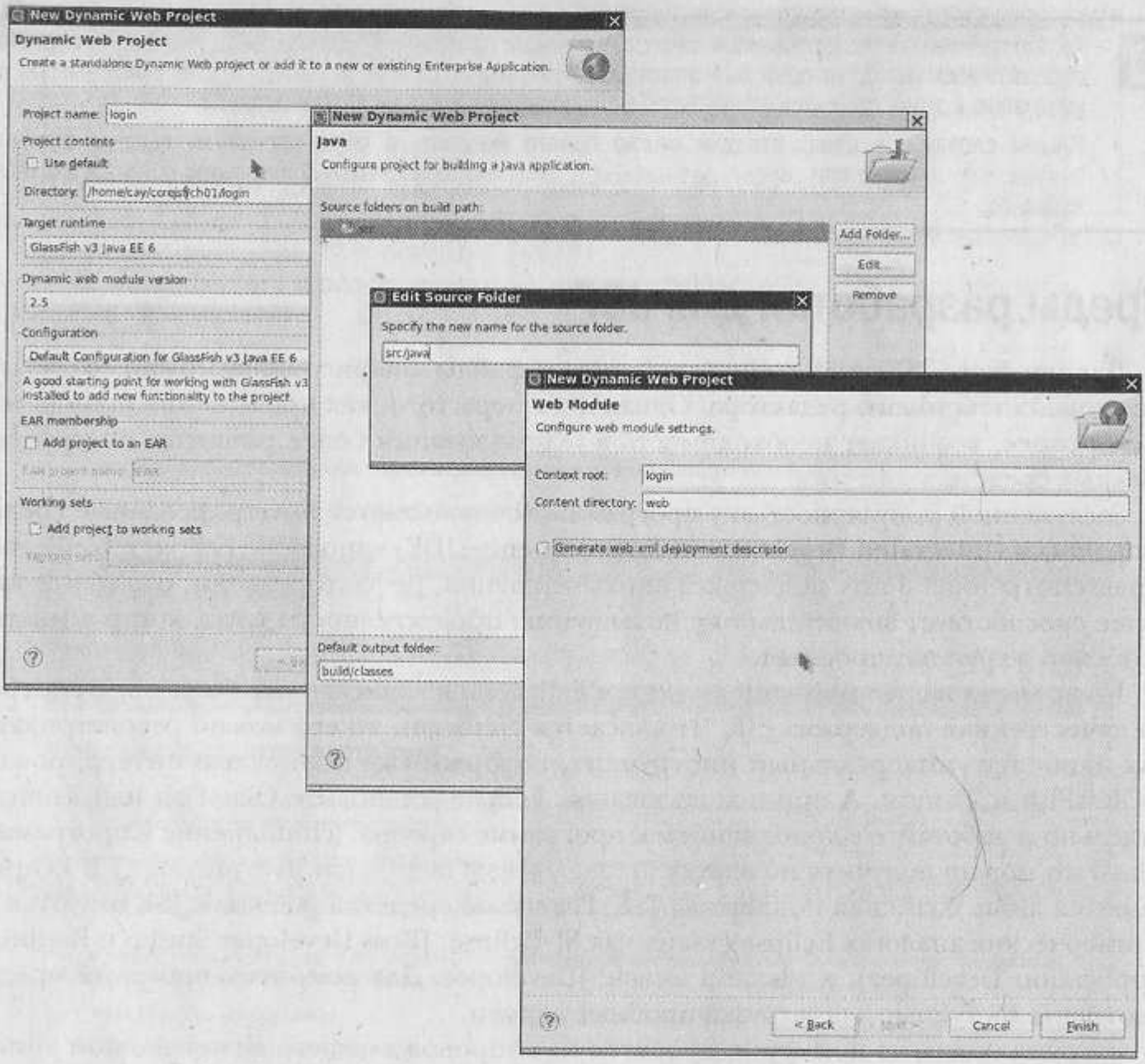


Рис. 1.8. Импорт проекта в программу Eclipse

Визуальные построители программы для JSF обычно оптимизируются для конкретного набора компонентов. Например, в среде JDeveloper используются компоненты ADF Faces. К сожалению, невозможно просто добавить свой предпочтительный набор компонентов к тому или иному визуальному инструментальному средству построителя программы. Стандарт JSF в настоящее время не регламентирует поведение компонентов во время разработки, и не предусмотрен также стандартный способ упаковки библиотек компонентов для использования в нескольких разных интегрированных средах разработки.

Анализ примера приложения

К этому времени мы представили простое приложение JSF и описали, как происходит его создание и выполнение. Теперь можно перейти к более подробному рассмотрению структуры приложения.

Веб-приложения состоят из двух частей: уровня представления и бизнес-логики. *Уровень представления* (presentation layer) отвечает за внешний вид приложения. В контексте приложения на основе браузера внешний вид определяется HTML-тегами,

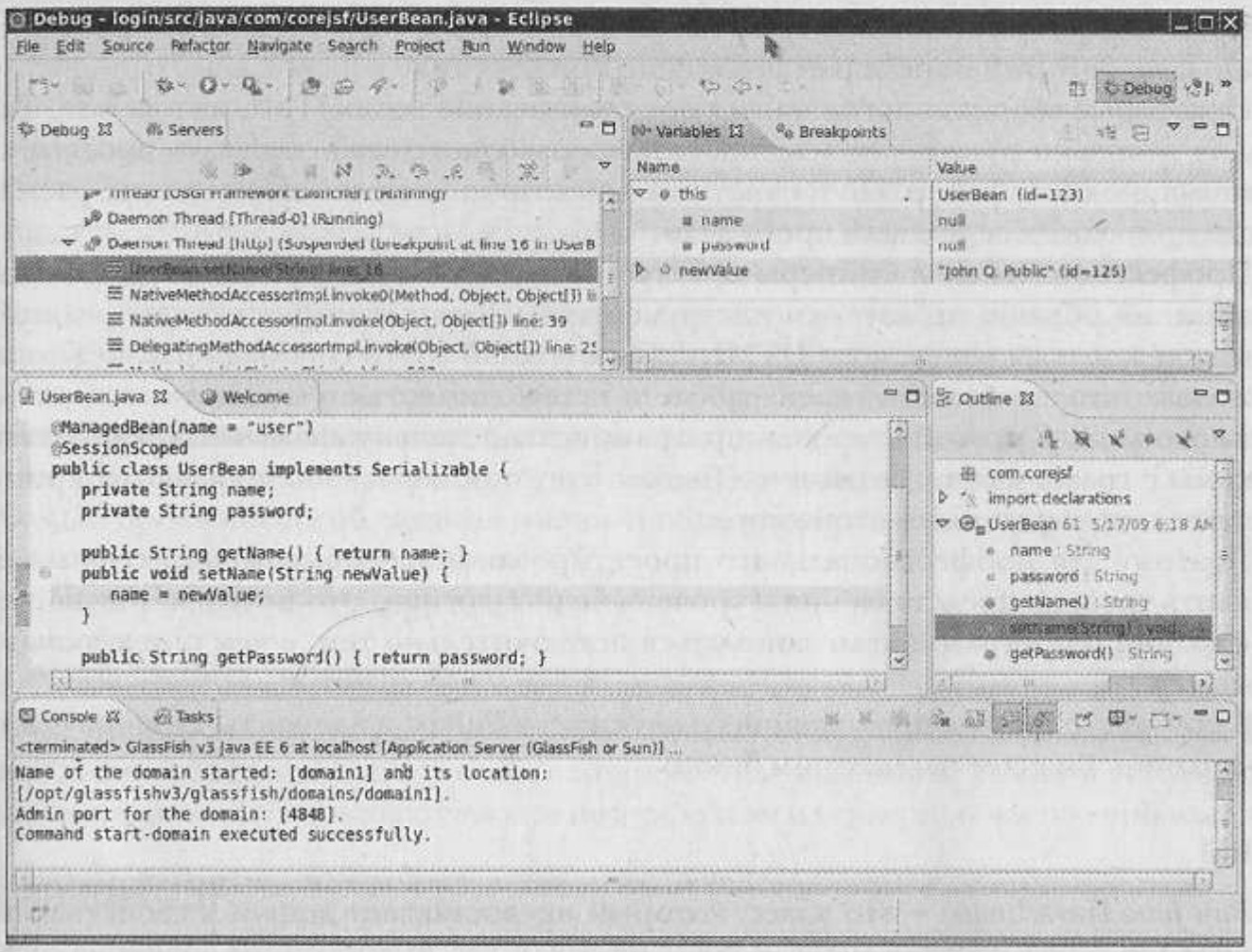


Рис. 1.9. Отладчик Eclipse

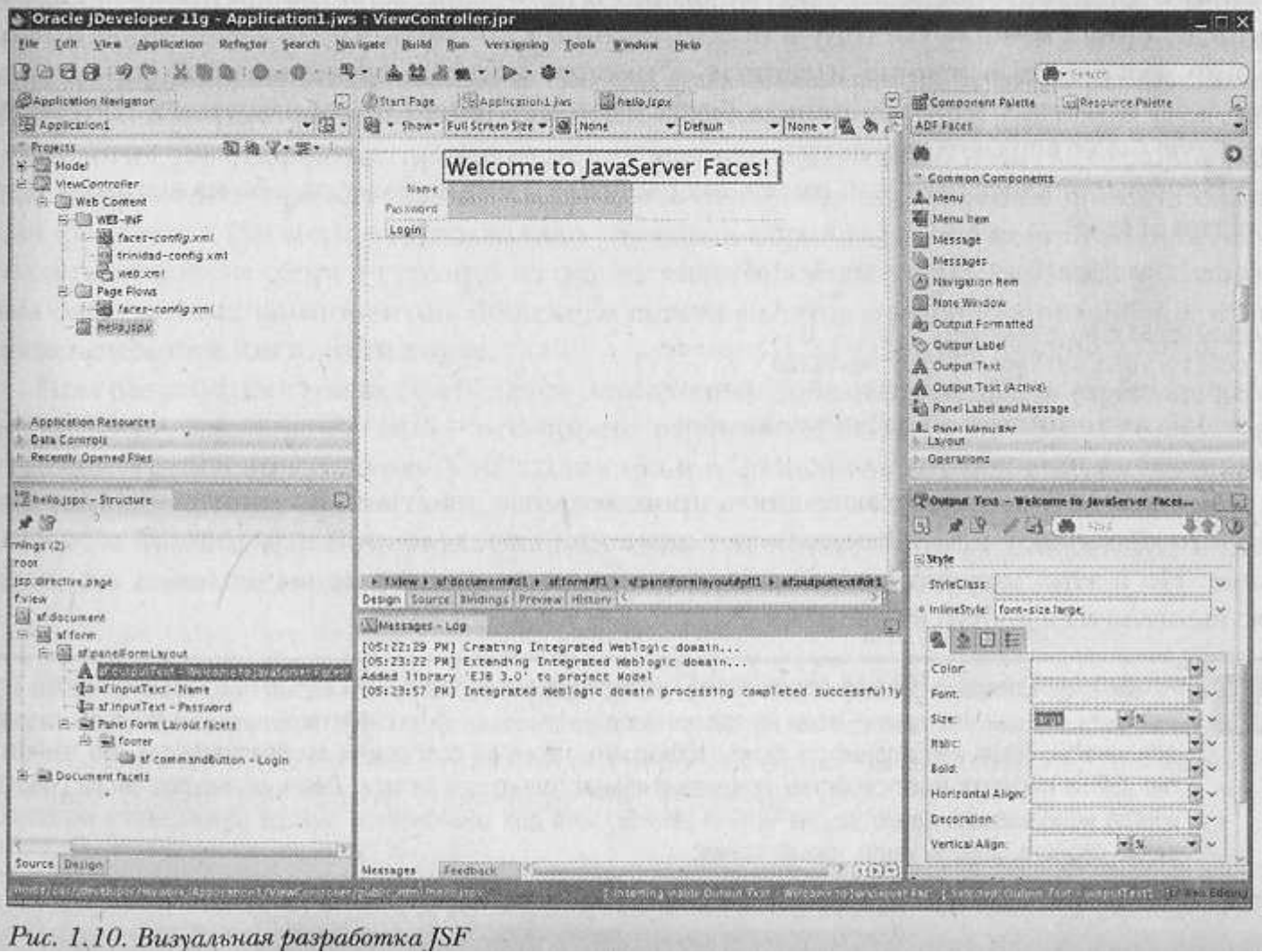


Рис. 1.10. Визуальная разработка JSF

которые задают компоновку, шрифты, изображения и т.д. *Бизнес-логика* реализуется в коде Java, который определяет поведение приложения.

Некоторые веб-технологии допускают смешивание кода HTML и кода Java. Такой подход довольно привлекателен, поскольку позволяет легко создавать простые приложения в одном файле. Но в ответственных приложениях смешивание разметки и кода приводит к серьезным проблемам.

Профессиональные дизайнеры веб-страниц знают о существовании графического дизайна, но обычно пользуются инструментами, которые позволяют им реализовывать свои замыслы в виде кода HTML самостоятельно. Эти специалисты, безусловно, не желают сталкиваться в своей работе с необходимостью учитывать наличие внедренного кода. С другой стороны, программисты печально знамениты тем, что мало знакомы с графическим дизайном. (Явным тому подтверждением являются примеры программ, приведенные в этой книге.)

Поэтому для профессионального проектирования веб-приложений очень важно отделить уровень представления и уровень бизнес-логики. Это позволяет дизайнерам веб-страниц и программистам заниматься исключительно тем, в чем они хорошо разбираются.

В контексте JSF код приложения содержится в бинах, а элементы дизайна – в коде веб-страниц. Вначале рассмотрим бины.

Бины

Бин Java (Java bean) – это класс, который предоставляет доступ к свойствам и событиям для такой платформы, как JSF. *Свойство* – это именованное значение определенного типа, которое может быть считано и (или) записано. Самый простой способ определения свойства состоит в использовании стандартного соглашения об именовании для методов чтения и записи, а именно широко применяемого соглашения `get/set`. В именах методов первая буква имени свойства преобразуется в соответствующую букву верхнего регистра.

Например, класс `UserBean` имеет два свойства, `name` и `password`, оба из которых имеют тип `String`:

```
public class UserBean implements Serializable {
  . . .
  public String getName() { . . . }
  public void setName(String newValue) { . . . }
  public String getPassword() { . . . }
  public void setPassword(String newValue) { . . . }
}
```

Методы `get/set` могут выполнять произвольные действия. Во многих случаях они просто извлекают или устанавливают значение определенной переменной в экземпляре. Но в этих методах могут также выполняться некоторые вычисления или даже осуществляться доступ к базе данных.

На заметку! Согласно спецификации бинов, метод задания свойства или метод получения свойства может быть опущен. Например, если не задан метод `getPassword`, то свойство `password` доступно только для записи. Такая необходимость может возникнуть также по соображениям безопасности. Но технология JSF не поддерживает свойства, предназначенные только для записи. Для компонентов ввода следует всегда использовать свойства для чтения-записи, хотя для компонентов вывода разрешается использование свойств, допускающих только чтение.

Управляемый бин — это бин Java, к которому можно обращаться с JSF-страницы. Для управляемого бина должны быть заданы имя и область определения. В нашем примере бин имеет имя `user`, а областью его определения является сеанс. Это означает, что объект бина доступен для одного пользователя, просматривающего много разных страниц. Другим пользователям, работающим с тем же веб-приложением, будут предоставляться другие экземпляры того же объекта бина. Различные области определения бина будут рассматриваться в главе 2.

Бины являются *управляемыми*. Под этим подразумевается следующее. Если на JSF-странице обнаруживается имя бина, реализация JSF определяет местонахождение объекта с этим именем или создает его, если он еще не существует в соответствующей области определения. Например, если к рассматриваемому нами образцу приложения подключится второй пользователь, будет создан еще один объект `UserBean`.

Самый простой способ задания имени и области определения управляемого бина состоит в использовании атрибутов:

```
@Named("user") // или @ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable
```

В JSF-приложениях управляемые бины используются для всех данных, доступ к которым должен предоставляться с той или иной страницы. Бины являются своего рода проводниками между пользовательским интерфейсом и серверной частью приложения.

Страницы JSF

Для каждого экрана, отображаемого в браузере, должна быть предусмотрена отдельная JSF-страница. Для разработки JSF-страниц предусмотрено несколько разных механизмов, что обусловлено историческими причинами. Версия JSF 1.x была основана на технологии JavaServer Pages (JSP), в связи с чем разработчикам приходилось решать некоторые трудоемкие технические задачи по созданию представления. Но технология JSF разрешает программистам выбирать для обработки JSF-страниц разные обработчики представлений. Именно эта цель была поставлена в проекте создания фейслетов (facelet), что позволило улучшить обработку сообщений об ошибках, создать механизм сборки страниц из общих частей, а также упростить задачу написания собственных компонентов. Фейслеты вошли в состав спецификации JSF 2.0, и мы используем их в настоящей книге.

При разработке страниц фейслетов необходимо добавить теги JSF к странице в коде XHTML. *Страница XHTML* — это просто страница HTML, код которой полностью отвечает требованиям к коду XML. Для страниц фейслетов мы используем расширение `.xhtml`.

Рассмотрим еще раз первую страницу примера приложения, приведенную в листинге 1.1. В начале страницы находится объявление пространства имен:

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
```

Во второй строке объявляется префикс `h:` для тегов HTML, применяемых на JSF-странице. Реализация JSF определяет также набор основных тегов, независимых от HTML. Если на создаваемой странице потребуется применить такой тег, то нужно будет ввести дополнительное объявление пространства имен:

```
xmlns:f="http://java.sun.com/jsf/core"
```

При использовании библиотек тегов, полученных от других поставщиков, достаточно ввести дополнительные объявления пространств имен.

На заметку! Можно выбирать по желанию любые префиксы тегов, например, задавать теги как `html:inputText` вместо `h:inputText`. В этой книге для базовых тегов всегда используется префикс `f`, а для HTML-тегов — префикс `h`.

JSF-страница во многом напоминает форму HTML, но имеет некоторые отличия.

- Код страницы должен представлять собой формально правильный код XHTML. В отличие от браузера реализация JSF при обнаружении синтаксических ошибок аварийно завершает обработку страницы.
- Вместо `head`, `body` и `form` используются теги в формате `h:head`, `h:body` и `h:form`.
- Вместо знакомых HTML-тегов ввода применяются теги `h:inputText`, `h:inputSecret` и `h:commandButton`.

На заметку! Такие теги JSF, как

```
<h:inputText value="#{user.name}"/>
```

могут быть заменены обычными тегами HTML с атрибутом `jsfc`:

```
<input type="text" jsfc="h:inputText" value="#{user.name}"/>
```

Эта возможность предназначена для упрощения разработки страниц в инструментальном средстве веб-дизайна. Но это применительно только к тем компонентам JSF, которые непосредственно соответствуют компонентам HTML. В настоящей книге мы всегда используем теги JSF.

На заметку! Читатели, знакомые с выпущенными ранее версиями JSF, возможно, видели JSF-страницы, которые определены с применением синтаксиса JSP:

```
<html>
 <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 <f:view>
  <head>...</head>
  <body>...</body>
 </f:view>
</html>
```

Возможность использования синтаксиса JSP в версии JSF 2.0 все еще предусмотрена, но авторы не рекомендуют это делать. Дело в том, что при этом обнаруживается один недостаток: если на странице имеется синтаксическая ошибка, то могут быть получены весьма загадочные сообщения об ошибках. Но еще более важно то, что некоторые функции JSF 2.0 (например, связанные с применением шаблонов) работают только с фейслетами.

Все стандартные JSF-теги и их атрибуты будут рассматриваться более подробно в главах 4 и 5. В первых трех главах будут описаны поля ввода и командные кнопки.

Значения полей ввода привязаны к свойствам бина с именем `user`:

```
<h:inputText value="#{user.name}"/>
```

Разграничители `#{...}` заключают в себе выражения на так называемом *языке выражений* JSF, который подробно рассматривается в главе 2.

При отображении страницы реализация JSF определяет местонахождение бина `user` и вызывает метод `getName` для получения текущего значения свойства. При передаче страницы реализация JSF вызывает метод `setName` для задания значения, которое было введено в форму.

Тег `h:commandButton` имеет атрибут `action`, значение которого указывает, какая страница должна быть отображена следующей:

```
<h:commandButton value="Login" action="welcome"/>
```

Конфигурация сервлета

При развертывании JSF-приложения в сервере приложений необходимо предоставить файл конфигурации web.xml. К счастью, для большинства JSF-приложений можно использовать один и тот же файл web.xml (листинг 1.4).

Листинг 1.4. Файл login\web\WEB-INF\web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   version="2.5">
   <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>faces/index.xhtml</welcome-file>
   </welcome-file-list>
   <context-param>
      <param-name>javax.faces.PROJECT_STAGE</param-name>
      <param-value>Development</param-value>
   </context-param>
</web-app>
```

Все JSF-страницы передаются сервлету Faces, который является частью кода реализации JSF. Для того чтобы при запросе JSF-страницы был активизирован требуемый сервлет, для URL-адресов, применяемых в JSF, предусмотрен специальный формат. В данной конфигурации эти адреса имеют префикс /faces. Это — элемент сопоставления с сервлетом, который гарантирует обработку сервлетом Faces всех URL с этим префиксом.

Например, нельзя просто ввести в браузере адрес http://localhost:8080/login/index.xhtml. Этот URL должен иметь форму http://localhost:8080/login/faces/index.xhtml. Согласно применяемому правилу сопоставления, произойдет активизация сервлета Faces, который является точкой входа в реализацию JSF. Реализация JSF удаляет префикс /faces, загружает страницу index.xhtml, обрабатывает теги и отображает результат.

Внимание! Если JSF-страница будет просматриваться без префикса /faces, то браузер отобразит на странице содержимое, соответствующее тегам HTML, а теги JSF просто пропустит.

На заметку! Можно также определить отображение расширения имени файла вместо отображения префикса /faces. Для этого в файл web.xml достаточно ввести следующую директиву:

```
<servlet-mapping>
 <servlet-name>Faces Servlet</servlet-name>
 <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

После этого появляется возможность использовать URL наподобие `http://localhost:8080/login/index.faces`. Этот URL также активизирует сервлет Faces. Реализация JSF удалит расширение `faces` и загрузит файл `/login/index.xhtml`.

На заметку! Строго говоря, JSF-страницы не представляют собой файлы XHTML; их задача состоит лишь в выработке таких файлов. Если желательно использовать для файлов JSF-страниц расширение `.jsf`, то нужно добавить следующую запись в файл `web.xml`:

```
<context-param>
 <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
 <param-value>.jsf</param-value>
</context-param>
```

Следует отметить, что такая конфигурация влияет только на работу веб-разработчиков, но никак не на работу пользователей данного веб-приложения. URL по-прежнему будут иметь расширение `.faces` или префикс `/faces`.

Файл `web.xml` определяет *начальную страницу*, т.е. страницу, которая загружается, когда пользователь вводит URL веб-приложения. Например, если пользователь вводит URL `http://localhost:8080/login`, то сервер приложений автоматически загружает страницу `/faces/index.xhtml`.

Наконец, определим параметр, который вводит в действие поддержку для отладки приложения JSF:

```
<context-param>
  <param-name>javax.faces.PROJECT_STAGE</param-name>
  <param-value>Development</param-value>
</context-param>
```

Вариантами для этой стадии проектирования являются `Development` (Разработка), `UnitTest` (Испытание компонентов), `SystemTest` (Испытание системы) и `Production` (Производство). На стадии разработки формируются более содержательные сообщения об ошибках.

На заметку! Параметр `PROJECT_STAGE` был впервые введен в версии JSF 2.0.

На заметку! Некоторые серверы приложений (включая GlassFish) автоматически обеспечивают сопоставление с сервлетом для шаблонов `/faces/*`, `*.faces` и `*.jsf` при условии соблюдения следующих требований.

- В каждом из классов в веб-приложении используется аннотация JSF.
- Все параметры инициализации начинаются с префикса `javax.faces`.
- Предусмотрен файл `WEB-INF/faces-config.xml`.

Предоставлять файл `web.xml` нет необходимости, если не требуется задавать дополнительные параметры. Но мы рекомендуем задавать `Development` в качестве стадии проектирования, поэтому во всех своих примерах предусматриваем файлы `web.xml`. Если известно, что применяемый сервер приложений автоматически обнаруживает приложения JSF, то можно опустить объявление сервлета Faces и сопоставление с сервлетом из применяемого файла `web.xml`.

Первое знакомство с Ajax JSF 2.0

Одной из технологий обновления веб-страницы в браузере клиента без передачи формы и вывода на экран результатов обработки ответа является Ajax (Asynchronous JavaScript and XML). При этом веб-страница содержит код JavaScript, который взаимодействует с сервером и вносит последовательные изменения в структуру страницы. Благодаря этому страница разворачивается перед глазами пользователя более плавно и не возникают неприятные "подергивания" страницы.

Технология Ajax будет подробно рассматриваться в главе 10. К счастью, версия JSF 2.0 позволяет использовать Ajax, не сталкиваясь со значительными сложностями, которые связаны с изучением канала связи Ajax. В настоящей главе рассматривается небольшое приложение, которое позволяет быстро оценить преимущества JSF.

Для этого структура приложения для входа в систему будет изменена так, чтобы с помощью кнопки Login (Вход) выполнялся запрос Ajax вместо передачи формы. Сразу после вхождения пользователя в систему появляется приветствие (рис. 1.11).

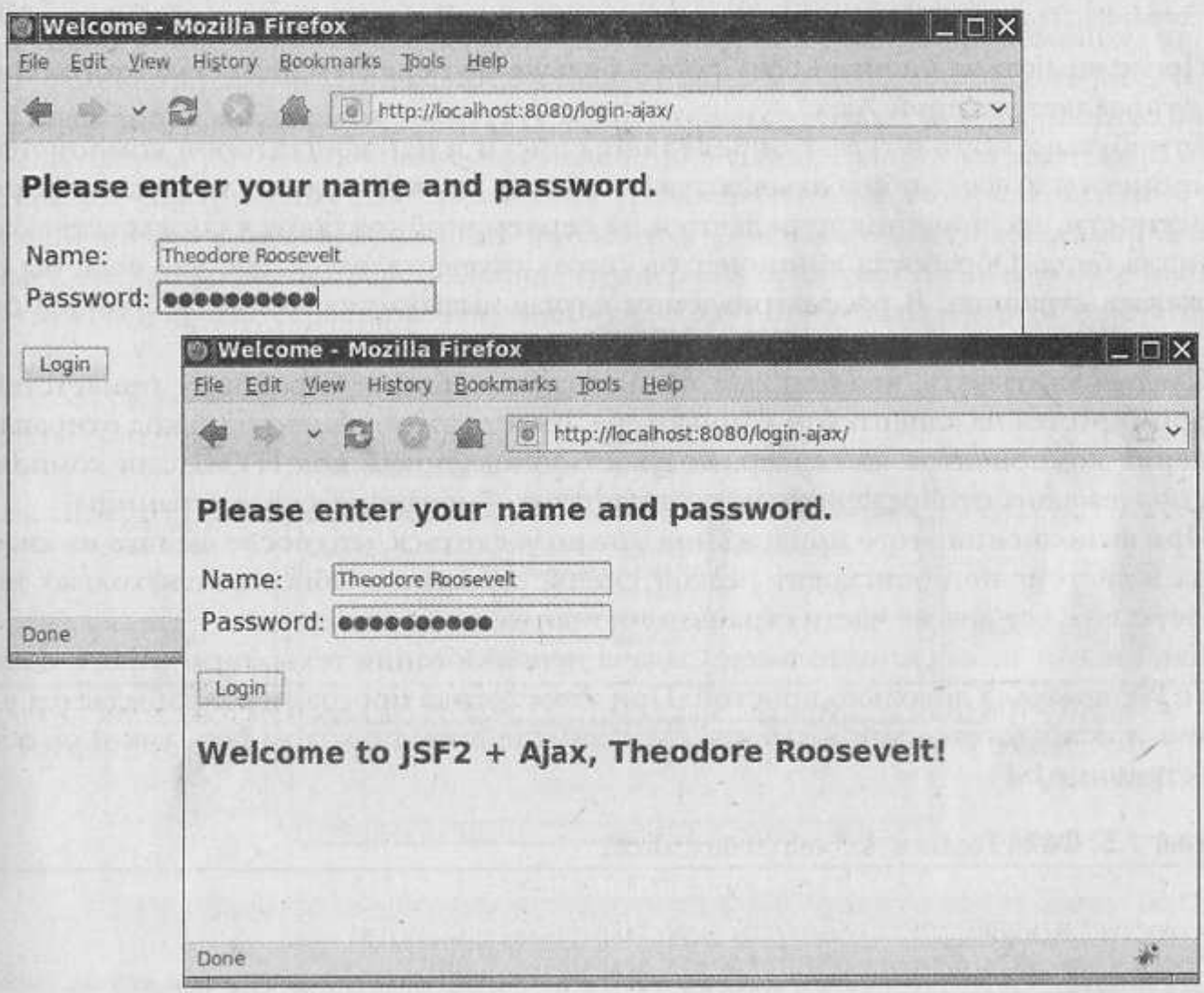


Рис. 1.11. Благодаря использованию Ajax приветственное сообщение развертывается без резкой смены страницы

Для каждого компонента, доступ к которому осуществляется в клиентском коде, требуется идентификатор, для объявления которого служит атрибут `id`, как показано ниже.

```
<h:outputText id="out" value="#{user.greeting}"/>
```

Идентификаторы присваиваются также полям ввода имени и пароля.

По умолчанию ко всем идентификаторам компонентов формы в качестве префи са добавляется идентификатор самой формы. В данном примере эта процедура отм нена в целях получения более простых имен идентификаторов; для этого атрибут prependId формы присваивается значение false (листинг 1.5).

К классу UserBean добавляется назначенное только для чтения свойство greeting:

```
public String getGreeting() {
if (name.length() == 0) return "";
  else return "Welcome to JSF2 + Ajax, " + name + "!";
}
```

Это свойство определяет текст приветствия, который отображается в текстово области.

Теперь можно приступать к реализации действий, определяемых технологие Ajax, для кнопки Login:

```
<h:commandButton value="Login">
  <f:ajax execute="name password" render="out" />
</h:commandButton>
```

После щелчка на кнопке Login форма больше не передается. Вместо этого на се вер отправляется запрос Ajax.

Атрибуты execute и render определяют списки идентификаторов компонентов Компоненты execute обрабатываются точно так же, как в случае передачи формы В частности, их значения передаются на сервер и обновляются соответствующи свойства бина. Обработка компонентов render осуществляется так, как если бы ото бражалась страница. В рассматриваемом случае вызывается метод getGreeting бин user, а полученный результат передается клиенту и отображается.

Следует учитывать, что бин user находится на сервере. Страница с приветствие не формируется на клиентском компьютере. Вместо этого клиентский код отправляе значения компонентов на сервер, получает обновленный код HTML для компонен тов, подлежащих отображению, и соединяет эти обновления в виде страницы.

При выполнении этого приложения можно убедиться, что после щелчка на кнопк входа в систему не происходит резкой смены страницы. Обновляется только тек приветствия; все другие части страницы остаются неизменными.

Как следует из сказанного выше, задача использования технологии Ajax в сочет нии с JSF является довольно простой. При этом логика программы оформляется в ко де Java и используется тот же механизм взаимодействия с кодом Java, как и на обы ной странице JSF.

Листинг 1.5. Файл login-ajax\web\index.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:h="http://java.sun.com/jsf/html"
6.       xmlns:f="http://java.sun.com/jsf/core">
7.    <h:head>
8.       <title>Welcome</title>
9.    </h:head>
10.   <h:body>
11.      <h:form prependId="false">
12.         <h3>Please enter your name and password.</h3>
13.         <table>
```

```
14.             <tr>
15.                <td>Name:</td>
16.                <td><h:inputText value="#{user.name}" id="name"/></td>
17.             </tr>
18.             <tr>
19.                <td>Password:</td>
20.                <td><h:inputSecret value="#{user.password}" id="password"/></td>
21.             </tr>
22.          </table>
23.          <p><h:commandButton value="Login">
24.             <f:ajax execute="name password" render="out"/>
25.          </h:commandButton></p>
26.          <h3><h:outputText id="out" value="#{user.greeting}"/></h3>
27.       </h:form>
28.    </h:body>
29. </html>
```

Службы платформы JSF

Теперь, когда читатель ознакомился с первым примером JSF-приложения, гораздо проще описать услуги, которые предоставляет разработчикам платформа JSF. На рис. 1.12 показано, как выглядит архитектура JSF в общих чертах. На этом рисунке видно, что платформа JSF обеспечивает взаимодействие с клиентскими устройствами и предоставляет инструменты для связывания уровней визуального представления, прикладной логики и бизнес-логики веб-приложения. Однако область действия JSF охватывает только уровень представления. Поддержка хранения данных в базе данных, веб-службы и другие серверные соединения выходят за пределы области действия JSF.

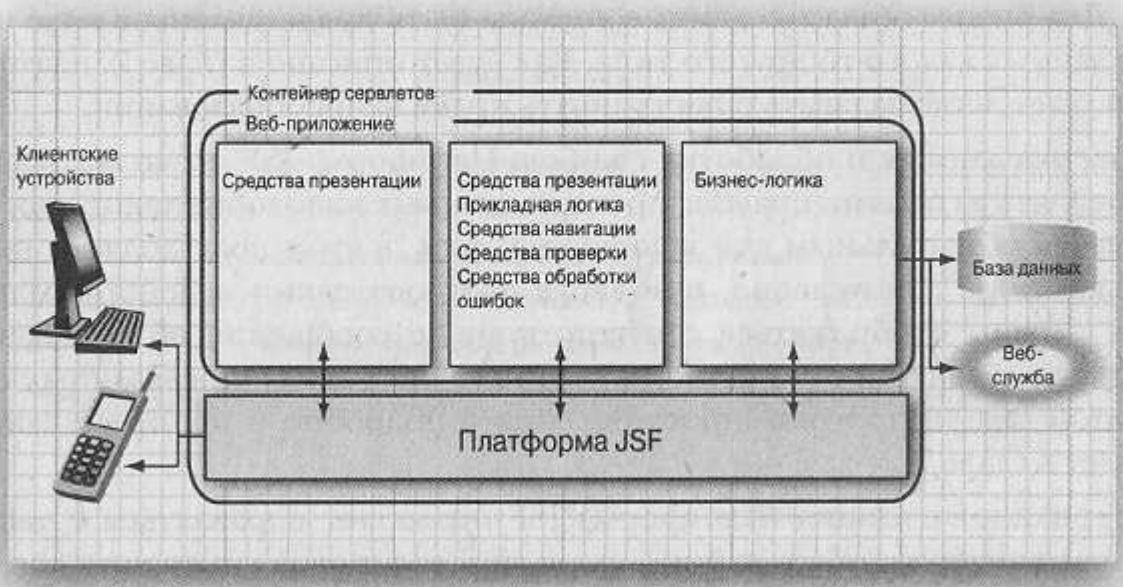


Рис. 1.12. Общий обзор платформы JSF

Ниже перечислены наиболее важные службы, предоставляемые платформой JSF.

- Архитектура "модель–представление–контроллер". Все программные приложения позволяют пользователям манипулировать определенными данными, в том числе данными тележек для покупок, маршрутов путешествий и прочими данными, которые требуются в конкретной предметной области. Данные кон-

кретного приложения принято называть *моделью*. По аналогии с тем, как художники рисуют картины, рассматривая модели в своей студии, разработчики веб-приложений формируют представления моделей данных. В веб-приложениях для отображения таких представлений применяется HTML (или аналогичная технология визуализации).

Платформа JSF соединяет представления и модели. Как показано выше, компонент представления может быть привязан к свойству бина объекта модели примерно таким образом:

```
<h:inputText value="#{user.name}"/>
```

Реализация JSF функционирует как контроллер, который реагирует на действия пользователя, обрабатывая последствия этих действий и события по изменению значений, а затем передавая полученные результаты в код, который обновляет модель или представление. Предположим, например, что необходимо вызвать метод, который проверяет, разрешено ли некоторому пользователю входить в систему. Для этого применяется следующий JSF-тег:

```
<h:commandButton value="Login" action="#{user.check}"/>
```

После того как пользователь щелкнет на кнопке и форма будет передана на сервер, реализация JSF вызывает метод `check` бина `user`. В этом методе могут быть выполнены все необходимые действия по обновлению модели, после чего возвращается идентификатор следующей страницы, которая должна быть выведена на экран. Этот механизм будет рассматриваться более подробно в главе 3.

Таким образом, платформа JSF реализует архитектуру "модель–представление–контроллер".

- Преобразование данных. Пользователи вводят данные в веб-формы в виде текста. Для бизнес-объектов данные должны быть представлены в виде чисел, дат или данных какого-то другого типа. Как будет описано в главе 7, платформа JSF позволяет легко задавать и настраивать правила преобразования.
- Проверка данных и обработка ошибок. Платформа JSF позволяет также легко закреплять за полями правила проверки данных наподобие таких, как "это поле является обязательным для заполнения" или "в этом поле должны находиться только числа". Безусловно, при вводе пользователями недействительных данных должны отображаться соответствующие сообщения об ошибках. JSF позволяет избавиться от множества утомительных деталей, связанных с решением этой задачи программирования. Более подробно о проверке данных речь пойдет в главе 7.
- Интернационализация. Платформа JSF позволяет справляться с такими проблемами интернационализации, как выбор кодировок символов и связок ресурсов. Более подробное описание связок ресурсов приведено в главе 2.
- Пользовательские компоненты. Разработчики компонентов имеют возможность разрабатывать сложные компоненты, чтобы дизайнеры страниц могли затем просто перетаскивать их на свои страницы. Предположим, например, что разработчик компонентов создал компонент календаря со всеми свойственными ему аксессуарами. После этого достаточно применить этот компонент на любой из своих страниц с помощью такой команды:

```
<acme:calendar value="#{flight.departure}" startOfWeek="Mon"/>
```

Более подробно пользовательские компоненты описаны в главе 11.

- Поддержка Ajax. Платформа JSF предоставляет стандартный канал связи Ajax, который незаметно для пользователя вызывает серверные действия и обновляет клиентские компоненты. Дополнительная информация по этому вопросу приведена в главе 10.
- Альтернативные средства визуализации. По умолчанию реализация JSF генерирует разметку для HTML-страниц. Но предусмотрена возможность расширить платформу JSF в целях выработки разметки для другого языка описания страниц, такого как WML или XUL. Сразу после разработки технологии JSF подобная гибкость казалась весьма привлекательной. Но сами авторы еще не встречали примеров эффективного использования такого универсального подхода, поэтому не описывают его в данной книге.

Внутренние процессы

Теперь, после знакомства с основными составляющими JSF и их назначением, читатель, по-видимому, захочет узнать о том, как реализация JSF выполняет свою работу.

Поэтому далее мы расскажем о том, что же происходит "за кулисами" нашего демонстрационного приложения. Начнем описание с того момента, когда браузер впервые подключается к адресу `http://localhost:8080/login/faces/index.xhtml`. Реализация JSF инициализирует код JSF и считывает страницу `index.xhtml`. На этой странице содержатся такие теги, как `h:form` и `h:inputText`. С каждым тегом связан класс *обработчика тега* (tag handler). При считывании страницы выполняются обработчики тегов. Обработчики тега JSF вступают во взаимодействие, в результате чего создается дерево компонентов (рис. 1.13).

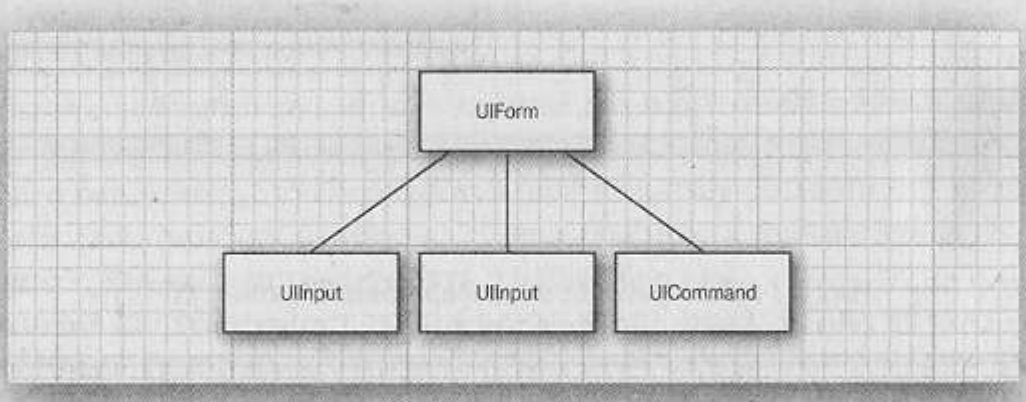


Рис. 1.13. Дерево компонентов для рассматриваемого примера приложения

Дерево компонентов — это структура данных, содержащая Java-объекты для всех элементов пользовательского интерфейса на конкретной JSF-странице. Например, два объекта `UIInput` в этом дереве соответствуют полям `h:inputText` и `h:inputSecret` в файле JSF.

Визуализация страниц

После этого визуализируется HTML-страница. Весь текст, который не обозначен тегами JSF, пропускается. Теги `h:form`, `h:inputText`, `h:inputSecret` и `h:commandButton` преобразуются в код HTML.

Как уже было сказано, обработка каждого из этих тегов приводит к созданию соо ветствующего компонента. Каждый компонент имеет *модуль подготовки к отображени* (renderer), который осуществляет вывод HTML, отражая состояние компонента. На пример, модуль подготовки к отображению для компонента, который соответствуе тегу `h:inputText`, производит следующий вывод:

```
<input type="text" name="unique ID" value="current value"/>
```

Этот процесс называется *кодированием* (encoding). Модуль подготовки к отображе нию объекта `UIInput` запрашивает реализацию JSF, чтобы был выполнен поиск уни кального идентификатора и текущего значения выражения `user.name`. По умолчани строки идентификаторов присваиваются реализацией JSF. Идентификаторы могу иметь довольно непривычный вид, такой как `_id_id12:_id_id21`.

Закодированная страница передается в браузер, который отображает ее обычны образом (рис. 1.14).

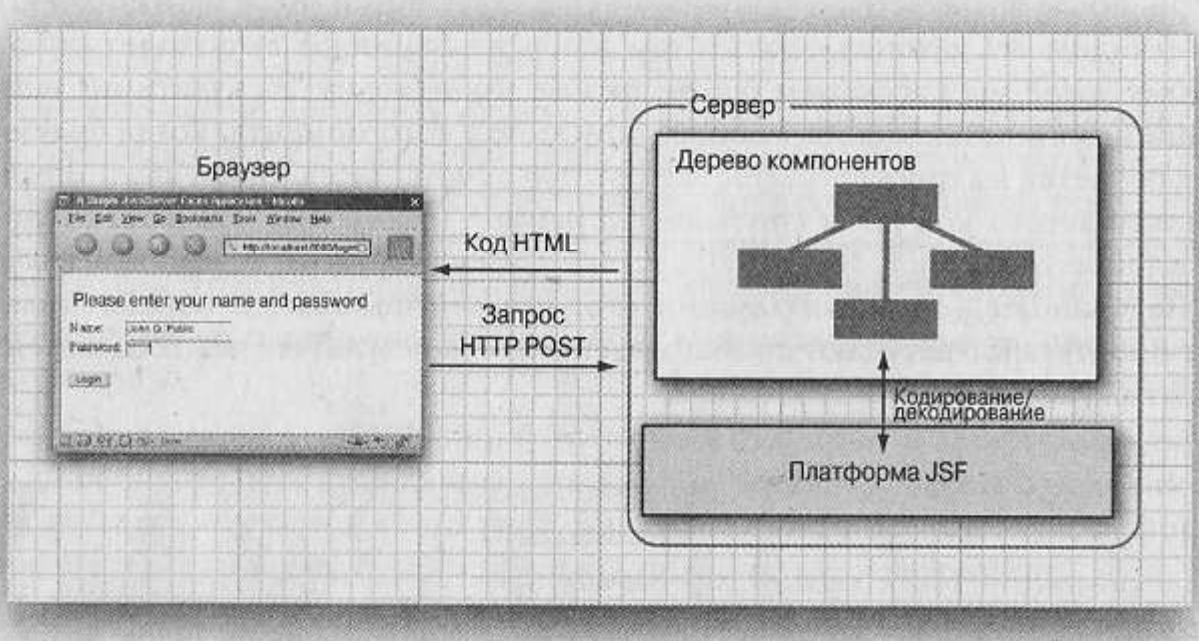


Рис. 1.14. Кодирование и декодирование страниц JSF

Совет. Выберите команду View⇨Page source в меню браузера для просмотра результатов вывода ко да HTML, полученного в процессе подготовки к отображению. На рис. 1.15 показан типичный вывод. Эт может быть полезно для устранения проблем JSF.

Декодирование запросов

После отображения страницы в окне браузера пользователь заполняет поля фор мы и щелкает на кнопке входа в систему. Затем браузер отправляет данные формы обратно веб-серверу в формате запроса `POST`. Это – специальный формат, являю щийся частью протокола HTTP. Запрос `POST` содержит URL формы (`/login/faces/index.xhtml`), а также данные формы.

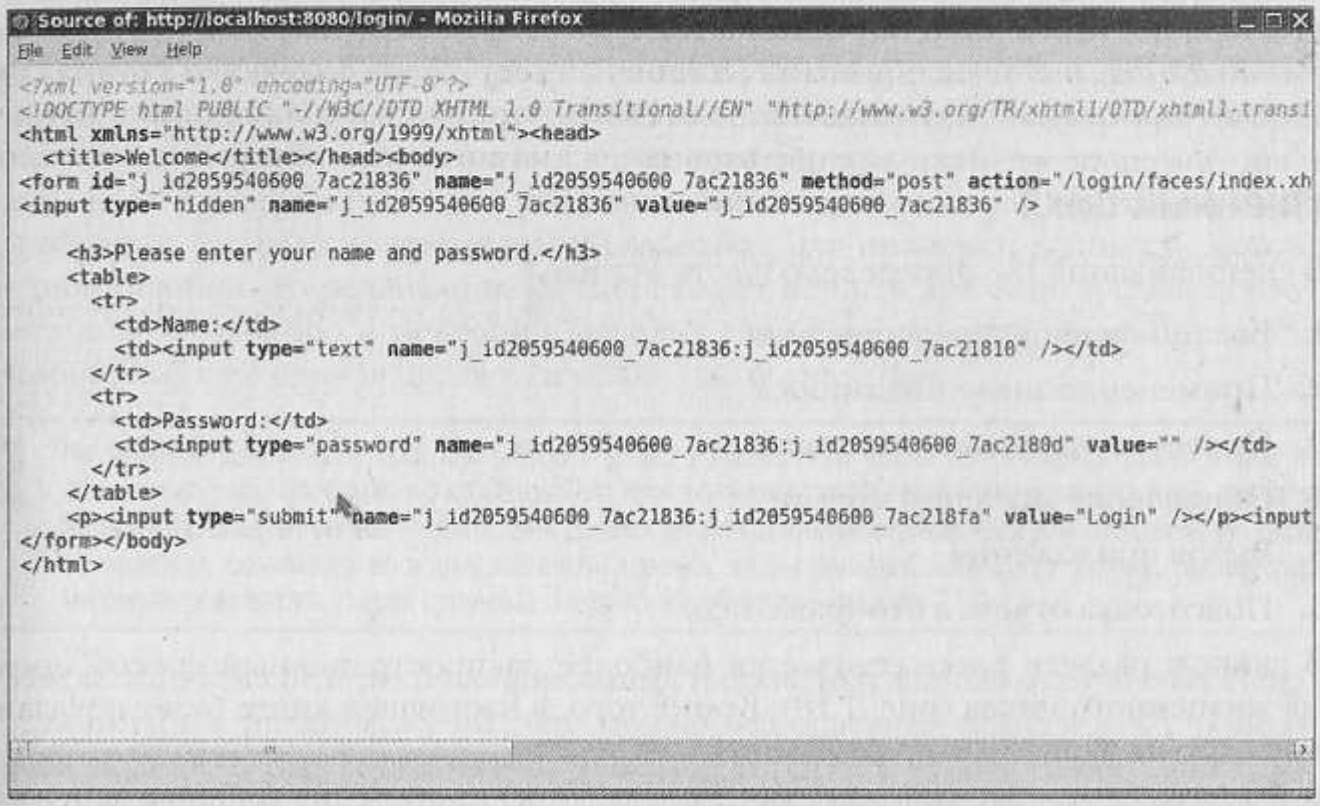


Рис. 1.15. Просмотр исходного кода страницы входа в систему

На заметку! URL для запроса `POST` совпадает с запросом, с помощью которого форма подготавливается к отображению. Переход на новую страницу происходит после отправки формы. (Поэтому в браузере обычно отображается URL, предшествующий URL страницы JSF, которая в текущий момент показана на экране.)

Данные формы имеют вид строки, состоящей из пар "идентификатор–значение", например:

```
id1=me&id2=secret&id3=Login
```

В ходе обычной обработки запроса данные формы помещаются в хеш-таблицу, к которой могут обращаться все компоненты.

После этого реализация JSF предоставляет каждому компоненту возможность просматривать эту хеш-таблицу в процессе, называемом *декодированием*. Каждый компонент сам решает, как интерпретировать данные формы.

Форма входа в систему имеет три объекта компонентов: два объекта `UIInput`, которые соответствуют текстовым полям формы, и объект `UICommand`, который соответствует кнопке отправки формы.

- Компоненты `UIInput` обновляют свойства бина, на которые имеются ссылки в атрибутах значений: они вызывают методы задания значений свойств и передают им значения, предоставленные пользователем.
- Компонент `UICommand` проверяет, был ли выполнен щелчок на кнопке, и в случае положительного ответа активизирует событие действия по запуску действия `login`, указанного в атрибуте `action`. Это событие указывает обработчику навигации, что нужно выполнить поиск следующей страницы, `welcome.xhtml`.

Далее цикл повторяется. Мы только что продемонстрировали две самые важные фазы в процессе обработки, осуществляемом реализацией JSF: кодирования и декодирования. Однако последовательность обработки (также называемая *жизненным циклом*) на самом деле является немного более сложной. Если все проходит нормально, то нет необходимости задумываться о деталях жизненного цикла. Однако в случае

возникновения какой-либо ошибки без четкого понимания действий, выполняем реализацией JSF, никак не обойтись. Поэтому в следующем разделе мы более подро но рассмотрим сам жизненный цикл.

Жизненный цикл

В спецификации JSF определено шесть этапов.

1. Восстановление представления.
2. Применение значений запроса.
3. Проверка правильности процесса.
4. Обновление значений модели.
5. Вызов приложения.
6. Подготовка ответа к отображению.

В данном разделе рассматривается наиболее распространенный способ прохо дения жизненного цикла (рис. 1.16). Кроме того, в настоящей книге будет показан ц лый ряд других вариантов прохождения.

В *фазе восстановления представления* (Restore View) происходит выборка дерева ко понентов для затребованной страницы, если эта страница уже отображалась ране либо создается новое дерево компонентов, если она отображается впервые.

Если запрос не требует получения данных, то реализация JSF переходит к *фазе по готовки ответа к отображению* (Render Response). Это происходит тогда, когда стр ница отображается впервые.

В противном случае следующей становится *фаза применения значений запроса* (App Request Values). В этой фазе реализация JSF обрабатывает в цикле все объекты ко понентов в дереве компонентов. Каждый объект компонента проверяет, какие и значений запроса принадлежат ему, и сохраняет эти значения.

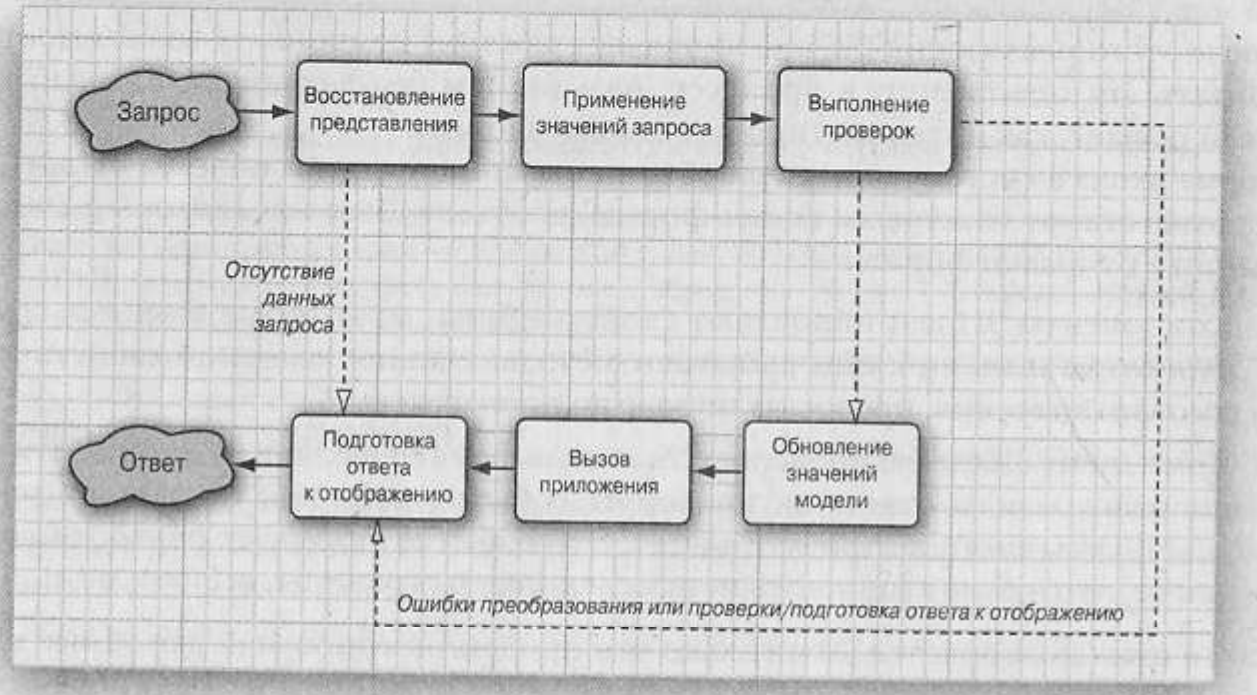


Рис. 1.16. Жизненный цикл JSF

Значения, сохраненные в компоненте, называются *локальными значениями*. При проектировании страницы JSF разработчик может закреплять за страницей *средства проверки*, выполняющие проверку правильности локальных значений. Эти средства проверки выполняют свои действия в *фазе проверки правильности процесса* (Validations Phase). Если проверка проходит успешно, жизненный цикл JSF продолжается обычным образом. Но если во время преобразования или проверки данных возникают какие-либо ошибки, то реализация JSF переходит непосредственно к фазе подготовки ответа к отображению и повторно выводит текущую страницу, чтобы предоставить пользователю еще один шанс ввести правильные значения.

На заметку! Для многих программистов эта часть жизненного цикла JSF является самой непостижимой: если какая-либо операция преобразования или проверки данных заканчивается ошибкой, текущая страница отображается снова. Разработчик должен ввести дополнительные теги для отображения сообщений об ошибках, возникших во время проверки данных, чтобы пользователи могли понять, почему перед ними снова появляется старая страница. Подробнее об этом — в главе 7.

После того как средства преобразования и проверки данных оканчивают свою работу, принимается предположение, что можно без опасений приступить к обновлению данных модели. В *фазе обновления значений модели* (Update Model Values) локальные значения используются для обновления бинов, которые привязаны к компонентам.

В *фазе вызова приложения* (Invoke Application) выполняется метод `action` компонента кнопки или ссылки, щелчок на которой привел к отправке формы. Этот метод позволяет осуществлять любую прикладную обработку. Он возвращает строку результата, которая передается обработчику навигации. Затем обработчик навигации отыскивает следующую страницу.

И наконец, в фазе подготовки ответа к отображению ответ кодируется и передается браузеру. Если пользователь отправляет форму, щелкает на ссылке или иным образом генерирует новый запрос, весь описанный цикл начинается заново.

На заметку! В примере с технологией Ajax происходит так, что запрос Ajax добавляет входные компоненты к *списку исполнения* и выходные компоненты — к *списку подготовки к отображению*. Для компонентов в списке исполнения выполняются все фазы за исключением подготовки ответа к отображению. Особым отличием является то, что бин модели обновляется в фазе обновления значений модели. И наоборот, для компонентов, находящихся в списке подготовки к отображению, выполняется фаза жизненного цикла подготовки ответа к отображению, и результат передается обратно в запрос Ajax.

Резюме

В настоящей главе были рассмотрены базовые механизмы, с помощью которых технология JSF творит свои чудеса. В следующих главах мы расскажем обо всех частях жизненного цикла JSF более подробно.

Управляемые бины

В этой главе...

- Определение бина
- Бины CDI CDI
- Связки сообщений
- Пример приложения
- Области действия бинов
- Настройка конфигурации бинов
- Синтаксис языка выражений

Глава 2

Центральной проблемой в дизайне веб-приложений является разделение уровня представления и уровня бизнес-логики. В технологии JSF такое разделение достигается с помощью бинов (beans). JSF-страницы ссылаются на свойства бинов, а логика программы закладывается в код реализации этих бинов. Поскольку бины играют такую фундаментальную роль в JSF-программировании, мы рассмотрим их подробно в настоящей главе.

В первой части главы будут описаны основные функциональные возможности бинов, о которых должен знать каждый разработчик JSF. Затем будет представлен пример программы, демонстрирующей все эти функциональные возможности в действии. В оставшейся части главы речь пойдет о средствах более низкого уровня, а именно о настройке конфигурации бинов и выражениях значений. При первом прочтении данной книги читатель может пропустить эти разделы и вернуться к ним после того, как в этом возникнет необходимость.

Определение бина

Согласно спецификации JavaBeans (которая доступна по адресу `http://java.sun.com/products/javabeans/`), *бин Java* — это "допускающий многократное использование программный компонент, которым можно манипулировать в инструментах-конструкторах". Данное определение является слишком общим, и, как будет показано в этой главе, в действительности бины используются для многих других целей.

На первый взгляд может показаться, что бин аналогичен *объекту*. Но бины имеют другое назначение. Создание и манипулирование объектами осуществляется в Java-программе, когда она запускает конструкторы и вызывает методы. С другой стороны, бины могут создаваться и подвергаться различному манипулированию без программирования.

На заметку! "Bean" (бин) дословно переводится как "боб". Все дело в том, что слово "Java" в США ассоциируется не только с островом Ява, но и с популярной маркой кофе, а кофе, как известно, готовится из кофе-бобов, которые и формируют его вкус. Кого-то эта аналогия забавляет, кого-то раздражает, но главное, что термин "бин" уже довольно прочно закрепился и вряд ли будет заменен каким-либо другим.

"Классическим" приложением для JavaBeans является конструктор пользовательских интерфейсов. В окне палитры этого конструктора содержатся бины компонентов, такие как текстовые поля, ползунки, флажки и т.д. Вместо написания кода Java разработчик с помощью конструктора пользовательских интерфейсов перетаскивает нужные бины из палитры в форму, а затем настраивает бины путем выбора свойств значений в диалоговом окне страницы свойств (рис. 2.1).

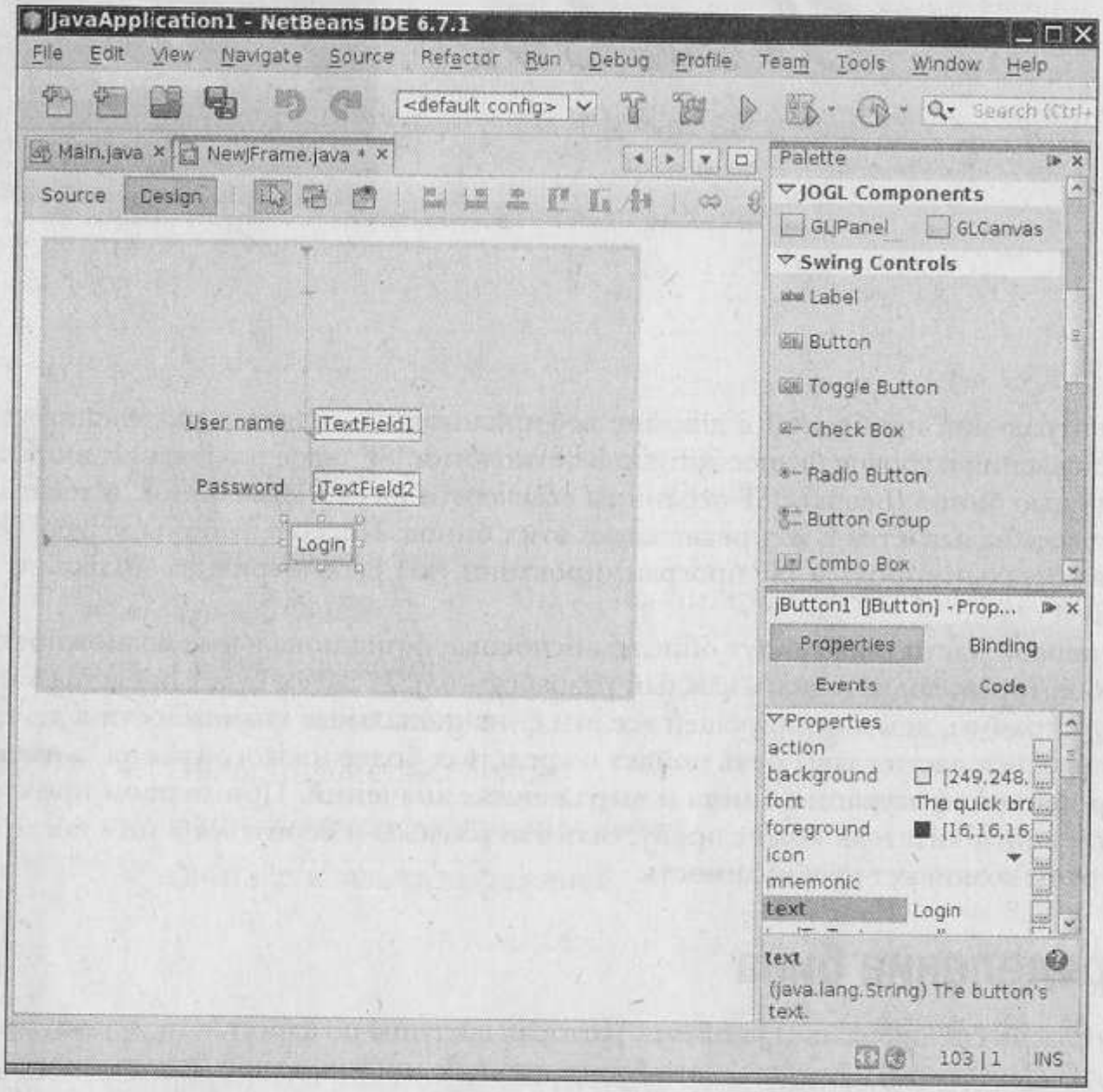


Рис. 2.1. Настройка бина в конструкторе с графическим интерфейсом пользователя

В технологии JSF бины применяются для хранения состояния веб-страниц. Создание бинов и манипулирование ими осуществляются под управлением реализации JSF, которая выполняет следующее.

- Создание и уничтожение бинов по мере необходимости (этим объясняется происхождение термина "управляемые бины").
- Считывание свойств бина при отображении веб-страницы.
- Задание свойств бина при отправке формы.

Рассмотрим приложение входа в систему, приведенное в главе 1, которое показано в разделе "Простой пример" на стр. 20. Чтение и обновление свойства пароля осуществляется с помощью поля ввода бина `user`:

```
<h:inputSecret value="#{user.password}"/>
```

Реализация JSF должна определить местонахождение класса бина для бина с именем user. В рассматриваемом примере класс UserBean был объявлен следующим образом:

```
@ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable {
...
}
```

Атрибут name в аннотации @ManagedBean можно опустить. В таком случае имя бина формируется из имени класса путем приведения первой буквы к нижнему регистру. В частности, если в приведенном выше примере будет исключено выражение (name = "user"), то бин получит имя userBean.

Внимание! Аннотация @ManagedBean находится в пакете javax.faces.bean. Платформа Java EE 6 определяет в пакете javax.annotation другую аннотацию @ManagedBean, которая не работает с JSF.

Впервые встретив выражение с именем user, реализация JSF создает объект класса UserBean. Этот объект продолжает существовать на протяжении всего сеанса; это означает, что все запросы, исходящие от одного и того же клиента, продолжают оставаться действительными до явного завершения сеанса или его прекращения по тайм-ауту. На протяжении всего сеанса имя бина user ссылается на ранее созданный объект.

В любой конкретный момент времени на сервере могут быть активными сеансы, которые принадлежат различным клиентам. В каждом из этих сеансов применяется свой собственный объект UserBean.

На заметку! Класс UserBean реализует интерфейс Serializable. Это требование для управляемого бина JSF не является обязательным, но его соблюдение настоятельно рекомендуется для бинов, действие которых охватывает весь сеанс. Серверы приложений могут обеспечивать лучшее управление для *сериализуемых* (serializable) бинов, например, поддерживать кластеризацию.

Вполне очевидно, что JSF-разработчику не приходится писать код Java для создания бина user и манипулирования им. Реализация JSF создает бины и обращается к ним в соответствии с тем, как описано выражениями на страницах JSF.

Свойства бинов

Для того чтобы классы бинов предоставляли функциональные возможности, пригодные для использования инструментальными средствами, при их создании должны соблюдаться определенные соглашения о программировании. Эти соглашения рассматриваются в настоящем разделе. Самыми важными функциональными возможностями любого бина являются предоставляемые им свойства. *Свойство* — это любой атрибут бина, который имеет следующие характеристики:

- имя;
- тип;
- методы извлечения и (или) задания значения свойства.

Например, класс UserBean, который рассматривался в предыдущей главе, имеет свойство password и тип String. Доступ к значению этого свойства осуществляется с помощью методов getPassword и setPassword.

В некоторых языках программирования, таких как Visual Basic и C#, предусмотрена непосредственная поддержка свойств. Однако в языке Java бин представляет собой просто класс, который следует определенным соглашениям о программировании.

В спецификации JavaBeans к классу бина предъявляется единственное требование: он должен иметь общедоступный конструктор без параметров. Однако, чтобы можно было определять его свойства, бин должен либо использовать шаблон именования для методов получения и задания свойств, либо определять сопровождающий класс `info`. (Классы `info` бина не нашли широкого распространения, поэтому мы не будем их рассматривать в данной книге. См. Cay Horstmann and Gary Cornell, *Core Java*™, Eighth Edition, Santa Clara, CA: Sun Microsystems Press/Prentice Hall, 2008, том 2, глава 8, чтобы получить дополнительную информацию.)

Задача определения свойств с помощью шаблонов именования является несложной. Рассмотрим следующую пару методов:

```
public T getFoo()
public void setFoo(T newValue)
```

Эта пара методов соответствует свойству для чтения и записи с типом `T` и именем `foo`. Если предусмотрен лишь первый метод, свойство становится доступным только для чтения. Если же имеется лишь второй метод, свойство предназначено только для записи.

Имена и сигнатуры методов должны точно соответствовать шаблону. Имя метода должно начинаться с префикса `get` или `set`. Метод `get` не должен иметь параметров, а метод `set` должен иметь один параметр, но не может возвращать значение. Класс бина может иметь другие методы, но их не следует применять для получения свойств бина.

Обратите внимание на то, что имя свойства представляет собой приведенную по определенным правилам к нижнему регистру часть имени метода, которая следует за префиксом `get` или `set`. Например, применение метода `getFoo` приводит к получению свойства `foo`, в котором первая буква преобразована в строчную. Но если прописными являются две первые буквы после префикса, то первая буква остается неизменной. Например, метод `getURL` определяет свойство `URL`, а не `uRL`.

Что касается свойств типа `boolean`, то для метода чтения этих свойств можно выбирать любой префикс. Допустимыми именами для метода чтения связанного свойства являются:

```
public boolean isConnected()
```

и

```
public boolean getConnected()
```

На заметку! Кроме того, спецификация JavaBeans определяет индексированные свойства, задаваемые с помощью набора методов, таких, как показано ниже.

```
public T[] getFoo()
public T getFoo(int index)
public void setFoo(T[] newArray)
public void setFoo(int index, T newValue)
```

Однако в технологии JSF поддержка доступа к индексированным значениям не предусмотрена.

В спецификации JavaBeans ничего не сказано о том, как должны действовать методы задания и получения свойств. Во многих случаях эти методы просто манипулируют одним из полей экземпляра. Но они с таким же успехом могут выполнять и более сложные операции, например, поиск значений в базе данных, преобразование данных, проверку данных и т.д.

Кроме методов получения и задания свойств, класс бина может иметь и другие методы. Безусловно, применение таких методов не приводит к созданию каких-либо свойств бина.

Выражения значений

Как уже было сказано в главе 1, для доступа к свойствам бина могут использоваться выражения, такие как `#{user.name}`. В технологии JSF подобные выражения называются *выражениями значения*. В качестве примера укажем, что страница `welcome.xhtml` содержит такой фрагмент:

```
Welcome to JavaServer Faces, #{user.name}!
```

Реализация JSF при подготовке этой страницы к просмотру вызывает метод `getName` объекта `user`. Выражение значения может использоваться и для чтения, и для записи значения. Рассмотрим следующий компонент ввода:

```
<h:inputText value="#{user.name}"/>
```

Реализация JSF при подготовке этого компонента к просмотру вызывает соответствующий метод получения свойства. А при передаче пользователям страницы реализация JSF вызывает метод задания свойства.

Выражения значения будут рассматриваться более подробно в разделе "Синтаксис языка выражений" на стр. 70.

На заметку! Язык выражений значения в технологии JSF напоминает язык выражений, используемый в технологии JSP. Для разграничения этих выражений используются конструкции `${...}`, а не `#{...}`. После выхода версий JSF 1.2 и JSP 2.1 синтаксис обоих языков выражений был унифицирован. (Подробное описание этого синтаксиса будет приведено в разделе "Синтаксис языка выражений" на стр. 70.)

Разграничитель `${...}` указывает, что оценка выражений должна осуществляться безотлагательно. При обработке страницы происходят вычисление и вставка значения выражения. С другой стороны, разграничитель `#{...}` обозначает отсроченную оценку. Если оценка отсрочена, то реализация JSF сохраняет выражение и вычисляет его каждый раз, когда потребуется его значение.

Как правило, выражения, вычисляемые с отсрочкой, используются для свойств компонентов JSF, а выражения, вычисляемые незамедлительно, — в простых конструкциях JSP или JSTL (JavaServer Pages Standard Template Library — библиотека стандартных шаблонов страниц JavaServer), (Необходимость в использовании этих конструкций на страницах JSF возникает редко.)

Вспомогательные бины

Иногда бывает удобно спроектировать бин, содержащий объекты некоторых или всех компонентов веб-формы. Бин такого типа называется *вспомогательным бином* веб-формы.

Например, мы можем определить вспомогательный бин для формы викторины, добавив свойства для компонента формы:

```
@ManagedBean(name="quizForm")
@SessionScoped
public class QuizFormBean {
  private UIInput answerComponent;
  private UIOutput scoreComponent;
  public UIInput getAnswerComponent() { return answerComponent; }
  public void setAnswerComponent(UIInput newValue) { answerComponent = newValue; }
  public UIOutput getScoreComponent() { return scoreComponent; }
  public void setScoreComponent(UIOutput newValue) { scoreComponent = newValue; }
  ...
}
```

Компоненты ввода принадлежат классу UIInput, а компоненты вывода — классу UIOutput. Эти классы будут описаны более подробно в главе 11.

Вспомогательные бины используются в некоторых вариантах визуальной среды разработки JSF. В таких средах методы получения и задания свойств создаются автоматически для всех компонентов, перетаскиваемых в форму.

Если используется вспомогательный бин, то возникает необходимость связывать компоненты в форме с теми, которые находятся в бине. Для этого применяется атрибут binding:

```
<h:inputText binding="#{quizForm.answerComponent}" .../>
```

При создании дерева компонентов для формы вызывается метод getAnswerComponent вспомогательного бина, но этот метод возвращает значение NULL. Поэтому создается компонент вывода и вставляется во вспомогательный бин с помощью вызова метода setAnswerComponent.

Бины CDI

В технологии JSF впервые введено понятие "управляемых бинов" для использования в веб-приложениях. Но возможности управляемых бинов JSF довольно ограничены. В технологии, которая регламентирована спецификацией JSR 299 (часто сокращенно обозначаемой как CDI — Contexts and Dependency Injection), определена более гибкая модель для бинов, которые управляются с помощью сервера приложений. Эти бины связаны с контекстом (таким как контекст текущего запроса, сеанса браузера или определяемого пользователем жизненного цикла). Технология CDI определяет механизмы для вставки бинов, перехвата и дополнения вызовов методов, а также активизации событий и наблюдения за ними. Технология CDI определяет гораздо более мощный механизм по сравнению с управляемыми бинами JSF, поэтому имеет смысл использовать бины CDI, если приложение развертывается на сервере приложений Java EE. Сервер приложений, совместимый с Java EE 6, такой как GlassFish, автоматически поддерживает технологию CDI.

На заметку! Можно также добавить справочную реализацию CDI к исполнителю сервлетов Tomcat. Дополнительные сведения приведены по адресу http://seamframework.org/Weld.

Бины CDI используются по такому же принципу, как управляемые бины JSF. Но для их объявления служит аннотация @Named, как показано ниже.

```
@Named("user")
@SessionScoped
public class UserBean implements Serializable {
   ...
}
```

После этого можно использовать выражения значения #{user} или #{user.name} по такому же принципу, как и управляемые бины JSF.

В данном случае аннотация @SessionScoped взята из пакета javax.enterprise.context, а не из пакета javax.faces.bean.

На заметку! Для активизации обработки бинов CDI необходимо включить файл WEB-INF/beans.xml. Этот файл может быть пустым или, в случае необходимости, содержать необязательные команды по настройке конфигурации бинов. Дополнительные сведения о файле beans.xml содержатся в спецификации CDI по адресу http://jcp.org/en/jsr/summary?id=299.

Следует отметить, что бины CDI с охватом сеанса должны реализовывать интерфейс Serializable.

Так сложилось исторически, что для поддержки бинов, которые могут использоваться на страницах JSF, предусмотрены два отдельных механизма: бины CDI и управляемые бины JSF. Мы рекомендуем разработчикам всегда использовать бины CDI, за исключением тех случаев, когда приложение должно работать в рамках такого простого исполнителя сервлетов, как Tomcat. Исходный код, прилагаемый к данной книге, подготовлен в двух версиях – с бинами CDI (для серверов приложений Java EE 6) и с управляемыми бинами JSF (для исполнителей сервлетов без поддержки CDI).

Связки сообщений

При реализации веб-приложения целесообразно собрать все строки сообщений в одном централизованном хранилище. Благодаря этому упрощается контроль за согласованностью сообщений, а что еще более важно, становится легче локализовать приложение для других языковых стандартов. В этом разделе будет показано, что предусмотрено в технологии JSF для упрощения организации сообщений. Ниже, в разделе "Пример приложения" на стр. 54 будет показано, как ввести в действие управляемые бины и связки сообщений.

Строки всех сообщений удобнее всего собрать в файле с проверенным временем форматом свойств:

```
guessNext=Guess the next number in the sequence!
answer=Your answer:
```

На заметку! Подробное описание этого формата файла можно найти в документации по API, в разделе, посвященном методу load класса java.util.Properties.

Этот файл необходимо сохранить вместе с применяемыми классами, например, под именем src/java/com/corejsf/messages.properties. Можно выбрать произвольные значения пути к каталогу и имени файла, но в качестве расширения допускается использование только .properties.

Объявить связку сообщений можно двумя способами. Простейший способ состоит в том, чтобы предусмотреть файл с именем faces-config.xml в каталоге WEB-INF конкретного приложения, имеющий следующее содержимое:

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
  version="2.0">
  <application>
    <resource-bundle>
      <base-name>com.corejsf.messages</base-name>
      <var>msgs</var>
  </resource-bundle>
  </application>
</faces-config>
```

Вместо использования глобального объявления связки ресурсов можно добавить элемент f:loadBundle к каждой странице JSF, для которой требуется доступ к связке:

```
<f:loadBundle basename="com.corejsf.messages" var="msgs"/>
```

На заметку! Файл `faces-config.xml` может использоваться для настройки конфигурации многих характеристик приложения JSF. При этом важно использовать правильную версию объявления схемы. В настоящем разделе показано объявление для JSF 2.0. Если используется более старая версия схемы, то в реализации JSF это может рассматриваться как указание по переходу в режим совместимости со старой версией JSF.

Так или иначе, сообщения из связки будут доступны через переменную карты `msgs`. (Базовое имя `com.corejsf.messages` выглядит как имя класса, и действительно этот файл свойств загружается с помощью загрузчика класса.)

После этого для получения доступа к строкам сообщений можно использовать выражения значения наподобие `#{msgs.guessNext}`.

Вот и все! Если возникнет необходимость локализовать приложение для другого языкового стандарта, достаточно будет предоставить локализованные файлы связок.

На заметку! Подход с применением элемента `resource-bundle` является более эффективным по сравнению с применением действия `f:loadBundle`, поскольку связка может создаваться единожды для всего приложения.

При локализации файла связки необходимо добавить суффикс локали к имени файла: символ подчеркивания, за которым следует двухбуквенный код языка по стандарту ISO-639, состоящий из строчных букв. Например, в случае локализации на немецкий язык имя файла должно иметь вид `com/corejsf/messages_de.properties`.

На заметку! Перечень всех двух- и трехбуквенных кодов языков в формате ISO-639 доступен по адресу `http://www.loc.gov/standards/iso639-2/`.

В языке Java предусмотрена поддержка интернационализации, которая, в частности, обеспечивает автоматическую загрузку связки, соответствующей текущей локали. Стандартная связка без префикса локали используется как средство перехода на аварийный режим, если соответствующая локализованная связка недоступна. См. Cay Horstmann and Gary Cornell, *Core Java*™, 8th ed., Santa Clara, CA: Sun Microsystems Press/Prentice Hall, 2008, том 2, глава 5, для ознакомления с подробным описанием средств интернационализации Java.

На заметку! При подготовке переводов для локализации необходимо учитывать одну особенность: файлы связок сообщений не закодированы с помощью кодовой страницы UTF-8. Вместо этого символы Юникода с кодами свыше 127 представляются как экранирующие последовательности `\uxxxx`. Для создания таких файлов может применяться утилита native2ascii, входящая в состав комплекта Java SDK.

Для каждой конкретной локали можно использовать много разных связок. Например, может потребоваться применение отдельной связки для обычно используемых сообщений об ошибках.

Сообщения с переменными частями

Часто сообщения имеют переменные части, которые требуют заполнения. Например, предположим, что требуется отобразить предложение `"You have n points"` (Вы заработали `n` очков), где `n` — значение, извлекаемое из бина. Можно создать строку ресурса с меткой-заполнителем:

```
currentScore=Your current score is {0}.
```

Метки-заполнители нумеруются как {0}, {1}, {2} и т.д. После этого на страницу JSF необходимо ввести тег h:outputFormat и предоставить значения для меток-заполнителей в виде дочерних элементов f:param следующим образом:

```
<h:outputFormat value="#{msgs.currentScore}">
  <f:param value="#{quizBean.score}"/>
</h:outputFormat>
```

При обработке тега h:outputFormat для форматирования строки сообщения используется класс MessageFormat из стандартной библиотеки. Этот класс предоставляет несколько функций форматирования с учетом локали.

Предусмотрена возможность форматировать числа как суммы в валюте путем добавления суффикса number,currency:

```
currentTotal=Your current total is {0,number,currency}.
```

После этого значение 1023.95, например, в США будет форматироваться и отображаться как $1,023.95, а в Германии — как €1.023,95 с использованием символа местной валюты и принятого в этом регионе десятичного разделителя.

Формат choice позволяет форматировать числа разными способами, например: "zero points" (нуль очков), "one point" (одно очко), "2 points" (2 очка), "3 points" (3 очка) и т.д. Строка формата, позволяющая добиться такого эффекта, приведена ниже.

```
currentScore=Your current score is {0,choice,0#zero points|1#one point|2#{0} points}.
```

Всего возможны три случая: 0, 1 и ≥ 2. Каждый случай определяет отдельную строку сообщения.

Обратите внимание на то, что метка-заполнитель 0 появляется дважды — для выбора любого варианта, т.е. формата choice, и для выбора третьего варианта, т.е. для получения результата "3 points".

В листинге 2.5 на стр. 60 и листинге 2.6 на стр. 60 показано применение формата choice в рассматриваемом примере приложения. При использовании английской локали не требуется задавать значение choice для выбора сообщения "Your score is...". А в немецком языке это значение выражается как "Sie haben ... Punkte" (У вас ... очков). Поэтому в данном случае использование формата choice является обязательным для получения варианта в единственном числе "einen Punkt" (одно очко).

Для получения дополнительных сведений о классе MessageFormat см. документацию API или книгу Cay Horstmann and Gary Cornell, *Core Java*™, Eighth Edition, Santa Clara, CA: Sun Microsystems Press/Prentice Hall, 2008, том 2, глава 5.

Установка локали в приложении

После подготовки связок сообщений необходимо принять решение о том, как задать определенную локаль в приложении. При этом возможны три варианта.

1. Можно позволить выбрать локаль браузеру. Задать используемые по умолчанию и поддерживаемые локали в файле WEB-INF/faces-config.xml:

```
<faces-config>
  <application>
<locale-config>
<default-locale>en</default-locale>
<supported-locale>de</supported-locale>
</locale-config>
</application>
</faces-config>
```

При создании соединения с приложением браузер обычно включает в HTT заголовок значение `Accept-Language` (подробнее об этом можно узнать по адре `http://www.w3.org/International/questions/qa-accept-lang-locales.html`). Реал зация JSF считывает этот заголовок и находит среди поддерживаемых локале наиболее подходящую. Эту функцию можно проверить, установив предпочт тельный язык в применяемом браузере (рис. 2.2).

2. Можно установить локаль программным путем. Для этого достаточно вызват метод `setLocale` объекта `UIViewRoot`:

```
UIViewRoot viewRoot = FacesContext.getCurrentInstance().getViewRoot();
viewRoot.setLocale(new Locale("de"));
```

Пример применения такого подхода приведен в разделе "Использование ко мандных ссылок" главы 4 на стр. 134.

3. Можно также установить локаль для отдельной страницы с использование элемента `f:view` с атрибутом `locale`, например:

```
<f:view locale="de">
```

Кроме того, локаль может быть установлена динамически:

```
<f:view locale="#{user.locale}"/>
```

После этого значение локали присваивается строке, возвращаемой методо `getLocale`. Такой подход является удобным для приложений, позволяющих вы бирать предпочтительный языковой стандарт самому пользователю.

Поместите всю свою страницу (включая теги `h:head` и `h:body`) в тег `f:view`.

JSF 2.0

На заметку! До выхода версии JSF 2.0 все страницы JSF должны были быть включены в теги `f:view`. Те перь в этом нет необходимости. Однако тег `f:view` все еще иногда бывает полезным, например, когд требуется установить локаль.

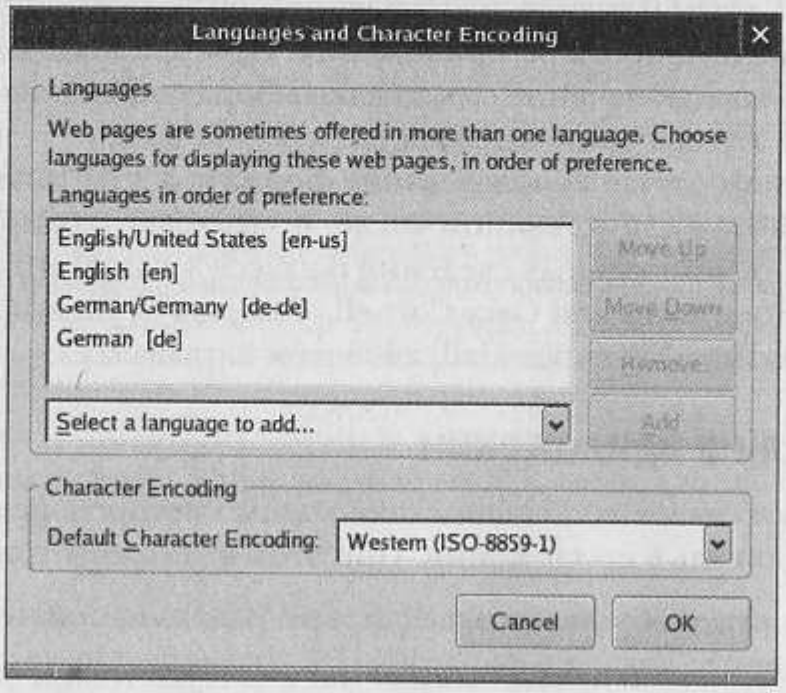


Рис. 2.2. Выбор предпочтительного языка

Пример приложения

После изложения всех этих довольно абстрактных правил и инструкций пришло время рассмотреть конкретный пример. Это приложение предоставляет пользовате

лю возможность ответить на несколько вопросов викторины. Рядом с каждым вопросом отображается последовательность чисел, после чего участнику предлагается угадать следующее число в последовательности.

Например, на рис. 2.3 показан вопрос к пользователю: каким является следующее число в последовательности?

```
3 1 4 1 5
```

Подобные задания часто встречаются в тестах, предназначенных для определения уровня интеллекта. Чтобы выполнить это задание, необходимо определить, по какому шаблону изменяется последовательность. В данном случае перед нами находятся первые цифры числа π.

После ввода следующей цифры в этой последовательности (т.е. 9) общий счет увеличится на одно очко.

На заметку! Для программистов на языке Java, который пронизан аналогиями с понятием "кофе", придумано следующее мнемоническое предложение для запоминания цифр числа π: "Can I have a small container of coffee? Thank you". После подсчета количества букв в каждом слове получаем последовательность цифр 3 1 4 1 5 9 2 6 5 3.

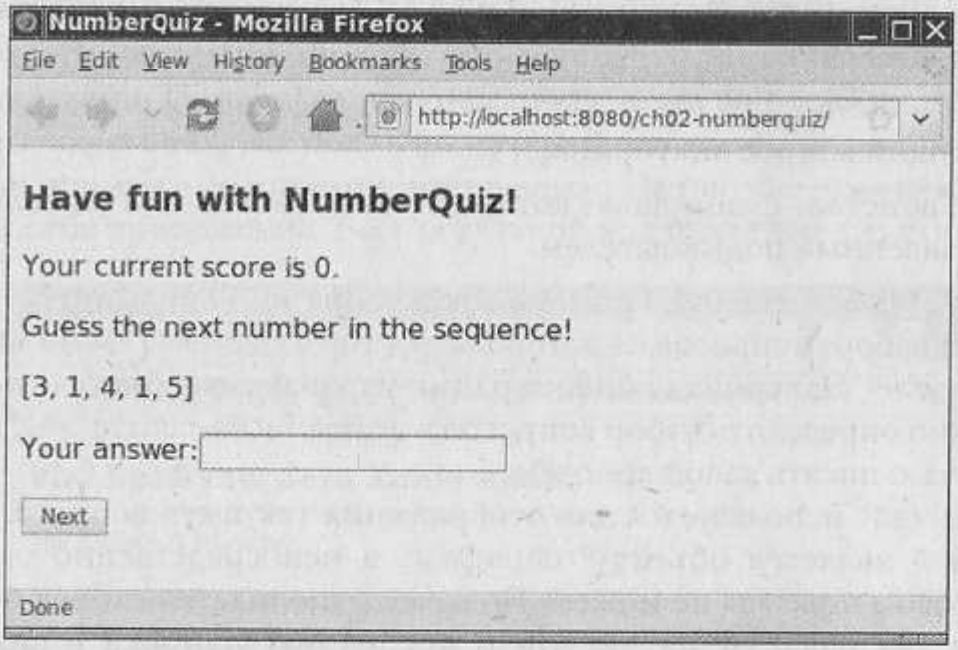


Рис. 2.3. Числовая викторина

В этом примере все вопросы викторины включены в класс `QuizBean`. Безусловно, в реальном приложении такая информация, по всей вероятности, была бы сохранена в базе данных. Но назначение данного примера состоит в демонстрации того, как используются бины, имеющие сложную структуру.

Начнем с класса `ProblemBean`. Класс `ProblemBean` имеет два свойства: свойство `solution` типа `int` и свойство `sequence` типа `ArrayList` (листинг 2.1).

Листинг 2.1. Файл `numberquiz\src\java\com\corejsf\ProblemBean.java`

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.util.ArrayList;
5.
6. public class ProblemBean implements Serializable {
```

```
7.    private ArrayList<Integer> sequence;
8.    private int solution;
9.
10.   public ProblemBean() {}
11.
12.   public ProblemBean(int[] values, int solution) {
13.      sequence = new ArrayList<Integer>();
14.      for (int i = 0; i < values.length; i++)
15.         sequence.add(values[i]);
16.      this.solution = solution;
17.   }
18.
19.   public ArrayList<Integer> getSequence() { return sequence; }
20.   public void setSequence(ArrayList<Integer> newValue) { sequence = newValue; }
21.
22.   public int getSolution() { return solution; }
23.   public void setSolution(int newValue) { solution = newValue; }
24. }
```

После этого определим бин для викторины со следующими свойствами:

- problems. Свойство, предназначенное только для записи и позволяющее формулировать вопросы викторины.
- score. Свойство, предназначенное только для чтения и позволяющее определить текущий счет.
- current. Свойство, предназначенное только для чтения и позволяющее определить текущий вопрос викторины
- answer. Свойство, с помощью которого извлекается и устанавливается ответ, предоставленный пользователем.

Свойство problems в данном примере программы не используется, поскольку для инициализации набора вопросов служит конструктор QuizBean. Тем не менее, как будет показано в разделе "Настройка конфигурации управляемых бинов с помощью XML" на стр. 66, можно определить набор вопросов в файле faces-config.xml, не сталкиваясь с необходимостью писать какой-либо код.

Свойство current используется для отображения текущего вопроса. Но значением свойства current является объект ProblemBean, а непосредственно отобразить этот объект в текстовом поле мы не можем. Поэтому выполняется вторая операция доступа к свойству для извлечения числовой последовательности в виде #{quizBean.current.sequence}.

Значением свойства sequence является ArrayList. При отображении оно преобразуется в строку путем вызова метода toString. Результатом становится строка следующего вида:

```
[3, 1, 4, 1, 5]
```

Наконец, нам придется проделать определенный объем черновой работы со свойством answer — привязать свойство answer к полю ввода:

```
<h:inputText value="#{quizBean.answer}"/>
```

При отображении поля ввода вызывается метод получения свойства, а метод getAnswer определяется так, чтобы он возвращал пустую строку.

При отправке формы вызывается метод задания свойства с тем значением, которое пользователь ввел в поле ввода. Метод setAnswer определен так, чтобы он проверял ответ, обновлял количество очков при правильном ответе и осуществлял переход к следующему вопросу.

```
public void setAnswer(String newValue) {
  try {
 int answer = Integer.parseInt(newValue.trim());
 if (getCurrent().getSolution() == answer) score++;
 currentIndex = (currentIndex + 1) % problems.size();
 }
catch (NumberFormatException ex) {
 }
}
```

Строго говоря, было бы лучше не размещать в методе задания свойства код, не связанный с задачей присваивания значения самому свойству. В действительности код определения количества очков и перехода к следующему вопросу должен находиться в обработчике действия кнопки. Но мы еще не рассматривали тему о том, как реагировать на действия с кнопками, поэтому используем в своих интересах дополнительные возможности метода задания свойства.

Еще один недостаток нашего примера приложения состоит в том, что в нем еще не предусмотрено завершение работы по окончании викторины. Вместо этого был предусмотрен просто возврат в начало, что позволяет пользователю набрать больше очков. Дополнительные сведения, позволяющие усовершенствовать это приложение, приведены в следующей главе. Напомним, что цель данного приложения состояла в демонстрации того, как настраивать конфигурацию и использовать бины.

Наконец, заслуживает внимания то, что мы использовали связки сообщений для интернационализации. Попытайтесь переключить язык браузера на немецкий, и программа будет выглядеть так, как показано на рис. 2.4.

На этом наш пример приложения завершается. На рис. 2.5 показано, как выглядит структура каталогов приложения. Весь остальной код приведен в листингах 2.2–2.6.

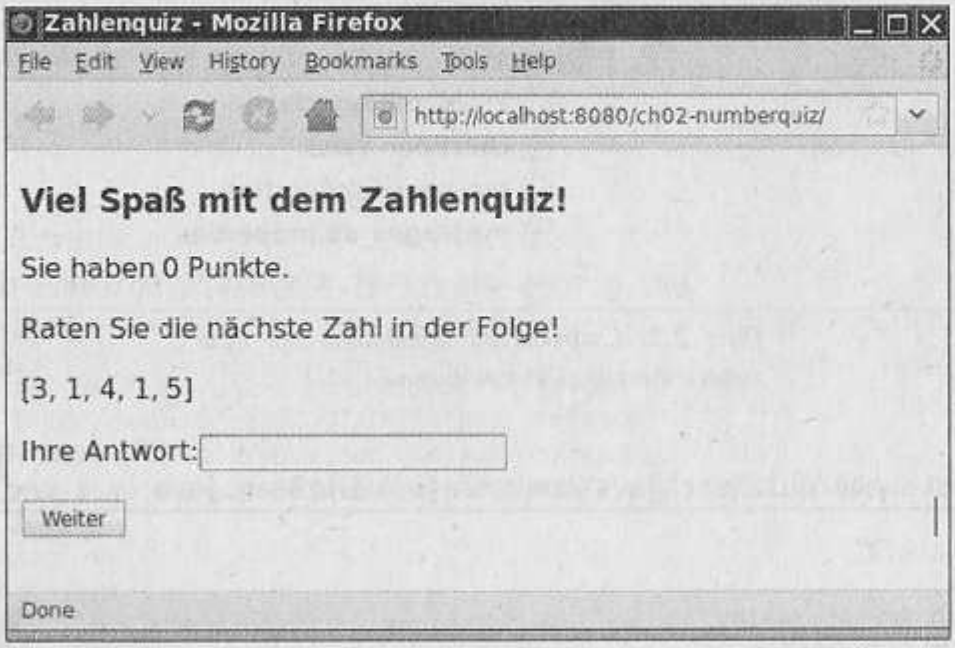


Рис. 2.4. Числовая викторина, в которой применяется немецкая локаль

Листинг 2.2. Файл `numberquiz\web\index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:f="http://java.sun.com/jsf/core"
6.       xmlns:h="http://java.sun.com/jsf/html">
```

```
    <h:head>
       <title>#{msgs.title}</title>
    </h:head>
    <h:body>
       <h:form>
          <h3>#{msgs.heading}</h3>
          <p>
             <h:outputFormat value="#{msgs.currentScore}">
                <f:param value="#{quizBean.score}"/>
             </h:outputFormat>
          </p>
          <p>#{msgs.guessNext}</p>
          <p>#{quizBean.current.sequence}</p>
          <p>
             #{msgs.answer}
             <h:inputText value="#{quizBean.answer}"/>
          </p>
          <p><h:commandButton value="#{msgs.next}"/></p>
       </h:form>
    </h:body>
</html>
```

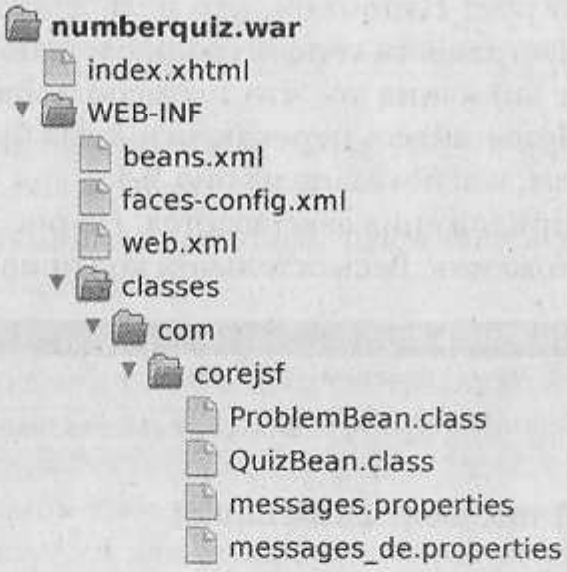


Рис. 2.5. Структура каталогов для примера с числовой викториной

Листинг 2.3. Файл `numberquiz\src\java\com\corejsf\QuizBean.java`

```
package com.corejsf;

import java.io.Serializable;
import java.util.ArrayList;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;

@Named // или @ManagedBean
@SessionScoped
public class QuizBean implements Serializable {
   private ArrayList<ProblemBean> problems = new ArrayList<ProblemBean>();
```

```
15.    private int currentIndex;
16.    private int score;
17.
18.    public QuizBean() {
19.       problems.add(
20.          new ProblemBean(new int[] { 3, 1, 4, 1, 5 }, 9)); // Цифры числа pi
21.       problems.add(
22.          new ProblemBean(new int[] { 1, 1, 2, 3, 5 }, 8)); // Числа Фибоначчи
23.       problems.add(
24.          new ProblemBean(new int[] { 1, 4, 9, 16, 25 }, 36)); // Квадраты
25.       problems.add(
26.          new ProblemBean(new int[] { 2, 3, 5, 7, 11 }, 13)); // Простые числа
27.       problems.add(
28.          new ProblemBean(new int[] { 1, 2, 4, 8, 16 }, 32)); // Степени 2
29.    }
30.
31.    public void setProblems(ArrayList<ProblemBean> newValue) {
32.       problems = newValue;
33.       currentIndex = 0;
34.       score = 0;
35.    }
36.
37.    public int getScore() { return score; }
38.
39.    public ProblemBean getCurrent() { return problems.get(currentIndex); }
40.
41.    public String getAnswer() { return ""; }
42.    public void setAnswer(String newValue) {
43.       try {
44.          int answer = Integer.parseInt(newValue.trim());
45.          if (getCurrent().getSolution() == answer) score++;
46.          currentIndex = (currentIndex + 1) % problems.size();
47.       }
48.       catch (NumberFormatException ex) {
49.       }
50.    }
51. }
52.
```

Листинг 2.4. Файл numberquiz\web\WEB-INF\faces-config.xml

```
1. <?xml version="1.0"?>
2. <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5.       http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
6.    version="2.0">
7.    <application>
8.       <locale-config>
9.          <default-locale>en</default-locale>
10.         <supported-locale>de</supported-locale>
11.      </locale-config>
12.      <resource-bundle>
13.         <base-name>com.corejsf.messages</base-name>
14.         <var>msgs</var>
15.      </resource-bundle>
16.   </application>
17. </faces-config>
```

Листинг 2.5. Файл numberquiz\src\java\com\corejsf\messages.properties

```
1. title=NumberQuiz
2. heading=Have fun with NumberQuiz!
3. currentScore=Your current score is {0}.
4. guessNext=Guess the next number in the sequence!
5. answer=Your answer:
6. next=Next
```

Листинг 2.6. Файл numberquiz\src\java\com\corejsf\messages_de.properties

```
1. title=Zahlenquiz
2. heading=Viel Spa\u00df mit dem Zahlenquiz!
3. currentScore=Sie haben {0,choice,0#0 Punkte|1#einen Punkt|2#{0} Punkte}.
4. guessNext=Raten Sie die n\u00e4chste Zahl in der Folge!
5. answer=Ihre Antwort:
6. next=Weiter
```

Области действия бинов

Для удобства программиста веб-приложений контейнер JSF предоставляет отдельные области действия, каждая из которых управляет отдельной таблицей привязок "имя–значение".

В этих областях действия обычно хранятся бины и другие объекты, которые должны быть доступны в различных компонентах веб-приложения.

При определении бина необходимо определить его область действия. Три области действия являются общими для бинов JSF и CDI:

- область действия сеанса;
- область действия запроса;
- область действия приложения.

В версии JSF 2.0 дополнительно предусмотрены область действия просмотра и пользовательские области действия. Эти области действия не поддерживаются в технологии CDI, вместо этого в ней предусмотрена намного более полезная область действия диалога.

В версии JSF 2.0 можно использовать аннотации, подобные приведенным ниже, для определения области действия бина.

```
@SessionScoped
@RequestScoped
@ApplicationScoped
```

Следует отметить, что эти аннотации находятся в пакете `javax.faces.bean` для управляемых бинов JSF и в пакете `javax.enterprise.context` для бинов CDI.

Эти области действия более подробно рассматриваются в следующих разделах.

Область действия сеанса

Напомним, что HTTP-протокол не поддерживает состояния. Браузер отправляет запрос серверу, сервер возвращает ответ, а затем ни браузер, ни сервер не берет на себя никаких обязательств хранить в памяти какие-либо сведения об этой транзакции. Такая простая организация работы вполне подходит для извлечения однозначно определяемых сведений, но не применима для серверных приложений. Например, в при-

ложении, предназначенном для электронной торговли, необходимо обеспечить, чтобы сервер запоминал содержимое электронной корзины для покупок.

Поэтому контейнеры сервлетов дополняют HTTP-протокол средствами отслеживания *сеансов*, под которыми подразумеваются повторные подключения одного и того же клиента. Для отслеживания сеансов существует несколько различных методов. Самый простой из них заключается в применении cookie-файлов, т.е. пар "имя–значение", которые сервер отправляет клиенту, чтобы получать их обратно при следующих запросах (рис. 2.6).

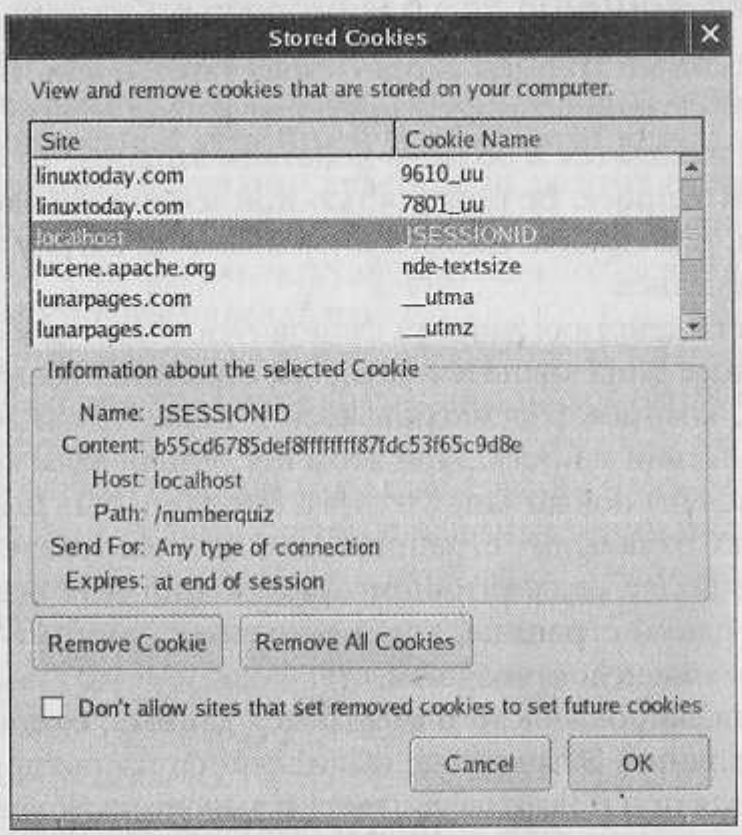


Рис. 2.6. Cookie-файл, отправленный приложением JSF

Если на клиенте не запрещены cookie-файлы, то сервер при каждом последующем запросе получает идентификатор сеанса.

Если же клиент не возвращает cookie-файлы, то на серверах приложений применяются так называемые *стратегии перехода на аварийный режим* наподобие перезаписи URL-адресов (URL rewriting). При перезаписи URL-адресов происходит добавление идентификатора сеанса к URL-адресу, который принимает примерно такой вид:

```
http://corejsf.com/login/faces/index.xhtml;jsessionid=b55cd6...d8e
```

На заметку! Чтобы увидеть этот способ в действии, настройте конфигурацию браузера таким образом, чтобы он отклонял cookie-файлы, полученные от сервера `localhost`, а затем перезапустите веб-приложение и отправьте страницу. Следующая страница будет иметь атрибут `jsessionid`.

Процесс отслеживания сеансов с помощью cookie-файлов полностью прозрачен для веб-разработчика, а если клиент не принимает cookie-файлы, то стандартные теги JSF обеспечивают перезапись URL-адресов автоматически.

Область действия сеанса сохраняется со времени установки сеанса до времени его завершения. Сеанс завершается после того, как веб-приложение вызывает метод `invalidate` применительно к объекту `HttpSession` или по истечении тайм-аута сеанса.

Например, бин UserBean может содержать информацию о пользователях, доступную на протяжении всего сеанса, а бин ShoppingCartBean может заполняться данными постепенно по мере поступления запросов, из которых складывается сеанс.

Следует помнить, что поддержка состояния сеанса в течение слишком долгого времени может стать причиной появления узкого места в работе и привести к снижению производительности. Сведения о том, какие альтернативные методы могут применяться для создания продолжительных сеансов, приведены в следующих разделах.

Область действия запроса

Область действия запроса (request scope) существует очень недолго, а точнее, со времени отправки HTTP-запроса и до момента отправки ответа обратно клиенту. Если управляемый бин находится в области действия запроса, то его новый экземпляр создается при каждом запросе. Возможностью применения области действия запроса следует пользоваться, если задача состоит в уменьшении затрат на выделение памяти для области действия сеанса.

Кроме того, область действия запроса становится очень удобной для организации работы, если все данные бина хранятся на одной странице. Например, в приложении для входа в систему, которое рассматривалось в главе 1, мы могли поместить бин UserBean в область действия запроса. При этом после передачи каждой страницы входа в систему создавался бы новый бин UserBean. Этот бин был бы доступен при подготовке к отображению начальной страницы, и имя пользователя отображалось бы правильно. Однако в более реалистичном приложении имя пользователя могло бы потребоваться не на одной странице, а на нескольких, поэтому применение области действия запроса было бы недостаточным.

В область действия запроса часто помещаются данные, относящиеся к сообщениям об ошибках и состоянии. Эти данные вычисляются при отправке клиенту данных формы и отображаются при подготовке ответа к выводу на экран. Аналогично с этим, тег f:loadBundle помещает в область действия запроса переменную связки, которая необходима только на этапе подготовки ответа к отображению в том же запросе.

При работе со сложными данными, такими как содержимое таблицы, область действия запроса может оказаться неприемлемой, поскольку при работе с ней приходится вновь формировать данные после каждого запроса.

Внимание! Только бины, находящиеся в области действия запроса, являются однопотоковыми, поэтому по самой своей сути, потокобезопасными. Это может показаться удивительным, но бины, находящиеся в области действия сеанса, однопотоковыми не являются. Например, ничто не мешает пользователю отправлять ответы одновременно из нескольких окон браузера. Каждый из этих ответов будет обрабатываться отдельным потоком запроса. Если необходимо обеспечить потокобезопасность, используя бины с областью действия сеанса, то невозможно обойтись без механизмов блокировки.

Область действия приложения

Область действия приложения (application scope) сохраняется на протяжении всего времени функционирования веб-приложения. Эта область действия используется одновременно всеми запросами и всеми сеансами. Управляемые бины помещаются в область действия приложения, если каждый отдельный бин должен использоваться совместно во всех экземплярах веб-приложения. В таком случае бин создается после первого запроса со стороны любого пользователя приложения и продолжает существовать до закрытия веб-приложения на сервере приложений.

Но если бин с областью действия приложения отмечен как eager, то должен быть создан до отображения даже первой страницы приложения. Для этого используется следующая аннотация:

```
@ManagedBean(eager=true)
```

Атрибут eager впервые введен в действие в версии JSF 2.0.

Область действия диалога CDI

Область действия диалога распространяется на набор взаимосвязанных страниц. Она позволяет обеспечить сохранение определенных данных до достижения какой-то конкретной цели, не вынуждая хранить эти данные на протяжении всего сеанса. Диалог закрепляется за конкретной страницей браузера или вкладкой. В каждом отдельном сеансе может проходить несколько диалогов на разных страницах. Обеспечение такой возможности является важным требованием с точки зрения практики. Пользователи часто открывают еще одну вкладку, чтобы можно было просматривать разные части конкретного приложения параллельно.

Чтобы убедиться в том, насколько важно обеспечивать поддержку нескольких диалогов, запустите приложение числовой викторины в двух окнах браузера и попытайтесь провести параллельно два раунда викторины. Вам явно не удастся это сделать. После щелчка на кнопке Next в любом окне станет ясно, что в сеансе имеется единственный экземпляр QuizBean, с одним счетом и одним текущим индексом.

Область действия диалога использовать несложно. Для этого достаточно придерживаться ряда правил.

- Использовать бин CDI, поскольку область действия диалога — это средство CDI, а не JSF.
- Использовать аннотацию @ConversationScoped.
- Добавить следующую переменную экземпляра:

  ```
  private @Inject Conversation conversation;
  ```

  Эта переменная экземпляра будет автоматически инициализирована вместе с объектом Conversation при создании бина.
- Вызвать метод conversation.begin(), чтобы перевести бин в более широкую область действия — из области действия запроса в область действия диалога.
- Вызвать метод conversation.end(), чтобы удалить бин из области действия диалога.

В качестве примера ниже показано, как можно применить область действия диалога в числовой викторине.

```
@Named
@ConversationScoped
public class QuizBean implements Serializable {
  @Inject Conversation conversation;
  ...
  public void setAnswer(String newValue) {
try {
if (currentIndex == 0) conversation.begin();
int answer = Integer.parseInt(newValue.trim());
if (getCurrent().getSolution() == answer) score++;
currentIndex = (currentIndex + 1) % problems.size();
if (currentIndex == 0) conversation.end();
}
```

```
 catch (NumberFormatException ex) {
}
 }
}
```

Снова попытайтесь открыть викторину в двух окнах браузера. Теперь в каждом з земпляре викторины поддерживается собственный счет и текущий индекс. Диалог з кончится после окончания викторины.

Область действия просмотра JSF 2.0

Области действия просмотра (view scope) были впервые введены в версии JSF 2. Бин в области действия просмотра сохраняется до тех пор, пока отображается одна та же страница JSF. (В спецификации JSF термин "view" применяется для обозначени страницы JSF, отсюда название "view scope".) После того как пользователь откро другую страницу, бин выходит из области действия просмотра.

Если в состав приложения входит страница, которая постоянно остается откр той, то можно поместить бины, хранящие данные для этой страницы, в область де ствия просмотра, что будет способствовать уменьшению затрат памяти для област действия сеанса. Это особенно важно для приложений Ajax.

Пользовательские области действия JSF 2.0

В конечном счете область действия — это просто карта, которая связывает имен с объектами. Основным признаком, по которому область действия одного типа отл чается от другого, является срок существования соответствующей карты. Срокам существования четырех стандартных областей действия JSF (сеанса, приложения просмотра и запроса) управляет реализация JSF. А с выходом версии JSF 2.0 появилас возможность создавать *пользовательские области действия* — карты, сроками существо вания которых можно управлять самостоятельно. Бин помещается в такую карту с по мощью следующей аннотации:

```
@CustomScoped("#{expr}")
```

Здесь используется выражение `#{expr}`, вычисление которого приводит к получе нию карты для области действия. При этом ответственность за удаление объектов и карты возлагается на приложение.

На заметку! Сами авторы относятся к возможности применения пользовательских областей действи скептически. В большинстве приложений лучшим выбором являются области действия диалога, поддер живаемые технологией CDI.

Настройка конфигурации бинов

В следующих разделах подробно описано, как настраивать конфигурацию бино с помощью аннотаций Java и тегов XML.

Встраивание бинов CDI CDI

В процессе проектирования приложения часто возникает необходимость связы вать бины друг с другом. Предположим, имеется бин `UserBean`, содержащий сведения о текущем пользователе, и какой-то бин с именем `EditBean`, для которого требуются эти

данные пользователя. Чтобы решить такую задачу, можно "встроить" экземпляр UserBean в EditBean. В простейшей форме этого подхода можно воспользоваться следующей аннотацией:

```
@Named
@SessionScoped
public class EditBean {
  @Inject private UserBean currentUser;
  ...
}
```

Вместо применения аннотации к полю можно также аннотировать метод задания свойства или конструктор.

При создании бина EditBean происходит поиск соответствующего экземпляра UserBean, в данном случае — бина UserBean, в текущем сеансе. Затем переменная экземпляра currentUser задается в соответствии с этим бином UserBean.

Такая простая организация работы превосходно подходит для несложных приложений, но спецификация CDI предоставляет более широкие возможности управления процессом встраивания. Используя аннотации и теги развертывания, можно встраивать разные бины для конкретных развертываний или тестирования. Для того чтобы читатель получил подробные сведения, рекомендуем обратиться к превосходной документации с примерами реализаций по адресу http://docs.jboss.org/weld/reference/1.0.0/en-US/html.

Встраивание управляемых бинов JSF 2.0

Для управляемых бинов JSF не предусмотрен богатый набор элементов управления встраиванием с учетом зависимостей, но имеется основной механизм — аннотация @ManagedProperty.

Предположим, в приложении применяется бин UserBean с именем user, который содержит сведения о текущем пользователе. Ниже показано, как можно его встроить в одно из полей другого бина.

```
@ManagedBean
@SessionScoped
public class EditBean implements Serializable {
@ManagedProperty(value="#{user}")
  private UserBean currentUser;
  public void setCurrentUser(UserBean newValue) { currentUser = newValue; }
...
}
```

Следует отметить, что аннотация относится к полю currentUser, но должен быть предусмотрен метод setCurrentUser. При создании экземпляра EditBean вычисляется значение выражения #{user} и результат передается методу setCurrentUser.

Внимание! У программистов, знакомых с аннотациями Java EE, может вызвать удивление тот факт, что аннотация @ManagedProperty применяется к полю, притом что фактически вызывается метод задания свойства. Если же имеется свойство, которое не соответствует полю, то придется создать фиктивное поле, чтобы можно было поместить в него аннотацию.

При задании какого-либо управляемого бина в качестве свойства другого необходимо убедиться в том, что их области действия совместимы. Область действия свойства не должна быть более узкой, чем область действия содержащего бина. В табл. 2.1 перечислены все допустимые комбинации.

Таблица 2.1. Совместимые области действия бинов

При определении бина из этой области действия...	... можно задать в качестве его свойств бины из следующих областей действия
отсутствует	отсутствует
приложение	отсутствует, приложение
сеанс	отсутствует, приложение, сеанс
просмотр	отсутствует, приложение, сеанс, просмотр
запрос	отсутствует, приложение, сеанс, просмотр, запрос

Аннотации для жизненного цикла бина

С помощью аннотаций `@PostConstruct` и `@PreDestroy` можно определить методы бина, автоматически вызываемые сразу после создания бина и непосредственно перед выходом бина из области действия:

```
public class MyBean {
 @PostConstruct
 public void initialize() {
// Код инициализации
}
 @PreDestroy
 public void shutdown() {
// Код завершения
}
// Другие методы бина
}
```

Аннотация `@PostConstruct` может применяться для бинов, которые должны собирать данные для отображения на странице. Еще одно распространенное направление использования — бины, которые устанавливают соединения с какими-либо внешними ресурсами, такими как базы данных.

На заметку! Аннотации `@PreDestroy` и `@PostDestruct` могут применяться и для управляемых бинов JSF, и для бинов CDI.

Настройка конфигурации управляемых бинов с помощью XML

До выхода версии JSF 2.0 настройка конфигурации всех бинов должна была осуществляться с помощью XML. Теперь разработчик может выбирать между аннотациями и конфигурацией XML. Описание конфигурации на языке XML является более подробным, но может оказаться полезным, если есть необходимость настраивать конфигурацию бинов во время развертывания. Читатель может без ущерба для дальнейшего понимания пропустить этот материал, если его не интересует настройка конфигурации бинов с помощью XML.

Для размещения сведений о конфигурации в коде XML могут применяться файлы, описанные ниже.

- Файл `WEB-INF/faces-config.xml`.
- Файлы, имеющие имя `faces-config.xml` или имя с суффиксом `.faces-config.xml` и находящиеся в каталоге `META-INF` файла JAR. (Такой механизм применяется при предоставлении компонентов многократного использования в файле JAR.)

- Файлы, перечисленные в параметре инициализации javax.faces.CONFIG_FILES в файле WEB-INF/web.xml, например:

```
<web-app>
  <context-param>
 <param-name>javax.faces.CONFIG_FILES</param-name>
 <param-value>WEB-INF/navigation.xml,WEB-INF/managedbeans.xml</param-value>
  </context-param>
  ...
</web-app>
```

Такой механизм более привлекателен для инструментальных средств создания программ, поскольку он позволяет отдельно определять способы навигации, бины и т.д.

Определение бинов

Для определения управляемого бина в файле конфигурации XML используется элемент managed-bean. Например:

```
<faces-config>
  <managed-bean>
<managed-bean-name>user</managed-bean-name>
<managed-bean-class>com.corejsf.UserBean</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

В качестве области действия может быть указано request (запрос), view (просмотр), session (сеанс), application (приложение), none (отсутствует) или задано выражение значения, вычисление которого приводит к получению пользовательского отображения области действия. (См. раздел "Пользовательские области действия" на стр. 64.)

Область действия none обозначает бин, который не определен ни в одном отображении области действия. Каждый раз, когда какой-либо объект из области действия none запрашивается в выражении значения, создается новый объект. Это может оказаться удобным для определения бинов, которые заданы как свойства других бинов.

Бин с областью действия приложения может быть обозначен как eager (см. стр. 63) с помощью следующего атрибута:

```
<managed-bean eager="true">
```

Задание значений свойств

Определение в конфигурации значений свойств может осуществляться с помощью XML, аналогично тому, как используется аннотация @Managed-Property. Ниже показано, как выполнить настройку конфигурации экземпляра UserBean.

```
<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
<property-name>name</property-name>
<value>troosevelt</value>
  </managed-property>
  <managed-property>
<property-name>password</property-name>
<value>jabberwock</value>
  </managed-property>
</managed-bean>
```

Когда в процессе работы приложения впервые осуществляется поиск бина user, этот бин создается с помощью конструктора UserBean(). После этого выполняются методы setName и setPassword.

Для инициализации свойств со значением null применяется элемент null-value. Например:

```
<managed-property>
  <property-name>password</property-name>
  <null-value/>
</managed-property>
```

В качестве элементов value могут применяться значения выражения:

```
<managed-bean>
  <managed-bean-name>editBean</managed-bean-name>
  <managed-bean-class>com.corejsf.EditBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
<property-name>user</property-name>
<value>#{user}</value
  </managed-property>
</managed-bean>
```

В данном примере вызывается метод setUser со значением выражения #{user}.

Инициализация списков и карт

Для инициализации значений типа List (список) или Map (карта) применяется специальный синтаксис. Ниже показан пример списка.

```
<list-entries>
  <value-class>java.lang.Integer</value.class>
  <value>3</value>
  <value>1</value>
  <value>4</value>
  <value>1</value>
  <value>5</value>
</list-entries>
```

В данном случае используется тип оболочки java.lang.Integer, поскольку объект List не может содержать значения примитивных типов.

Список может содержать смесь элементов value и null-value. Элемент value-class является необязательным. Если этот элемент опущен, создается список объектов java.lang.String.

С картой дело обстоит сложнее. Вначале необходимо задать необязательные элементы key-class и value-class (опять-таки с помощью применяемых по умолчанию объектов java.lang.String), а затем предусматривается последовательность элементов map-entry, каждый из которых имеет элемент key со следующим за ним элементом value или null-value.

Ниже приведен пример.

```
<map-entries>
  <key-class>java.lang.Integer</key-class>
  <map-entry>
<key>1</key>
<value>George Washington</value>
  </map-entry>
  <map-entry>
<key>3</key>
<value>Thomas Jefferson</value>
  </map-entry>
```

```
  <map-entry>
<key>16</key>
<value>Abraham Lincoln</value>
  </map-entry>
  <map-entry>
<key>26</key>
<value>Theodore Roosevelt</value>
  </map-entry>
</map-entries>
```

Элементы `list-entries` и `map-entries` можно использовать для инициализации свойства `managed-bean` или `managed-property`, при условии, что бин или свойство имеет тип `List` или `Map`.

На рис. 2.7 показана синтаксическая диаграмма элемента `managed-bean` и всех его дочерних элементов. Следуя по стрелкам, можно видеть, какие конструкции допустимы в элементе `managed-bean`. Например, во втором графе показано, что элемент `managed-property` начинается с нуля или большего количества элементов `description`, за которыми следуют от нуля или более элементов `display-name`, нуль или более пиктограмм, за ними находится обязательный элемент `property-name`, необязательный элемент `property-class` и одно и только одно значение из элементов `value`, `null-value`, `map-entries` или `list-entries`.

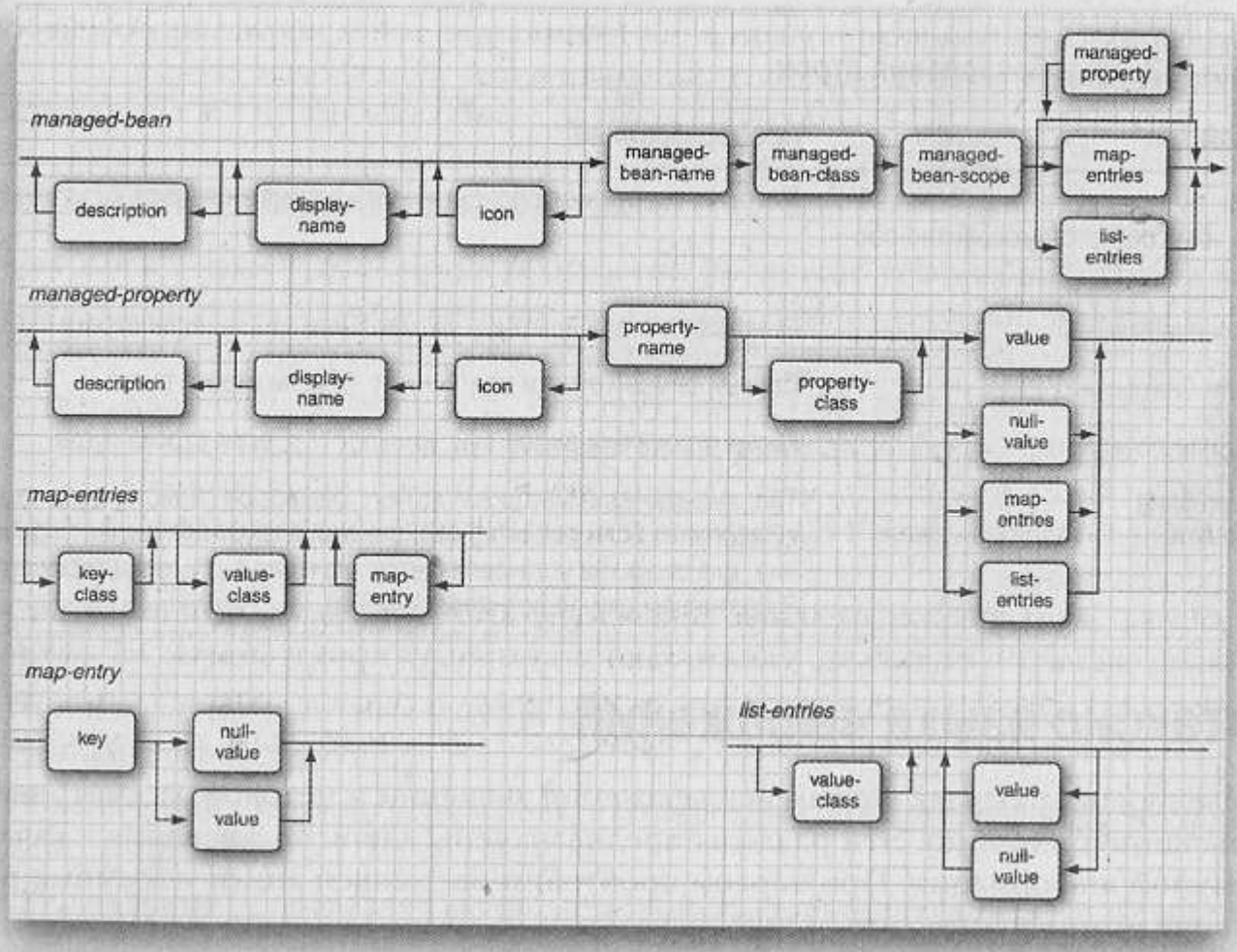


Рис. 2.7. Синтаксическая диаграмма элемента `managed-bean`

Преобразование строк

Для задания значений свойств и элементов списков или карт применяется элемент `value`, содержащий строку. Вложенная строка должна быть преобразована в тип свой-

ства или элемента. В случае простых типов это преобразование выполняется легко. Например, можно определить такую строку, как `10` или `true`, и преобразовать ее в число или булево значение.

Начиная с версии 1.2 поддерживаются также и значения перечислимых типов. В этом случае преобразование выполняется путем вызова метода `Enum.valueOf (propertyClass, valueText)`.

Что касается других типов свойств, то реализация JSF пытается найти соответствующий класс редактора свойств, `PropertyEditor`. Если существует требуемый класс редактора свойств, вызывается его метод `setAsText` для преобразования строк в значения свойств. Задача определения редактора свойства является довольно затруднительной, поэтому рекомендуем заинтересованному читателю обратиться к книге Cay Horstmann and Gary Cornell, *Core Java*™, 8th ed., Santa Clara, CA: Sun Microsystems Press/Prentice Hall, 2008, том 2, глава 8.

В табл. 2.2 приведен краткий перечень всех этих правил преобразования. Они идентичны правилам действия `jsp:setProperty`, перечисленным в спецификации JSP.

Внимание! Правила преобразования строки являются довольно ограничительными. Например, если применяется свойство типа `URL`, то нельзя просто указывать URL-адрес в виде строки, даже несмотря на наличие конструктора `URL(String)`. Вместо этого необходимо предоставить редактор свойств для типа `URL` или заново реализовать этот тип свойства как `String`.

Таблица 2.2. Преобразования строк

Целевой тип	Преобразование
Тип `int`, `byte`, `short`, `long`, `float`, `double` или соответствующий тип оболочки	Метод `valueOf` типа оболочки или `0`, если строка пуста
`boolean` или `Boolean`	Результат метода `Boolean.valueOf` или `false`, если строка пуста
`char` или `Character`	Первый символ строки или `(char) 0`, если строка пуста
`String` или `Object`	Копия строки или вызов `new String("")`, если строка пуста
Свойство бина	Тип, вызывающий метод `setAsText` редактора свойств, если таковой существует. Если редактор свойств отсутствует или выдает исключение, для свойства устанавливается значение `null`, если строка пуста. В противном случае возникает ошибка

Синтаксис языка выражений

В следующих разделах синтаксис выражений значений и методов рассматривается чрезвычайно подробно. Эти разделы предназначены для использования в качестве справочной информации. При первом прочтении настоящей книги указанные разделы вполне можно пропустить.

Режимы лево- и правосторонних выражений

Начнем с выражения, имеющего вид `a.b`. На данный момент предположим, что объект, на который ссылается компонент `a`, уже известен. Если `a` представляет собой массив, список или карту, то действуют специальные правила (подробнее об этом — ниже, в разделе "Использование квадратных скобок" на стр. 71). Если компонент `a`

представляет собой какой-либо другой объект, то компонент b должен представлять собой имя свойства объекта a. Точный смысл выражения a.b зависит от того, используется ли выражение в лево- или правостороннем режиме.

Эта терминология применяется в теории языков программирования для указания на то, что выражение, находящееся справа от оператора присваивания, трактуется иначе, чем выражение, находящееся слева от него.

Рассмотрим следующую операцию присваивания:

```
left = right;
```

Компилятор генерирует различный код для выражения слева и выражения справа. Выражение справа вычисляется в правостороннем режиме, что приводит к получению значения, а выражение слева — в левостороннем режиме и сохраняет значение в определенном месте.

То же самое происходит и в случае использования выражения значения в компоненте пользовательского интерфейса:

```
<h:inputText value="#{user.name}"/>
```

При подготовке текстового поля к отображению выражение user.name вычисляется в правостороннем режиме, после чего вызывается метод getName. Во время декодирования то же самое выражение вычисляется в левостороннем режиме и вызывается метод setName.

Вообще говоря, вычисление выражения a.b в правостороннем режиме осуществляется путем вызова метода получения свойства, тогда как в левостороннем режиме применительно к a.b вызывается метод задания свойства.

Использование квадратных скобок

Так же как и в языке JavaScript, в технологии JSF вместо уточняющей записи через точку разрешено использовать квадратные скобки. Иными словами, все три следующие выражения будут иметь одно и то же значение:

```
a.b
a["b"]
a['b']
```

Например, выражения user.password, user["password"] и user['password'] являются эквивалентными.

Для чего кому-либо может потребоваться использовать выражение user["password"], когда слова user.password ввести гораздо легче? Для этого существует несколько причин.

- При получении доступа к массиву или карте обозначение [] является более доступным для восприятия.
- Обозначение [] можно использовать со строками, которые содержат точки или тире, например msgs["error.password"].
- Обозначение [] позволяет вычислять свойства динамически: a[b.propname].

Совет. Если требуется ввести в атрибуты разграничители в виде двойных кавычек, то в выражениях значения следует использовать одинарные кавычки: value="#{user['password']}". Еще один вариант состоит в том, что можно поменять одинарные и двойные кавычки местами: value='#{user["password"]}'.

Выражения карт и списков

Возможности языка выражений значения выходят за рамки доступа к свойствам бинов. Например, представим, что `m` — это объект какого-либо класса, который реализует интерфейс `Map`. В таком случае выражение `m["key"]` (или эквивалентное ему `m.key`) представляет собой привязку к ассоциированному с ним значению. В правостороннем режиме выбирается значение:

```
m.get("key")
```

В левостороннем режиме выполняется инструкция:

```
m.put("key", right);
```

где `right` — правостороннее значение, которое присваивается выражению `m.key`.

Можно также получить доступ к значению любого объекта класса, который реализует интерфейс `List` (такому как `ArrayList`). Для этого требуется просто указать целочисленный индекс позиции этого объекта в списке. Например, `a[i]` (или, если угодно, `a.i`) привязывает `i`-й элемент списка `a`. Здесь `i` может быть либо целым числом, либо строкой, допускающей преобразование в целое число. То же правило распространяется и на типы массивов. Отсчет значений индексов, как обычно, начинается с нуля.

Итоговые сведения об этих правилах вычисления приведены в табл. 2.3.

Таблица 2.3. Вычисление выражения значения `a.b`

Тип `a`	Тип `b`	Левосторонний режим	Правосторонний режим
`null`	Любой	Ошибка	`null`
Любой	`null`	Ошибка	`null`
Карта	Любой	`a.put(b, right)`	`a.get(b)`
Список	Преобразуемый в `int`	`a.set(b, right)`	`a.get(b)`
Массив	Преобразуемый в `int`	`a[b] = right`	`a[b]`
Бин	Любой	Вызвать метод задания свойства с именем `b.toString()`	Вызвать метод получения свойства с именем `b.toString()`

Внимание! К сожалению, выражения значения не могут применяться к индексированным свойствам. Если `p` — индексированное свойство бина `b`, а `i` — целое число, то `b.p[i]` не получает доступ к `i`-му значению свойства. Появление в программе такой конструкции рассматривается просто как синтаксическая ошибка. Данный недостаток унаследован в технологии и перекочевал в JSF из языка выражений JSP.

Вызов методов и функций JSF 2.0

Начиная с версии JSF 2.0 появилась возможность вызывать методы в выражениях значения. Для этого достаточно указать имя метода и параметры. Например, если бин `stockQuote` имеет метод `double price(String)`, то можно использовать следующее выражение:

```
#{stockQuote.price("ORCL")}
```

Перегруженные методы не поддерживаются. Бин должен иметь единственный метод с конкретным именем. Можно также вызывать полезные функции из библиотеки функций JSTL (табл. 2.4). При использовании таких функций следует помнить, что

нужно добавлять `xmlns:fn="http://java.sun.com/jsp/jstl/functions"` к элементу `html` на конкретной странице.

Таблица 2.4. Функции JSTL

Функция	Описание
`fn:contains(str, substr)`	Возвращает `true`, если строка `str` содержит подстроку `substr`
`fn:containsIgnoreCase(str, substr)`	Возвращает `true`, если строка `str` содержит подстроку `substr`; при этом регистр символов игнорируется
`fn:startsWith(str, substr)`	Возвращает `true`, если строка `str` начинается с подстроки `substr`
`fn:endsWith(str, substr)`	Возвращает `true`, если строка `str` заканчивается подстрокой `substr`
`fn:length(str)`	Возвращает длину строки `str`
`fn:indexOf(str, substr)`	Возвращает индекс первого вхождения подстроки `substr` в строку `str` или `-1`, если подстрока `substr` не найдена
`fn:join(strArray, separator)`	Соединяет строки из заданного строкового массива, размещая между ними разделительную строку
`fn:split(str, separator)`	Разбивает строку на массив строк, удаляя все вхождения разделителя
`fn:substring(str, start, pastEnd)`	Возвращает подстроку строки `str`, начиная с позиции `start` и заканчивая позицией `pastEnd - 1`
`fn:substringAfter(str, separator)`	Возвращает подстроку строки `str` после первого вхождения разделителя
`fn:substringBefore(str, separator)`	Возвращает подстроку строки `str` перед первым вхождением разделителя
`fn:replace(str, from, to)`	Возвращает результат замены всех вхождений подстроки `from` в строке `str` подстрокой `to`
`fn:toLowerCase(str)`	Возвращает строку `str`, приведенную к нижнему регистру
`fn:toUpperCase(str)`	Возвращает строку `str`, приведенную к верхнему регистру
`fn:trim(str)`	Возвращает строку `str` с удаленными начальными и конечными пробельными символами
`fn:escapeXml(str)`	Возвращает строку `str` с символами <, > и &, экранированными как сущности XML

Разрешение первого члена

Выше было описано, как происходит разрешение выражений в форме `a.b`. Те же правила можно применять и к выражениям наподобие `a.b.c.d` (или, разумеется, `a['b'].c["d"]`) столько раз, сколько потребуется. Тем не менее необходимо дополнительно рассмотреть смысл начального члена `a`.

В примерах, приведенных выше, начальный член ссылался либо на бин, либо на карту связки сообщений. Такие ситуации действительно являются наиболее распространенными. Но с помощью начального члена могут быть также заданы другие имена.

Предусмотрен целый ряд предопределенных объектов, называемых в спецификации JSF неявными объектами. В табл. 2.4 приведен их полный перечень. Например,

```
header['User-Agent']
```

представляет собой значение параметра User-Agent HTTP-запроса, который идентифицирует браузер пользователя.

Если начальный член не является одним из предопределенных объектов, то реализация JSF пытается выполнить его поиск в области действия запроса, просмотра, сеанса и приложения (в указанном порядке). В частности, в этих картах области действия находятся все экземпляры управляемых бинов.

Если указанный процесс поиска все еще не приносит успеха, реализация JSF предпринимает попытки создать управляемый бин или связку ресурсов с указанным именем.

Предусмотрена также возможность определять в приложениях специализированные "модули разрешения", которые распознают дополнительные имена. Эта тема будет кратко рассматриваться в разделе "Расширение языка выражений JSF" главы 13 на стр. 511.

Таблица 2.5. Предопределенные объекты в языке выражений

Имя переменной	Значение
header	Карта (Map) параметров HTTP-заголовка, содержащая только первое значение для каждого имени
headerValues	Карта (Map) параметров HTTP-заголовка, применение которой приводит к получению массива String[] со всеми значениями для данного имени
param	Карта (Map) параметров HTTP-запроса, содержащая только первое значение для каждого имени
paramValues	Карта (Map) параметров HTTP-запроса, применение которой приводит к получению массива String[] со всеми значениями для данного имени
cookie	Карта (Map) имен cookie-файлов и значений текущего запроса
initParam	Карта (Map) параметров инициализации данного веб-приложения. Сведения о параметрах инициализации приведены в главе 12
requestScope	Карта (Map) всех атрибутов области действия запроса
viewScope JSF 2.0	Карта (Map) всех атрибутов области действия просмотра
sessionScope	Карта (Map) всех атрибутов области действия сеанса
applicationScope	Карта (Map) всех атрибутов области действия приложения
flash JSF 2.0	Карта (Map), предназначенная для перенаправления объектов в следующую область действия просмотра. См. главу 3
resource JSF 2.0	Карта (Map) ресурсов приложения. Ресурсы рассматриваются в главе 4
facesContext	Экземпляр класса FacesContext для данного запроса. Сведения об этом классе приведены в главе 7
view	Экземпляр класса UIViewRoot для данного запроса. Сведения об этом классе приведены в главе 8
component JSF 2.0	Текущий компонент (подробнее о нем — в главе 9)
cc JSF 2.0	Текущий составной компонент (также рассматривается в главе 9)

Рассмотрим, например, следующее выражение:

```
#{user.password}
```

Член user не является одним из предопределенных объектов. Встретившись впервые, он не представляет собой имя атрибута в области действия запроса, просмотра, сеанса или приложения.

Поэтому реализация JSF определяет местонахождение управляемого бина user и вызывает конструктор соответствующего класса без параметров, а затем добавляет требуемую ассоциацию в карту sessionScope. Наконец, созданный объект возвращается в качестве результата поиска.

Если во время того же сеанса снова возникает необходимость в разрешении члена user, соответствующий объект обнаруживается уже в области действия сеанса.

Сложные выражения

В выражениях значений разрешается использовать ограниченный набор операторов.

- Арифметические операторы: +, -, *, /, %. Последние два оператора имеют эквиваленты в виде ключевых слов div и mod.
- Операторы сравнения <, <=, >, >=, ==, != и их эквиваленты в виде ключевых слов lt, le, gt, ge, eq, ne. Первые четыре символьных варианта среди перечисленных операторов при работе с кодом XML должны обязательно заменяться ключевыми словами.
- Логические операторы &&, ||, ! и их эквиваленты в виде ключевых слов and, or, not. Первый символьный вариант среди перечисленных операторов при работе с кодом XML должен обязательно заменяться ключевым словом.
- Операция empty. Выражение empty a является истинным, если a имеет значение null, представляет собой массив или значение типа String с нулевой длиной или объект Collection или Map с нулевым размером.
- Трехзначный ?: оператор выбора.

Для определения приоритета операций применяются такие же правила, как в языке Java. Например, оператор empty имеет такой же приоритет, как и унарные операторы - и !.

Вообще говоря, следует избегать большого объема вычислений с помощью выражений, находящихся на веб-страницах, поскольку при этом нарушается принцип разделения уровня представления и уровня бизнес-логики. Однако иногда уровень представления может быть усовершенствован с помощью выражений с операторами. Например, предположим, что нужно обеспечить сокрытие некоторого компонента путем присваивания значения true свойству hide бина. Скрыть компонент можно путем задания для его атрибута rendered значения false, а для инвертирования значения бина требуется использование оператора ! (или not):

```
<h:inputText rendered="#{!bean.hide}" ... />
```

Наконец, можно просто сцеплять строки и выражения значений, размещая их рядом. Рассмотрим следующий пример:

```
<h:commandButton value="#{msgs.clickHere}, #{user.name}!"/>
```

С помощью этой инструкции происходит соединение четырех строк: строки, возвращенной выражением #{messages.greeting}; строки, состоящей из запятой и пробела; строки, возвращенной выражением #{user.name}, и строки, состоящей из восклицательного знака.

Выше были показаны в действии все правила, которые применяются для разрешения выражений значений. Безусловно, на практике большинство выражений имеет вид `#{bean.property}`. Те читатели, которые лишь бегло просмотрели данный раздел, всегда смогут к нему вернуться, если им придется столкнуться с более сложными выражениями.

Выражения методов

Выражение метода (method expression) обозначает объект и метод, который может к нему применяться.

В качестве примера ниже приведен типичный вариант использования выражения метода.

```
<h:commandButton action="#{user.checkPassword}"/>
```

Здесь предполагается, что `user` — значение типа `UserBean`, а `checkPassword` — метод этого класса. Выражение метода является удобным способом описания вызова метода, который должен быть выполнен в какое-то время в будущем.

При вычислении этого выражения метод применятся к объекту.

В данном примере компонент, представляющий командную кнопку, вызывает метод `user.checkPassword()` и передает возвращенную строку обработчику навигации.

Синтаксические правила для выражений методов аналогичны таковым для выражений значений. Все компоненты, кроме последнего, используются для определения объекта. Последним компонентом должно быть имя метода, который может применяться к этому объекту. В табл. 2.6 показаны атрибуты, для которых требуются выражения методов.

Таблица 2.6. Атрибуты выражений методов

Тег	Атрибут	Тип выражения метода	См. главу
Кнопки и связи	`action`	`String action()`	3
	`actionListener`	`void listener(ActionEvent)`	8
Входные компоненты	`valueChangeListener`	`void listener(ValueChangeEvent)`	9
	`validator`	`void validator(FacesContext, UIComponent, Object)`	7
`f:event`	`listener`	`void listener(ComponentSystemEvent)`	8
`f:ajax`	`listener`	`void listener(AjaxBehaviorEvent)`	10

На заметку! Синтаксис языка выражений для типов выражений методов (см. табл. 2.6) содержит не только параметры метода и возвращаемые типы, но и имя метода, как в примере `void listener(ActionEvent)`. Эти имена предназначены для документирования назначения метода.

Параметры выражения метода JSF 2.0

Начиная с версии JSF 2.0 в выражениях методов можно применять значения параметров. Такая возможность становится удобной, если есть необходимость предоставлять параметры для действий кнопок и связей. Например:

```
<h:commandButton value="Previous" action="#{formBean.move(-1)}"/>
<h:commandButton value="Next" action="#{formBean.move(1)}"/>
```

Затем метод действия должен быть объявлен с параметром:

```
public class FormBean {
  ...
  public String move(int amount) { ... }
}
```

При обработке ссылки на метод параметры вычисляются и передаются методу.

Внимание! Параметры метода не могут применяться в версии Tomcat 6, в которой предусмотрены только JAR-файлы JSF. Язык выражений (expression language — EL) определен не в спецификации JSF, а в спецификации JSR 245 (JavaServer Pages). Параметры метода относятся к средствам EL 2.2, которые не поддерживаются в Tomcat 6. Для того чтобы иметь возможность использовать средства EL 2.2, загрузите JAR-файлы с сайта `http://uel.dev.java.net`, добавьте их в каталог `WEB-INF/lib` конкретного веб-приложения, а также добавьте следующее в применяемый файл `web.xml`:

```
<context-param>
        <param-name>com.sun.faces.expressionFactory</param-name>
        <param-value>com.sun.el.ExpressionFactoryImpl</param-value>
      </context-param>
```

Резюме

На этом наше обсуждение управляемых бинов заканчивается. Мы хотим заверить читателей, что бо́льшая часть отмеченных здесь технических трудностей не препятствует достижению успехов при разработке для JSF. Начните с использования связок сообщений и именованных бинов с областью действия сеанса и возвращайтесь к этой главе по мере того, как будет возникать необходимость в применении более развитых средств.

В следующей главе будет показано, как в приложениях JSF осуществляется переход между страницами.

Навигация

В этой главе...

- Статическая навигация
- Динамическая навигация
- Перенаправление
- Навигация с поддержкой метода REST и применение URL, обеспечивающих формирование закладок JSF 2.0
- Расширенные правила навигации

Глава 3

В этой краткой главе речь пойдет о настройке конфигурации механизма навигации в веб-приложении. В частности, будет показано, как в приложении обеспечить переход с одной страницы на другую в зависимости от действий пользователя и выходных данных, получаемых в результате решений, которые принимаются на уровне бизнес-логики.

Статическая навигация

Прежде всего рассмотрим, что происходит, когда пользователь приложения заполняет веб-страницу. При этом пользователь может вводить данные в текстовые поля, выбирать переключатели или выделять записи в списках.

Все эти операции редактирования осуществляются в браузере пользователя. После того как пользователь щелкнет на кнопке, отвечающей за отправку данных формы, все внесенные им изменения передаются на сервер.

Затем веб-приложение анализирует входные данные пользователя и принимает решение о том, какую из JSF-страниц ему следует использовать для подготовки ответа к отображению. За выбор этой очередной JSF-страницы отвечает *обработчик навигации* (navigation handler).

В простом веб-приложении навигация является статической. Это означает, что щелчок на определенной кнопке всегда приводит к выбору для подготовки к отображению конкретной (заранее заданной) JSF-страницы. В разделе "Простой пример" главы 1 на стр. 20 показан наиболее простой механизм обеспечения статической навигации между страницами JSF.

К каждой кнопке добавляется атрибут `action`:

```
<h:commandButton label="Login" action="welcome"/>
```

На заметку! Как будет показано в главе 4, действия по навигации могут быть также закреплены за гиперссылками.

Значение атрибута `action` называется *результатом*. Ниже, в разделе "Отображение результатов в идентификаторы представлений" на стр. 81 указано, что результат может быть, по желанию, сопоставлен идентификатору представления. (В спецификации JSF *представлением* называется страница JSF.)

Если подобное отображение для какого-то конкретного результата не предусмотрено, то результат преобразуется в идентификатор представления с использованием следующих шагов.

1. Если в результате не предусмотрено расширение имени файла, то добавляется расширение текущего представления.
2. Если результат не начинается со знака /, то задается префикс в виде пути текущего представления.

В качестве примера можно указать, что результат с именем `welcome` в представлении `/index.xhtml` приводит к получению идентификатора целевого представления `/welcome.xhtml`.

На заметку! Начиная с версии JSF 2.0 применение отображений из результатов в идентификаторы представлений является необязательным. До выхода версии JSF 2.0 необходимо было задавать явные правила навигации для каждого результата.

Динамическая навигация

В большинстве веб-приложений навигация не является статической. Поток страниц зависит не только от того, на какой кнопке выполнен щелчок, но и от того, какие входные данные предоставлены. Например, отправка страницы входа в систему может иметь два результата: успешный и неудачный. Результат вычисляется в коде и в данном случае указывает, являются ли допустимыми имя пользователя и пароль.

Для обеспечения динамической навигации кнопка отправки должна иметь *выражение метода* (method expression), такое, как

```
<h:commandButton label="Login" action="#{loginController.verifyUser}"/>
```

В данном примере `loginController` ссылается на бин некоторого класса, а этот класс должен иметь метод `verifyUser`.

Выражение метода в атрибуте `action` не имеет параметров. Возвращаемый им тип может быть любым. Возвращаемое значение преобразуется в строку путем вызова метода `toString`.

На заметку! В версии JSF 1.1 метод действия должен был иметь только возвращаемый тип `String`. А начиная с версии 1.2 разрешается использовать любой возвращаемый тип. Удобной альтернативой, в частности, являются перечисления, поскольку это позволяет использовать компилятор для перехвата опечаток в именах действий.

Ниже показан пример метода действия.

```
String verifyUser() {
  if (...)
return "success";
  else
return "failure";
}
```

На заметку! Метод действия может также возвращать и значение `null`, указывающее, что должно быть снова отображено то же представление. В этом случае сохраняется область действия представления, которая рассматривалась в главе 2. Получение любого результата, отличного от `null`, приводит к очистке области действия представления, даже если в конечном итоге должно быть получено представление, совпадающее с текущим.

Этот метод возвращает строку результата, такую как "success" или "failure", используемую для определения следующего представления.

Ниже вкратце перечислены шаги, которые выполняются каждый раз, когда пользователь щелкает на командной кнопке, в атрибуте action которой указано выражение метода.

1. Извлечение указанного бина.
2. Вызов метода, указанного в ссылке, и возврат строки результата.
3. Преобразование строки результата в идентификатор представления.
4. Отображение страницы, соответствующей идентификатору представления.

Таким образом, для реализации поведения с ветвлением предоставляется ссылка на метод в соответствующем классе бина. Существует много вариантов размещения этого метода. Наилучший подход состоит в том, чтобы найти класс, имеющий все данные, необходимые для принятия решения.

Отображение результатов в идентификаторы представлений

Одна из ключевых целей проектирования на основе технологии JSF состоит в отделении уровня представления от уровня бизнес-логики. Если решения по навигации принимаются динамически, то в коде, в котором вычисляется результат, не требуются точные сведения об именах веб-страниц. Технология JSF предоставляет механизм отображения логических результатов, таких как успешное или неудачное завершение, в действительные веб-страницы.

Это достигается путем добавления записей navigation-rule (правило навигации) в файл faces-config.xml. Ниже приведен типичный пример.

```
<navigation-rule>
  <from-view-id>/index.xhtml</from-view-id>
  <navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/welcome.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Согласно данному правилу, результат success должен приводить к переходу на страницу /welcome.xhtml, если он возникает во время работы со страницей /index.xhtml.

На заметку! Строки идентификаторов представлений начинаются со знака /. Если используется отображение расширения (такое как суффикс .faces), то расширение должно согласовываться с расширением имени файла (наподобие .xhtml), а не с расширением URL.

Тщательный выбор строк результата позволяет собирать многочисленные правила навигации в группы. Например, может оказаться так, что в приложении на многих страницах имеются кнопки с действием logout. С помощью одного-единственного правила можно сделать так, чтобы щелчок на любой из них приводил к переходу на страницу loggedOut.xhtml:

```
<navigation-rule>
  <navigation-case>
<from-outcome>logout</from-outcome>
<to-view-id>/loggedOut.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Это правило будет действовать для всех страниц, поскольку не был задан элемент `from-view-id`.

Правила навигации с одинаковым элементом `from-view-id` можно объединять:

```
<navigation-rule>
  <from-view-id>/index.xhtml</from-view-id>
  <navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/welcome.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
<from-outcome>failure</from-outcome>
<to-view-id>/newuser.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

На заметку! В простых приложениях необходимость в использовании правил навигации может не возникнуть. А по мере усложнения приложений появляется смысл использовать логические результаты в управляемых бинах наряду с правилами навигации для отображения результатов в целевые представления.

Приложение JavaQuiz

В настоящем разделе показано, как ввести в действие средства навигации в программе, которая предъявляет пользователю ряд вопросов викторины (рис. 3.1).

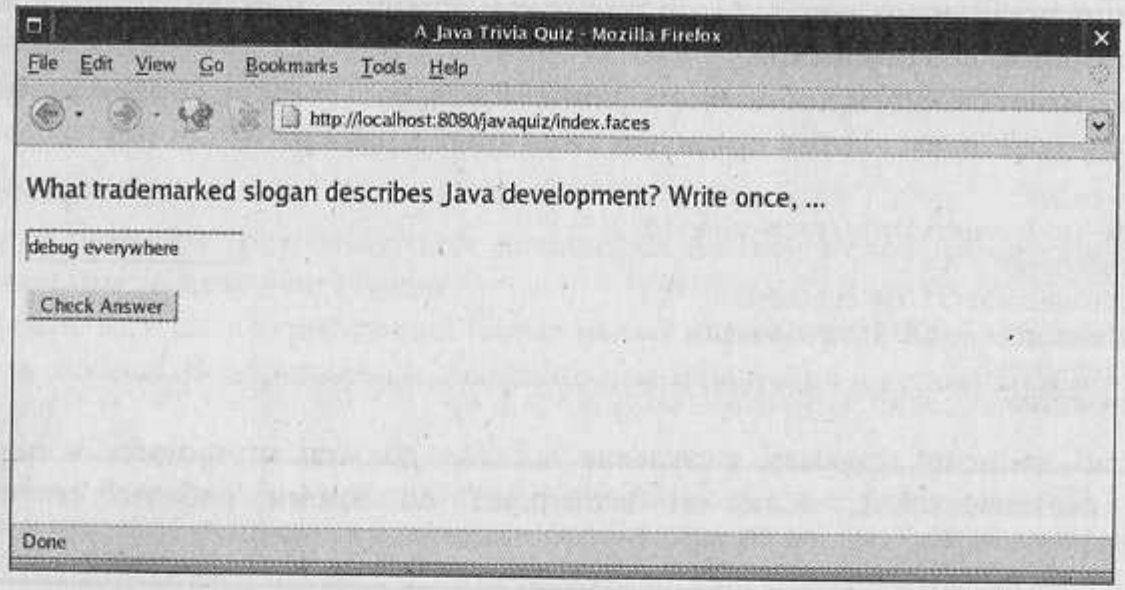


Рис. 3.1. Вопрос викторины

После того как пользователь щелкнет на кнопке Check Answer (Проверить ответ), приложение проверяет, правильным ли является предоставленный им ответ. В случае неправильного ответа пользователю дается еще один дополнительный шанс ответить на тот же вопрос (рис. 3.2).

После двух неправильных ответов отображается следующий вопрос (рис. 3.3).

Разумеется, следующий вопрос отображается также после правильного ответа. Наконец, после последнего вопроса отображается итоговая страница с результатами, которая предлагает пользователю попытаться еще раз пройти викторину (рис. 3.4).

В рассматриваемом приложении имеются два класса. Класс `Problem`, показанный в листинге 3.1, описывает одну задачу с вопросом, ответом и методом проверки того, является ли предъявленный ответ правильным.

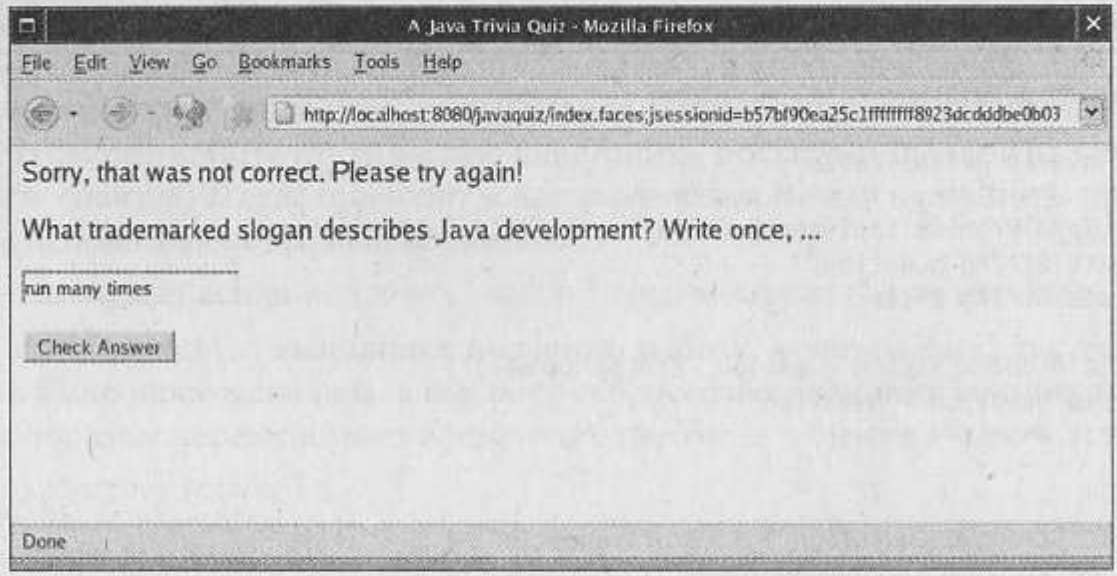


Рис. 3.2. Один неправильный ответ: сделать еще одну попытку

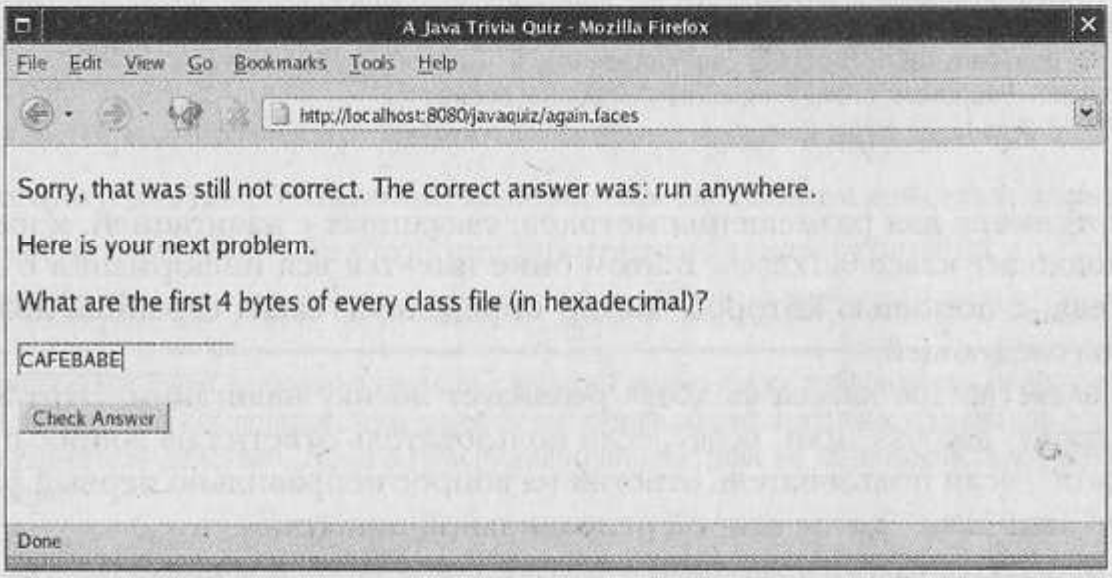


Рис. 3.3. Два неправильных ответа: перейти к следующему

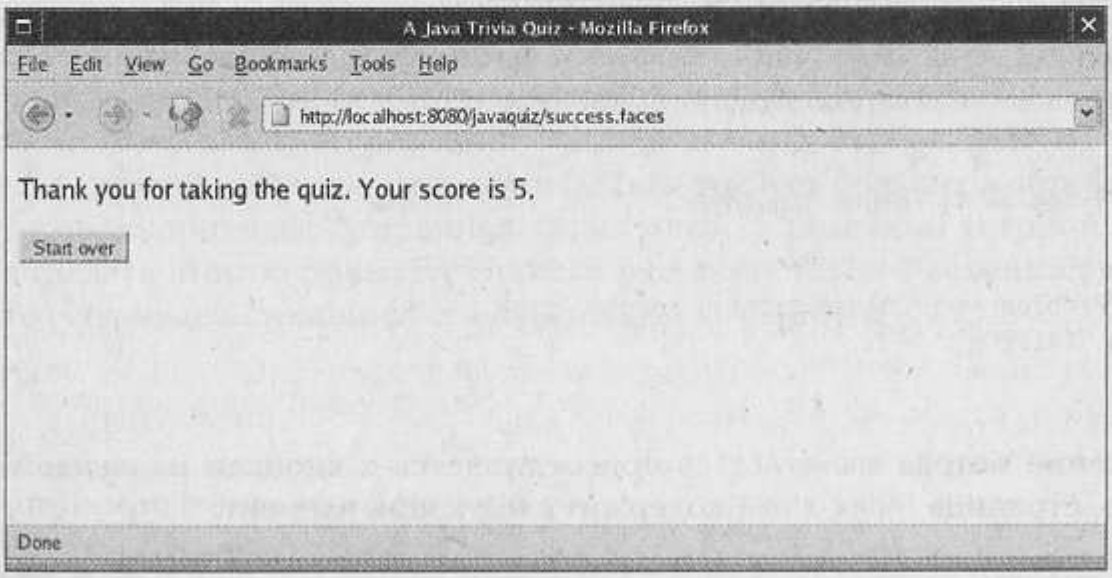


Рис. 3.4. Завершение викторины

Класс `QuizBean` содержит описание викторины, которая состоит из ряда вопросов. Кроме того, экземпляр `QuizBean` дополнительно отслеживает текущую задачу и общее количество очков, заработанное пользователем. Полный код приложения приведен в листинге 3.2.

Листинг 3.1. Файл javaquiz/src/java/com/corejsf/Problem.java

```
 1. package com.corejsf;
 2.
 3. import java.io.Serializable;
 4.
 5. public class Problem implements Serializable {
 6.    private String question;
 7.    private String answer;
 8.
 9.    public Problem(String question, String answer) {
10.       this.question = question;
11.       this.answer = answer;
12.    }
13.
14.    public String getQuestion() { return question; }
15.
16.    public String getAnswer() { return answer; }
17.
18.    // Переопределить в целях более сложной проверки
19.    public boolean isCorrect(String response) {
20.       return response.trim().equalsIgnoreCase(answer);
21.    }
22. }
```

В этом примере для размещения методов, связанных с навигацией, в наибольшей степени подходит класс QuizBean. В этом бине имеется вся информация о действиях пользователя, с помощью которой можно определить, какая страница должна быть отображена следующей.

Метод answerAction класса QuizBean реализует логику навигации. Этот метод возвращает строку "success" или "done", если пользователь ответил на вопрос правильно, строку "again", если пользователь ответил на вопрос неправильно первый раз, и строку "failure" или "done" после второй неправильной попытки.

```
public String answerAction() {
  tries++;
  if (problems.get(currentProblem).isCorrect(response)) {
score++;
nextProblem();
if (currentProblem == problems.size()) return "done";
else return "success";
 }
  else if (tries == 1) return "again";
  else {
nextProblem();
if (currentProblem == problems.size()) return "done";
else return "failure";
 }
}
```

Выражение метода answerAction присоединяется к кнопкам на каждой странице. Например, страница index.xhtml содержит следующий элемент:

```
<h:commandButton value="#{msgs.checkAnswer}" action="#{quizBean.answerAction}"/>
```

На рис. 3.5 показана структура каталогов данного приложения, а в листинге 3.3 приведен код главной страницы викторины index.xhtml. Код страниц success.xhtml и failure.xhtml мы решили не приводить в данной книге, поскольку он отличается от кода index.xhtml только сообщением, отображаемым в верхней части.

На странице done.xhtml, которая рассматривается в листинге 3.4, отображаются окончательная оценка и приглашение пользователю вернуться к началу игры. На этой странице заслуживает внимания командная кнопка. Создается впечатление, будто можно было бы применить статическую навигацию, поскольку щелчок на кнопке Start Over (Начать сначала) всегда приводит к возврату пользователя к странице index.xhtml. Однако мы использовали выражение метода:

```
<h:commandButton value="#{msgs.startOver}" action="#{quizBean.startOverAction}"/>
```

Метод startOverAction выполняет полезную работу, необходимую для того, чтобы игру можно было начать сначала, а именно — сбрасывает значение количества очков и случайным образом перемешивает элементы ответов:

```
public String startOverAction() {
  Collections.shuffle(problems);
  currentProblem = 0;
  score = 0;
  tries = 0;
response = "";
  return "startOver";
}
```

В целом методы действия применяются для достижения двух целей.

- Внесение в модель обновлений, являющихся следствием действий пользователя.
- Передача указания обработчику навигации, какая страница должна отображаться далее.

На заметку! Как будет показано в главе 8, к кнопкам можно также присоединять прослушиватели действий. После того как пользователь щелкнет на кнопке, будет выполнен код метода processAction прослушивателя действий. Однако прослушиватели действий не взаимодействуют с обработчиком навигации.

В листинге 3.5 приведен файл конфигурации приложения с правилами навигации. Чтобы лучше понять эти правила, рассмотрите переходы со страницы на страницу, показанные на рис. 3.6.

Для трех из применяемых результатов ("success", "again" и "done") правила навигации не предусмотрены. Эти результаты всегда приводят к страницам /success.xhtml, /again.xhtml и /done.xhtml. А результат "startOver" мы отображаем на странице /index.xhtml. С другой стороны, результат failure требует большего объема работы. Он первоначально приводит к странице /again.xhtml, с помощью которой пользователь может сделать вторую попытку. Но если результат failure возникает и на этой странице, то следующей страницей становится /failure.xhtml:

```
<navigation-rule>
  <from-view-id>/again.xhtml</from-view-id>
  <navigation-case>
<from-outcome>failure</from-outcome>
<to-view-id>/failure.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <navigation-case>
<from-outcome>failure</from-outcome>
<to-view-id>/again.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

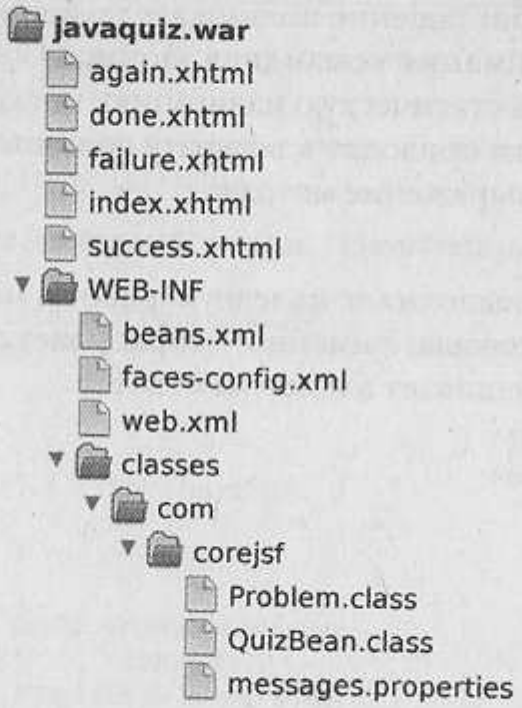


Рис. 3.5. Структура каталогов приложения JavaQuiz

Обратите внимание на то, что имеет значение порядок правил. Второе правило согласуется, если текущей страницей не является `/again.xhtml`.

Наконец, в листинге 3.6 показаны строки сообщений.

Листинг 3.2. Файл `javaquiz/src/java/com/corejsf/QuizBean.java`

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. import java.util.ArrayList;
6. import java.util.Arrays;
7. import java.util.Collections;
8.
9. import javax.inject.Named;
10.    // или import javax.faces.bean.ManagedBean;
11. import javax.enterprise.context.SessionScoped;
12.    // или import javax.faces.bean.SessionScoped;
13.
14. @Named // или @ManagedBean
15. @SessionScoped
16. public class QuizBean implements Serializable {
17.    private int currentProblem;
18.     private   int tries;
19.     private   int score;
20.     private   String response = "";
21.     private   String correctAnswer;
22.
23.     // Здесь формулировки задач приведены в коде. В реальном приложении
24.     // они должны считываться из базы данных
25.     private ArrayList<Problem> problems = new ArrayList<Problem>(Arrays.asList(
26.        new Problem(
27.           "What trademarked slogan describes Java development? Write once, ...",
28.           "run anywhere"),
29.        new Problem(
30.           "What are the first 4 bytes of every class file (in hexadecimal)?",
```

```
31.            "CAFEBABE"),
32.         new Problem(
33.            "What does this statement print? System.out.println(1+\"2\");",
34.            "12"),
35.         new Problem(
36.            "Which Java keyword is used to define a subclass?",
37.            "extends"),
38.         new Problem(
39.            "What was the original name of the Java programming language?",
40.            "Oak"),
41.         new Problem(
42.            "Which java.util class describes a point in time?",
43.            "Date")));
44.
45.      public String getQuestion() { return problems.get(currentProblem).getQuestion(); }
46.
47.      public String getAnswer() { return correctAnswer; }
48.
49.      public int getScore() { return score; }
50.
51.      public String getResponse() { return response; }
52.      public void setResponse(String newValue) { response = newValue; }
53.
54.      public String answerAction() {
55.         tries++;
56.         if (problems.get(currentProblem).isCorrect(response)) {
57.            score++;
58.            nextProblem();
59.            if (currentProblem == problems.size()) return "done";
60.            else return "success";
61.         }
62.         else if (tries == 1) return "again";
63.         else {
64.            nextProblem();
65.              if (currentProblem == problems.size()) return "done";
66.              else return "failure";
67.          }
68.      }
69.
70.      public String startOverAction() {
71.         Collections.shuffle(problems);
72.         currentProblem = 0;
73.         score = 0;
74.         tries = 0;
75.         response = "";
76.         return "startOver";
77.      }
78.
79.      private void nextProblem() {
80.         correctAnswer = problems.get(currentProblem).getAnswer();
81.         currentProblem++;
82.         tries = 0;
83.         response = "";
84.      }
85. }
```

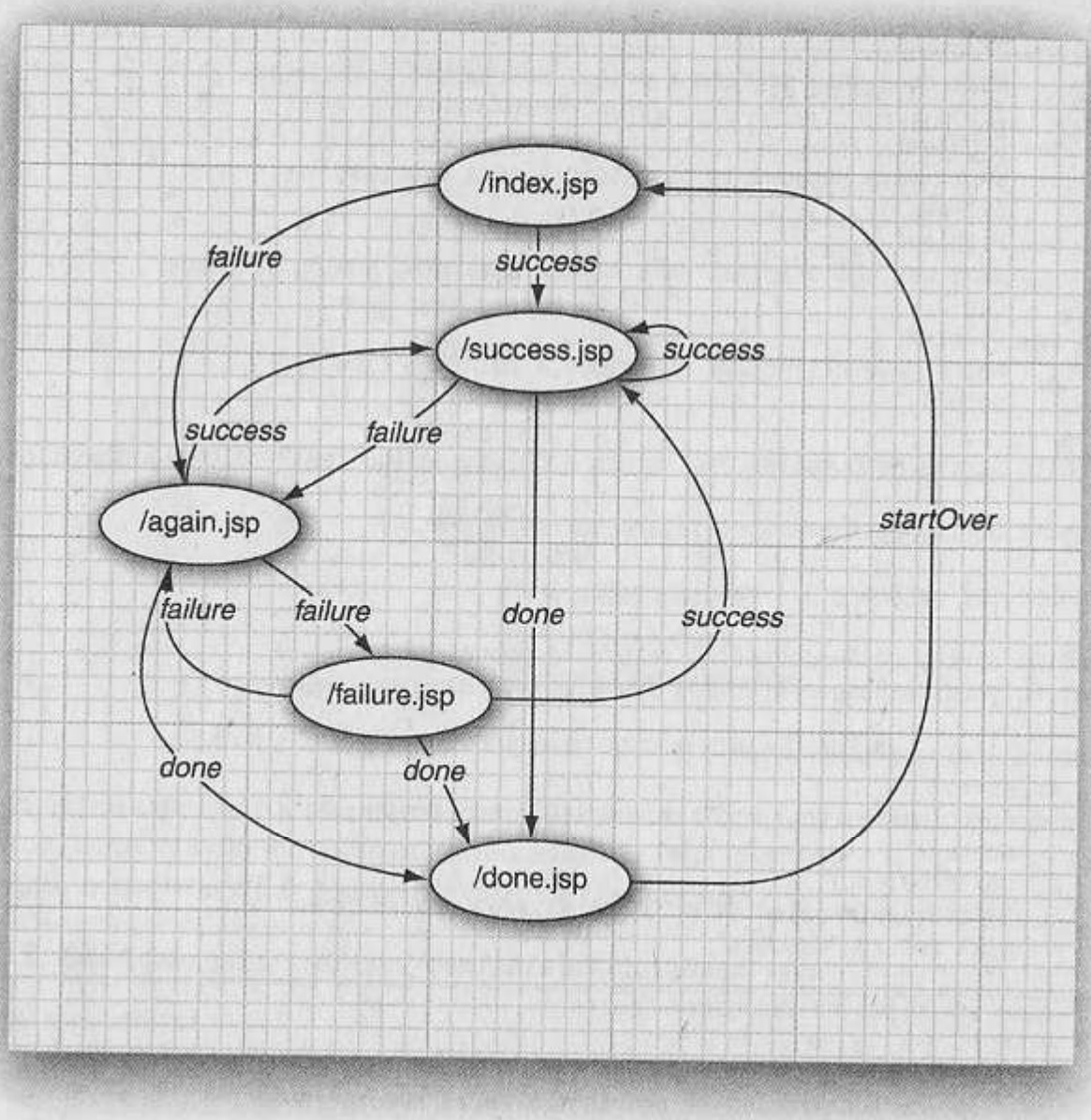


Рис. 3.6. Схема перехода со страниц на страницы в приложении JavaQuiz

Листинг 3.3. Файл `javaquiz/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <p>#{quizBean.question}"/></p>
         <p><h:inputText value="#{quizBean.response}"/></p>
         <p>
            <h:commandButton value="#{msgs.checkAnswer}"
                             action="#{quizBean.answerAction}"/>
         </p>
      </h:form>
   </h:body>
</html>
```

Листинг 3.4. Файл javaquiz/web/done.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <p>
            #{msgs.thankYou}
            <h:outputFormat value="#{msgs.score}">
               <f:param value="#{quizBean.score}"/>
            </h:outputFormat>
         </p>
         <p>
            <h:commandButton value="#{msgs.startOver}"
                             action="#{quizBean.startOverAction}"/>
         </p>
      </h:form>
   </h:body>
</html>
```

Листинг 3.5. Файл javaquiz/web/WEB-INF/faces-config.xml

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
   version="2.0">
   <navigation-rule>
      <navigation-case>
         <from-outcome>startOver</from-outcome>
         <to-view-id>/index.xhtml</to-view-id>
      </navigation-case>
   </navigation-rule>
   <navigation-rule>
      <from-view-id>/again.xhtml</from-view-id>
      <navigation-case>
         <from-outcome>failure</from-outcome>
         <to-view-id>/failure.xhtml</to-view-id>
      </navigation-case>
   </navigation-rule>
   <navigation-rule>
      <navigation-case>
         <from-outcome>failure</from-outcome>
         <to-view-id>/again.xhtml</to-view-id>
      </navigation-case>
   </navigation-rule>

   <application>
      <resource-bundle>
         <base-name>com.corejsf.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
```

```
32.    </application>
33. </faces-config>
```

Листинг 3.6. Файл `javaquiz/src/java/com/corejsf/messages.properties`

```
1. title=A Java Trivia Quiz
2. checkAnswer=Check Answer
3. startOver=Start over
4. correct=Congratulations, that is correct.
5. notCorrect=Sorry, that was not correct. Please try again!
6. stillNotCorrect=Sorry, that was still not correct.
7. correctAnswer=The correct answer was: {0}.
8. score=Your score is {0}.
9. nextProblem=Here is your next problem.
10. thankYou=Thank you for taking the quiz.
```

Перенаправление

Предусмотрена возможность передать реализации JSF запрос на перенаправление к новому представлению. Затем реализация JSF передает клиенту код HTTP перенаправления. Ответ перенаправления `redirect` сообщает клиенту, какой URL должен использоваться для следующей страницы. После этого клиент выполняет запрос `GET` к указанному URL.

Перенаправление происходит медленно, поскольку требует дополнительного цикла обмена данными между сервером и клиентом (браузером). Однако перенаправление дает браузеру шанс обновить свое поле адреса.

На рис. 3.7 показано, как изменяется поле адреса при использовании перенаправления.

Если перенаправление не используется, то исходный URL-адрес (`localhost:8080/javaquiz/faces/index.xhtml`) остается неизменным, когда пользователь переходит со страницы `/index.xhtml` на страницу `/success.xhtml`. А в случае перенаправления браузер отображает новый URL-адрес (`localhost:8080/javaquiz/faces/success.xhtml`).

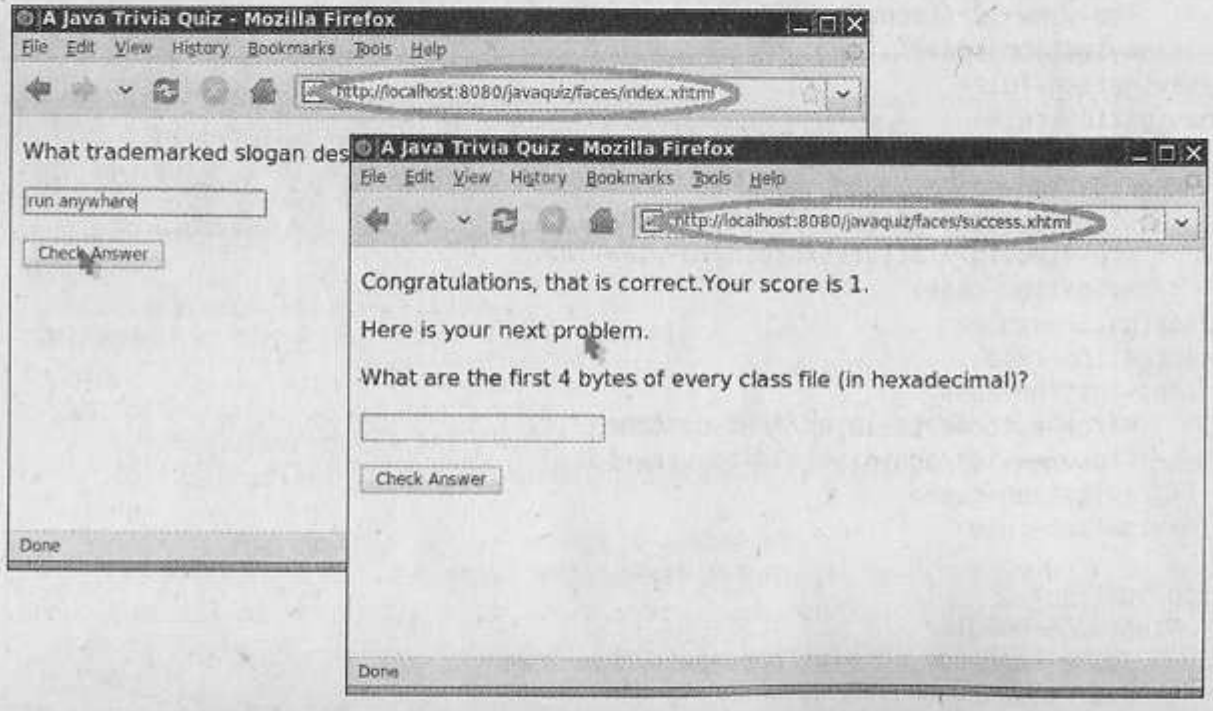


Рис. 3.7. Перенаправление с помощью обновления URL в браузере

Если правила навигации не используются, то необходимо добавить строку

```
?faces-redirect=true
```

к строке результата:

```
<h:commandButton label="Login" action="welcome?faces-redirect=true"/>
```

В правиле навигации элемент `redirect` добавляется после элемента `to-view-id` следующим образом:

```
<navigation-case>
  <from-outcome>success</from-outcome>
  <to-view-id>/success.xhtml</to-view-id>
  <redirect/>
</navigation-case>
```

Перенаправление и флеш-память JSF 2.0

Чтобы уменьшить до предела заполнение ненужными элементами области действия сеанса, имеет смысл в максимально возможной степени использовать область действия запроса. Не прибегая к использованию элемента `redirect`, можно применять бины с областью действия запроса для данных, отображаемых в следующем представлении.

Однако рассмотрим, что происходит при перенаправлении.

1. Клиент отправляет запрос серверу.
2. Отображение области действия запроса заполняется бинами, для которых областью действия служит запрос.
3. Сервер отправляет клиенту код состояния HTTP 302 (Moved temporarily), указывающий на временное перемещение страницы наряду со сведениями о месте перенаправления. На этом обработка текущего запроса заканчивается, и происходит удаление бинов с областью действия запроса.
4. Клиент выполняет запрос `GET` к новому местоположению.
5. Сервер подготавливает к отображению следующее представление. Однако бины с применявшейся ранее областью действия запроса становятся недоступными.

В целях преодоления этой проблемы в версии JSF 2.0 предусмотрен объект флеш-памяти, который может заполняться в одном запросе и использоваться в другом. (Понятие *флеш-памяти* заимствовано из терминологии, применяющейся на веб-платформе Ruby on Rails.) Обычно флеш-память используется для сообщений. Например, поместить сообщение во флеш-память может обработчик кнопки:

```
ExternalContext.getFlash().put("message", "Your password is about to expire");
```

Метод `ExternalContext.getFlash()` возвращает объект класса `Flash`, который реализует интерфейс `Map<String, Object>`.

На странице JSF ссылка на объект флеш-памяти осуществляется с помощью переменной флеш-памяти. Например, сообщение можно отобразить следующим образом:

```
#{flash.message}
```

После подготовки сообщения к отображению и передачи подготовленного представления клиенту строка сообщения автоматически удаляется из флеш-памяти. Значение во флеш-памяти можно даже сохранить больше чем для одного запроса. Выражение

```
#{flash.keep.message}
```

отправляет значение ключа сообщения во флеш-память и добавляет его снова для еще одного цикла запроса.

На заметку! Если в ходе разработки обнаруживается, что между флеш-памятью и бином перебрасываются большие объемы данных, то вместо этого можно рассмотреть возможность использования области действия сеанса.

Навигация с поддержкой метода REST и применение URL, обеспечивающих формирование закладок JSF 2.0

По умолчанию приложение JSF выполняет последовательность запросов POST, передаваемых на сервер. Каждый запрос POST содержит данные формы. Это имеет смысл для приложения, которое собирает большой объем ввода от пользователя. Но большинство веб-приложений не действует подобным образом. Рассмотрим пример, в котором пользователь просматривает каталог товаров для совершения покупок, с помощью щелчков переходя от одной ссылки к другой. Пользователь не вводит никаких данных, а лишь выбирает ссылки для последующего щелчка на них. Эти ссылки должны обеспечивать формирование закладок, чтобы пользователь мог возвращаться к одним и тем же страницам при повторном посещении данного URL. Кроме того, страницы должны быть кэшируемыми. Кэширование – это важная часть создания эффективных веб-приложений. Безусловно, запросы POST не могут применяться для решения задач создания закладок или кэширования.

Архитекторы веб-приложений применяют стиль, получивший название REST (Representational State Transfer – передача представительного состояния), который основан на использовании в приложениях протокола HTTP по такому принципу, который был заложен в нем первоначально. В операциях поиска должны использоваться запросы GET. Запросы PUT, POST и DELETE должны служить для создания, модификации и удаления.

Сторонники подхода на основе REST, как правило, предпочитают применять URL примерно такого типа:

```
http://myserver.com/catalog/item/1729
```

но архитектура REST не требует применения подобных URL, которые принято называть "привлекательными". Запрос GET с параметром

```
http://myserver.com/catalog?item=1729
```

также вполне можно рассматривать как поддерживающий метод REST.

Следует учитывать, что запросы GET ни в коей мере не должны использоваться для обновления информации. Например, запрос GET для добавления товара в корзину

```
http://myserver.com/addToCart?cart=314159&item=1729
```

был бы неподходящим. Запросы GET должны быть идемпотентными. Под этим подразумевается то, что, допустим, двухкратная выдача запроса не должна приводить к иным последствиям, чем однократная. Кэшируемыми могут быть только такие запросы, которые подчиняются этому требованию. Запрос на "добавление товара к корзине" не является идемпотентным, поскольку после его двухкратного выполнения в корзине появятся два экземпляра одного и того же товара. Но в этом контексте запросы POST определенно являются приемлемыми. Таким образом, даже в веб-приложении, которое поддерживает метод REST, должны в определенной степени использоваться запросы POST.

В настоящее время в технологии JSF не предусмотрен стандартный механизм создания или применения "привлекательных URL", но начиная с версии JSF 2.0 имеется поддержка для запросов GET. Такая поддержка будет описана в следующих разделах настоящей главы.

Параметры просмотра

Рассмотрим запрос GET на отображение сведений об определенном товаре:

```
http://myserver.com/catalog?item=1729
```

Идентификатор товара задан как параметр запроса. После получения запроса значение этого параметра должно быть передано соответствующему бину. С этой целью могут использоваться параметры представления.

В верхней части страницы добавьте теги наподобие следующих:

```
<f:metadata>
  <f:viewParam name="item" value="#{catalog.currentItem}"/>
</f:metadata>
```

При обработке запроса значение параметра запроса сведений о товаре передается методу setCurrentItem бина каталога.

Страница JSF может иметь любое количество параметров представления. Параметры представления могут проверяться и преобразовываться, как и любые другие параметры запроса. (В главе 7 изложены подробные сведения о преобразовании и проверке правильности.)

Часто возникает необходимость в получении дополнительных данных после задания параметров представления. Например, после того как будет задан параметр представления определенного товара, может потребоваться выбрать данные о свойствах товара из базы данных для дальнейшей подготовки к отображению страницы с описанием товара. В главе 8 будет показано, как возложить обязанности по выполнению этой работы на обработчик события preRenderView.

Ссылки запросов GET

В предыдущем разделе было показано, как в веб-технологии JSF обрабатываются запросы GET. В приложении, поддерживающем метод REST, может потребоваться предоставить пользователям возможность перемещаться по страницам с помощью запросов GET. В связи с этим возникает необходимость в добавлении на страницы кнопок и ссылок, после щелчков на которых вырабатываются запросы GET. С этой целью используются теги h:button и h:link. (С другой стороны, для выработки запросов POST служат теги h:commandButton и h:commandLink.)

В связи с этим при работе с такими запросами может потребоваться управлять идентификаторами целевого представления и параметрами запроса. Идентификатор целевого представления задается с помощью атрибута outcome. Он может представлять собой фиксированную строку:

```
<h:button value="Done" outcome="done"/>
```

Еще один вариант состоит в том, что может быть указано выражение значения:

```
<h:button value="Skip" outcome="#{quizBean.skipOutcome}"/>
```

Вызывается метод getSkipOutcome. Он должен выдать строку результата. Затем строка результата обычным образом передается в обработчик навигации, выдавая идентификатор целевого представления.

В этом состоит существенное различие между атрибутом outcome тега h:button и атрибутом action тега h:commandButton. Значение атрибута outcome вычисляется до подготовки страницы к отображению, что позволяет встроить требуемую ссылку в содержимое страницы. Однако вычисление значения атрибута action происходит только после фактически выполненного пользователем щелчка на кнопке. По этой причине в спецификации JSF для описания процесса вычисления идентификаторов целевых представлений для запросов GET используется термин "навигация с вытеснением".

Внимание! Выражение языка выражений в атрибуте outcome является выражением значения, а не выражением метода. В принципе действие кнопки можно применять для изменения состояния приложения определенным образом. Однако вычисление результата обработки ссылки с запросом GET не должно приводить к каким-либо изменениям, ведь в конечном итоге ссылка вычисляется только для потенциального использования в будущем.

Определение параметров запроса

Часто возникает необходимость включать параметры в связи с обработкой ссылки с запросом GET. Эти параметры могут быть получены из трех источников.

- Строка результата.
- Параметры просмотра.
- Вложенные теги f:param.

Если один и тот же параметр задан более одного раза, то приоритет получает последнее значение в этом списке присваивания.

Рассмотрим более подробно, как осуществляется выбор применяемых значений.

Параметры могут быть заданы в строке результатов примерно так:

```
<h:link outcome="index?q=1" value="Skip">
```

Обработчик навигации удаляет параметры из результата, вычисляет идентификатор целевого представления и добавляет параметры. В этом примере идентификатором целевого представления является /index.xhtml?q=1.

При передаче многочисленных параметров следует обязательно предусмотреть экранирование с помощью разделителя &:

```
<h:link outcome="index?q=1&amp;score=0" value="Skip">
```

Безусловно, в строке результата можно использовать выражение значения, как в следующем примере:

```
<h:link outcome="index?q=#{quizBean.currentProblem + 1}" value="Skip">
```

Предусмотрен удобный сокращенный способ включения всех параметров представления в строку запроса. Он состоит в том, что добавляется один атрибут:

```
<h:link outcome="index" includeViewParams="true" value="Skip">
```

С помощью этого способа можно переносить все параметры представления с одной страницы на другую, что представляет собой обычное требование к приложению, поддерживающему метод REST.

Для определения параметров представления может использоваться тег f:param. Например:

```
<h:link outcome="index" includeViewParams="true" value="Skip">
  <f:param name="q" value="#{quizBean.currentProblem + 1}"/>
</h:link>
```

Аналогичные преимущества, связанные с включением параметров представления, можно получить при работе со ссылками перенаправления, которые также являются запросами GET. Но вместо задания значения атрибута в теге следует добавлять параметр в результат:

```
<h:commandLink action="index?faces-redirect=true&amp;includeViewParams=true"
  value="Skip"/>
```

К сожалению, вложенные теги f:param не включаются в запрос.

Если для задания правил навигации служат файлы конфигурации в коде XML, используйте атрибут include-viewparams и вложенные теги view-param:

```
<redirect include-view-params=true>
  <view-param>
<name>q</name>
<value>#{quizBean.currentProblem + 1}</value>
  </view-param>
</redirect>
```

В применяемом синтаксисе имеются некоторые несогласованности, но они не должны вызывать обеспокоенность. Программист просто должен работать более внимательно, а это способствует повышению уровня его квалификации.

Добавление ссылок, обеспечивающих формирование закладок, в приложение для викторины

Рассмотрим приложение для викторины, которое использовалось ранее в этой главе для демонстрации способов навигации. Можно ли добиться того, чтобы это приложение в большей степени поддерживало метод REST?

Запрос GET не подходит для передачи ответа, поскольку такие запросы не являются идемпотентными. Передача ответа приводит к изменению счета. Но ссылки, поддерживающие метод REST, можно применить для перехода от одного задания викторины к другому.

Чтобы чрезмерно не усложнять это приложение, мы предусмотрим единственную ссылку, применимую для формирования закладки, в целях перехода к следующему заданию. Для этого используется параметр представления:

```
<f:metadata>
  <f:viewParam name="q" value="#{quizBean.currentProblem}"/>
</f:metadata>
```

Ссылка определяется следующим образом:

```
<h:link outcome="#{quizBean.skipOutcome}" value="Skip">
  <f:param name="q" value="#{quizBean.currentProblem + 1}"/>
</h:link>
```

Метод getSkipOutcome бина QuizBean возвращает индекс или значение "done", в зависимости от того, доступны ли дополнительные задания:

```
public String getSkipOutcome() {
if (currentProblem < problems.size() - 1) return "index";
  else return "done";
}
```

Ссылка, полученная в итоге, выглядит примерно так (рис. 3.8):

```
http://localhost:8080/javaquiz-rest/faces/index.xhtml?q=1
```

Теперь имеется возможность создать закладку на эту ссылку, что позволяет вернуться к любому заданию викторины.

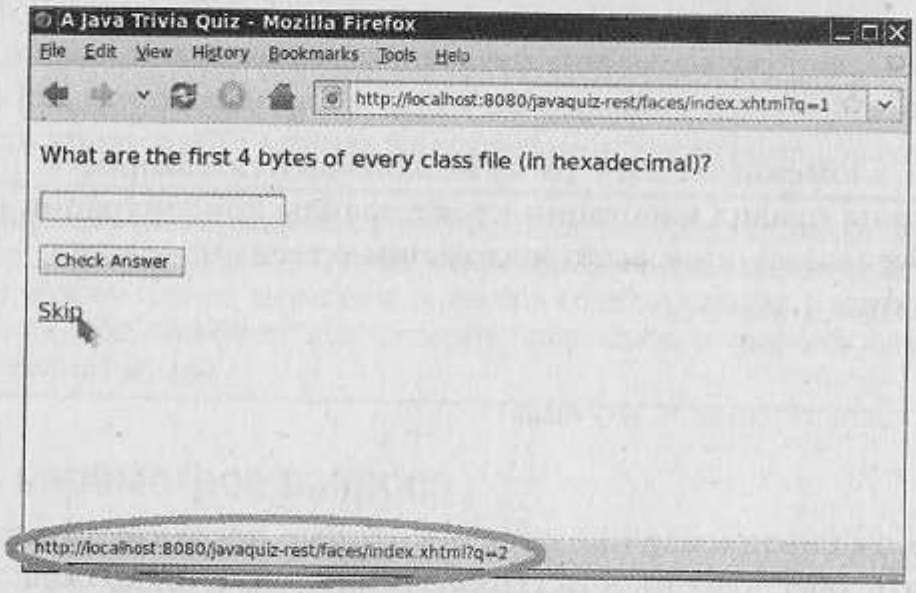


Рис. 3.8. Ссылка, поддерживающая метод REST

В листинге 3.7 показана страница `index.xhtml` с параметром представления и тегом `h:link`. В листинге 3.8 приведен модифицированный код бина `QuizBean`. Мы добавили метод `setCurrentProblem` и изменили механизм вычисления количества очков. Поскольку теперь появилась возможность снова и снова возвращаться к одному и тому же заданию, мы должны исключить возможность того, чтобы пользователь зарабатывал дополнительные очки, отвечая на один и тот же вопрос несколько раз.

Листинг 3.7. Файл `javaquiz-rest/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <f:metadata>
      <f:viewParam name="q" value="#{quizBean.currentProblem}"/>
   </f:metadata>
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <p>#{quizBean.question}</p>
         <p><h:inputText value="#{quizBean.response}"/></p>
         <p><h:commandButton value="#{msgs.checkAnswer}"
         action="#{quizBean.answerAction}"/></p>
         <p><h:link outcome="#{quizBean.skipOutcome}" value="Skip">
            <f:param name="q" value="#{quizBean.currentProblem + 1}"/>
         </h:link>
         </p>
      </h:form>
   </h:body>
</html>
```

Листинг 3.8. Файл javaquiz-rest/src/java/com/corejsf/QuizBean.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. import java.util.ArrayList;
6. import java.util.Arrays;
7. import java.util.Collections;
8. import javax.inject.Named;
9.    // или import javax.faces.bean.ManagedBean;
10. import javax.enterprise.context.SessionScoped;
11.    // или import javax.faces.bean.SessionScoped;
12.
13. @Named // или @ManagedBean
14. @SessionScoped
15. public class QuizBean implements Serializable {
16.    private int currentProblem;
17.    private int tries;
18.    private String response = "";
19.    private String correctAnswer;
20.
21.    // Здесь формулировки задач приведены в коде. В реальном приложении
22.    // они должны считываться из базы данных
23.    private ArrayList<Problem> problems = new ArrayList<Problem>(Arrays.asList(
24.       new Problem(
25.          "What trademarked slogan describes Java development? Write once, ...",
26.          "run anywhere"),
27.       new Problem(
28.          "What are the first 4 bytes of every class file (in hexadecimal)?",
29.          "CAFEBABE"),
30.       new Problem(
31.          "What does this statement print? System.out.println(1+\"2\");",
32.          "12"),
33.        new Problem(
34.           "Which Java keyword is used to define a subclass?",
35.           "extends"),
36.        new Problem(
37.           "What was the original name of the Java programming language?",
38.           "Oak"),
39.        new Problem(
40.           "Which java.util class describes a point in time?",
41.           "Date")));
42.
43.    private int[] scores = new int[problems.size()];
44.
45.    public String getQuestion() {
46.       return problems.get(currentProblem).getQuestion();
47.    }
48.
49.    public String getAnswer() { return correctAnswer; }
50.
51.    public int getScore() {
52.       int score = 0;
53.       for (int s : scores) score += s;
54.       return score;
55.    }
56.
57.    public String getResponse() { return response; }
58.    public void setResponse(String newValue) { response = newValue; }
59.
```

```
60.     public int getCurrentProblem() { return currentProblem; }
61.     public void setCurrentProblem(int newValue) { currentProblem = newValue; }
62.
63.     public String getSkipOutcome() {
64.        if (currentProblem < problems.size() - 1) return "index";
65.        else return "done";
66.     }
67.
68.     public String answerAction() {
69.        tries++;
70.        if (problems.get(currentProblem).isCorrect(response)) {
71.           scores[currentProblem] = 1;
72.           nextProblem();
73.           if (currentProblem == problems.size()) return "done";
74.           else return "success";
75.        }
76.        else {
77.           scores[currentProblem] = 0;
78.           if (tries == 1) return "again";
79.             else {
80.                nextProblem();
81.                if (currentProblem == problems.size()) return "done";
82.                else return "failure";
83.             }
84.         }
85.     }
86.
87.     public String startOverAction() {
88.        Collections.shuffle(problems);
89.        currentProblem = 0;
90.        for (int i = 0; i < scores.length; i++)
91.           scores[i] = 0;
92.        tries = 0;
93.        response = "";
94.        return "startOver";
95.     }
96.
97.     private void nextProblem() {
98.        correctAnswer = problems.get(currentProblem).getAnswer();
99.        currentProblem++;
100.       tries = 0;
101.       response = "";
102.    }
103.}
```

Расширенные правила навигации

Приемы, описанные в предыдущих разделах, должны удовлетворять большинству практических потребностей, связанных с навигацией. В этом разделе рассматриваются оставшиеся правила для элементов навигации, которые могут присутствовать в файле `faces-config.xml`. На рис. 3.9 показана синтаксическая диаграмма для допустимых элементов.

На заметку! Как уже было описано в разделе "Настройка конфигурации бинов" главы 2 на стр. 64, информация о навигации также может размещаться и в других файлах конфигурации, а не только в стандартном файле `faces-config.xml`.

Синтаксическая диаграмма, приведенная на рис. 3.9, показывает, что каждый элемент, `navigation-rule` и `navigation-case`, может иметь произвольное описание, а также элементы `display-name` и `icon`. Эти элементы предназначены для использования в инструментальных средствах разработки программ, поэтому более подробно здесь рассматриваться не будут.

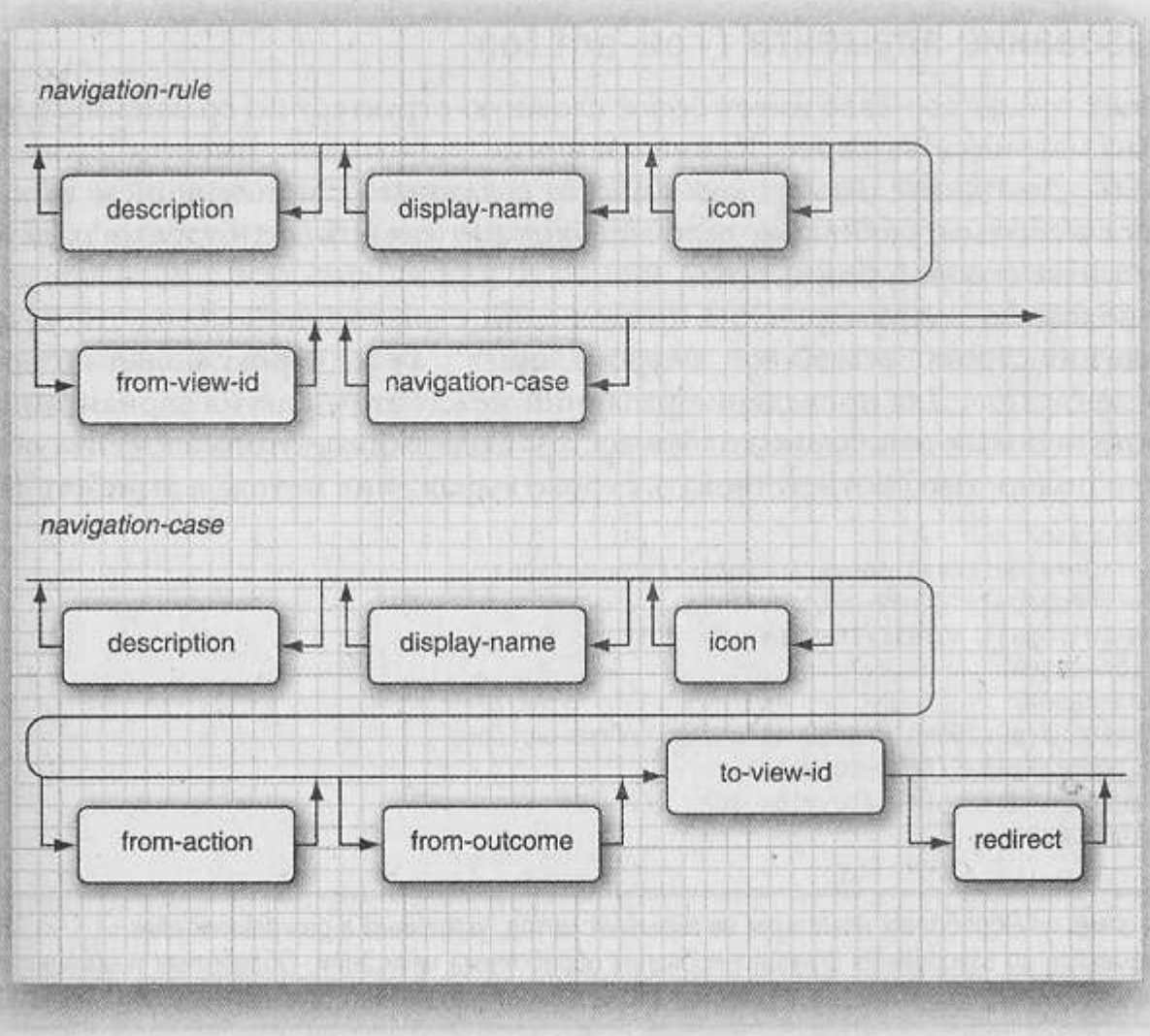


Рис. 3.9. Синтаксическая диаграмма для элементов навигации

Подстановочные знаки

В элементе `from-view-id` правила навигации можно использовать подстановочные знаки, например:

```
<navigation-rule>
  <from-view-id>/secure/*</from-view-id>
  <navigation-case>
    . . .
  </navigation-case>
</navigation-rule>
```

Это правило будет действовать для всех страниц, имена которых начинаются с префикса `/secure/`. Допускается применение только одного символа `*`, который должен находиться в конце строки идентификатора.

Если с шаблоном согласуется несколько правил с подстановочными знаками, то выбирается правило с наибольшей длиной.

На заметку! Вместо того чтобы не учитывать элемент `from-view-id`, можно также использовать один из следующих подходов для указания правила, распространяющегося на все страницы:

```
<from-view-id>/*</from-view-id>
```

или

```
<from-view-id>*</from-view-id>
```

Использование элемента `from-action`

Элемент `navigation-case` имеет более сложную структуру по сравнению с теми, которые рассматривались ранее. Кроме элемента `from-outcome`, имеется также элемент `from-action`. Дополнительные возможности, связанные с применением двух элементов, могут оказаться удобными, если необходимо связать два отдельных действия с одной и той же строкой результата.

Предположим, например, что в приложении с викториной метод `startOverAction` возвращает не строку `"startOver"`, а строку `"again"`. Ту же строку может возвращать и метод `answerAction`. Для проведения различий между этими двумя вариантами навигации можно использовать элемент `from-action`. Содержимое этого элемента обязательно должно полностью соответствовать строке выражения метода в атрибуте `action`.

```
<navigation-case>
  <from-action>#{quizBean.answerAction}</from-action>
  <from-outcome>again</from-outcome>
  <to-view-id>/again.xhtml</to-view-id>
</navigation-case>
<navigation-case>
  <from-action>#{quizBean.startOverAction}</from-action>
  <from-outcome>again</from-outcome>
  <to-view-id>/index.xhtml</to-view-id>
</navigation-case>
```

На заметку! Обработчик навигации не вызывает метод, указанный в разграничителях `#{...}`. Этот метод вызывался до завершения функционирования обработчика навигации. Обработчик навигации просто использует строку `from-action` в качестве ключа для поиска соответствующего варианта навигации.

Возможности условной навигации JSF 2.0

Начиная с версии JSF 2.0 предусмотрена возможность задавать элемент `if`, который активизирует определенный вариант навигации только при выполнении заданного условия. Для этого условия должно быть предусмотрено выражение значения. Ниже приведен соответствующий пример.

```
<navigation-case>
  <from-outcome>previous</from-outcome>
  <if>#{quizBean.currentQuestion != 0}</if>
  <to-view-id>/main.xhtml</to-view-id>
</navigation-case>
```

Динамические идентификаторы целевого представления JSF 2.0

Элемент `to-view-id` может представлять собой выражение значения, и в этом случае происходит его вычисление. Полученный результат используется как идентификатор представления. Рассмотрим пример:

```
<navigation-rule>
  <from-view-id>/main.xhtml</from-view-id>
  <navigation-case>
<to-view-id>#{quizBean.nextViewID}</to-view-id>
  </navigation-case>
</navigation-rule>
```

В этом примере вызывается метод `getNextViewID` бина викторины для получения идентификатора целевого представления.

Резюме

Выше были описаны все средства управления навигацией, предусмотренные в технологии JSF. Следует помнить, что в наиболее простом случае эти средства являются весьма несложными: действия кнопок и ссылок могут просто возвращать результат, который определяет следующую страницу. Но если необходимо обеспечить более развитое управление, платформа JSF предоставит все необходимые инструментальные средства.

В следующей главе будут приведены все необходимые сведения о стандартных компонентах JSF.

Стандартные теги JSF

В этой главе...

- Общие сведения об основных тегах, применяемых в технологии JSF
- Общие сведения о тегах HTML, применяемых в технологии JSF
- Панели
- Теги head, body и form
- Текстовые поля и области
- Кнопки и ссылки
- Теги выбора
- Сообщения

Глава 4

Для разработки привлекательных приложений JSF требуется хорошее освоение библиотек тегов JSF. В версии JSF 1.2 были предусмотрены две библиотеки тегов: основная и HTML. Со времени выхода версии JSF 2.0 появилось шесть библиотек более чем со 100 тегами (табл. 4.1). В настоящей главе рассматриваются основная библиотека и большая часть библиотеки HTML. Один из компонентов библиотеки HTML (таблица данных) является настолько сложным, что его описание приведено отдельно в главе 6.

Таблица 4.1. Библиотеки тегов JSF

Библиотека	Идентификатор пространства имен	Обычно используемый префикс	Число тегов	См. главу
Основная	`http://java.sun.com/jsf/core`	`f:`	27	См. табл. 4.2
HTML	`http://java.sun.com/jsf/html`	`h:`	31	4 и 6
Facelets JSF 2.0	`http://java.sun.com/jsf/facelets`	`ui:`	11	5
Составные компоненты JSF 2.0	`http://java.sun.com/jsf/composite`	`composite:`	12	9
Основная библиотека JSTL JSF 2.0	`http://java.sun.com/jsp/jstl/core`	`c:`	7	13
Функции JSTL JSF 2.0	`http://java.sun.com/jsp/jstl/functions`	`fn:`	16	2

Общие сведения об основных тегах, применяемых в технологии JSF

Основная библиотека содержит теги, независимые от средств подготовки к отображению языка HTML.

Основные теги перечислены в табл. 4.2.

Таблица 4.2. Основные теги JSF

Тег	Описание	См. главу
attribute	Задает атрибут ("ключ-значение") в его родительском компоненте	4
param	Добавляет дочерний компонент параметра к его родительскому компоненту	4
facet	Добавляет аспект к компоненту	4
actionListener	Добавляет прослушиватель action к компоненту	8
setPropertyActionListener JSF 1.2	Добавляет прослушиватель action, который задает свойство	8
valueChangeListener JSF 1.2	Добавляет прослушиватель изменения значения к компоненту	8
phaseListener	Добавляет прослушиватель этапа к родительскому представлению	8
event JSF 2.0	Добавляет прослушиватель системного события компонента	8
converter	Добавляет произвольный преобразователь к компоненту	7
convertDateTime	Добавляет преобразователь типов данных даты и времени к компоненту	7
convertNumber	Добавляет преобразователь чисел к компоненту	7
validator	Добавляет средство проверки к компоненту	7
validateDoubleRange	Проверяет достоверность диапазона double для значения компонента	7
validateLength	Проверяет достоверность длины значения компонента	7
validateLongRange	Проверяет достоверность диапазона long для значения компонента	7
validateRequired JSF 2.0	Проверяет, присутствует ли значение	7
validateRegex JSF 2.0	Проверяет достоверность значения с помощью регулярного выражения	7
validateBean JSF 2.0	Использует API-интерфейс Bean Validation (JSR 303) для проверки правильности	7
loadBundle	Загружает связку ресурсов и сохраняет свойство в качестве объекта Map	2
selectitems	Определяет элементы для выбора одного или нескольких компонентов	4
selectitem	Определяет элемент для выбора одного или нескольких компонентов	4
verbatim	Преобразует текст, содержащий разметку, в компонент	4
viewParam JSF 2.0	Определяет так называемый параметр представления, который может быть инициализирован с помощью параметра запроса	3
metadata JSF 2.0	Сохраняет параметры представления. В будущем не исключена возможность его применения для хранения других метаданных	3

Окончание табл. 4.2

Тег	Описание	См. главу
ajax JSF 2.0	Допускает поведение Ajax для компонентов	11
view	Используется для определения локали страницы или прослушивателя этапа	2 и 7
subview	Не требуется при работе с фейслетами	

Большинство основных тегов представляет объекты, добавляемые к компоненту, наподобие следующих:

- атрибуты
- параметры
- аспекты
- прослушиватели
- преобразователи;
- средства проверки;
- элементы выбора.

Все основные теги подробно обсуждаются в различных главах этой книги (см. табл. 4.1).

Атрибуты, параметры и аспекты

Теги `f:attribute`, `f:param` и `f:facet` представляют собой универсальные теги, предназначенные для добавления данных к компоненту. Любой компонент может хранить произвольные пары "имя–значение" в своей карте атрибутов. Предусмотрена возможность задать любой атрибут на странице, а затем осуществлять его выборку программным путем. Например, в разделе "Предоставление атрибутов для преобразователей" главы 7 на стр. 255 показано, как задать символ разделителя для групп цифр кредитной карточки:

```
<h:outputText value="#{payment.card}">
  <f:attribute name="separator" value="-" />
</h:outputText>
```

Преобразователь, который форматирует вывод, осуществляет выборку этого атрибута из компонента. Тег `f:param` также позволяет определять пары "имя–значение", но значение размещается в отдельном дочернем компоненте, поэтому создаваемый при этом механизм памяти является более громоздким. Однако дочерние компоненты формируют список, а не карту. Тег `f:param` можно использовать, если требуется предусмотреть хранение нескольких значений под одним и тем же именем (или вообще без имени). Один из соответствующих примеров см. в разделе "Сообщения с переменными частями" главы 2 на стр. 52, в котором компонент `h:output-Format` содержит список дочерних тегов `f:param`.

Наконец, тег `f:facet` добавляет именованный компонент к карте аспектов компонента. Аспект – это не дочерний компонент; каждый компонент имеет и список дочерних компонентов, и карту именованных компонентов аспекта. Компоненты аспекта обычно подготавливаются к отображению в специальной области. Корневой каталог страницы Facelets имеет два аспекта с именами `"head"` и `"body"`. В разделе "Заголовки, нижние колонтитулы и надписи" главы 6 на стр. 192 будет показано, как использовать аспекты `"header"` и `"footer"` в таблицах данных.

На заметку! Компонент `h:commandlink` превращает свои дочерние теги `f:param` в пары "имя–значение" HTTP-запроса. После этого в прослушивателе событий, который активизируется, когда пользователь щелкает на ссылке, можно осуществлять выборку пар "имя–значение" из карты запроса. Мы продемонстрируем эту методику в главе 8.

В табл. 4.3 показаны атрибуты для тегов `f:attribute`, `f:param` и `f:facet`.

Таблица 4.3. Атрибуты тегов `f:attribute`, `f:param` и `f:facet`

Атрибут	Описание
`name`	Атрибут, компонент параметра или имя аспекта
`value`	Атрибут или значение компонента параметра (не относится к тегу `f:facet`),
`binding, id`	См. табл. 4.5 на стр. 108 (только тег `f:param`)

На заметку! Все атрибуты тегов, описанные в этой главе, за исключением `var` и `id`, принимают значение или выражения метода. Атрибут `var` должен быть представлен строкой. Атрибут `id` может быть строкой или непосредственным выражением `${...}`.

Общие сведения о тегах HTML, применяемых в технологии JSF

В табл. 4.4 перечислены все теги HTML. Мы можем сгруппировать их по категориям.

- Теги ввода (`input...`).
- Теги вывода (`output...`, `graphicImage`).
- Команды (`commandButton` и `commandLink`).
- Запросы `GET` (`button`, `link`, `outputLink`).
- Теги выбора (`checkbox`, `listbox`, `menu`, `radio`).
- Страницы HTML (`head`, `body`, `form`, `outputStylesheet`, `outputScript`).
- Макеты (`panelGrid`, `panelGroup`).
- Таблицы данных (`dataTable` и `column`); см. главу 6.
- Ошибки и сообщения (`message`, `messages`).

Теги HTML, применяемые в технологии JSF, совместно используют общие атрибуты, передаваемые атрибуты HTML и атрибуты, поддерживающие динамически формируемый код HTML.

Таблица 4.4. Теги HTML, применяемые в технологии JSF

Тег	Описание
`head` JSF 2.0	Подготавливает к отображению заголовок страницы
`body` JSF 2.0	Подготавливает к отображению текст страницы
`form`	Подготавливает к отображению форму HTML
`outputStylesheet` JSF 2.0	Добавляет таблицу стилей к странице

Окончание табл. 4.4

Тег	Описание
`outputScript` JSF 2.0	Добавляет сценарий к странице
`inputText`	Однострочный элемент управления вводом текста
`inputTextarea`	Многострочный элемент управления вводом текста
`inputSecret`	Элемент управления вводом пароля
`inputHidden`	Скрытое поле
`outputLabel`	Метка для другого компонента, обеспечивающая доступ
`outputLink`	Ссылка на другой веб-сайт
`outputFormat`	Аналогичен `outputText`, но форматирует составные сообщения
`outputText`	Однострочный вывод текста
`commandButton`	Кнопка: передачи, сброса или нажимная
`commandLink`	Ссылка, которая действует как нажимная кнопка
`button` JSF 2.0	Кнопка для выдачи запроса `GET`
`link` JSF 2.0	Ссылка для выдачи запроса `GET`
`message`	Отображает последнее по времени сообщение для компонента
`messages`	Отображает все сообщения
`graphicImage`	Показывает изображение
`selectOneListbox`	Окно списка с единственным выбором
`selectOneMenu`	Меню с единственным выбором
`selectOneRadio`	Набор переключателей
`selectBooleanCheckbox`	Флажок
`selectManyCheckbox`	Набор флажков
`selectManyListbox`	Окно списка с множественным выбором
`selectManyMenu`	Меню с множественным выбором
`panelGrid`	Табличный макет
`panelGroup`	Два или несколько компонентов, которые обрабатываются в макете как один
`dataTable`	Многофункциональный элемент управления таблицей (см. главу 6)
`column`	Столбец в таблице данных `dataTable` (см. главу 6)

На заметку! На первый взгляд кажется, что теги HTML носят слишком длинные имена, например, `selectManyListbox` может быть с таким же успехом обозначен как `multiList`. Но эти подробные имена соответствуют сочетанию имен компонента и модуля подготовки к отображению, поэтому становится ясно, что тег с именем `selectManyListbox` представляет собой компонент `selectMany`, соединенный с модулем подготовки к отображению окна списка (`listbox`). Сведения о том, какой тип компонента представлен тегом, становятся очень важными, когда возникает необходимость обращаться к компонентам программным путем.

Общие атрибуты

Общими для многих тегов компонентов HTML являются три типа атрибутов:

- базовые атрибуты;
- атрибуты HTML 4.0;
- атрибуты событий DHTML.

Ниже каждый из этих типов рассматривается отдельно.

Базовые атрибуты

Как показано в табл. 4.5, базовые атрибуты являются общими для большей части тегов HTML, применяемых в технологии JSF.

Таблица 4.5. Базовые атрибуты тегов HTML[1]

Атрибут	Типы компонентов	Описание
`id`	`A (31)`	Идентификатор компонента
`binding`	`A (31)`	Связывает этот компонент со свойством вспомогательного бина
`rendered`	`A (31)`	Значение типа `Boolean`; `false` указывает на подавление подготовки к отображению
`value`	`I, O, C (21)`	Значение компонента, как правило, выражение значения
`valueChangeListener`	`I (11)`	Выражение метода для такого метода, который отвечает на изменения значения
`converter`	`I, O (15)`	Имя класса преобразователя
`validator`	`I (11)`	Имя класса средства проверки, которое создается и закрепляется за компонентом
`required`	`I (11)`	Значение типа `Boolean`; `true` указывает, что в соответствующее поле должно быть введено значение
`converterMessage`, `validatorMessage`, `requiredMessage` JSF 1.2	`I (11)`	Определяемое пользователем сообщение, которое отображается при возникновении ошибки преобразования или при проверке правильности, а также если обязательный ввод пропущен

[1] `A` = все, `I` = ввод, `O` = вывод, `C` = команды, `(n)` = число тегов, применяемых с атрибутом

Все компоненты могут иметь атрибуты `id`, `binding` и `rendered`, которые будут описаны в следующих разделах. Атрибуты `value` и `converter` позволяют задать значение компонента и предназначены для преобразования его из строки в объект или наоборот.

Атрибуты `validator`, `required` и `valueChangeListener` предназначены для компонентов ввода и позволяют проверять правильность значений и реагировать на изменения этих значений. В главе 7 приведена дополнительная информация о средствах проверки и преобразователях.

Идентификаторы и средства связывания

Многофункциональный атрибут `id` позволяет выполнять указанные ниже действия.

- Обращаться к компонентам JSF из других тегов JSF.

- Получать справочную информацию о компонентах в коде Java.
- Обращаться к элементам HTML в сценариях.

В настоящем разделе рассматриваются первые две упомянутые выше задачи. В разделе "Элементы формы и код JavaScript" на стр. 118 вы найдете подробные сведения о последней задаче.

Атрибут `id` позволяет авторам страниц ссылаться на компоненты из других тегов. Например, сообщение об ошибке для компонента может быть отображено следующим образом:

```
<h:inputText id="name" .../>
<h:message for="name"/>
```

Идентификаторы компонентов можно также использовать в коде Java для получения доступа к компоненту в целях выборки справочной информации о нем. Например, к компоненту `name` в прослушивателе можно обратиться примерно так:

```
UIComponent component = event.getComponent().findComponent("name");
```

При использовании приведенного выше вызова метода `findComponent` необходимо учитывать один нюанс: компонент, в котором было сформировано событие, и компонент `name` должны находиться в одной и той же форме. Предусмотрен еще один способ доступа к компоненту в коде Java-приложения: определить компонент как поле экземпляра класса и предусмотреть методы получения и задания свойства для компонента. После этого можно воспользоваться атрибутом `binding`, задаваемым на странице JSF, следующим образом:

```
<h:inputText binding="#{form.nameField}" .../>
```

Атрибут `binding` определяется с помощью выражения значения. Это выражение ссылается на свойство бина для чтения и записи, как в следующем примере:

```
private UIComponent nameField = new UIInput();
public UIComponent getNameField() { return nameField; }
public void setNameField(UIComponent newValue) { nameField = newValue; }
```

Дополнительную информацию об атрибуте `binding` см. в разделе "Вспомогательные бины" главы 2 на стр. 49. Реализация JSF задает свойства для компонентов, что позволяет манипулировать компонентами программным путем.

Значения, преобразователи и средства проверки

Все объекты, подобные полям ввода, полям вывода, командам и таблицам данных, имеют значения. Связанные с ними теги в библиотеке HTML, такие как `h:inputText` и `h:dataTable`, имеют в своем составе атрибут `value`. Значение можно задавать с помощью строки:

```
<h:commandButton value="Logout" .../>
```

Но чаще всего используется выражение значения, например:

```
<h:inputText value="#{customer.name}"/>
```

Атрибут `converter`, который является общим для объектов ввода и вывода, позволяет закреплять преобразователь за компонентом. Входные теги имеют также атрибут `validator`, который можно использовать для закрепления средства проверки за компонентом. Преобразователи и средства проверки подробно описаны в главе 7.

Подготовка к отображению по условию

Для включения или исключения компонента в зависимости от условия можно использовать атрибут `rendered`. Например, может потребоваться подготовить к отображению кнопку Logout (Выход), только если пользователь в настоящее время находится в системе:

```
<h:commandButton ... rendered="#{user.loggedIn}"/>
```

Чтобы иметь возможность включать группу компонентов по условию, определите их в дескриптере `h:panelGrid` с атрибутом `rendered`. Чтобы получить дополнительную информацию, обратитесь к разделу "Панели" на стр. 114.

Совет. Следует помнить, что в выражениях значения можно использовать операторы. Например, предположим, что имеется представление, которое действует как область окна с вкладками, на которой по условию подготавливается к отображению та или иная панель в зависимости от выбранной вкладки. В этом случае можно использовать тег `h:panelGrid` следующим образом:

```
<h:panelGrid rendered="#{bean.selectedTab == 'Movies'}"/>
```

В приведенном выше коде осуществляется подготовка к отображению панели с фильмами, после того как пользователь выбирает вкладку `Movies`.

На заметку! Иногда можно видеть, что для подготовки к отображению по условию используется конструкция `c:if` библиотеки JSTL. Но этот способ менее эффективен по сравнению с применением атрибута `rendered`.

Атрибуты HTML 4.0

Теги HTML, применяемые в технологии JSF, имеют соответствующие передаваемые атрибуты HTML 4.0. Значения этих атрибутов передаются в сформированный элемент HTML. Например, тег `<h:inputText value="#{form.name.last}" size="25".../>` вырабатывает следующий код HTML: `<input type="text" size="25".../>`. Обратите внимание на то, что в HTML передается атрибут `size`.

Атрибуты HTML 4.0 перечислены в табл. 4.6.

Таблица 4.6. Передаваемые атрибуты HTML 4.0[1]

Атрибут	Описание
`accesskey (16)`	Ключ, как правило, объединенный с определяемым системой метаключом, который переводит фокус на элемент
`accept (1)`	Разделенный запятыми список типов содержимого для формы
`acceptcharset (1)`	Разделенный запятыми или пробелами список символьных кодировок для формы. Атрибут `accept-charset` языка HTML задается с помощью атрибута JSF с именем `acceptcharset`
`alt (5)`	Альтернативный текст для нетекстовых элементов, таких как изображения или аплеты
`border (4)`	Значение ширины границы элемента в пикселях
`charset (3)`	Символьная кодировка для ресурса, обозначенного ссылкой
`coords (3)`	Координаты элемента, имеющего форму прямоугольника, круга или многоугольника
`dir (26)`	Направление для текста. Допустимыми значениями являются `"ltr"` (left to right — слева направо) и `"rtl"` (right to left — справа налево)

Окончание табл. 4.6

Атрибут	Описание
disabled (14)	Отмененное состояние элемента ввода или кнопки
hreflang (3)	Базовый язык ресурса, заданный с помощью атрибута href; атрибут hreflang может использоваться только в сочетании с href
lang (26)	Базовый язык атрибутов и текста элемента
maxlength (2)	Максимальное количество символов для текстовых полей
readonly (11)	Состояние поля ввода, допускающее только чтение; в поле только для чтения текст можно выбирать, но не редактировать
rel (3)	Связь между текущим документом и ссылкой, заданной с помощью атрибута href
rev (3)	Обратная ссылка от анкера, указанного атрибутом href, на текущий документ. Значение атрибута представляет собой разделенный пробелами список типов ссылок
rows (1)	Количество видимых строк в текстовой области. Тег h:dataTable имеет атрибут rows, но это — не передаваемый атрибут HTML
shape (3)	Форма геометрической области. Допустимые значения: default, rect, circle, poly (default обозначает всю геометрическую область)
size (4)	Размер поля ввода
style (26)	Встроенные данные о стиле
styleClass (26)	Класс стиля; подготавливается к отображению как атрибут класса HTML
tabindex (16)	Числовое значение, определяющее индекс вкладки
target (5)	Имя фрейма, в котором открывается документ
title (25)	Заголовок, используемый для обеспечения доступности, который описывает элемент. В браузерах для визуального отображения обычно создаются подсказки для этих значений заголовков
type (4)	Тип ссылки, например "stylesheet"
width (3)	Ширина элемента

[1] (n) = количество тегов, применяемых с атрибутом

Атрибуты, перечисленные в табл. 4.6, определены в спецификации HTML, к которой можно обратиться в оперативном режиме по адресу http://www.w3.org/TR/REC-html40. Этот веб-сайт представляет собой превосходный источник ресурсов для глубокого изучения языка HTML.

Стили

Для оказания влияния на то, как происходит подготовка компонентов к отображению, могут использоваться стили CSS, либо встроенные (style), либо основанные на классах (styleClass):

```
<h:outputText value="#{customer.name}" styleClass="emphasis"/>
<h:outputText value="#{customer.id}" style="border: thin solid blue"/>
```

Атрибуты стиля CSS могут представлять собой выражения значения, что позволяет получить программный контроль над стилями.

Ресурсы JSF 2.0

Предусмотрена возможность включить таблицу стилей обычным образом, с помощью тега ссылки HTML. Но этот способ становится трудоемким, если страницы приложения находятся на разных уровнях вложенности каталогов, поскольку после перехода к любой странице всегда будет требоваться задавать по-новому каталог таблицы стилей. Но еще более важно то, что при сборке страниц, получаемых из различных источников (как описано в главе 5), часто даже не известно, где в конечном итоге находятся те или иные источники.

Начиная с версии JSF 2.0 предусмотрен лучший способ. Он позволяет помещать таблицы стилей, файлы JavaScript, изображения и другие файлы в каталог ресурсов корневого каталога конкретного веб-приложения. Подкаталоги этого каталога называются *библиотеками*. Предусмотрена возможность создавать любые необходимые библиотеки. В настоящей книге часто используются библиотеки `css`, `images` и `javascript`.

Для включения таблицы стилей применяется следующий тег:

```
<h:outputStylesheet library="css" name="styles.css"/>
```

Этот тег добавляет ссылку на форму

```
<link href="/context-root/faces/javax.faces.resource/styles.css?ln=css"
  rel="stylesheet" type="text/css"/>
```

в заголовок страницы.

Для включения ресурса сценария вместо этого следует использовать тег `outputScript`:

```
<h:outputScript name="jsf.js" library="javascript" target="head" />
```

Если целевым атрибутом является `head` или `body`, то сценарий добавляется к аспекту `"head"` или `"body"` компонента корневого каталога, а это означает, что он появляется в конце заголовка или текста в сформированном коде HTML. Если целевой элемент отсутствует, то сценарий вставляется в текущем местоположении.

Чтобы включить изображение из библиотеки, можно воспользоваться тегом `graphicImage`:

```
<h:graphicImage name="logo.png" library="images"/>
```

Для библиотек ресурсов и отдельных ресурсов предусмотрен механизм управления версиями, позволяющий добавлять к каталогам библиотек подкаталоги и размещать в них новые версии файлов. Имена подкаталогов представляют собой просто номера версий. Например, предположим, что имеются следующие каталоги:

```
resources/css/1_0_2
resources/css/1_1
```

В таком случае будет использоваться новейшая версия (`resources/css/1_1`). Следует учитывать, что имеется возможность добавлять новые версии библиотек в ходе работы программы. Аналогичным образом, можно добавлять новые версии отдельных ресурсов, но применяемая при этом схема именования является довольно странной. Ресурс заменяется каталогом с тем же именем, после чего имя версии используется в качестве имени файла. По желанию можно добавить расширение. Например:

```
resources/css/styles.css/1_0_2.css
resources/css/styles.css/1_1.css
```

Номера версий должны состоять из десятичных чисел, разделенных символами подчеркивания. Они сравниваются обычным образом, причем вначале сравниваются

основные номера версий, а дополнительные номера используются для устранения неопределенности.

Предусмотрен также механизм, обеспечивающий применение локализованных версий ресурсов. К сожалению, этот механизм сложно поддается освоению и не очень полезен. Локализованные ресурсы имеют префикс, такой как `resources/de_DE/images`, но он трактуется иначе по сравнению с суффиксом связки. При этом механизм возобновления нормальной работы не предусмотрен. Иными словами, если какое-то изображение не будет найдено с помощью префикса `resources/de_DE/images`, то префиксы `resources/de/images` и `resources/images` не проверяются.

Кроме того, префикс локали не рассматривается просто как текущая локаль. Вместо этого ее получение происходит с помощью необычного поиска, который вводится в действие с помощью приведенных ниже шагов.

1. Добавляется строка

   ```
   <message-bundle>name of a resource bundle used in your application</message-bundle>
   ```

   в элемент `application` файла `faces-config.xml`

2. В каждую локализованную версию рассматриваемой связки ресурсов вводится пара "имя–значение"

   ```
   javax.faces.resource.localePrefix=prefix
   ```

3. Соответствующие ресурсы размещаются в каталоге `resources/prefix/library/...`

Например, если используется связка сообщений `com.corejsf.messages`, а файл `com.corejsf.messages_de` содержит запись

```
javax.faces.resource.localePrefix=german
```

то ресурсы для немецкого языка помещаются в каталог `resources/german`. (В этом префиксе не обязательно должны использоваться стандартные коды языка и страны, а фактически даже не рекомендуется их задавать, чтобы у пользователей не возникали неоправданные ожидания в отношении применения стандартных ресурсов.)

Внимание! К сожалению, эта схема локализации не очень привлекательна с практической точки зрения. После определения префикса локали этот префикс используется для всех ресурсов. Предположим, что необходимо предусмотреть разные изображения для версий сайта на немецком и английском языках. В таком случае вам придется также продублировать все прочие ресурсы. Хотелось бы надеяться, что это положение будет исправлено в одной из будущих версий JSF.

События DHTML

Клиентская сценарная поддержка может применяться для решения задач любых типов, будь то проверка правильности синтаксиса или формирование ролловерных изображений, и может быть легко реализована с помощью технологии JSF. Атрибуты HTML, которые обеспечивают сценарную поддержку, такие как `onclick` и `onchange`, называются атрибутами событий динамического кода HTML (Dynamic HTML – DHTML). Технология JSF поддерживает атрибуты событий DHTML почти для всех тегов HTML, предназначенных для JSF. Эти атрибуты перечислены в табл. 4.7.

Атрибуты событий DHTML, перечисленные в табл. 4.7, позволяют связывать клиентские сценарии с событиями. Как правило, в качестве языка сценариев используется JavaScript, но можно использовать любой желаемый язык сценариев. Для того чтобы ознакомиться с дополнительными сведениями, обратитесь к спецификации HTML.

Таблица 4.7. Атрибуты событий DHTML[1]

Атрибут	Описание
onblur (16)	Элемент теряет фокус
onchange (11)	Изменяется значение элемента
onclick (17)	Кнопка мыши нажата над элементом
ondblclick (21)	Кнопка мыши дважды нажата над элементом
onfocus (16)	Элемент получает фокус
onkeydown (21)	Нажата клавиша
onkeypress (21)	Клавиша нажата, а затем отпущена
onkeyup (21)	Клавиша отпущена
onload (1)	Страница загружена
onmousedown (21)	Кнопка мыши нажата над элементом
onmousemove (21)	Указатель мыши перемещается над элементом
onmouseout (21)	Указатель мыши выходит из области элемента
onmouseover (21)	Указатель мыши перемещается в область элемента
onmouseup (21)	Кнопка мыши отпущена
onreset (1)	Произошел сброс формы
onselect (11)	Текст выбран в поле ввода
onsubmit (1)	Форма передана
onunload (1)	Страница выгружена

[1] (n) = количество тегов, применяемых с атрибутом

Совет. Многие разработчики, вскоре после того, как впервые начинают использовать JSF, приступают к расширению возможностей своих страниц JSF с помощью клиентских сценариев. Один из распространенных способов использования клиентских сценариев состоит в передаче запроса после изменения какого-то входного значения, чтобы прослушиватели изменений значений немедленно получали уведомления об изменениях, примерно таким образом:

```
<h:selectOneMenu onchange="submit()"...>
```

Панели

До сих пор для описания расположения компонентов в этой книге использовались таблицы HTML. Но задача создания разметки таблицы вручную является трудоемкой, поэтому теперь рассмотрим способ определенного упрощения этой сложной процедуры с помощью тега `h:panelGrid`, который создает разметку HTML, позволяющую располагать компоненты по строкам и столбцам.

На заметку! В теге `h:panelGrid` для создания макета используются таблицы HTML, но некоторые дизайнеры веб-страниц находят такой способ мало приемлемым. Безусловно, вместо тега `h:panelGrid` можно применять макет CSS. Возможно, в одной из будущих версий тега `h:panelGrid` появится вариант, позволяющий использовать также макет CSS.

Количество столбцов можно определить с помощью атрибута columns следующим образом:

```
<h:panelGrid columns="3">
  ...
</h:panelGrid>
```

Атрибут columns не является обязательным; если он не задан, то количество столбцов по умолчанию устанавливается равным 1. В теге h:panelGrid предусмотрено, что компоненты размещаются в столбцах слева направо и сверху вниз. Например, если определена сетка панели с тремя столбцами и девятью компонентами, то в конечном итоге будут созданы три строки, каждая из которых охватывает три столбца. Если определены три столбца и десять компонентов, то появятся четыре строки, а в последней строке только первый столбец будет содержать десятый компонент.

В табл. 4.8 перечислены атрибуты h:panelGrid.

Таблица 4.8. Атрибуты тега h:panelGrid

Атрибуты	Описание
bgcolor	Цвет фона для таблицы
border	Ширина границы таблицы
cellpadding	Заполнение вокруг ячеек таблицы
cellspacing	Интервал между ячейками таблицы
columnClasses	Разделенный запятыми список классов CSS для столбцов
columns	Количество столбцов в таблице
footerClass	Класс CSS для нижнего колонтитула таблицы
frame	Спецификация для сторон рамки, окружающей таблицу, которая должна быть изображена; допустимые значения: none, above, below, hsides, vsides, lhs, rhs, box, border
headerClass	Класс CSS для заголовка таблицы
rowClasses	Разделенный запятыми список классов CSS для строк
rules	Спецификация для линий, проведенных между ячейками; допустимые значения: groups, rows, columns, all
summary	Итоговое описание назначения таблицы и структуры, используемые для невизуальной обратной связи, такой как речь
captionClass JSF 1.2, captionStyle JSF 1.2	Класс CSS или стиль для надписи; для определения надписи на панели служит необязательный аспект "caption"
binding, id, rendered, value	Базовые атрибуты[1]
dir, lang, style, styleClass, title, width	HTML 4.0[2]
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	События DHTML[3]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.
[3] Дополнительные сведения об атрибутах событий DHTML см. в табл. 4.7 на стр. 114.

Предусмотрена возможность задавать классы CSS для различных частей таблицы: header, footer, rows и columns. Классы columnClasses и rowClasses определяют списки классов CSS, которые применяются к столбцам и строкам соответственно. Если эти списки содержат меньше имен классов по сравнению с количеством строк или столбцов, то классы CSS используются повторно. Это позволяет задавать классы так:

```
rowClasses="evenRows, oddRows"
```

и

```
columnClasses="evenColumns, oddColumns"
```

Атрибуты cellpadding, cellspacing, frame, rules и summary относятся к типу передаваемых атрибутов HTML, которые применяются только к таблицам. Дополнительную информацию см. в спецификации HTML 4.0.

Тег h:panelGrid часто используется вместе с тегом h:panelGroup, который группирует два или более компонента так, чтобы они рассматривались как один. Например, можно сгруппировать поле ввода и связанное с ним сообщение об ошибке, как в следующем примере:

```
<h:panelGrid columns="2">
  ...
  <h:panelGroup>
 <h:inputText id="name" value="#{user.name}">
 <h:message for="name"/>
  </h:panelGroup>
  ...
</h:panelGrid>
```

Группирование текстового поля и сообщения об ошибке приводит к тому, что они помещаются в одну и ту же ячейку таблицы. (Тег h:message будет рассматриваться в разделе "Сообщения" на стр. 158.) Тег h:panelGroup является несложным и имеет лишь небольшое количество атрибутов, которые приведены в табл. 4.9.

Таблица 4.9. Атрибуты для тега h:panelGroup

Атрибуты	Описание
layout JSF 1.2	Если атрибут layout имеет значение "block", то для размещения дочерних элементов используется тег div языка HTML; в противном случае используется тег span
binding, id, rendered	Базовые атрибуты[1]
style, styleClass	HTML 4.0[2]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.

Теги head, body и form

В табл. 4.10 перечислены атрибуты тегов h:head и h:body. Все они являются базовыми атрибутами или атрибутами HTML/DHTML.

Таблица 4.10. Атрибуты тегов h:head и h:body

Атрибуты	Описание
id, binding, rendered	Базовые атрибуты[1]

Окончание табл. 4.10

Атрибуты	Описание
dir, lang, только h:body — style, styleClass, target, title	Атрибуты HTML 4.0[2]
Только h:body: onclick, ondblclick, onkeydown, onkeypress, onkeyup, onload, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onunload	События DHTML[3]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.
[3] Дополнительные сведения об атрибутах событий DHTML см. в табл. 4.7 на стр. 114.

Веб-приложения действуют по принципу передачи формы, и приложения JSF не составляют исключения. В табл. 4.11 перечислены все атрибуты тега h:form.

Таблица 4.11. Атрибуты тега h:form

Атрибуты	Описание
prependId	true (значение по умолчанию), если идентификатор этой формы используется в качестве префикса для идентификаторов ее компонентов; false, если требуется предотвратить применение идентификатора формы как префикса (необходимость в этом возникает при использовании идентификатора в коде JavaScript)
binding, id, rendered	Базовые атрибуты[1]
accept, acceptcharset, dir, enctype, lang, style, styleClass, target, title	Атрибуты HTML 4.0[2]
onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onreset, onsubmit	События DHTML[3]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.
[3] Дополнительные сведения об атрибутах событий DHTML см. в табл. 4.7 на стр. 114.

Тег form языка HTML имеет атрибуты method и action, а тег h:form таковых не имеет. Предусмотрена возможность сохранять состояние на клиенте (этот вариант реализуется с помощью скрытого поля), поэтому отправка форм с применением метода GET запрещена. Для хранения содержимого этого скрытого поля может потребоваться довольно много места, что приведет к переполнению буфера для параметров запроса, поэтому все операции передачи форм JSF реализуются с помощью метода POST.

Необходимость в использовании атрибута anchor отсутствует, поскольку операции передачи форм JSF всегда приводят к поступлению данных на текущую страницу. (Переход к новой странице происходит после отправки данных формы.)

Тег h:form генерирует элемент формы HTML. Например, если на странице JSF с именем /index.xhtml используется тег h:form без атрибутов, то модуль подготовки формы к отображению вырабатывает такой код HTML:

```
<form id="_id0" method="post" action="/faces/index.xhtml"
  enctype="application/x-www-form-urlencoded">
```

Если атрибут id не задан явно, то значение вырабатывается реализацией JSF, точно так же, как и в отношении всех формируемых элементов HTML. Можно явно определить атрибут id для форм, чтобы на них можно было ссылаться в таблицах стилей или сценариях.

Элементы формы и код JavaScript

Технология JavaServer Faces полностью рассчитана на работу с серверными компонентами, но в ней также предусмотрена возможность работать с такими языками сценариев, как JavaScript. Например, в приложении, показанном на рис. 4.1, код JavaScript используется для подтверждения того, что содержимое поля пароля совпадает с содержимым поля подтверждения пароля. Если эти поля не согласуются, то отображается диалоговое окно JavaScript. В случае же совпадения их содержимого происходит передача формы.

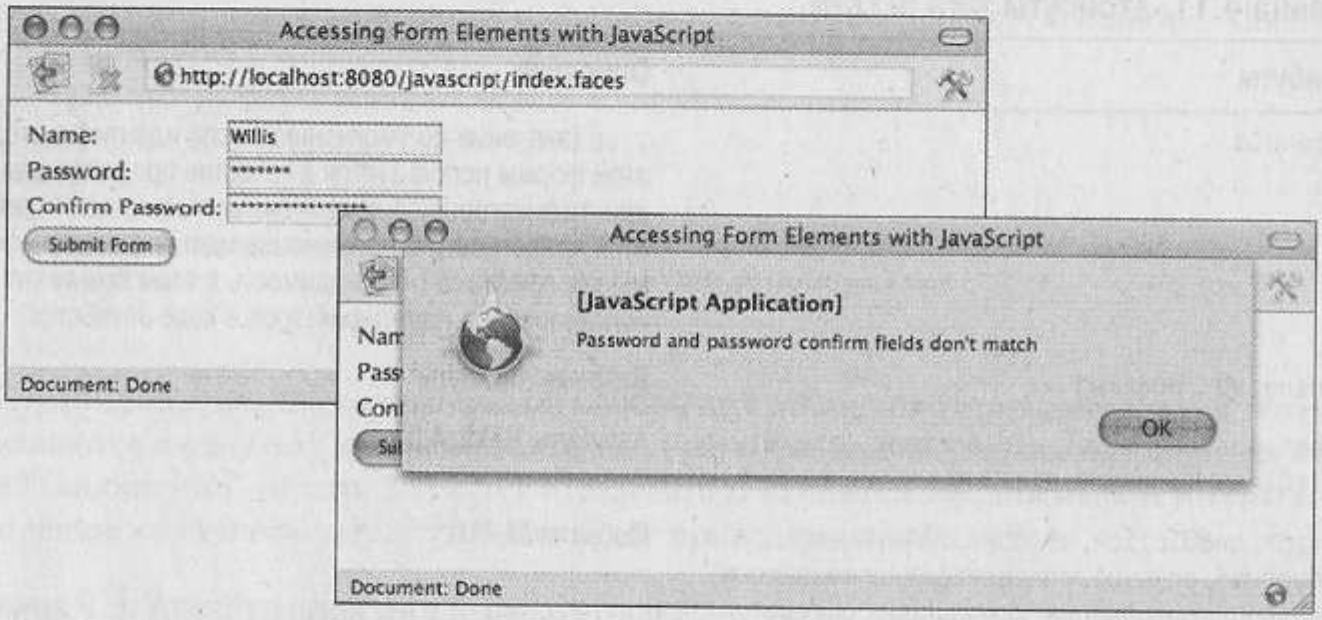


Рис. 4.1. Использование кода JavaScript для доступа к элементам формы

В рассматриваемом примере атрибут id применяется для назначения имен необходимым элементам HTML, чтобы к ним можно было обращаться с помощью кода JavaScript:

```
<h:form>
  ...
  <h:inputSecret id="password" .../>
  <h:inputSecret id="passwordConfirm" .../>
  ...
  <h:commandButton type="button" onclick="checkPassword(this.form)"/>
  ...
</h:form>
```

После того как пользователь щелкнет на кнопке, вызывается функция checkPassword языка JavaScript. Реализация этой функции приведена ниже.

```
function checkPassword(form) {
  var password = form[form.id + ":password"].value;
  var passwordConfirm = form[form.id + ":passwordConfirm"].value;
  if (password == passwordConfirm)
form.submit();
  else
 alert("Password and password confirm fields don't match");
}
```

Чтобы проще было понять синтаксис, используемый для доступа к элементам формы, необходимо рассмотреть фрагмент HTML, выработанный в показанном выше коде.

```
<form id="_id0" method="post"
  action="/javascript/faces/index.xhtml"
  enctype="application/x-www-form-urlencoded">
  ...
  <input id="_id0:password"
type="text" name="registerForm:password"/>
  ...
  <input type="button" name="_id0:_id5"
value="Submit Form" onclick="checkPassword(this.form)"/>
  ...
</form>
```

Все элементы управления формой, сгенерированные реализацией JSF, имеют имена, соответствующие следующему образцу:

```
formName:componentName
```

где `formName` представляет имя формы элемента управления, а `componentName` обозначает имя самого элемента управления. Если атрибуты `id` не заданы, реализация JSF создает идентификаторы автоматически. В рассматриваемом случае не было предусмотрено задание атрибутов `id` для формы. Поэтому для доступа к полю пароля в предыдущем примере в сценарий включено следующее выражение:

```
form[form.id + ":password"]
```

На заметку! Создается впечатление, что с каждой новой версией JSF значения идентификаторов, сгенерированных реализацией JSF, становятся все более сложными. В предыдущих выпусках они были достаточно простыми (наподобие `_id0`), но в более новых версиях уже используются такие идентификаторы, как `j_id2059540600_7ac21823`. Чтобы было проще разбираться в коде, в приведенных примерах применяются более простые идентификаторы.

Структура каталогов для приложения, показанного на рис. 4.1, приведена на рис. 4.2. Код страницы JSF содержится в листинге 4.1. Код сценария JavaScript, таблиц стилей и связки ресурсов приведен в листингах 4.2–4.4.

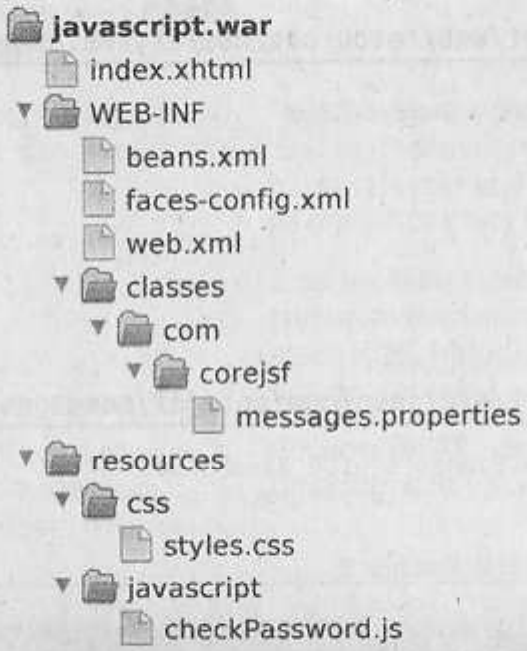


Рис. 4.2. Структура каталогов для примера применения JavaScript

Листинг 4.1. Файл javascript/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.windowTitle}</title>
      <h:outputStylesheet library="css" name="styles.css"/>
      <h:outputScript library="javascript" name="checkPassword.js"/>
   </h:head>
   <h:body>
      <h:form>
         <h:panelGrid columns="2" columnClasses="evenColumns, oddColumns">
            #{msgs.namePrompt}
            <h:inputText/>
            #{msgs.passwordPrompt}
            <h:inputSecret id="password"/>
            #{msgs.confirmPasswordPrompt}
            <h:inputSecret id="passwordConfirm"/>
         </h:panelGrid>
         <h:commandButton type="button" value="Submit Form"
                          onclick="checkPassword(this.form)"/>
      </h:form>
   </h:body>
</html>
```

Листинг 4.2. Файл javascript/web/resources/javascript/checkPassword.js

```
function checkPassword(form) {
   var password = form[form.id + ":password"].value;
   var passwordConfirm = form[form.id + ":passwordConfirm"].value;

   if (password == passwordConfirm)
      form.submit();
   else
      alert("Password and password confirm fields don't match");
}
```

Листинг 4.3. Файл javascript/web/resources/css/styles.css

```
.evenColumns {
   font-style: italic;
}

.oddColumns {
   padding-left: 1em;
}
```

Листинг 4.4. Файл javascript/src/java/com/corejsf/messages.properties

```
windowTitle=Accessing Form Elements with JavaScript
namePrompt=Name:
passwordPrompt=Password:
confirmPasswordPrompt=Confirm Password:
```

Текстовые поля и области

В основе большинства веб-приложений лежит ввод текста. Технология JSF поддерживает три разновидности средств ввода текста, представленные следующими тегами:

- `h:inputText`.
- `h:inputSecret`.
- `h:inputTextarea`.

Атрибуты, используемые в этих трех тегах, являются аналогичными, поэтому в табл. 4.12 перечислены атрибуты всех указанных тегов.

Таблица 4.12. Атрибуты тегов `h:inputText`, `h:inputSecret`, `h:inputTextarea` и `h:inputHidden`

Атрибуты	Описание
`cols`	Только для тега `h:inputTextarea` — количество столбцов
`immediate`	Запуск средств проверки правильности на раннем этапе жизненного цикла
`redisplay`	Только для тега `h:inputSecret` — при значении, равном `true`, значение поля ввода повторно отображается после перезагрузки веб-страницы
`required`	Требует ввода в компоненте при передаче формы
`rows`	Только для тега `h:inputTextarea` — количество строк
`valueChangeListener`	Заданный прослушиватель, который получает уведомление об изменении значений
`label` JSF 1.2	Описание компонента, предназначенное для использования в сообщениях об ошибках. Не относится к тегу `h:inputHidden`
`binding`, `converter`, `id` `converterMessage` JSF 1.2, `rendered`, `required`, `value`, `requiredMessage` JSF 1.2, `validator`, `validatorMessage` JSF 1.2	Базовые атрибуты[1]
`accesskey`, `alt`, `dir`, `disabled`, `lang`, `maxlength`, `readonly`, `size`, `style`, `styleClass`, `tabindex`, `title`	Передаваемые атрибуты HTML 4.0[2] — атрибуты `alt`, `maxlength` и `size` не относятся к тегу `h:inputTextarea`. Ни один из указанных атрибутов не относится к тегу `h:inputHidden`
`Autocomplete`	Если значение равно `"off"`, подготовить к отображению нестандартный атрибут `autocomplete="off"` языка HTML (только для тегов `h:inputText` и `h:inputSecret`)
`onblur`, `onchange`, `onclick`, `ondblclick`, `onfocus`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `onselect`	События DHTML. Ни один из указанных атрибутов не относится к тегу `h:inputHidden`[3]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.
[3] Дополнительные сведения об атрибутах событий DHTML см. в табл. 4.7 на стр. 114.

Все три тега имеют атрибуты immediate, required, value и valueChangeListener. Атрибут immediate используется в основном для отслеживания изменений значений, которые затрагивают пользовательский интерфейс, и редко используется в трех указанных тегах. Вместо этого он чаще всего применяется в других компонентах ввода, таких как меню и окна списка. Дополнительная информация об атрибуте immediate приведена в разделе "Непосредственные компоненты" главы 8 на стр. 281.

Каждый из следующих трех атрибутов, рассматриваемых в табл. 4.12, применим только к одному тегу: cols, rows и redisplay. Атрибуты rows и cols используются в теге h:inputTextarea для задания соответственно количества строк и столбцов в текстовой области. Атрибут redisplay, используемый в теге h:inputSecret, является логическим и определяет, должно ли секретное поле сохранять свое значение (и поэтому повторно отображать его) после повторной передачи формы с этим полем.

В табл. 4.13 приведены примеры использования тегов h:inputSecret и h:inputText.

Таблица 4.13. Примеры использования тегов h:inputText и h:inputSecret

Пример	Результат
`<h:inputText value="#{form.testString}" readonly="true"/>`	12345678901234567890
`<h:inputSecret value="#{form.passwd}" redisplay="true"/>`	********** (отображается после неудачной передачи формы)
`<h:inputSecret value="#{form.passwd}" redisplay="false"/>`	(отображается после неудачной передачи формы)
`<h:inputText value="inputText" style="color: Yellow; background: Teal;"/>`	inputText
`<h:inputText value="1234567" size="5"/>`	12345
`<h:inputText value="1234567890" maxlength="6" size="10"/>`	123456

В первом примере из табл. 4.13 вырабатывается следующий код HTML:

```
<input type="text" name="_id0:_id4" value="12345678901234567890" readonly="readonly"/>
```

Поле ввода предназначено только для чтения, поэтому для рассматриваемого бина формы определен лишь метод получения свойства:

```
private String testString = "12345678901234567890";
public String getTestString() {
  return testString;
}
```

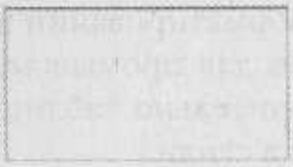
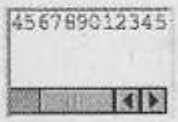
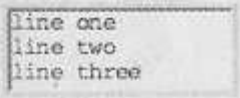
В примерах с тегом h:inputSecret иллюстрируется использование атрибута redisplay. Если этот атрибут имеет значение true, то текстовое поле сохраняет свое значение от одного запроса к другому, поэтому содержимое этого поля повторно отображается после перезагрузки страницы. Если значение атрибута redisplay равно false, то значение поля отбрасывается и не отображается повторно.

Атрибут size определяет количество видимых символов в текстовом поле. Но большинство шрифтов не являются равноширинными, поэтом атрибут size не может быть точным, как показано в пятом примере в табл. 4.13, где задан размер 5, но отображается шесть символов. Атрибут maxlength определяет максимальное количество символов, отображаемых в текстовом поле. Этот атрибут является точным. И size, и maxlength относятся к типу передаваемых атрибутов HTML.

Примеры использования тега h:inputTextarea приведены в табл. 4.14.

Тег h:inputTextarea имеет атрибуты cols и rows, позволяющие определить количество столбцов и строк в текстовой области. Атрибут cols напоминает атрибут size тега h:inputText и также является неточным.

Таблица 4.14. Примеры использования тега h:inputTextarea

Пример	Результат
`<h:inputTextarea rows="5"/>`	
`<h:inputTextarea cols="5"/>`	
`<h:inputTextarea value="123456789012345" rows="3" cols="10"/>`	
`<h:inputTextarea value="#{form.dataInRows}" rows="2" cols="15"/>`	

Если для размещения значения для тега h:inputTextarea задана одна длинная строка, то она полностью размещается в одной строке текста, как показано в третьем примере в табл. 4.14. Если же требуется разбить строковые данные на несколько строк текста, то можно вставить символы обозначения конца строки (\n) для принудительного задания конца текстовых строк. В частности, в последнем примере табл. 4.14 демонстрируется доступ к свойству dataInRows вспомогательного бина. Это свойство реализовано следующим образом:

```
private String dataInRows = "line one\nline two\nline three";
public void setDataInRows(String newValue) {
  dataInRows = newValue;
}
public String getDataInRows() {
  return dataInRows;
}
```

Скрытые поля

В технологии JSF предусмотрена поддержка для скрытых полей на основе тега h:inputHidden. Скрытые поля часто используются в действиях, реализованных с помощью кода JavaScript, которые предназначены для отправки данных обратно на сервер. Тег h:inputHidden имеет такие же атрибуты, как и другие теги ввода, но не поддерживает стандартные теги HTML и DHTML.

Использование текстовых полей и областей

Далее будет рассматривается законченный пример, в котором используются текстовые поля и области. В приложении, показанном на рис. 4.3, применяются теги h:inputText, h:inputSecret и h:inputTextarea для сбора персональной информации, предоставляемой пользователем. Значения этих компонентов привязаны к свойствам

бина, доступ к которым предоставляется на странице thankYou.xhtml, повторно отображающей сведения, введенные пользователем.

В приведенном ниже приложении заслуживают внимания три особенности. Во-первых, страницы JSF ссылаются на бин user (com.corejsf.UserBean). Во-вторых, тег h:inputTextarea передает текст, введенный в текстовой области, в модель (в данном случае в бин user) как одну строку со вставленными символами обозначения новой строки (\n). В приложении эта строка отображается с использованием элемента <pre> языка HTML в целях сохранения ее форматирования. В-третьих, для иллюстрации при форматировании вывода используется атрибут style. В приложении, предназначенном для промышленной эксплуатации, предпочтительным является применение исключительно таблиц стилей, что позволяет упростить внесение глобальных изменений в стили.

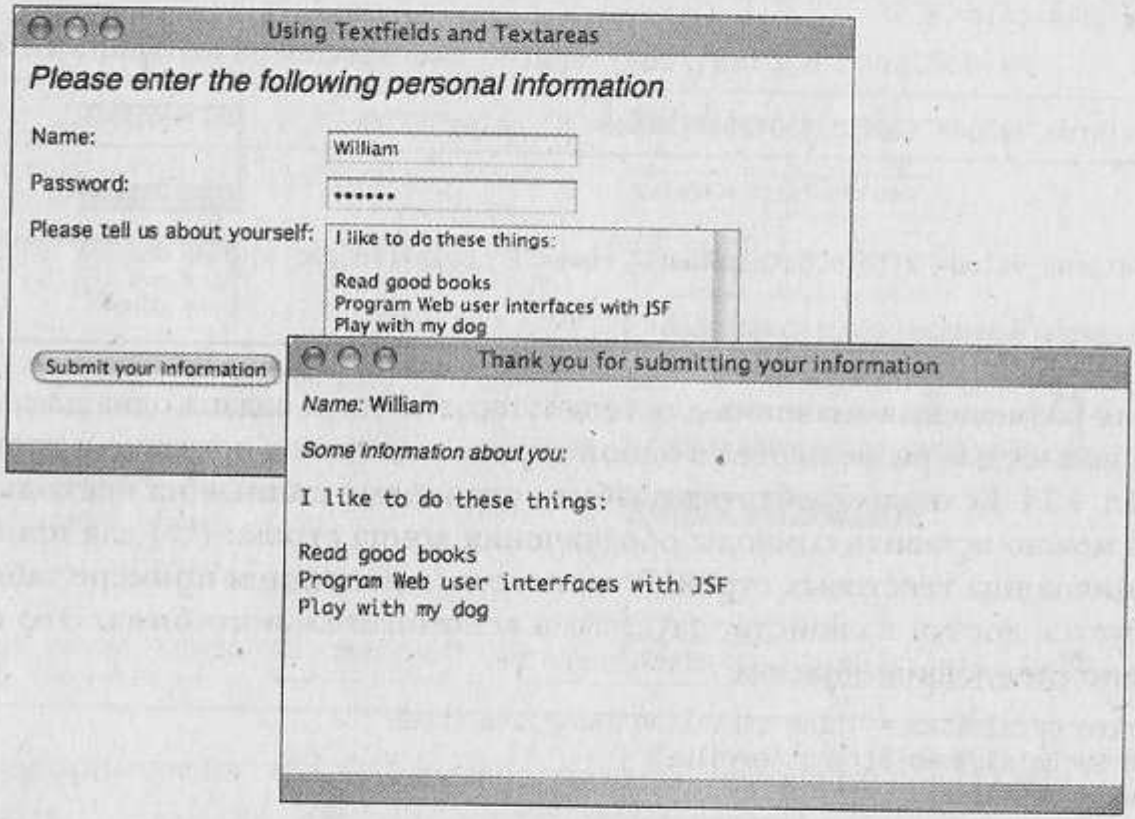


Рис. 4.3. Использование текстовых полей и областей

На рис. 4.4 показана структура каталогов для приложения, приведенного на рис. 4.3. В листингах 4.5–4.8 содержится код соответствующих страниц JSF, управляемых бинов, файла конфигурации faces и связки ресурсов.

Листинг 4.5. Файл personalData/web/index.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:h="http://java.sun.com/jsf/html">
6.    <h:head>
7.       <title>#{msgs.indexWindowTitle}</title>
8.    </h:head>
9.    <h:body>
10.      <h:outputText value="#{msgs.indexPageTitle}"
11.                    style="font-style: italic; font-size: 1.5em"/>
12.      <h:form>
13.         <h:panelGrid columns="2">
```

```
            #{msgs.namePrompt}
            <h:inputText value="#{user.name}"/>
            #{msgs.passwordPrompt}
            <h:inputSecret value="#{user.password}"/>
            #{msgs.tellUsPrompt}
            <h:inputTextarea value="#{user.aboutYourself}" rows="5" cols="35"/>
          </h:panelGrid>
          <h:commandButton value="#{msgs.submitPrompt}" action="thankYou"/>
        </h:form>
    </h:body>
</html>
```

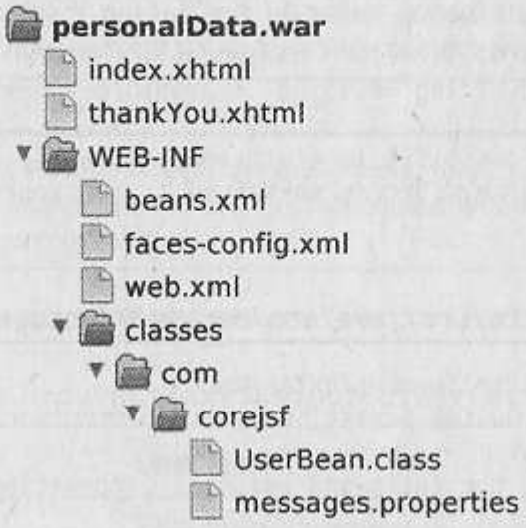


Рис. 4.4. Структура каталогов для примера с текстовыми полями и областями

Листинг 4.6. Файл personalData/web/thankYou.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
      <title>#{msgs.thankYouWindowTitle}</title>
    </h:head>
    <h:body>
      <h:outputText value="#{msgs.namePrompt}" style="font-style: italic"/>
      #{user.name}
      <br/>
      <h:outputText value="#{msgs.aboutYourselfPrompt}" style="font-style: italic"/>
      <br/>
      <pre>#{user.aboutYourself}</pre>
    </h:body>
</html>
```

Листинг 4.7. Файл personalData/src/java/com/corejsf/UserBean.java

```
package com.corejsf;

import java.io.Serializable;

import javax.inject.Named;
```

```
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;

@Named("user") // или @ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable {
   private String name;
   private String password;
   private String aboutYourself;

   public String getName() { return name; }
   public void setName(String newValue) { name = newValue; }

   public String getPassword() { return password; }
   public void setPassword(String newValue) { password = newValue; }

   public String getAboutYourself() { return aboutYourself; }
   public void setAboutYourself(String newValue) { aboutYourself = newValue; }
}
```

Листинг 4.8. Файл personalData/src/java/com/corejsf/messages.properties

```
indexWindowTitle=Using Textfields and Textareas
thankYouWindowTitle=Thank you for submitting your information
thankYouPageTitle=Thank you!
indexPageTitle=Please enter the following personal information
namePrompt=Name:
passwordPrompt=Password:
tellUsPrompt=Please tell us about yourself:
aboutYourselfPrompt=Some information about you:
submitPrompt=Submit your information
```

Вывод на экран текста и изображений

Для вывода на экран текста и изображений в приложениях JSF используются следующие теги:

- h:outputText.
- h:outputFormat.
- h:graphicImage.

Тег h:outputText является одним из наиболее простых тегов JSF. Он обладает лишь небольшим количеством атрибутов и, как правило, не формирует элемент HTML. Вместо этого он вырабатывает простой текст, за одним исключением: если задан атрибут style или styleClass, то тег h:outputText формирует элемент span языка HTML.

После выхода версии JSF 2.0 необходимость в использовании тега h:outputText возникает редко, поскольку теперь в код страницы можно просто вставлять выражения значения, такие как #{msgs.namePrompt}. Как правило, тег h:outputText применяется при следующих обстоятельствах.

- Для формирования стилизованного вывода.
- В сетке панели, чтобы сформированный текст рассматривался как содержимое одной ячейки сетки.
- Для формирования разметки HTML.

Теги h:outputText и h:outputFormat имеют один атрибут, escape, являющийся уникальным среди всех тегов JSF. По умолчанию атрибут escape имеет значение true, что вызывает преобразование символов <, > и & в символьные сущности <, > и & соответственно. Преобразование этих символов позволяет предотвратить атаки на основе сценарной поддержки, направленные с одного сайта на другой. (В документе http://www.cert.org/advisories/CA-2000-02.html содержится дополнительная информация об атаках на основе сценарной поддержки, направленных с одного сайта на другой.) Если формирование разметки HTML должно осуществляться программным путем, то следует присвоить этому атрибуту значение false.

На заметку! Атрибут value тега h:outputText не может содержать символы <. Единственный способ выработки разметки HTML с помощью тега h:outputText состоит в использовании выражения значения.

На заметку! Если на страницу включено выражение значения, такое как #{msgs.namePrompt}, то результирующее значение всегда экранируется. Если есть необходимость формировать разметку HTML, то следует использовать тег h:outputText.

В табл. 4.15 перечислены все атрибуты тега h:outputText.

Таблица 4.15. Атрибуты тегов h:outputText и h:outputFormat

Атрибуты	Описание
escape	Если задано значение true (значение по умолчанию), то символы <, > и & экранируются
binding, converter, id, rendered, value	Базовые атрибуты[1]
style, styleClass, title, dir JSF 1.2, lang JSF 1.2	HTML 4.0[2]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.

Тег h:outputFormat форматирует составное сообщение с помощью параметров, указанных в тексте этого тега, например:

```
<h:outputFormat value="{0} is {1} years old">
  <f:param value="Bill"/>
  <f:param value="38"/>
</h:outputFormat>
```

В приведенном выше фрагменте кода составным сообщением "is {0} is {1} years old" и параметрами, заданными с помощью тегов f:param, являются Bill и 38. С помощью предыдущего фрагмента кода формируется следующий вывод: Bill is 38 years old. В теге h:outputFormat используется экземпляр java.text.MessageFormat для форматирования создаваемого им вывода.

Тег h:graphicImage формирует элемент img языка HTML. Предусмотрена возможность указать местонахождение изображения с помощью атрибута url или value как путь относительно контекста, иными словами, относительно корневого каталога контекста веб-приложения. Со времени выхода версии JSF 2.0 появилась возможность размещать изображения в каталоге resources и задавать библиотеку и имя:

```
<h:graphicImage library="images" name="de_flag.gif"/>
```

В этом примере изображение находится в файле resources/images/de_flag.gif. Еще один вариант состоит в применении примерно такого кода:

```
<h:graphicImage url="/resources/images/de_flag.gif"/>
```

Можно также использовать карту ресурсов:

```
<h:graphicImage value="#{resources['images:de_flag.gif']}"/>
```

В табл. 4.16 перечислены все атрибуты тега h:graphicImage.

Таблица 4.16. Атрибуты тега h:graphicImage

Атрибуты	Описание
binding, id, rendered, value	Базовые атрибуты[1]
alt, dir, height, ismap, lang, longdesc, style, styleClass, title, url, usemap, width	HTML 4.0[2]
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	События DHTML[3]
library, name JSF 2.0	Библиотека ресурсов и имя для этого изображения

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.
[3] Дополнительные сведения об атрибутах событий DHTML см. в табл. 4.7 на стр. 114.

В табл. 4.17 показаны некоторые примеры использования тегов h:outputText и h:graphicImage.

Таблица 4.17. Примеры использования тегов h:outputText и h:graphicImage

Пример	Результат
`<h:outputText value="#{form.testString}"/>`	12345678901234567890
`<h:outputText value="Number #{form.number}"/>`	Number 1000
`<h:outputText value="#{form.htmlCode}" escape="false"/>`, где метод `getHtmlCode` возвращает строку `"<input type='text' value='hello'/>"`	hello
`<h:outputText value="#{form.htmlCode}"/>` где метод `getHtmlCode` возвращает строку `"<input type='text' value='hello'/>"`	`<input type="text" value="hello">`
`<h:graphicImage value="/tjefferson.jpg"/>`	
`<h:graphicImage library="images" name="tjefferson.jpg" style="border: thin solid black"/>`	

В третьем и четвертом примерах табл. 4.17 иллюстрируется использование атрибута escape. Если значением тега h:outputText является `<input type='text' value='hello'/>`, а атрибут escape имеет значение false (как и в третьем примере табл. 4.17), то тег h:outputText формирует элемент input языка HTML. Иногда элементы HTML формируются вопреки намерениям разработчика, а это — именно та разновидность злоупотребления, которая позволяет недобросовестным лицам проводить атаки на основе

сценарной поддержки, направленные с одного сайта на другой. Если атрибут escape задан равным true (как в четвертом примере табл. 4.17), то вывод преобразуется в безвредный текст, что исключает возможность совершения потенциальной атаки.

Последние два примера табл. 4.17 показывают, как использовать тег h:graphicImage.

Кнопки и ссылки

В веб-приложениях широко применяются кнопки и ссылки, поэтому в технологии JSF предусмотрены следующие теги для их поддержки:

- h:commandButton.
- h:commandLink.
- h:button.
- h:link.
- h:outputLink.

Теги h:commandButton и h:commandLink являются основными компонентами, применяемыми для создания средств навигации в приложении JSF. После активизации кнопки или ссылки происходит передача данных формы обратно на сервер с помощью запроса POST.

В версии JSF 2.0 впервые введены компоненты h:link и h:button. Эти компоненты также подготавливают кнопки и ссылки к отображению, но щелчки на создаваемых при этом элементах, в отличие от указанного выше, приводят к созданию запроса GET, обеспечивающего формирование закладки. Об этом механизме см. в главе 3.

Тег h:outputLink формирует элемент anchor языка HTML, который указывает на ресурс, такой как изображение или веб-страница. Щелчок на этой сформированной ссылке приводит к переходу к указанному ресурсу, не требуя дальнейшего взаимодействия с платформой JSF. Такие ссылки являются в наибольшей степени приемлемыми для перехода на другие веб-сайты.

В табл. 4.18 перечислены атрибуты, совместно используемые в тегах h:commandButton, h:commandLink, h:button и h:link.

Таблица 4.18. Атрибуты тегов h:commandButton, h:commandLink, h:button и h:link

Атрибут	Описание
action (только теги h:commandButton и h:commandLink)	Если значение задано в виде строки: непосредственно задает результат, используемый обработчиком навигации, для определения страницы JSF, подлежащей загрузке в следующую очередь в результате активизации кнопки или ссылки. Если указано как выражение метода, то метод имеет следующую сигнатуру: String methodName(). Строка представляет результат. Если опущено, то активизация кнопки или ссылки приводит к повторному отображению текущей страницы
Outcome (только h:button и h:link)	Значение outcome, используемое обработчиком навигации для определения целевого представления при подготовке компонента к отображению
fragment (только h:button и h:link)	Фрагмент, который должен быть добавлен к целевому URL. Разделитель # применяется автоматически и не должен быть включен во фрагмент

Окончание табл. 4.1

Атрибут	Описание
actionListener	Выражение метода, которое ссылается на метод с этой сигнатурой: `void methodName(ActionEvent)`
image (только h:commandButton и h:button)	Путь к изображению, показанному на кнопке. Если этот атрибут задан, то типом ввода HTML становится изображение. Если путь начинается с символа /, то в качестве префикса применяется корневой каталог контекста приложения
immediate	Значение Boolean. Если задано значение false (значение по умолчанию), то прослушиватели actions и action вызываются в конце жизненного цикла обработки запроса; при значении true прослушиватели actions и action вызываются в начале жизненного цикла. Дополнительная информация об атрибуте immediate приведена в главе 8
type	Для h:commandButton — тип сформированного элемента input: button, submit или reset. Значением по умолчанию, если не определен атрибут image, является submit. Для h:commandLink и h:link — тип содержимого связанного ресурса; например, text/html, image/gif или audio/basic
value	Метка, отображаемая кнопкой или ссылкой. Предусмотрена возможность определить строку или выражение значения
binding, id, rendered	Базовые атрибуты[1]
accesskey, charset (только h:commandLink и h:link), coords (только h:commandLink и h:link), dir JSF 1.1, disabled (h:commandButton только в JSF 1.1), hreflang (только h:commandLink и h:link), lang, rel (только h:commandLink и h:link), rev (только h:commandLink и h:link), shape (только h:commandLink и h:link), style, styleClass, tabindex, target (только h:commandLink и h:link), title	HTML 4.0[2]
onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	События DHTML[3]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.
[3] Дополнительные сведения об атрибутах событий DHTML см. в табл. 4.7 на стр. 114.

Использование кнопок

Теги h:commandButton и h:button формируют элемент input языка HTML, типом которого становится button, image, submit или reset, в зависимости от заданного атрибута. В табл. 4.19 приведены некоторые способы использования этих тегов.

В третьем примере табл. 4.19 формируется нажимная кнопка (элемент input языка HTML, типом которого является кнопка), не применяемая для передачи форм. Единственный способ закрепления определенного поведения за нажимной кнопкой со-

стоит в задании сценария для одного из атрибутов событий DHTML, как было сделано в примере для события onclick.

Таблица 4.19. Примеры применения тегов h:commandButton и h:button

Пример	Результат
`<h:commandButton value="submit" type="submit" action="#{form.submitAction}"/>`	submit
`<h:commandButton value="reset" type="reset"/>`	reset
`<h:commandButton value="click this button..." onclick="alert('button clicked')" type="button"/>`	click this button to execute JavaScript
`<h:commandButton value="disabled" disabled="#{not form.buttonEnabled}"/>`	disabled
`<h:button value="#{form.buttonText}" outcome="#{form.pressMeOutcome}"/>`	press me

Внимание! В версии JSF 1.1 была допущена несогласованность в обработке путей к изображениям между тегами h:graphicImage и h:commandButton. Тег h:graphicImage, в отличие от h:commandButton, автоматически добавляет корневой каталог контекста. В качестве примера ниже показано, как в приложении myApp нужно было определять одно и то же изображение для каждого из тегов.

```
<h:commandButton image="/myApp/imageFile.jpg"/> <!-- JSF 1.1 -->
      <h:graphicImage value="/imageFile.jpg"/>
```

Этот недостаток приводил к тому, что в процессе разработки создавались дополнительные трудности, поскольку необходимо было задавать корневой каталог контекста для каждой такой страницы. В версии JSF 1.2 поведение тега h:commandButton изменилось. Теперь корневой каталог контекста добавляется автоматически, если обозначение пути начинается с символа /.

Но определенные трудности остались, поскольку эта функция плохо взаимодействует с картой ресурсов. Невозможно использовать следующую конструкцию:

```
<h:commandButton image="#{resources['images:imageFile.jpg']}"/>
```

поскольку строка, возвращенная картой ресурсов, начинается с префикса /context-root.

При непосредственном использовании этой строки был бы получен такой результат:

```
<input type="image" src="/context-root/context-root/..."/>
```

Теги h:commandLink и h:link формируют элемент anchor языка HTML, который действует как кнопка передачи формы. В табл. 4.20 приведены некоторые примеры их применения.

Таблица 4.20. Примеры применения тегов h:commandLink и h:link

Пример	Результат
`<h:commandLink>register</h:commandLink>`	register

`<h:commandLink style="font-style: italic">`
`  #{msgs.linkText}`
`</h:commandLink>`
	click here to register

`<h:commandLink>`
` #{msgs.linkText}`
` <h:graphicImage value="/registration.jpg"/>`
`</h:commandLink>`
	click here to register

Окончание табл. 4.20

Пример	Результат
`<h:commandLink value="welcome" actionListener="#{form.useLinkValue}" action="#{form.followLink}"/>`	welcome
`<h:link value="welcome" outcome="#{form.welcomeOutcome}"> <f:param name="id" value="#{form.userId}"/> </h:link>`	welcome

Теги h:commandLink и h:link формируют код JavaScript, в результате применения которого ссылки действуют как кнопки. В качестве образца ниже приведен код HTML, сформированный в первом примере из табл. 4.20.

```
<a href="#" onclick="document.forms['_id0']['_id0:_id2'].value='_id0:_id2';
  document.forms['_id0'].submit()">register</a>
```

Если пользователь щелкает на этой ссылке, то элементу anchor присваивается значение идентификатора клиента из тега h:commandLink и происходит передача включающей формы. Эта передача приводит к запуску жизненного цикла JSF, а поскольку атрибут href задан равным "#", то перегружается текущая страница, если только действие, связанное с этой ссылкой, не возвращает ненулевой результат.

В текст тега h:commandLink можно поместить произвольное количество дочерних тегов, причем каждый соответствующий элемент HTML является частью ссылки. Поэтому, например, после щелчка либо на тексте, либо на изображении в третьем примере табл. 4.20 происходит передача формы, связанной со ссылкой.

В предпоследнем примере, приведенном в табл. 4.20, за ссылкой закрепляется не только действие, но и прослушиватель action. Прослушиватели action описаны в разделе "События действий" главы 8 на стр. 274.

В последнем примере табл. 4.20 в текст тега h:link внедряется тег f:param. После щелчка на этой ссылке происходит создание с помощью ссылки параметра запроса, имя и значение которого задаются тегом f:param. В главе 2 было показано, как может осуществляться обработка параметров запроса. В сочетании с тегами h:commandLink или h:commandButton также можно использовать параметры запроса. Соответствующие примеры приведены в разделе "Передача данных из пользовательского интерфейса на сервер" главы 8 на стр. 285.

Как и теги h:commandLink и h:link, тег h:outputLink формирует элемент anchor языка HTML. Но в отличие от h:commandLink тег h:outputLink не формирует код JavaScript, который бы обеспечил выполнение ссылкой действий, присущих для кнопки передачи формы. Величина атрибута значения h:outputLink используется для формирования атрибута href анкера, а содержимое текста тега h:outputLink служит для заполнения текста элемента anchor. В табл. 4.21 перечислены все атрибуты тега h:outputLink, а в табл. 4.22 показаны некоторые примеры применения тега h:outputLink.

Таблица 4.21. Атрибуты тега h:outputLink

Атрибуты	Описание
binding, converter, id, lang, rendered, value	Базовые атрибуты[1]
accesskey, charset, coords, dir, disabled JSF 1.2, hreflang, lang, rel, rev, shape, style, styleClass, tabindex, target, title, type	HTML 4.0[2]

Окончание табл. 4.21

Атрибуты	Описание
onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	События DHTML[3]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.
[3] Дополнительные сведения об атрибутах событий DHTML см. в табл. 4.7 на стр. 110.

Таблица 4.22. Примеры применения тега `h:outputLink`

Пример	Результат

`<h:outputLink value="http://java.net">`
`<h:graphicImage value="java-dot-net.jpg"/>`
`<h:outputText value="java.net"/>`
`</h:outputLink>`
	java.net

`<h:outputLink value="#{form.welcomeURL}">`
`#{form.welcomeLinkText}`
`</h:outputLink>`
	go to welcome page

`<h:outputLink value="#introduction">`
`<h:outputText value="Introduction"`
`style="font-style: italic"/>`
`</h:outputLink>`
	Introduction

`<h:outputLink value="#conclusion"`
`title="Go to the conclusion">`
`Conclusion`
`</h:outputLink>`
	Conclusion
	Go to the conclusion

`<h:outputLink value="#toc"`
`title="Go to the table of contents">`
`<h2>Table of Contents</h2>`
`</h:outputLink>`
	Table of Contents

Первый пример, приведенный в табл. 4.22, показывает создание ссылки на `http://java.net`. Во втором примере используются свойства, хранимые в бине для URL и текста ссылки. Эти свойства реализуются так:

```
private String welcomeURL = "/outputLinks/faces/welcome.jsp";
public String getWelcomeURL() {
  return welcomeURL;
}
private String welcomeLinkText = "go to welcome page";
public String getWelcomeLinkText() {
  return welcomeLinkText;
}
```

В последних трех примерах табл. 4.22 показаны ссылки на именованные анкеры на той же странице JSF. Эти анкеры выглядят следующим образом:

```
<a name="introduction">Introduction</a>
...
<a name="conclusion">Conclusion</a>
...
<a name="toc">Table of Contents</a>
...
```

Внимание! Если используется версия JSF 1.1, то при возникновении необходимости разместить текст в теге приходится прибегать к тегу `f:verbatim`. В частности, последний пример табл. 4.22 должен был бы иметь такой вид:

```
<h:outputLink...><f:verbatim>Table of Contents</f:verbatim></h:outputLink>
```

В противном случае при использовании версии JSF 1.1 текст появился бы вне ссылки. Чтобы выйти из этого положения, необходимо было размещать текст в другом компоненте, таком как `h:outputText` или `f:verbatim`. Этот недостаток был устранен в версии JSF 1.2.

Использование командных ссылок

Выше были приведены подробные сведения о тегах JSF для кнопок и ссылок, поэтому теперь можно перейти к рассмотрению законченного примера. На рис. 4.5 показано приложение, описанное в разделе "Использование текстовых полей и текстовых областей" на стр. 123, с двумя ссылками, которые позволяют выбирать одну из двух локалей: для немецкого или английского языка. После активизации одной из ссылок определенное действие изменяет локаль представления, а реализация JSF перезагружает текущую страницу.

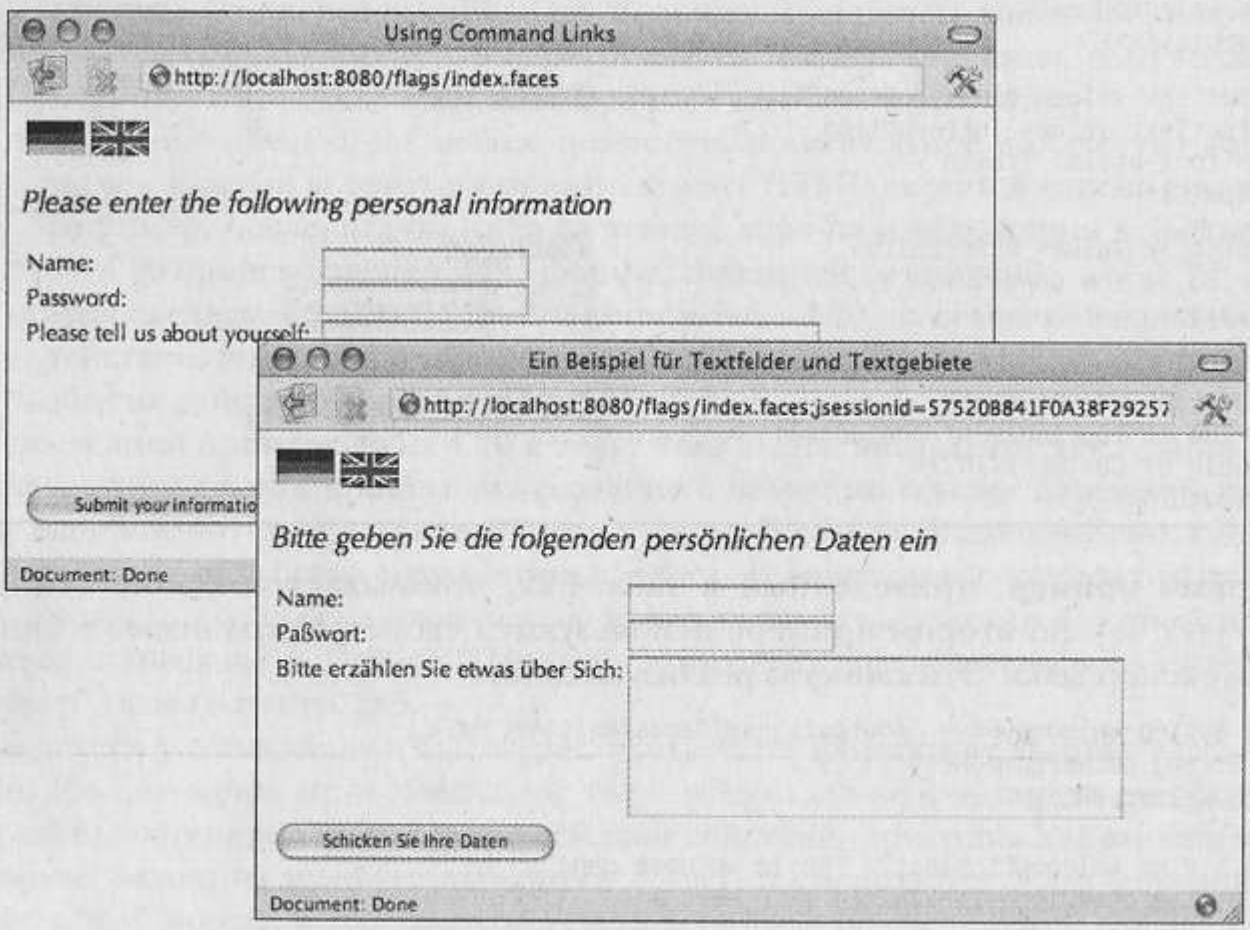


Рис. 4.5. Использование командных ссылок для изменения локали

Реализация ссылок осуществляется так:

```
<h:commandLink action="#{localeChanger.englishAction}">
  <h:graphicImage library="images" name="en_flag.gif" style="border: 0px" />
</h:commandLink>
```

Обе ссылки определяют изображение и метод действия. Метод перехода на локаль английского языка выглядит следующим образом:

```
public class LocaleChanger {
   ...
```

```
    public String englishAction() {
        FacesContext context = FacesContext.getCurrentInstance();
        context.getViewRoot().setLocale(Locale.ENGLISH);
        return null;
    }
}
```

Для этого действия не были предусмотрены какие-либо средства навигации, поэтому реализация JSF перезагружает текущую страницу после передачи формы. После перезагрузки страницы происходит ее локализация для английского или немецкого языка и соответствующим образом осуществляется повторное отображение страницы.

На рис. 4.6 показана структура каталогов для приложения, а в листингах 4.9–4.11 приведены соответствующие страницы JSF и классы Java.

Рис. 4.6. Структура каталогов для примера с флагами

Листинг 4.9. Файл `flags/web/index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:h="http://java.sun.com/jsf/html">
6.    <h:head>
7.       <title>#{msgs.indexWindowTitle}</title>
8.    </h:head>
9.    <h:body>
10.      <h:form>
11.         <h:commandLink action="#{localeChanger.germanAction}">
12.            <h:graphicImage library="images" name="de_flag.gif"
13.                            style="border: 0px; margin-right: 1em;"/>
14.         </h:commandLink>
15.         <h:commandLink action="#{localeChanger.englishAction}">
16.            <h:graphicImage library="images"
17.                            name="en_flag.gif" style="border: 0px"/>
```

```
18.          </h:commandLink>
19.          <p><h:outputText value="#{msgs.indexPageTitle}"
20.          style="font-style: italic; font-size: 1.3em"/></p>
21.          <h:panelGrid columns="2">
22.             #{msgs.namePrompt}
23.             <h:inputText value="#{user.name}"/>
24.             #{msgs.passwordPrompt}
25.             <h:inputSecret value="#{user.password}"/>
26.             #{msgs.tellUsPrompt}
27.             <h:inputTextarea value="#{user.aboutYourself}" rows="5" cols="35"/>
28.          </h:panelGrid>
29.          <h:commandButton value="#{msgs.submitPrompt}" action="thankYou"/>
30.       </h:form>
31.    </h:body>
32. </html>
```

Листинг 4.10. Файл flags/src/java/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. import javax.inject.Named;
6.    // или import javax.faces.bean.ManagedBean;
7. import javax.enterprise.context.SessionScoped;
8.    // или import javax.faces.bean.SessionScoped;
9.
10. @Named("user") // или @ManagedBean(name="user")
11. @SessionScoped
12. public class UserBean implements Serializable {
13.    private String name;
14.    private String password;
15.    private String aboutYourself;
16.
17.    public String getName() { return name; }
18.    public void setName(String newValue) { name = newValue; }
19.
20.    public String getPassword() { return password; }
21.    public void setPassword(String newValue) { password = newValue; }
22.
23.    public String getAboutYourself() { return aboutYourself; }
24.    public void setAboutYourself(String newValue) { aboutYourself = newValue; }
25. }
```

Листинг 4.11. Файл flags/src/java/com/corejsf/LocaleChanger.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.util.Locale;
5.
6. import javax.inject.Named;
7.    // или import javax.faces.bean.ManagedBean;
8. import javax.enterprise.context.SessionScoped;
9.    // или import javax.faces.bean.SessionScoped;
10. import javax.faces.context.FacesContext;
11.
12. @Named // или @ManagedBean
13. @SessionScoped
14. public class LocaleChanger implements Serializable {
```

```
15.    public String germanAction() {
16.       FacesContext context = FacesContext.getCurrentInstance();
17.       context.getViewRoot().setLocale(Locale.GERMAN);
18.       return null;
19.    }
20.
21.    public String englishAction() {
22.       FacesContext context = FacesContext.getCurrentInstance();
23.       context.getViewRoot().setLocale(Locale.ENGLISH);
24.       return null;
25.    }
26. }
```

Теги выбора

В технологии JSF предусмотрено семь тегов для выполнения выбора:

- h:selectBooleanCheckbox.
- h:selectManyCheckbox.
- h:selectOneRadio.
- h:selectOneListbox.
- h:selectManyListbox.
- h:selectOneMenu.
- h:selectManyMenu.

Примеры применения каждого тега приведены в табл. 4.23.

Таблица 4.23. Примеры применения тегов выбора

Тег	Сформированный код HTML	Примеры
h:selectBooleanCheckbox	`<input type="checkbox">`	Receive email: ☑

h:selectManyCheckbox	`<table>`
	`...`
	`<label>`
	`<input type="checkbox"/>`
	`</label>`
	`...`
	`</table>`
		☐ Red ☑ Blue ☐ Yellow

h:selectOneRadio	`<table>`
	`...`
	`<label>`
	`<input type="radio"/>`
	`</label>`
	`...`
	`</table>`
		○ High School
		◉ Bachelor's
		○ Master's
		○ Doctorate

h:selectOneListbox	`<select>`
	`<option value="Cheese">`
	`Cheese`
	`</option>`
	`...`
	`</select>`
		Cheese
		Pickle
		Mustard
		Lettuce

h:selectManyListbox	`<select multiple>`
	`<option value="Cheese">`
	`Cheese`
	`</option>`
	`...`
	`</select>`
		Cheese
		Pickle
		Mustard
		Lettuce
		Onions

Окончание табл. 4.23

Тег	Сформированный код HTML	Примеры
h:selectOneMenu	<select size="1"> <option value="Cheese"> Cheese </option> ... </select>	
h:selectManyMenu	<select multiple size="1"> <option value="Sunday"> Sunday </option> ... </select>	

Тег h:selectBooleanCheckbox является самым простым тегом выбора; он подготавливает к отображению флажок, который можно привязать к одному из булевых свойств бина. Можно также подготовить к отображению набор флажков с помощью тега h:selectManyCheckbox.

Теги, имена которых начинаются с selectOne, позволяют выбирать один элемент из коллекции. Теги selectOne подготавливают к отображению наборы переключателей, меню с единственным выбором или окна списка. Теги selectMany подготавливают к отображению наборы флажков, меню с множественным выбором или окна списка.

Все теги выбора совместно используют почти идентичный набор атрибутов, которые приведены в табл. 4.24.

Таблица 4.24. Атрибуты тегов h:selectBooleanCheckbox, h:selectManyCheckbox, h:selectOneRadio, h:selectOneListbox, h:selectManyListbox, h:selectOneMenu и h:selectManyMenu

Атрибуты	Описание
enabledClass, disabledClass JSF 2.0	Класс CSS для разрешенных или запрещенных элементов; это касается только тегов h:selectOneRadio и h:selectManyCheckbox
selectedClass, unselectedClass JSF 2.0	Класс CSS для выбранных или невыбранных элементов; это касается только тега h:selectManyCheckbox
layout	Спецификация, определяющая способ расположения элементов: атрибуты lineDirection (горизонтальное расположение) или pageDirection (вертикальное расположение); применяются только для тегов h:selectOneRadio и h:selectManyCheckbox
label JSF 1.2	Описание компонента, предназначенное для использования в сообщениях об ошибках
collectionType JSF 2.0	(Только для тегов selectMany.) Строка или выражение значения, вычисление которого приводит к получению полного имени класса коллекции, такого как java.util.TreeSet. См. раздел "Атрибут value и множественные выборы" на стр. 151
hideNoSelectionOption JSF 2.0	Скрывают любой элемент, который отмечен как "вариант пустого выбора". См. раздел "Тег f:selectItem" на стр. 144

Окончание табл. 4.24

Атрибуты	Описание
binding, converter, converterMessage JSF 1.2, requiredMessage JSF 1.2, id, immediate, required, rendered, validator, validatorMessage JSF 1.2, value, valueChangeListener	Базовые атрибуты[1]
accesskey, border, dir, disabled, lang, readonly, style, styleClass, size, tabindex, title	HTML 4.0[2] — понятие границы применимо только к тегам h:selectOneRadio и h:selectManyCheckbox; понятие размера применимо только к тегам h:selectOneListbox и h:selectManyListbox
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	События DHTML[3]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.
[3] Дополнительные сведения об атрибутах событий DHTML см. в табл. 4.7 на стр. 114.

Флажки и переключатели

Флажки представляют два тега JSF:

- h:selectBooleanCheckbox;
- h:selectManyCheckbox.

Тег h:selectBooleanCheckbox представляет единственный флажок, который может быть привязан к одному из булевых свойств бина.

Ниже приведен пример его применения (рис. 4.7).

Рис. 4.7. Единственный флажок

На странице JSF необходимо применить следующий код:

```
<h:selectBooleanCheckbox value="#{form.contactMe}"/>
```

А во вспомогательном бине должно быть предусмотрено свойство для чтения и записи:

```
private boolean contactMe;
public void setContactMe(boolean newValue) { contactMe = newValue; }
public boolean getContactMe() { return contactMe; }
```

Сформированный код HTML выглядит так:

```
<input type="checkbox" name="_id2:_id7"/>
```

Предусмотрена возможность создать группу флажков с помощью тега h:selectManyCheckbox. Как следует из самого имени тега, он позволяет выбрать один или несколько флажков в группе. Сама группа определяется в тексте тега h:selectManyCheckbox, либо с помощью одного или нескольких тегов f:selectItem, либо с помощью одного тега f:selectItems. С дополнительной информацией об этих основных тегах можно ознакомиться в разделе "Элементы" на стр. 144. Например, ниже приведена группа флажков, предназначенная для выбора цветов (рис. 4.8).

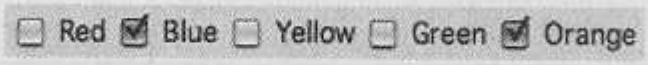


Рис. 4.8. Группа флажков

Тег `h:selectManyCheckbox` выглядит следующим образом:

```
<h:selectManyCheckbox value="#{form.colors}">
  <f:selectItem itemValue="Red" itemLabel="Red"/>
  <f:selectItem itemValue="Blue" itemLabel="Blue"/>
  <f:selectItem itemValue="Yellow" itemLabel="Yellow"/>
  <f:selectItem itemValue="Green" itemLabel="Green"/>
  <f:selectItem itemValue="Orange" itemLabel="Orange"/>
</h:selectManyCheckbox>
```

Флажки задаются с помощью тега `f:selectItem` (стр. 144) или `f:selectItems` (стр. 145).

Тег `h:selectManyCheckbox` формирует элемент таблицы HTML; ниже приведен код HTML, сформированный для рассматриваемого примера с выбором цветов.

```
<table>
  <tr>
    <td>
      <label for="_id2:_id14">
        <input name="_id2:_id14" value="Red" type="checkbox"> Red</input>
      </label>
    </td>
  </tr>
  ...
</table>
```

Каждый цвет задан с помощью элемента `input`, снабженного меткой для упрощения доступа. Такая метка размещается в элементе `td`.

Переключатели реализованы с помощью тега `h:selectOneRadio`. Соответствующий пример приведен на рис. 4.9.

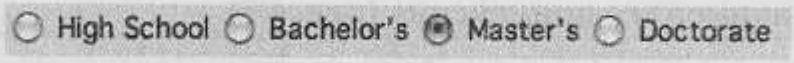


Рис. 4.9. Переключатели

Атрибут `value` тега `h:selectOneRadio` определяет элемент, выбранный в настоящее время. Еще раз отметим, что для заполнения переключателей используется несколько тегов `f:selectItem`:

```
<h:selectOneRadio value="#{form.education}">
  <f:selectItem itemValue="High School" itemLabel="High School"/>
  <f:selectItem itemValue="Bachelor's" itemLabel="Bachelor's"/>
  <f:selectItem itemValue="Master's" itemLabel="Master's"/>
  <f:selectItem itemValue="Doctorate" itemLabel=Doctorate"/>
</h:selectOneRadio>
```

Как и `h:selectManyCheckbox`, тег `h:selectOneRadio` формирует таблицу HTML. Ниже приведена таблица, сформированная с помощью указанного тега.

```
<table>
  <tr>
    <td>
      <label for="_id2:_id14">
        <input name="_id2:_id14" value="High School" type="radio">
          High School
        </input>
      </label>
    </td>
```

```
    </tr>
    ...
</table>
```

Теги `h:selectOneRadio` и `h:selectManyCheckbox` сходны не только тем, что формируют таблицы HTML, но и имеют другие общие особенности — несколько атрибутов, присущих только этим двум тегам:

- `border`;
- `enabledClass`;
- `disabledClass`;
- `layout`.

Атрибут `border` определяет ширину границы. В качестве примера на рис. 4.10 показаны переключатели и флажки с границами 1 и 2 соответственно.

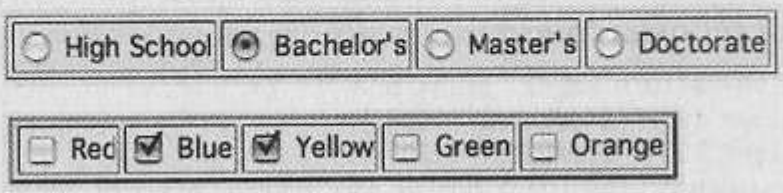


Рис. 4.10. Атрибут `border`

Атрибуты `enabledClass` и `disabledClass` определяют классы CSS, используемые соответственно, когда флажки или переключатели разрешены или запрещены. Например, на рис. 4.11 показан разрешенный класс с курсивным начертанием шрифта, синим цветом переднего плана и желтым фоном.

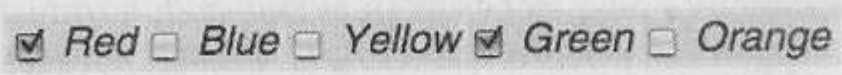


Рис. 4.11. Разрешенный класс с курсивным начертанием шрифта, синим цветом переднего плана и желтым фоном

Атрибут `layout` может иметь значение `lineDirection` (горизонтальное расположение) или `pageDirection` (вертикальное расположение). Например, на рис. 4.12 флажки слева имеют макет `pageDirection`, а флажки справа — `lineDirection`.

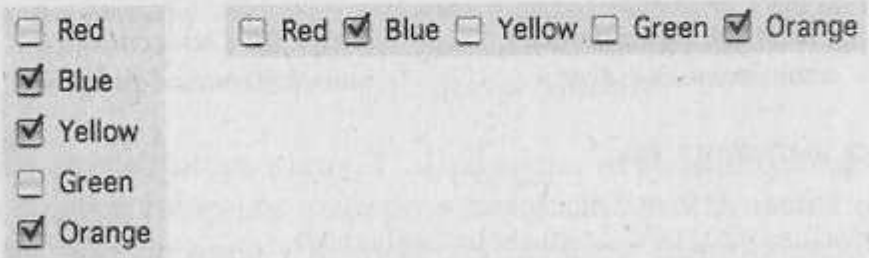


Рис. 4.12. Флажки с разными макетами

На заметку! Читателя может заинтересовать вопрос, почему значения атрибута `layout` задаются как `lineDirection` и `pageDirection`, а не `horizontal` и `vertical`? Дело в том, что расположения `lineDirection` и `pageDirection` действительно соответствуют горизонтальному и вертикальному в национальных языках с латинской графикой, но это не всегда имеет место в других языках. Например, в браузере для китайского языка, в котором текст отображается сверху вниз, расположение `lineDirection` может рассматриваться как вертикальное, а `pageDirection` — как горизонтальное.

Меню и окна списка

Меню и окна списка представлены следующими тегами:

- h:selectOneListbox;
- h:selectManyListbox;
- h:selectOneMenu;
- h:selectManyMenu.

Атрибуты для указанных выше тегов перечислены в табл. 4.24 на стр. 138, поэтому здесь мы не будем повторять эти сведения. Теги меню и окон списка формируют элементы select языка HTML. В тегах меню к элементу select добавляется атрибут size="1". В этом обозначении размера заключается все различие между меню и окнами списка.

Ниже показано окно списка с единственным выбором (рис. 4.13).

Соответствующий тег окна списка выглядит следующим образом:

```
<h:selectOneListbox value="#{form.year}" size="5">
  <f:selectItem itemValue="1900" itemLabel="1900"/>
  <f:selectItem itemValue="1901" itemLabel="1901"/>
  ...
</h:selectOneListbox>
```

Обратите внимание на то, что для определения числа видимых элементов использовался атрибут size. Сформированный код HTML выглядит следующим образом:

```
<select name="_id2:_id11" size="5">
  <option value="1900">1900</option>
  <option value="1901">1901</option>
  ...
</select>
```

Тег h:selectManyListbox служит для формирования окон списка с множественным выбором, как в примере, приведенном на рис. 4.14.

Рис. 4.13. Окно списка с единственным выбором

Рис. 4.14. Окно списка с множественным выбором

Тег окна списка выглядит так:

```
<h:selectManyListbox value="#{form.languages}">
  <f:selectItem itemValue="English" itemLabel="English"/>
  <f:selectItem itemValue="French" itemLabel="French"/>
  <f:selectItem itemValue="Italian" itemLabel="Italian"/>
  <f:selectItem itemValue="Spanish" itemLabel="Spanish"/>
  <f:selectItem itemValue="Russian" itemLabel="Russian"/>
</h:selectManyListbox>
```

На сей раз мы не определяем атрибут size, поэтому окно списка увеличивается так, чтобы вместить все свои элементы. Сформированный код HTML показан ниже.

```
<select name="_id2:_id11" multiple>
  <option value="English">English</option>
  <option value="French">French</option>
```

```
  ...
</select>
```

Теги `h:selectOneMenu` и `h:selectManyMenu` используются для формирования меню. Меню с единственным выбором выглядит примерно так, как показано на рис. 4.15.

Для создания указанного меню применялся тег `h:selectOneMenu`:

```
<h:selectOneMenu value="#{form.day}">
  <f:selectItem itemValue="1" itemLabel="Sunday"/>
  <f:selectItem itemValue="2" itemLabel="Monday"/>
  <f:selectItem itemValue="3" itemLabel="Tuesday"/>
  <f:selectItem itemValue="4" itemLabel="Wednesday"/>
  <f:selectItem itemValue="5" itemLabel="Thursday"/>
  <f:selectItem itemValue="6" itemLabel="Friday"/>
  <f:selectItem itemValue="7" itemLabel="Saturday"/>
</h:selectOneMenu>
```

Сформированный код HTML показан ниже.

```
<select name="_id2:_id17" size="1">
  <option value="1">Sunday</option>
  ...
</select>
```

Тег `h:selectManyMenu` используется для меню с множественным выбором. Этот тег формирует код HTML, который выглядит следующим образом:

```
<select name="_id2:_id17" multiple size="1">
  <option value="1">Sunday</option>
  ...
</select>
```

Но применение этого кода HTML в разных браузерах не приводит к получению одинаковых результатов. В качестве примера на рис. 4.16 показано, как выглядит результат использования тега `h:selectManyMenu` в браузерах Internet Explorer (слева) и Netscape (справа).

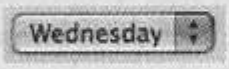


Рис. 4.15. Меню с единственным выбором

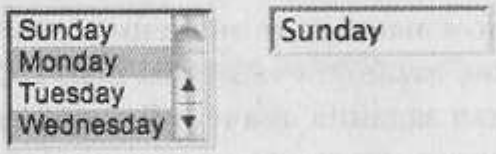


Рис. 4.16. Результаты применения тега `h:selectManyMenu` в различных браузерах – Internet Explorer (слева) и Netscape (справа)

Меню всегда имеют атрибут `size="1"`. Браузеры неизменно подготавливают к отображению меню с единственным выбором как раскрывающиеся списки, в соответствии с ожиданиями. Но браузеры не подготавливают единообразно к отображению меню с множественным выбором, заданные с помощью атрибута `size="1"`, а также меню с несколькими атрибутами. Вместо подготовки меню к отображению в виде раскрывающегося списка с множественным выбором, как можно было бы ожидать, некоторые браузеры формируют нелепые графические элементы наподобие крошечных полос прокрутки, которыми почти невозможно манипулировать (Internet Explorer), или вообще не формируют полосу прокрутки, вынуждая пользователя передвигаться по элементам меню с помощью клавиш со стрелками (Firefox).

Элементы

В разделе "Флажки и переключатели" на стр. 139 мы приступили к рассмотрению способов использования множественных тегов f:selectItem для заполнения компонентов выбора. Итак, в настоящей книге уже показано, как выглядят на экране результаты применения тегов выбора, поэтому можно перейти к более внимательному рассмотрению тегов f:selectItem и связанных с ними тегов f:selectItems.

Тег f:selectItem

Тег f:selectItem используется для определения единственных элементов выбора, как в следующем примере:

```
<h:selectOneMenu value="#{form.condiments}">
  <f:selectItem itemValue="Cheese" itemLabel="Cheese"/>
  <f:selectItem itemValue="Pickle" itemLabel="Pickle"/>
  <f:selectItem itemValue="Mustard" itemLabel="Mustard"/>
  <f:selectItem itemValue="Lettuce" itemLabel="Lettuce"/>
  <f:selectItem itemValue="Onions" itemLabel="Onions"/>
</h:selectOneMenu>
```

Значения (в данном случае Cheese, Pickle и т.д.) передаются как значения параметра запроса после выполнения выбора в меню, а затем происходит передача самой формы с меню. Значения itemLabel используются в качестве меток для пунктов меню. Иногда возникает необходимость определять различные данные для использования в качестве значений параметров запроса и меток элементов:

```
<h:selectOneMenu value="#{form.condiments}">
  <f:selectItem itemValue="1" itemLabel="Cheese"/>
  <f:selectItem itemValue="2" itemLabel="Pickle"/>
  <f:selectItem itemValue="3" itemLabel="Mustard"/>
  <f:selectItem itemValue="4" itemLabel="Lettuce"/>
  <f:selectItem itemValue="5" itemLabel="Onions"/>
</h:selectOneMenu>
```

В приведенном выше коде значения элементов представляют собой строки. В разделе "Связывание атрибута value" на стр. 150 показано, как использовать данные различных типов для задания значений элементов. Кроме меток и значений, можно также предусматривать применение описаний элементов и задавать запрещенное состояние элемента:

```
<f:selectItem itemLabel="Cheese" itemValue="#{form.cheeseValue}"
  itemDescription="used to be milk"
  itemDisabled="true"/>
```

Описания элементов предназначены только для тех или иных инструментальных средств и не оказывают влияния на формирование кода HTML. С другой стороны, значение атрибута itemDisabled передается в код HTML. Тег f:selectItem имеет атрибуты, приведенные в табл. 4.25.

Со времени выхода версии JSF 2.0 появилась возможность использовать атрибут noSelectionOption для отметки элемента, включаемого в целях обеспечения навигации, такого как "Select a condiment" (Выберите приправу) в рассматриваемом приложении. Этот атрибут используется также при проверке правильности. Если запись является обязательной, а пользователь отмечает "вариант пустого выбора", то происходит ошибка проверки правильности.

Таблица 4.25. Атрибуты тега `f:selectItem`

Атрибут	Описание
`binding, id`	Базовые атрибуты[1]
`itemDescription`	Описание, используемое только инструментальными средствами
`itemDisabled`	Значение типа `Boolean`, которое задает заблокированный атрибут элемента HTML
`itemLabel`	Текст, отображаемый элементом
`itemValue`	Значение элемента, передаваемое на сервер в качестве параметра запроса
`value`	Выражение значения, которое указывает на экземпляр `SelectItem`
`escape` JSF 1.2	Равен `true`, если специальные символы в значении должны быть преобразованы в символьные сущности (значение по умолчанию); равен `false`, если значение должно передаваться без изменений
`noSelectionOption` JSF 2.0	Равен `true`, если текущий элемент соответствует варианту "пустого выбора", который, будучи выбранным пользователем, указывает, что пользователь намеренно отказывается от конкретного выбора

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.

Атрибут `value` тега `f:selectItem` можно использовать для доступа к экземплярам `SelectItem`, созданным в бине:

```
<f:selectItem value="#{form.cheeseItem}"/>
```

Выражение значения для атрибута `value` указывает метод, который возвращает экземпляр `javax.faces.model.SelectItem`:

```
public SelectItem getCheeseItem() { return new SelectItem("Cheese"); }
```

`javax.faces.model.SelectItem`

- `SelectItem(Object value)`

  Создает тег `SelectItem` со значением `value`. Для получения метки элемента необходимо применить метод `toString()` к значению.

- `SelectItem(Object value, String label)`

  Создает тег `SelectItem` со значениями `value` и `label`.

- `SelectItem(Object value, String label, String description)`

  Создает тег `SelectItem` со значениями `value`, `label` и `description`.

- `SelectItem(Object value, String label, String description, boolean disabled)`

  Создает тег `SelectItem` со значениями `value`, `label`, `description` и состоянием `disabled`.

- `SelectItem(Object value, String label, String description, boolean disabled, boolean noSelectionOption)` JSF 2.0

  Создает тег `SelectItem` со значениями `value`, `label`, `description`, состоянием `disabled` и флагом "вариант пустого выбора".

Тег `f:selectItems`

Как было показано в разделе "Тег `f:selectItem`" на стр. 144, тег `f:selectItem` предоставляет широкие возможности, но с его применением становится затруднительным определение большого количества элементов. Первый фрагмент кода, показанный в этом разделе, можно привести к следующему с помощью тега `f:selectItems`:

```
<h:selectOneRadio value="#{form.condiments}>
  <f:selectItems value="#{form.condimentItems}"/>
</h:selectOneRadio>
```

Выражение значения `#{form.condimentItems}` может указывать на массив экземпляров `SelectItem`:

```
private static SelectItem[] condimentItems = {
  new SelectItem(1, "Cheese"),
  new SelectItem(2, "Pickle"),
  new SelectItem(3, "Mustard"),
  new SelectItem(4, "Lettuce"),
  new SelectItem(5, "Onions")
};
public SelectItem[] getCondimentItems() {
  return condimentItems;
}
```

Атрибут `value` тега `f:selectItems` должен представлять собой выражение значения, которое указывает на одно из следующего:

- единственный экземпляр `SelectItem`;
- коллекция;
- массив;
- карта, записи которой представляют метки и значения.

Первый вариант вряд ли может принести значительную пользу, а другие варианты рассматриваются в следующих разделах.

На заметку! Иногда бывает сложно запомнить, какие объекты применяются в сочетании с атрибутом `value` тега `f:selectItems`. В таком случае можно воспользоваться аббревиатурой SCAM: Single select item, Collection, Array, Map (Элемент единственного выбора, коллекция, массив, карта).

На заметку! Обычно лучше применить один тег `f:selectItems`, чем несколько тегов `f:selectItem`. В случае изменения количества элементов приходится корректировать только код Java, если используется тег `f:selectItems`, тогда как при использовании тега `f:selectItem` может потребоваться исправить и код Java, и страницы JSF.

В табл. 4.26 приведены итоговые сведения об атрибутах тега `f:selectItems`.

Таблица 4.26. Атрибуты тега `f:selectItems`

Атрибут	Описание
`binding, id`	Базовые атрибуты[1]
`value`	Выражение значения, которое указывает на экземпляр `SelectItem`, массив, коллекцию или карту
`var`	Имя переменной, используемой в указанных ниже выражениях значения при переборе массива или коллекции объектов, отличных от `SelectItem`
`itemLabel`	Выражение значения, вычисление которого приводит к получению текста, отображаемого на элементе, на который ссылается переменная `var`
`itemValue`	Выражение значения, вычисление которого приводит к получению значения элемента, на который ссылается переменная `var`

Окончание табл. 4.26

Атрибут	Описание
itemDescription	Выражение значения, вычисление которого приводит к получению описания элемента, на который ссылается переменная var; описание предназначено для использования инструментальными средствами
itemDisabled	Выражение значения, вычисление которого приводит к получению отмененного атрибута HTML элемента, на который ссылается переменная var
itemLabelEscaped	Выражение значения, вычисление которого приводит к получению значения true, если специальные символы в значении элемента должны быть преобразованы в символьные сущности (значение по умолчанию), или значения false, если значение должно передаваться без изменений
noSelectionOption	Выражение значения, вычисление которого приводит к получению элемента "вариант пустого выбора" или строки, эквивалентной значению элемента "вариант пустого выбора"

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.

Использование коллекций и массивов с тегом f:selectItems

До выхода версии JSF 2.0 применяемые коллекции и массивы должны были содержать экземпляры SelectItem. Это было неудобно, поскольку в бизнес-логике приложения приходилось учитывать особенности API-интерфейса JSF. А со времени выхода версии JSF 2.0 значением тега f:selectItems может быть коллекция или массив, содержащие объекты любого типа.

Если эти объекты представляют собой экземпляры SelectItem, то какая-либо дополнительная обработка не требуется. В противном случае должны быть получены метки путем вызова метода toString применительно к каждому объекту.

Еще один вариант состоит в том, что можно использовать атрибут var для определения переменной, с помощью которой выполняются итерации по массиву или коллекции. После этого можно передавать выражения значения для метки и значения в атрибутах itemLabel и itemValue.

Например, предположим, что пользователи должны выбирать объекты следующего класса:

```
public class Weekday {
   public String getDayName() { ... } // День недели в текущей локали, скажем, "Monday"
   public int getDayNumber() { ... } // Числовое значение, допустим, Calendar.MONDAY (2)
   ...
}
```

Для этого используется следующий тег:

```
<f:selectItems value="#{form.daysOfTheWeek}"
  var="w"
  itemLabel="#{w.dayName}"
  itemValue="#{w.dayNumber}" />
```

В данном случае вычисление выражения значения #{form.daysOfTheWeek} приводит к получению массива или коллекции объектов Weekday. Переменная w задается равной каждому из этих элементов. Затем создается объект SelectItem с результатами выражений itemLabel и itemValue.

На заметку! Атрибут var тега f:selectItems концептуально аналогичен по своему назначению атрибуту var тега h:dataTable, который будет рассматриваться в главе 6.

Использование карт с тегом f:selectItems

Если вычисление атрибута value тега f:selectItems приводит к получению карты, то реализация JSF создает по одному экземпляру SelectItem для каждой записи в карте. Ключ записи используется как метка элемента, а значение записи — как значение элемента. В качестве примера ниже показано, как задавать описание необходимых компонентов блюд с помощью карты.

```
private static Map<String, Object> condimentItems;
static {
   condimentItems = new LinkedHashMap<String, Object>();
   condimentItems.put("Cheese", 1); // Метка, значение
   condimentItems.put("Pickle", 2);
   condimentItems.put("Mustard", 3);
   condimentItems.put("Lettuce", 4);
   condimentItems.put("Onions", 5);
}

public Map<String, Object> getCondimentItems() {
   return condimentItems;
}
```

Следует отметить, что при использовании карты исключена возможность задавать описания элементов или указывать отмененное состояние.

При использовании карты необходимо учитывать наличие двух особенностей.

1. Обычно лучше использовать карты типа LinkedHashMap, а не TreeMap или HashMap. Карта типа LinkedHashMap позволяет управлять порядком элементов, поскольку перебор элементов происходит в той последовательности, в которой они были вставлены. Если применяется карта типа TreeMap, метки, предоставляемые пользователю (которые являются ключами карты), отсортированы в алфавитном порядке. Это может совпадать или не совпадать с намерениями разработчика приложения. Например, может оказаться, что дни недели аккуратно отсортированы по алфавиту как Friday, Monday, Saturday, Sunday, Thursday, Tuesday, Wednesday. Если же используется карта типа HashMap, то упорядочение элементов происходит случайным образом.
2. Ключи карты преобразуются в метки элементов, а значения карты — в значения элементов. После того как пользователь выбирает элемент, вспомогательный бин приложения получает из карты значение, а не ключ. В частности, как показано в приведенном выше примере, если вспомогательный бин получает значение 5, то должны быть выполнены итерации по записям в случае необходимости найти соответствующее значение "Onions". Тем не менее такое значение, по-видимому, является более важным для приложения, чем метка, поэтому обычно в связи с этим не возникают какие-либо проблемы, просто об этом следует знать.

Группы элементов

Предусмотрена возможность группировать элементы меню или окна списка, как показано на рис. 4.17.

Теги JSF, которые определяют это окно списка, приведены ниже.

```
<h:selectManyListbox>
  <f:selectItems value="#{form.menuItems}"/>
</h:selectManyListbox>
```

Рис. 4.17. Группирование элементов

Свойство `menuItems` представляет собой массив `SelectItem`:

```
public SelectItem[] getMenuItems() { return menuItems; }
```

Создание экземпляра массива `menuItems` происходит следующим образом:

```
private static SelectItem[] menuItems = { burgers, beverages, condiments };
```

Переменные `burgers`, `beverages` и `condiments` представляют собой экземпляры `SelectItemGroup`, создаваемые так:

```
private SelectItemGroup burgers =
   new SelectItemGroup("Burgers",          //  Значение
      "burgers on the menu",               //  Описание
      false,                               //  Отменено
      burgerItems);                        //  Выбранные элементы
private SelectItemGroup beverages =
   new SelectItemGroup("Beverages",        //  Значение
      "beverages on the menu",             //  Описание
      false,                               //  Отменено
      beverageItems);                      //  Выбранные элементы

private SelectItemGroup condiments =
   new SelectItemGroup("Condiments",       //  Значение
      "condiments on the menu",            //  Описание
      false,                               //  Отменено
      condimentItems);                     //  Выбранные элементы
```

Обратите внимание на то, что используются экземпляры `SelectItemGroups` для заполнения массива `SelectItems`. Такая возможность имеется в связи с тем, что класс `SelectItemGroup` является расширением класса `SelectItem`. Создание и инициализация групп происходят таким образом:

```
private SelectItem[] burgerItems = {
   new SelectItem("Qwarter pounder"),
   new SelectItem("Single"),
   new SelectItem("Veggie"),
};
private SelectItem[] beverageItems = {
   new SelectItem("Coke"),
   new SelectItem("Pepsi"),
   new SelectItem("Water"),
   new SelectItem("Coffee"),
   new SelectItem("Tea"),
};
private SelectItem[] condimentItems = {
   new SelectItem("cheese"),
   new SelectItem("pickle"),
```

```
    new SelectItem("mustard"),
    new SelectItem("lettuce"),
    new SelectItem("onions"),
};
```

В экземплярах `SelectItemGroup` элементы `optgroup` языка HTML кодируются. Например, при обработке приведенного выше кода формируется следующий код HTML:

```
<select name="_id0:_id1" multiple size="16">
   <optgroup label="Burgers">
      <option value="1" selected>Qwarter pounder</option>
      <option value="2">Single</option>
      <option value="3">Veggie</option>
   </optgroup>
   <optgroup label="Beverages">
      <option value="4" selected>Coke</option>
      <option value="5">Pepsi</option>
      <option value="6">Water</option>
      <option value="7">Coffee</option>
      <option value="8">Tea</option>
   </optgroup>
   <optgroup label="Condiments">
      <option value="9">cheese</option>
      <option value="10">pickle</option>
      <option value="11">mustard</option>
      <option value="12">lettuce</option>
      <option value="13">onions</option>
   </optgroup>
</select>
```

На заметку! Спецификация HTML 4.01 не допускает применения вложенных элементов `optgroup`, что было бы полезно для создания таких объектов, как каскадные меню. Но в этой спецификации есть упоминание о том, что в будущих версиях HTML может быть предусмотрена поддержка такого поведения.

`javax.faces.model.SelectItemGroup`

- `SelectItemGroup(String label)`

  Создает группу с меткой, но без элементов выбора.

- `SelectItemGroup(String label, String description, boolean disabled, SelectItem[] items)`

  Создает группу с меткой, описанием (которое игнорируется реализацией JSF Reference), логическим значением, которое отменяет все элементы, будучи равным `true`, и массивом элементов выбора, используемым для заполнения группы.

- `setSelectItems(SelectItem[] items)`

  Задает массив `SelectItems` группы.

Связывание атрибута `value`

Если в приложении используется набор флажков, меню или окно списка, возникает необходимость следить за тем, какой элемент (или элементы) выбран пользователем. С этой целью применяется атрибут `value` тегов `selectMany` и `selectOne`. Рассмотрим следующий пример:

```
<h:selectOneMenu value="#{form.bestDay}">
  <f:selectItems value="#{form.weekdays}"/>
</h:selectOneMenu>
```

Атрибут value тега h:selectOneMenu ссылается на значение, выбранное пользователем. Атрибут value тега f:selectItems определяет все возможные значения.

Предположим, что наряду с массивом объектов SelectItem, содержащим следующее, заданы переключатели:

```
new SelectItem(1, "Sunday"),    // Значение, метка
new SelectItem(2, "Monday"),
...
```

Пользователь видит метки (воскресенье, понедельник, ...), а в приложении используются значения (1, 2, ...).

Значения типов, применяемых в языке Java, имеют одну важную, но весьма тонкую особенность. На веб-странице эти значения всегда являются строковыми:

```
<option value="1">Sunday</option>
<option value="2">Monday</option>
```

После передачи страницы сервер получает выбранную строку и должен преобразовать ее в соответствующий тип. В реализации JSF предусмотрены средства преобразования строк в числа и перечислимые типы, а для других типов необходимо определять преобразователи. (Преобразователи будут описаны в главе 7.)

В рассматриваемом примере выражение значения #{form.bestDay} должно ссылаться на свойство типа int или Integer. В листинге 4.13 показан пример, в котором значение относится к перечислимому типу.

Внимание! Значение SelectItem относится к типу Object, поэтому может возникнуть стремление задать его равным значению, которое фактически требуется в приложении. Однако следует помнить, что при передаче клиенту это значение преобразуется в строковое. В качестве примера рассмотрим значение SelectItem(Color.RED, "Red"). После отправки этого значения клиент принимает строку "java.awt.Color[r=255,g=0,b=0]". Возврат этой строки происходит, если пользователь выбирает вариант с меткой "Red". Чтобы снова получить обозначение цвета, необходимо провести синтаксический анализ строки. Вместо этого проще отправить значение RGB цвета.

Атрибут value и множественные выборы

Для отслеживания множественных выборов может применяться тег selectMany. Такие теги имеют атрибут value, который задает количество выбранных элементов от нуля и более, используя массив или коллекцию.

Рассмотрим тег h:selectManyListbox, который позволяет пользователю выбирать несколько приправ:

```
<h:selectManyListbox value="#{form.condiments}">
  <f:selectItems value="#{form.condimentItems}"/>
</h:selectManyListbox>
```

Свойства condimentItems и condiments являются таковыми:

```
private static SelectItem[] condimentItems = {
   new SelectItem(1, "Cheese"),
   new SelectItem(2, "Pickle"),
   new SelectItem(3, "Mustard"),
   new SelectItem(4, "Lettuce"),
   new SelectItem(5, "Onions"),
};
public SelectItem[] getCondimentItems() {
   return condimentItems;
}
private int[] condiments;
public void setCondiments(int[] newValue) {
```

```
    condiments = newValue;
}
public int[] getCondiments() {
    return condiments;
}
```

Для хранения значений свойства `condiments` вместо массива `int[]` можно использовать массив `Integer[]`.

Значение тега `selectMany` может представлять собой коллекцию, а не массив, однако необходимо учитывать две технических проблемы. Наиболее важной из них является то, что элементы коллекции не могут быть преобразованы, поскольку тип элементов коллекции не известен во время выполнения. (Это — одна из неудобных особенностей универсальных типов данных Java. Во время выполнения объект `ArrayList<Integer>` или `ArrayList<String>` представляет собой всего лишь бесформатный объект `ArrayList`, а способ определения типа его элементов отсутствует. Напротив, `Integer[]` и `String[]` во время выполнения рассматриваются как различные типы.) Это означает, что коллекции следует использовать только для хранения строк.

Вторая проблема является более трудноуловимой. После получения приложением JSF данных о выборе, сделанном пользователем, возникает необходимость создать новый экземпляр коллекции, заполнить его и передать коллекцию методу задания свойства. Однако предположим, что типом свойства является `Set<String>`. Какой объект `Set` должен быть создан?

До выхода версии JSF 2.0 ответ на этот вопрос не был однозначным. А в версии JSF 2.0 сформулированы следующие правила.

1. Если тег имеет атрибут `collectionType`, его значение должно быть строкой или выражением значения, вычисление которого приводит к получению полного имени класса, такого как `java.util.TreeSet`. Необходимо создать экземпляр этого класса.
2. В противном случае необходимо возвратить существующее значение и попытаться его клонировать и очистить.
3. Если эта попытка окончится неудачей (возможно, в связи с тем, что существующее значение было равно нулю или оказалось не клонируемым), то следует рассмотреть тип выражения значения. Если этим типом является `SortedSet`, `Set` или `Queue`, необходимо создать `TreeSet`, `HashSet` или `LinkedList`.
4. В противном случае должен быть создан `ArrayList`.

Например, предположим, что определяется свойство `languages`:

```
private Set<String> languages; // Инициализировано значением null
public Set<String> getLanguages() {
    return languages;
}
public void setLanguages(Set<String> newValue) {
    languages = newValue;
}
```

После того как форма передается впервые, метод задания свойства вызывается со значением `HashSet`, которое содержит данные о выборе, сделанном пользователем (шаг 3). При последующих вызовах это значение клонируется (шаг 2). Но предположим, что была выполнена инициализация указанного набора:

```
private Set<String> languages = new TreeSet();
```

В таком случае всегда происходит возврат значения `TreeSet`.

Итоговые сведения: флажки, переключатели, меню и окна списка

Завершая этот раздел, посвященный тегам выбора, рассмотрим пример, в котором применяются почти все эти теги. В данном примере, показанном на рис. 4.18, реализуется форма, запрашивающая личную информацию. Тег `h:selectBooleanCheckbox` используется для определения того, желает ли пользователь вступать в контакт, а тег `h:selectOneMenu` позволяет пользователю выбрать самый удобный день недели, в который мы могли бы это сделать.

Окно списка с календарными данными реализуется с помощью тега `h:selectOneMenu` и демонстрирует использование элемента "пустого выбора". Флажки выбора языков реализованы с помощью тега `h:selectManyCheckbox`, а для выбора уровня образования применяется тег `h:selectOneRadio`.

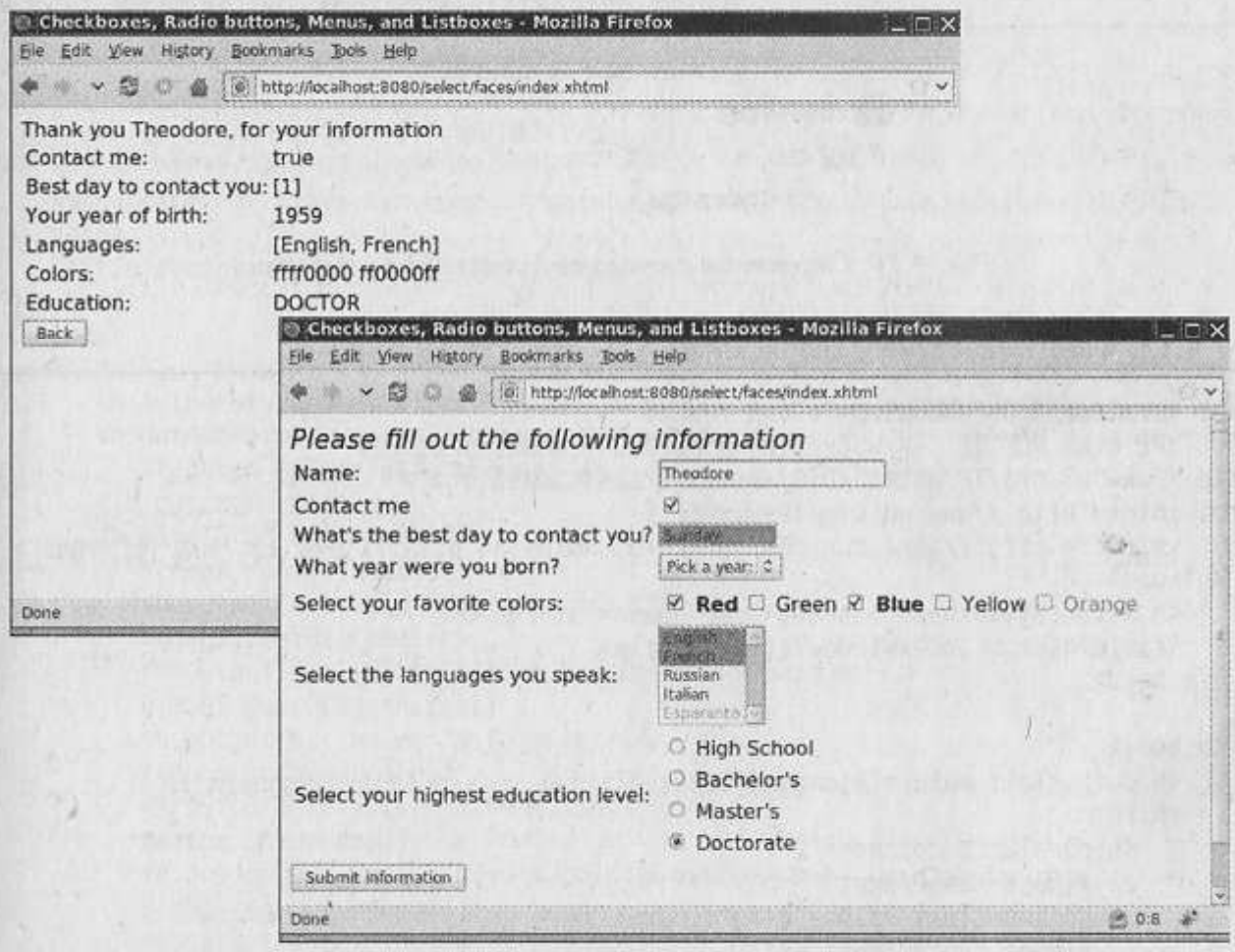


Рис. 4.18. Использование флажков, переключателей, меню и окон списка

Необходимо учитывать, что данные о языках собраны в объекте `Set<String>`. Важно также обратить внимание на стили в средстве выбора цвета. Отмененный вариант `Orange` закрашен серым цветом, а выбранные цвета выделены полужирным шрифтом. Для немедленного обновления стилей после выбора используется атрибут `onchange="submit()"`.

После того как пользователь передает форму, средства навигации JSF осуществляют переход к странице JSF, которая показывает данные, введенные пользователем.

Структура каталогов для приложения, показанного на рис. 4.18, приведена на рис. 4.19. Страницы JSF, бин `RegisterForm`, файл конфигурации `faces` и связка ресурсов показаны в листингах 4.12–4.16.

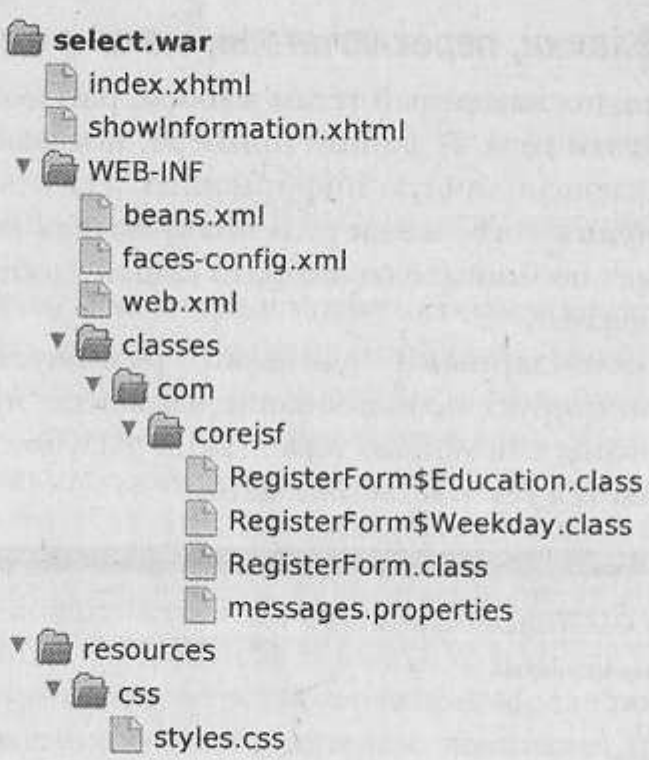


Рис. 4.19. Структура каталогов для примера с выбором

Листинг 4.12. Файл `select/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.indexWindowTitle}</title>
   </h:head>

   <h:body>
      <h:outputText value="#{msgs.indexPageTitle}" styleClass="emphasis"/>
      <h:form>
         <h:panelGrid columns="2">
            #{msgs.namePrompt}
            <h:inputText value="#{form.name}"/>
            #{msgs.contactMePrompt}
            <h:selectBooleanCheckbox value="#{form.contactMe}"/>
            #{msgs.bestDayPrompt}
            <h:selectManyMenu value="#{form.bestDaysToContact}">
               <f:selectItems value="#{form.daysOfTheWeek}" var="w"
                              itemLabel="#{w.dayName}" itemValue="#{w.dayNumber}"/>
            </h:selectManyMenu>
            #{msgs.yearOfBirthPrompt}
            <h:selectOneMenu value="#{form.yearOfBirth}" required="true">
               <f:selectItems value="#{form.yearItems}"/>
            </h:selectOneMenu>
            #{msgs.colorPrompt}
            <h:selectManyCheckbox value="#{form.colors}"
                                  selectedClass="selected" disabledClass="disabled"
                                  onchange="submit()">
               <f:selectItems value="#{form.colorItems}"/>
            </h:selectManyCheckbox>
            #{msgs.languagePrompt}
            <h:selectManyListbox size="5" value="#{form.languages}">
```

```
            <f:selectItems value="#{form.languageItems}"/>
         </h:selectManyListbox>
         #{msgs.educationPrompt}
         <h:selectOneRadio value="#{form.education}"
                           selectedClass="selected" layout="pageDirection">
            <f:selectItems value="#{form.educationItems}"/>
         </h:selectOneRadio>
      </h:panelGrid>
      <h:commandButton value="#{msgs.buttonPrompt}" action="showInformation"/>
   </h:form>
   <h:messages/>
</h:body>
</html>
```

Листинг 4.13. Файл select/web/showInformation.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.indexWindowTitle}</title>
   </h:head>
   <h:body>
      <h:form>
         <h:outputStylesheet library="css" name="styles.css" target="head"/>
         <h:outputFormat value="#{msgs.thankYouLabel}">
            <f:param value="#{form.name}"/>
         </h:outputFormat>
         <h:panelGrid columns="2">
            #{msgs.contactMeLabel}
            <h:outputText value="#{form.contactMe}"/>
            #{msgs.bestDayLabel}
            <h:outputText value="#{form.bestDaysConcatenated}"/>
            #{msgs.yearOfBirthLabel}
            <h:outputText value="#{form.yearOfBirth}"/>
            #{msgs.languageLabel}
            <h:outputText value="#{form.languages}"/>
            #{msgs.colorLabel}
            <h:outputText value="#{form.colorsConcatenated}"/>
            #{msgs.educationLabel}
            <h:outputText value="#{form.education}"/>
         </h:panelGrid>
         <h:commandButton value="#{msgs.backPrompt}" action="index"/>
      </h:form>
   </h:body>
</html>
```

Листинг 4.14. Файл select/src/java/com/corejsf/RegisterForm.java

```
package com.corejsf;

import java.awt.Color;
import java.io.Serializable;
import java.text.DateFormatSymbols;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Calendar;
import java.util.Collection;
```

```
10. import java.util.LinkedHashMap;
11. import java.util.Map;
12. import java.util.Set;
13. import java.util.TreeSet;
14.
15. import javax.inject.Named;
16.    // или import javax.faces.bean.ManagedBean;
17. import javax.enterprise.context.SessionScoped;
18.    // или import javax.faces.bean.SessionScoped;
19. import javax.faces.model.SelectItem;
20.
21. @Named("form") // или @ManagedBean(name="form")
22. @SessionScoped
23. public class RegisterForm implements Serializable {
24.    public enum Education { HIGH_SCHOOL, BACHELOR, MASTER, DOCTOR };
25.
26.    public static class Weekday {
27.       private int dayOfWeek;
28.       public Weekday(int dayOfWeek) {
29.          this.dayOfWeek = dayOfWeek;
30.       }
31.
32.       public String getDayName() {
33.          DateFormatSymbols symbols = new DateFormatSymbols();
34.          String[] weekdays = symbols.getWeekdays();
35.          return weekdays[dayOfWeek];
36.       }
37.
38.       public int getDayNumber() {
39.          return dayOfWeek;
40.       }
41.    }
42.
43.    private String name;
44.    private boolean contactMe;
45.     private   int[] bestDaysToContact;
46.     private   Integer yearOfBirth;
47.     private   int[] colors;
48.     private   Set<String> languages = new TreeSet<String>();
49.     private   Education education = Education.BACHELOR;
50.
51.     public String getName() { return name; }
52.     public void setName(String newValue) { name = newValue; }
53.
54.     public boolean getContactMe() { return contactMe; }
55.     public void setContactMe(boolean newValue) { contactMe = newValue; }
56.
57.    public int[] getBestDaysToContact() { return bestDaysToContact; }
58.    public void setBestDaysToContact(int[] newValue) { bestDaysToContact = newValue; }
59.
60.     public Integer getYearOfBirth() { return yearOfBirth; }
61.     public void setYearOfBirth(Integer newValue) { yearOfBirth = newValue; }
62.
63.     public int[] getColors() { return colors; }
64.     public void setColors(int[] newValue) { colors = newValue; }
65.
66.     public Set<String> getLanguages() { return languages; }
67.     public void setLanguages(Set<String> newValue) { languages = newValue; }
68.
69.     public Education getEducation() { return education; }
70.     public void setEducation(Education newValue) { education = newValue; }
71.
```

```
72.     public Collection<SelectItem> getYearItems() { return birthYears; }
73.
74.     public Weekday[] getDaysOfTheWeek() { return daysOfTheWeek; }
75.
76.     public SelectItem[] getLanguageItems() { return languageItems; }
77.
78.     public SelectItem[] getColorItems() { return colorItems; }
79.
80.     public Map<String, Education> getEducationItems() { return educationItems; }
81.
82.     public String getBestDaysConcatenated() {
83.        return Arrays.toString(bestDaysToContact);
84.     }
85.
86.     public String getColorsConcatenated() {
87.        StringBuilder result = new StringBuilder();
88.        for (int color : colors) result.append(String.format("%06x ", color));
89.        return result.toString();
90.     }
91.
92.     private SelectItem[] colorItems = {
93.        new SelectItem(Color.RED.getRGB(), "Red"), // Значение, метка
94.        new SelectItem(Color.GREEN.getRGB(), "Green"),
95.        new SelectItem(Color.BLUE.getRGB(), "Blue"),
96.        new SelectItem(Color.YELLOW.getRGB(), "Yellow"),
97.        new SelectItem(Color.ORANGE.getRGB(), "Orange", "", true) //  Отменено
98.     };
99.
100.    private static Map<String, Education> educationItems;
101.    static {
102.       educationItems = new LinkedHashMap<String, Education>();
103.       educationItems.put("High School", Education.HIGH_SCHOOL); // Метка, значение
104.       educationItems.put("Bachelor's", Education.BACHELOR);
105.       educationItems.put("Master's", Education.MASTER);
106.       educationItems.put("Doctorate", Education.DOCTOR);
107.    };
108.
109.    private static SelectItem[] languageItems = {
110.       new SelectItem("English"),
111.       new SelectItem("French"),
112.       new SelectItem("Russian"),
113.       new SelectItem("Italian"),
114.       new SelectItem("Esperanto", "Esperanto", "", true) //  Отменено
115.    };
116.
117.    private static Collection<SelectItem> birthYears;
118.    static {
119.       birthYears = new ArrayList<SelectItem>();
120.       // Первым элементом является "вариант пустого выбора"
121.       birthYears.add(new SelectItem(null, "Pick a year:", "", false, false, true));
122.       for (int i = 1900; i < 2020; ++i) birthYears.add(new SelectItem(i));
123.    }
124.
125.    private static Weekday[] daysOfTheWeek;
126.    static {
127.       daysOfTheWeek = new Weekday[7];
128.       for (int i = Calendar.SUNDAY; i <= Calendar.SATURDAY; i++) {
129.          daysOfTheWeek[i - Calendar.SUNDAY] = new Weekday(i);
130.       }
131.    }
132. }
```

Листинг 4.15. Файл `select/src/java/com/corejsf/messages.properties`

```
indexWindowTitle=Checkboxes, Radio buttons, Menus, and Listboxes
indexPageTitle=Please fill out the following information

namePrompt=Name:
contactMePrompt=Contact me
bestDayPrompt=What's the best day to contact you?
yearOfBirthPrompt=What year were you born?
buttonPrompt=Submit information
backPrompt=Back
languagePrompt=Select the languages you speak:
educationPrompt=Select your highest education level:
emailAppPrompt=Select your email application:
colorPrompt=Select your favorite colors:

thankYouLabel=Thank you {0}, for your information
contactMeLabel=Contact me:
bestDayLabel=Best day to contact you:
yearOfBirthLabel=Your year of birth:
colorLabel=Colors:
languageLabel=Languages:
educationLabel=Education:
```

Листинг 4.16. Файл `select/web/resources/css/styles.css`

```
.emphasis {
   font-style: italic;
   font-size: 1.3em;
}
.disabled {
   color: gray;
}
.selected {
   font-weight: bold;
}
```

Сообщения

На протяжении жизненного цикла JSF любой объект может создать сообщение и добавить его к очереди сообщений, поддерживаемой контекстом `faces`. В конце жизненного цикла (на этапе подготовки ответа к отображению) можно отобразить эти сообщения в представлении. Как правило, сообщения связаны с конкретными компонентами и показывают, встречались ли ошибки преобразования или проверки правильности.

Безусловно, сообщения об ошибках относятся к самому распространенному типу сообщений в приложении JSF, но в целом сообщения подразделяются на четыре разновидности:

- информационные сообщения;
- предупреждения;
- сообщения об ошибках;
- сообщения о неисправимых ошибках.

Все сообщения могут содержать резюме и подробные сведения. Например, в качестве резюме может выступать "Недопустимая запись", а подробные сведения могут иметь вид: "Введенное число больше максимального".

В приложениях JSF используются два тега для отображения сообщений на страницах JSF: `h:messages` и `h:message`.

Тег `h:messages` отображает все сообщения, которые были сохранены в контексте `faces` в течение жизненного цикла JSF. Можно ограничить эти сообщения глобальными сообщениями, т.е. сообщениями, не связанными с конкретным компонентом, задавая значение атрибута `globalOnly` тега `h:message`, равное `true`. По умолчанию этот атрибут имеет значение `false`.

Тег `h:message` отображает единственное сообщение для конкретного компонента. Данный компонент определяется с помощью тега `h:message`, обязательного для атрибута. Если в отношении какого-то компонента было сформировано больше чем одно сообщение, то тег `h:message` показывает только последнее.

На заметку! Если используется версия JSF 2.0, а стадией проектирования является разработка, то к создаваемой странице автоматически добавляется тег, дочерний по отношению к `h:messages`, при условии, если таковой не добавлен самим разработчиком.

Теги `h:message` и `h:messages` имеют много общих атрибутов. Все атрибуты, применяемые в обоих тегах, перечислены в табл. 4.27.

Таблица 4.27. Атрибуты тегов `h:message` и `h:messages`

Атрибуты	Описание
`errorClass`	Класс CSS, применяемый для сообщений об ошибках
`errorStyle`	Стиль CSS, применяемый для сообщений об ошибках
`fatalClass`	Класс CSS, применяемый для сообщений о неисправимых ошибках
`fatalStyle`	Стиль CSS, применяемый для сообщений о неисправимых ошибках
`for`	Идентификатор компонента, к которому относится отображаемое сообщение (только `h:message`)
`globalOnly`	Указание, согласно которому должны отображаться только глобальные сообщения, применимое только к тегу `h:messages`. Значение по умолчанию — `false`
`infoClass`	Класс CSS, применяемый для информационных сообщений
`infoStyle`	Стиль CSS, применяемый для информационных сообщений
`layout`	Спецификация для макета сообщения: `"table"` или `"list"`, которая относится только к тегу `h:messages`
`showDetail`	Значение типа `Boolean`, которое определяет, должны ли отображаться подробные сведения, относящиеся к сообщению. Значения по умолчанию — `false` для тега `h:messages`, `true` — для тега `h:message`
`showSummary`	Значение типа `Boolean`, которое определяет, должны ли отображаться резюме, относящиеся к сообщениям. Значения по умолчанию — `true` для тега `h:messages`, `false` — для тега `h:message`
`tooltip`	Значение типа `Boolean`, которое определяет, должны ли подробные сведения, относящиеся к сообщению, подготавливаться к отображению в виде подсказки; такая подсказка формируется, только если значения `showDetail` и `showSummary` равны `true`
`warnClass`	Класс CSS для предупреждающих сообщений
`warnStyle`	Стиль CSS для предупреждающих сообщений
`binding, id, rendered`	Базовые атрибуты[1]

Окончание табл. 4.27

Атрибуты	Описание
style, styleClass, title, dir, lang	HTML 4.0[2]

[1] Дополнительные сведения о базовых атрибутах см. в табл. 4.5 на стр. 108.
[2] Дополнительные сведения об атрибутах HTML 4.0 см. в табл. 4.6 на стр. 110.

Большая часть атрибутов, приведенных в табл. 4.27, представляет классы или стили CSS, применяемые тегами `h:message` и `h:messages` к конкретным типам сообщений.

Можно также указать, следует ли отображать резюме или подробные сведения сообщения, или то и другое, с помощью атрибутов `showSummary` и `showDetail` соответственно.

Атрибут `layout` тега `h:messages` может использоваться для определения того, в каком виде должны располагаться сообщения, — в виде списка или таблицы. Если для атрибута `tooltip` задано значение `true` и значения `true` заданы также для атрибутов `showDetail` и `showSummary`, то подробные сведения, относящиеся к сообщению, будут оформлены в виде подсказки, отображаемой при перемещении курсора мыши над сообщением об ошибке.

Выше были приведены основные данные о том, по каким принципам обрабатываются сообщения, а теперь рассмотрим приложение, в котором используются теги `h:messages` и `h:message`. Приложение, показанное на рис. 4.20, содержит простую форму с двумя текстовыми полями. Оба текстовых поля имеют обязательные атрибуты.

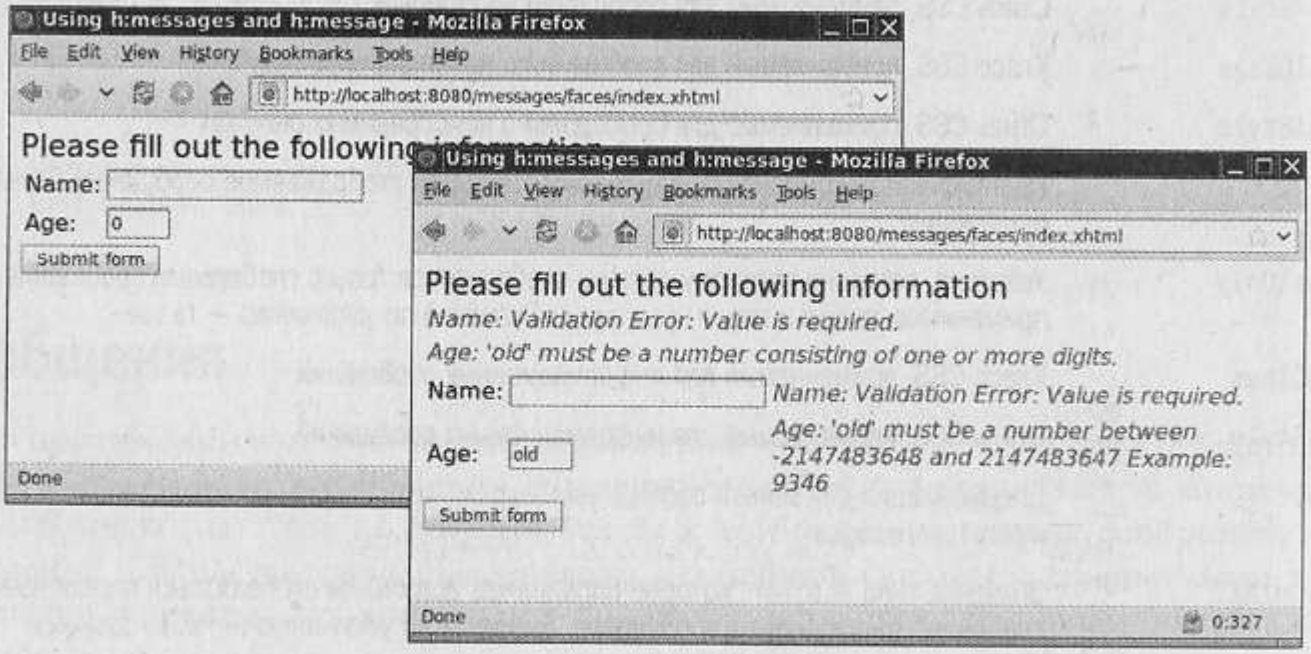


Рис. 4.20. Отображение сообщений

Кроме того, текстовое поле "Age" привязано к целочисленному свойству, поэтому его значение преобразуется платформой JSF автоматически. На рис. 4.20 показаны сообщения об ошибках, вырабатываемые платформой JSF, если пользователь забывает задать значение для поля "Name" и предоставляет значение неправильного типа для поля Age.

В верхней части страницы JSF отображаются все сообщения с применением тега `h:messages`. Тег `h:message` служит для отображения сообщений для каждого поля ввода:

```
<h:form>
  <h:messages layout="table" errorClass="errors"/>
  ...
```

```
    <h:inputText id="name"
      value="#{user.name}" required="true" label="#{msgs.namePrompt}"/>
    <h:message for="name" errorClass="errors"/>
      ...
    <h:inputText id="age"
      value="#{form.age}" required="true" label="#{msgs.agePrompt}"/>
    <h:message for="age" errorClass="errors"/>
    ...
</h:form>
```

Следует отметить, что поля ввода имеют атрибуты `label`, которые описывают эти поля. Такие метки используются в сообщениях об ошибках, например, как метка `Age` (сформированная с помощью выражения значения `#{msgs.agePrompt}`) в следующем сообщении:

```
Age: 'old' must be a number between -2147483648 and 2147483647 Example: 9346
```

Оба тега сообщений в рассматриваемом примере задают класс CSS с именем `errors`, который определен в файле `styles.css`. Определение этого класса выглядит следующим образом:

```
.errors {
    font-style: italic;
    color: red;
}
```

Кроме того, для тега `h:messages` задано значение `layout="table"`. Если бы этот атрибут был опущен (или применен альтернативный вариант `layout="list"`), то вывод был бы аналогичен тому, который показан на рис. 4.21.

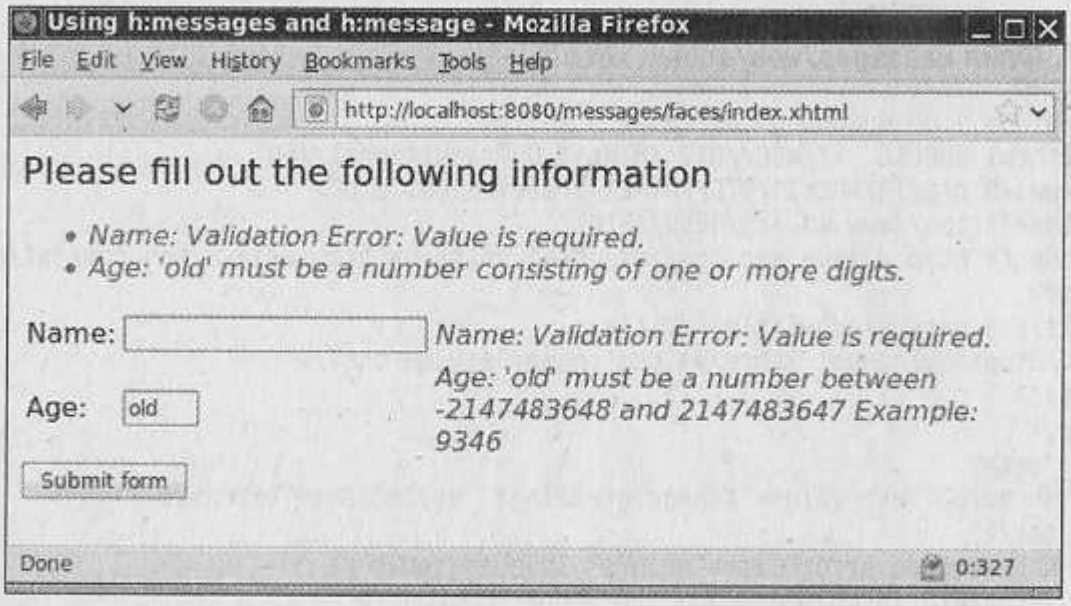


Рис. 4.21. Сообщения, отображенные в виде списка

С помощью макета списка сообщения об ошибках представляются в виде ненумерованного списка (внешним видом которого можно управлять с применением стилей).

Внимание! В версии JSF 1.1 стиль `"list"` предусматривал размещение сообщений одного за другим, без каких-либо разделителей, а это было не очень удобно.

На рис. 4.22 показана структура каталогов для приложения, приведенного на рис. 4.20. В листингах 4.17–4.19 представлены страница JSF, связка ресурсов и таблица стилей для приложения. По условиям этого примера были добавлены методы `getAge` и `setAge` к классу `UserBean`.

На заметку! По умолчанию тег h:messages показывает резюме сообщения, но не подробные сведения. Тег h:message, с другой стороны, отображает подробные сведения, но не резюме. При совместном использовании тегов h:messages и h:message, как было сделано в предыдущем примере, резюме появляются в верхней части страницы, а подробные сведения — рядом с соответствующим полем ввода.

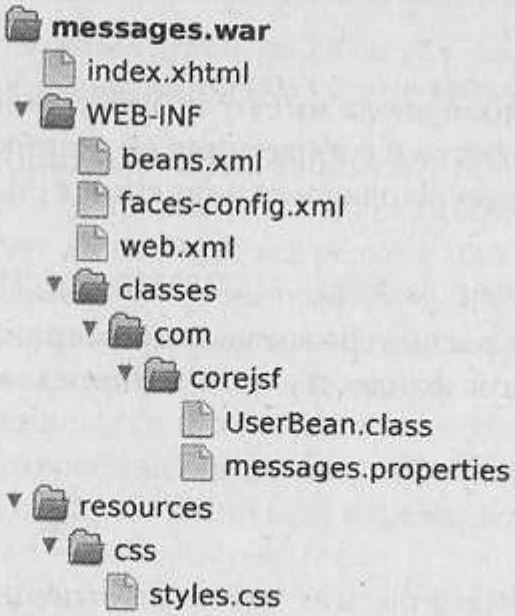


Рис. 4.22. Структура каталогов для примера применения сообщений

Листинг 4.17. Файл messages/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.windowTitle}</title>
      <h:outputStylesheet library="css" name="styles.css"/>
   </h:head>
   <h:body>
      <h:form>
         <h:outputText value="#{msgs.greeting}" styleClass="emphasis"/>
         <br/>
         <h:messages errorClass="errors" layout="table"/>
         <h:panelGrid columns="3">
            #{msgs.namePrompt}:
            <h:inputText id="name" value="#{user.name}" required="true"
                         label="#{msgs.namePrompt}"/>
            <h:message for="name" errorClass="errors"/>
            #{msgs.agePrompt}:
            <h:inputText id="age" value="#{user.age}" required="true"
                         size="3" label="#{msgs.agePrompt}"/>
            <h:message for="age" errorClass="errors"/>
         </h:panelGrid>
         <h:commandButton value="#{msgs.submitPrompt}"/>
      </h:form>
   </h:body>
</html>
```

Листинг 4.18. Файл messages/src/java/com/corejsf/messages.properties

```
1. windowTitle=Using h:messages and h:message
2. greeting=Please fill out the following information
3. namePrompt=Name
4. agePrompt=Age
5. submitPrompt=Submit form
```

Листинг 4.19. Файл messages/web/resources/css/styles.css

```
1. .errors {
2.    font-style: italic;
3.    color: red;
4. }
5. .emphasis {
6.    font-size: 1.3em;
7. }
```

Резюме

Выше были описаны все теги HTML, определенные в стандартной библиотеке, за исключением тегов, используемых для таблиц, которые рассматриваются в главе 6. В следующей главе будет показано, как использовать теги Facelets.

Технология Facelets

В этой главе...

- Теги Facelets
- Применение шаблонов на основе платформы Facelets
- Специализированные теги
- Нерассмотренные вопросы

Глава 5

Как правило, пользовательские интерфейсы в ходе разработки веб-приложений подвержены наиболее частым изменениям и во многих случаях состоят из кода, на разработку которого затрачиваются большие усилия, с трудом допускающего корректировку, поэтому процесс создания пользовательских интерфейсов становится дорогостоящим. В настоящей главе показано, как можно реализовать гибкие пользовательские интерфейсы на основе платформы Facelets.

Теги Facelets

Технология Facelets была первоначально разработана как альтернатива обработчику представлений на основе JSP, применявшемуся в версиях JSF 1.x. В версии JSF 2.0 технология Facelets заменила JSP в качестве применявшейся по умолчанию в JSF технологии представления. Платформа Facelets не только является лучшим обработчиком представлений, но и поддерживает целый ряд тегов, предназначенных для реализации шаблонов и других целей. Эти теги служат темой настоящей главы.

Теги Facelets могут быть сгруппированы по нескольким категориям.

- Включение содержимого из других страниц XHTML (`ui:include`).
- Формирование страниц из шаблонов (`ui:composition`, `ui:decorate`, `ui:insert`, `ui:define`, `ui:param`).
- Создание пользовательских компонентов без написания кода Java (`ui:component`, `ui:fragment`).
- Различные утилиты (`ui:debug`, `ui:remove`, `ui:repeat`).

Чтобы иметь возможность использовать теги Facelets, необходимо добавить следующее объявление пространства имен к конкретным страницам JSF:

```
xmlns:ui="http://java.sun.com/jsf/facelets"
```

В табл. 5.1 приведены краткие сведения о тегах Facelets. Затем эти теги будут подробно описаны в настоящей главе, за исключением тега `ui:repeat`, который рассматривается в главе 6.

Таблица 5.1. Теги Facelets

Тег	Описание
`ui:include`	Включает содержимое из другого файла XML
`ui:composition`	Будучи используемым без атрибута `template`, тег `ui:composition` определяет последовательность элементов, которая может быть вставлена в другом месте. Композиция может иметь переменные части (указанные с помощью дочерних тегов `ui:insert`). Если тег `ui:composition` используется с атрибутом `template`, загружается шаблон. Дочерние теги этого тега определяют переменные части шаблона. Содержимое шаблона заменяет этот тег
`ui:decorate`	Будучи используемым без атрибута `template`, тег `ui:decorate` определяет страницу, в которую могут быть вставлены части. Переменные части задаются с помощью дочерних тегов `ui:insert`. Если тег `ui:composition` используется с атрибутом `template`, загружается шаблон. Дочерние теги этого тега определяют переменные части шаблона
`ui:define`	Определяет содержимое, которое вставляется в шаблон с помощью соответствующих тегов `ui:insert`
`ui:insert`	Вставляет содержимое в шаблон. Это содержимое определяется в теге, который загружает шаблон
`ui:param`	Задает параметр, передаваемый во включенный файл или шаблон
`ui:component`	Этот тег идентичен `ui:composition`, за исключением того, что создает компонент, добавляемый к дереву компонентов
`ui:fragment`	Этот тег идентичен `ui:decorate`, за исключением того, что создает компонент, добавляемый к дереву компонентов
`ui:debug`	Тег `ui:debug` позволяет пользователю с помощью определенной комбинации клавиш вывести на экран окно отладки, в котором показаны иерархия компонентов для текущей страницы и переменные с областью действия приложения
`ui:remove`	Реализация JSF удаляет все, что находится в тегах `ui:remove`
`ui:repeat`	Выполняет итерации по списку, массиву, результирующему набору или отдельному объекту. (Подробнее об этом речь пойдет в главе 6.)

Применение шаблонов на основе платформы Facelets

Большинство веб-приложений написано по одинаковому образцу, характеризующемуся тем, что все страницы имеют общий макет и в них применяются одни и те же стили. Например, общепринято, чтобы страницы имели однотипные заголовки, нижние колонтитулы и боковые меню.

Технология Facelets позволяет представить эти общие особенности в виде одного шаблона, чтобы можно было работать над совершенствованием внешнего вида сайта, внося изменения в шаблон, а не на отдельные страницы.

На заметку! Шаблоны Facelets заключают в себе функциональные средства, общие для многих страниц, поэтому отпадает необходимость задавать эти функции отдельно для каждой страницы. Такая инкапсуляция является краеугольным камнем и объектно-ориентированного программирования, и широко применяемого принципа DRY (Don't Repeat Yourself — не делать дважды одно и то же).

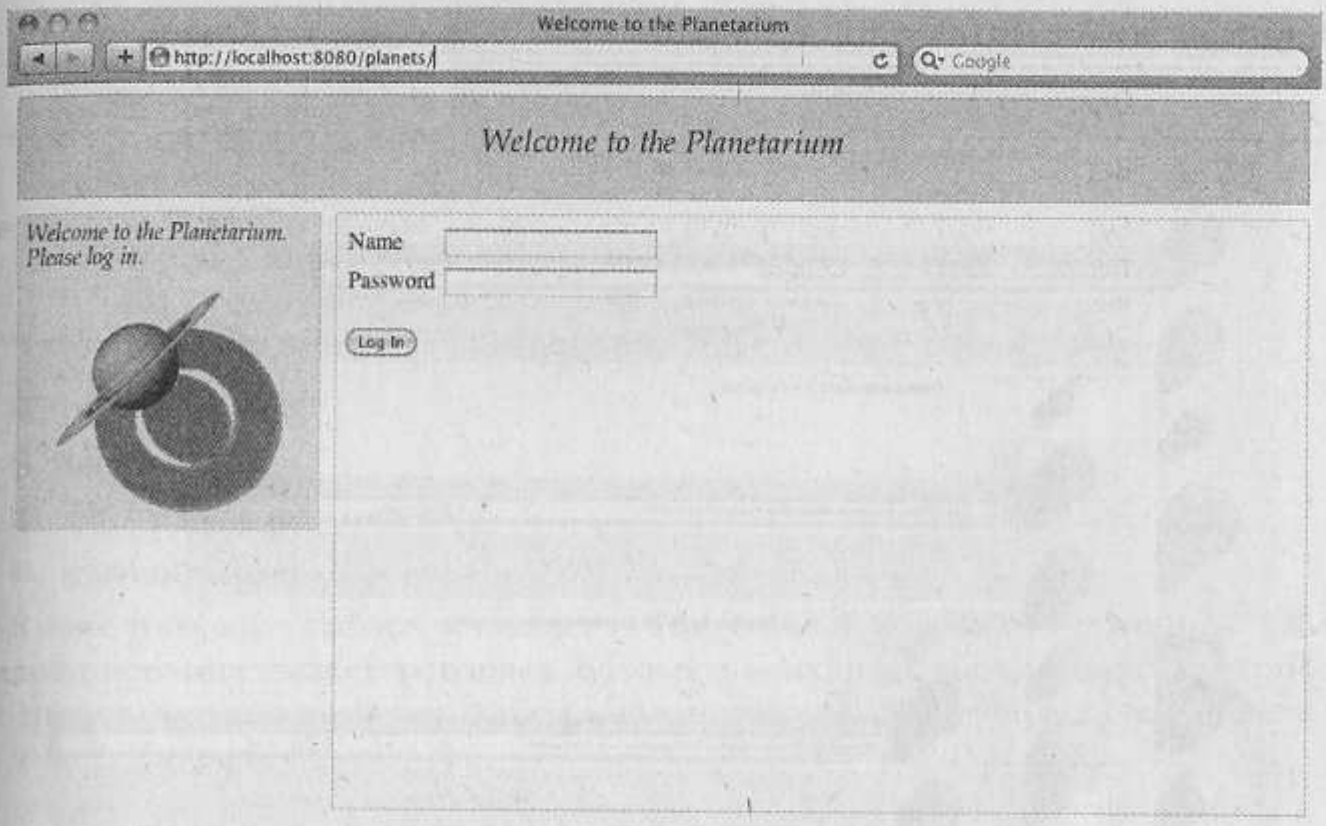


Рис. 5.1. Вход в программу планетария

В качестве простого примера рассмотрим приложение, которое отображает сведения о планетах Солнечной системы (рис. 5.1 и 5.2).

Приложение planets в общей сложности состоит из десяти страниц: страница входа, начальная страница и страница для каждой из планет. На всех этих страницах совместно используется общий макет, с заголовком сверху, боковым меню слева и областью содержимого справа от бокового меню. На страницах также совместно используются некоторое содержимое (все страницы планет имеют идентичные заголовки и боковые меню) и таблица стилей CSS.

Формирование страниц исходя из общих шаблонов

В листинге 5.1 показан шаблон для приложения planets.

Листинг 5.1. Файл `planets/web/templates/masterLayout.xhtml`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">

   <h:head>
      <title><ui:insert name="windowTitle"/></title>
      <h:outputStylesheet library="css" name="styles.css"/>
   </h:head>

   <h:body>
```

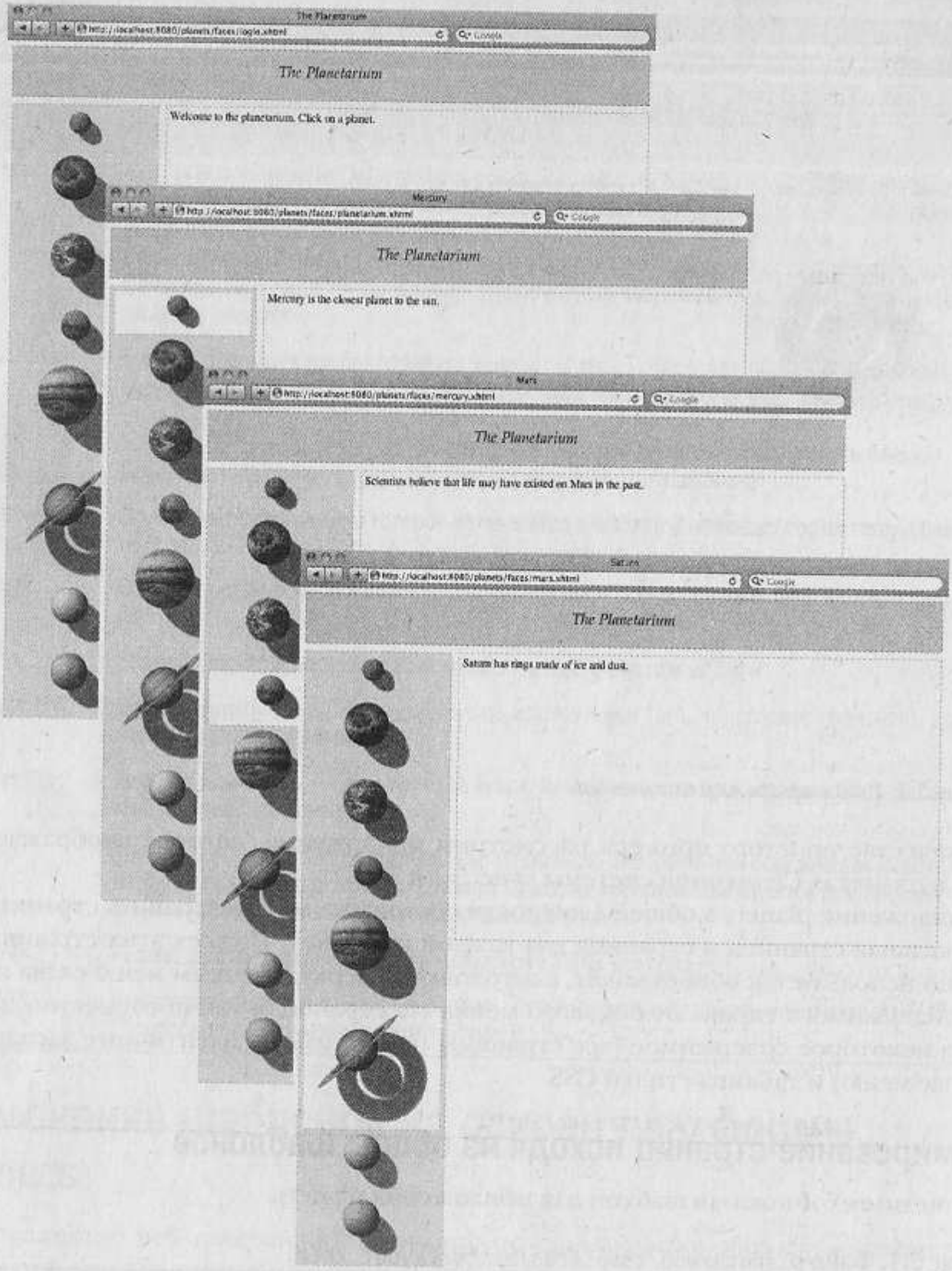


Рис. 5.2. Общий шаблон, совместно используемый во всех представлениях планетария

```
14.     <div id="heading">
15.        <ui:insert name="heading">
16.           <ui:include src="/sections/planetarium/header.xhtml"/>
17.        </ui:insert>
18.     </div>
19.
20.
21.     <div id="sidebarLeft">
22.        <ui:insert name="sidebarLeft">
23.           <ui:include src="/sections/planetarium/sidebarLeft.xhtml"/>
24.        </ui:insert>
```

```
      </div>

      <div id="content">
         <ui:insert name="content"/>
      </div>
      <ui:debug/>
   </h:body>
</html>
```

В этом шаблоне четыре раза используется тег `ui:insert` для вставки

- названия окна;
- заголовка;
- левого бокового меню;
- главного содержимого.

Кроме того, этот шаблон вставляет таблицу стилей в заголовок страницы. Таблица стилей определяет макет заголовка, бокового меню и главного содержимого. Предусмотрена возможность задать содержимое, применяемое по умолчанию, в тексте тега `ui:insert`. Например:

```
<ui:insert name="header">
  Default header goes here
</ui:insert>
```

Это значение по умолчанию используется, если при вызове шаблона не задается какая-либо замена для заголовка.

Общепринято использовать тег `ui:include` для включения применяемого по умолчанию содержимого из другого файла:

```
<ui:insert name="header">
  <ui:include src="/sections/planetarium/header.xhtml">
</ui:insert>
```

Мы используем предусмотренное по умолчанию содержимое для заголовка и левого бокового меню.

Для ввода в действие шаблона применяется тег `ui:composition` с атрибутом `template`, как показано в листинге 5.2.

Листинг 5.2. Файл `planets/web/saturn.xhtml`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
   <head><title>IGNORED</title></head>
   <body>
      <ui:composition template="/templates/masterLayout.xhtml">
         <ui:define name="windowTitle">
            #{msgs.saturn}
         </ui:define>

         <ui:define name="content">
            Saturn has rings made of ice and dust.
         </ui:define>
      </ui:composition>
   </body>
</html>
```

Реализация Facelets удаляет все теги вне тега ui:composition; иными словами, объявления типа документа, а также теги html, head, title и body. Необходимость в этом обусловлена тем, что тег ui:composition заменяется шаблоном, содержащим собственный набор тегов html, head, title и body.

Фактически вместо этого можно было бы просто подготовить файл saturn.xhtml со следующим кодом XML:

```
<ui:composition template="/templates/masterLayout.xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
   <ui:define name="windowTitle">
      #{msgs.saturn}
   </ui:define>

   <ui:define name="content">
      Saturn has rings made of ice and dust.
   </ui:define>
</ui:composition>
```

Эта форма короче, а также, возможно, вызывает меньше путаницы. Однако при использовании подобного файла разработчик не может получить такой объем помощи от применяемой интегрированной среды разработки, как при редактировании обычного файла Facelets. По этой причине имеет смысл включать композицию в отформатированную должным образом страницу XHTML, как было сделано в листинге 5.2.

Теги ui:define в теге ui:composition соответствуют тегам ui:insert шаблона, который показан в листинге 5.1. Например,

```
<ui:define name="content">
   Saturn has rings made of ice and dust.
</ui:define>
```

в композиции соответствует

```
<ui:insert name="content"/>
```

в шаблоне.

Во время загрузки шаблона каждый тег ui:insert заменяется содержимым соответствующего тега ui:define.

На всех страницах приложения planets используется одинаковый шаблон. В следующих примерах показано, как этот шаблон применяется для некоторых из указанных страниц. (Различия отмечены полужирным шрифтом.)

Файл mars.xhtml

```
<ui:composition template="/templates/masterLayout.xhtml">
   <ui:define name="windowTitle">
      #{msgs.mars}
   </ui:define>

   <ui:define name="content">
      Scientists believe that life may have existed on Mars in the past.
   </ui:define>
</ui:composition>
```

Файл planetarium.xhtml

```
<ui:composition template="/templates/masterLayout.xhtml">
   <ui:define name="windowTitle">

   <ui:define name="content">
      #{msgs.planetariumWelcome}
   </ui:define>
</ui:composition>
```

Файл login.xhtml

```
<ui:composition template="/templates/masterLayout.xhtml">
  <ui:define name="windowTitle">
    #{msgs.loginTitle}
  </ui:define>

  <ui:define name="heading">
    <ui:include src="/sections/login/header.xhtml"/>
  </ui:define>

  <ui:define name="sidebarLeft">
    <ui:include src="/sections/login/sidebarLeft.xhtml"/>
  </ui:define>

  <ui:define name="content">
    <h:form>
      <h:panelGrid columns="2">
        #{msgs.namePrompt}
        <h:inputText id="name" value="#{user.name}"/>
        #{msgs.passwordPrompt}
        <h:inputSecret id="password" value="#{user.password}"/>
      </h:panelGrid>
      <p>
        <h:commandButton value="#{msgs.loginButtonText}"
                         action="planetarium"/>
      </p>
    </h:form>
  </ui:define>
</ui:composition>
```

Обратите внимание на то, что на странице `login.xhtml` происходит замена заданного по умолчанию содержимого для заголовка и бокового меню.

Организация конкретных представлений

По существу, задание шаблона приводит к разбиению представления на две страницы XHTML: на ту, которая определяет общие функциональные средства (шаблон), и ту, с помощью которой задаются функции, различающиеся между представлениями (композиция).

Несмотря на то что эта методика применения шаблонов достаточно проста, она позволяет создавать весьма доступные для корректировки и расширяемые пользовательские интерфейсы. Рассмотрим приложение planets на более высоком уровне, чтобы узнать, как это делается.

На рис. 5.3 показаны файлы, из которых состоят шаблон и представления для приложения planets.

Мало того, что представления приложения planets разбиты на общий шаблон и композиции, но и каждая часть содержимого также вынесена в результате разбиения в собственный файл; например, в представлениях входа в систему и планетария имеется по одному файлу и для заголовка, и для бокового меню. Эти отдельные части содержимого включаются каждым представлением с помощью тега `ui:include`. Например, на странице входа включение заголовка происходит следующим образом:

```
<ui:define name="heading">
  <ui:include src="sections/login/header.xhtml"/>
</ui:define>
```

Благодаря тому, что отдельные части содержимого определяются в своих собственных файлах, становится проще определять местонахождение кода при внесении изменений в то или иное представление. Например, если потребуется что-то изменить в боковом меню представления входа в систему, то будет известно, что для этого достаточно отредактировать файл `sections/login/sidebarLeft.xhtml`, не затрудняя себя поиском этого определения бокового меню в одном большом файле. Разбиение на разделы способствует упрощению чтения, понимания и внесения изменений на конкретные страницы, поскольку каждый файл содержит небольшой объем разметки.

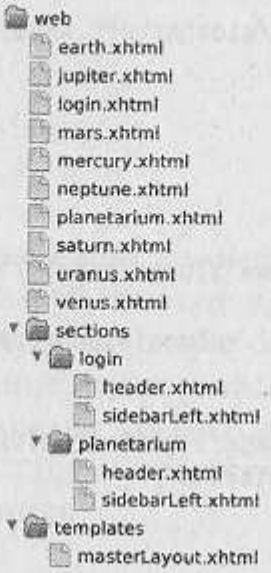


Рис. 5.3. Страницы приложения planets, разделы и шаблон

На страницах рассматриваемого приложения не используются отдельные файлы для разделов содержимого. Но по желанию можно вынести раздел содержимого в отдельный файл Facelets, такой как `/sections/login/content.xhtml`.

На заметку! Один из принципов, лежащих в основе языка Smalltalk, состоит в использовании подхода к проектированию, называемому *методом составления*. Этот принцип побуждает разработчика создавать до предела упрощенные методы с небольшим объемом кода, из которых составляется конкретное приложение. Задача написания, изучения и совершенствования таких простейших методов с небольшим объемом кода намного проще по сравнению с тем подходом, когда методы воплощают в себе большое количество функций и требуют для реализации значительный объем кода. Кроме того, становится проще заменять или модифицировать функциональные средства, если они состоят из функций, представленных в виде небольших фрагментов кода.

С выходом версии JSF 2.0 появилась возможность использовать метод составления для реализации конкретных представлений. Если представления компонуются из небольших файлов XHTML, каждый из которых выполняет единственную четко заданную функцию, такую как вывод формы регистрации, то задача реализации, сопровождения и расширения представлений, из которых состоит приложение, намного упрощается.

Чтобы завершить наше исследование того, как в приложении planets применяются средства поддержки шаблонов Facelets, рассмотрим файлы, находящиеся в каталоге `sections`. Файлы в этом каталоге содержат разделы страниц, включаемые в шаблон.

В листингах 5.3–5.5 показана разметка XHTML, с помощью которой создаются заголовок представления входа в систему, меню и содержимое.

Заголовок представления входа в систему приведен на рис. 5.4.

Рис. 5.4. Заголовок представления входа в систему

Реализация этого заголовка показана в листинге 5.3.

Обратите внимание на то, что содержимое (в данном случае текст заголовка) размещается в теге `ui:composition`, который не задает шаблон. В этом примере тег `ui:composition` применяется по тактической причине: он позволяет уничтожить окружающие теги XHTML.

Листинг 5.3. Файл `planets/web/sections/login/header.xhtml`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
   <head><title>IGNORED</title></head>
   <body>
      <ui:composition>
         <div class="header">
            #{msgs.loginHeading}
         </div>
      </ui:composition>
   </body>
</html>
```

Если бы не было тега `ui:composition`, то в конечном итоге сформированный код включал бы несколько тегов `<html>`, поскольку следующий далее тег `<ui:include src="/sections/login/header.xhtml">` предусматривает включение всего файла.

Как правило, каждый раз при включении содержимого с использованием тега `ui:include` необходимо ограничивать включенное содержимое пределами тега `ui:composition`.

На заметку! Как уже было сказано, нет необходимости помещать окружающие теги XHTML во включаемый файл. Но все еще требуется помещать содержимое (которое обычно состоит из последовательности тегов) в тег `ui:composition`, чтобы включенный файл представлял собой формально правильный код XML.

Ниже показано боковое меню представления входа в систему (рис. 5.5).

Боковое меню входа в систему реализуется так, как показано в листинге 5.4.

Рис. 5.5. Боковое меню представления входа в систему

Листинг 5.4. Файл planets/web/sections/login/sidebarLeft.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
   <head><title>IGNORED</title></head>
   <body>
      <ui:composition>
         <div class="welcome">
            #{msgs.loginWelcome}
            <div class="welcomeImage">
               <h:graphicImage library="images" name="Saturn.gif"/>
            </div>
         </div>
      </ui:composition>
   </body>
</html>
```

Наконец, область содержимого страницы входа приведена на рис. 5.6.

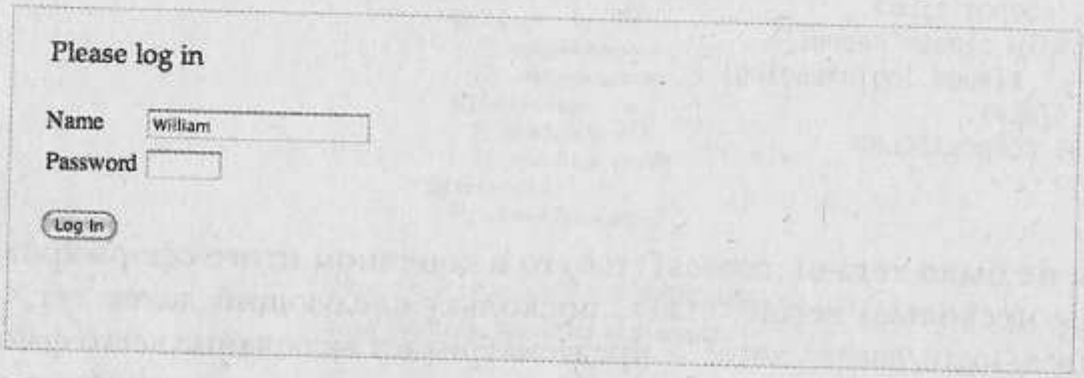


Рис. 5.6. Область содержимого страницы входа

Раздел содержимого включен на страницу login.xhtml, как показано в листинге 5.5.

Листинг 5.5. Файл planets/web/login.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
   <head><title>IGNORED</title></head>
   <body>
      <ui:composition template="/templates/masterLayout.xhtml">
         <ui:define name="windowTitle">
            #{msgs.loginTitle}
         </ui:define>

         <ui:define name="heading">
            <ui:include src="/sections/login/header.xhtml"/>
         </ui:define>

         <ui:define name="sidebarLeft">
            <ui:include src="/sections/login/sidebarLeft.xhtml"/>
         </ui:define>

         <ui:define name="content">
            <h:form>
               <h:panelGrid columns="2">
```

```
                #{msgs.namePrompt}
                <h:inputText id="name" value="#{user.name}"/>
                #{msgs.passwordPrompt}
                <h:inputSecret id="password" value="#{user.password}"/>
            </h:panelGrid>
            <p>
                <h:commandButton value="#{msgs.loginButtonText}"
                                 action="planetarium"/>
            </p>
        </h:form>
    </ui:define>
</ui:composition>
</body>
</html>
```

Теперь рассмотрим содержимое приложения планетария.

В листингах 5.6–5.8 показана разметка XHTML, применяемая для создания заголовка представления планетария, меню и содержимого.

Листинг 5.6. Файл planets/web/sections/planetarium/header.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
  <head><title>IGNORED</title></head>
  <body>
    <ui:composition>
      <div class="header">
        #{msgs.planetariumHeading}
      </div>
    </ui:composition>
  </body>
</html>
```

Листинг 5.7. Файл planets/web/sections/planetarium/sidebarLeft.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:corejsf="http://corejsf.com/facelets">
  <head><title>IGNORED</title></head>
  <body>
    <ui:composition>
      <h:form>
        <corejsf:planet name="mercury"
                        image="#{resource['images:Mercury.gif']}"/>
        <corejsf:planet name="venus"
                        image="#{resource['images:Venus.gif']}"/>
        <corejsf:planet name="earth"
                        image="#{resource['images:Earth.gif']}"/>
        <corejsf:planet name="mars"
                        image="#{resource['images:Mars.gif']}"/>
        <corejsf:planet name="jupiter"
                        image="#{resource['images:Jupiter.gif']}"/>
        <corejsf:planet name="saturn"
                        image="#{resource['images:Saturn.gif']}"/>
        <corejsf:planet name="uranus"
                        image="#{resource['images:Uranus.gif']}"/>
```

```
25.             <corejsf:planet name="neptune"
26.                             image="#{resource['images:Neptune.gif']}"/>
27.          </h:form>
28.       </ui:composition>
29.    </body>
30. </html>
```

Листинг 5.8. Файл planets/web/planetarium.xhtml

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3. <html xmlns="http://www.w3.org/1999/xhtml"
4.       xmlns:ui="http://java.sun.com/jsf/facelets">
5.    <head><title>IGNORED</title></head>
6.    <body>
7.       <ui:composition template="/templates/masterLayout.xhtml">
8.          <ui:define name="windowTitle">
9.             #{msgs.planetariumTitle}
10.         </ui:define>
11.
12.         <ui:define name="content">
13.            #{msgs.planetariumWelcome}
14.         </ui:define>
15.      </ui:composition>
16.   </body>
17. </html>
```

Разделы страниц планетария аналогичны разделам страницы входа в систему — каждый раздел представляет собой композицию, которая определяет часть представления планетария.

Заслуживает внимания то, как используется тег `corejsf:planet` в листинге 5.7. Данный тег, а также все специализированные теги Facelets в целом будут рассматриваться в разделе "Специализированные теги" на стр. 178.

Декораторы

Шаблон, показанный в предыдущем разделе, определял страницу по принципу размещения отдельных частей. При использовании шаблона задается содержимое каждой части. В этом просматривается аналогия с платформой Tiles (`http://tiles.apache.org`), которая может применяться с реализациями Struts и JSF 1.x.

Если применяемый набор страниц является достаточно сложным, то подход на основе платформы Tiles предоставляет разработчику более широкие возможности. А что касается простого приложения, то трактовка каждой страницы как сборки разделов, по-видимому, является чрезмерно усложненной. В большей степени сосредоточенным на содержимом является подход на основе декораторов. При этом страницы разрабатываются как обычно, но содержимое заключается в теги `ui:decorate`, которые имеют атрибут `template`. Подход на основе декораторов — это применяемый в технологии Facelets аналог платформы Sitemesh (`http://www.opensymphony.com/sitemesh/`). При использовании платформы Sitemesh, как и в подходе на основе декораторов, вначале осуществляется проектирование содержимого, а затем его оформление.

В своей простейшей форме декоратор может использоваться примерно так:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
```

```
      xmlns:ui="http://java.sun.com/jsf/facelets">
   <head><title>#{msgs.loginTitle}</title></head>
   <body>
      <ui:decorate template="/templates/masterDecorator.xhtml">
         <!-- Содержимое, подлежащее оформлению -->
         <h:form>
            <h:panelGrid columns="2">
#{msgs.namePrompt}
<h:inputText id="name" value="#{user.name}"/>
#{msgs.passwordPrompt}
<h:inputSecret id="password" value="#{user.password}"/>
            </h:panelGrid>
            <p>
               <h:commandButton value="#{msgs.loginButtonText}"
                  action="planetarium"/>
            </p>
         </h:form>
      </ui:decorate>
   </body>
</html>
```

С помощью шаблона осуществляется оформление его содержимого определенным способом, скажем, с применением заголовка и левого бокового меню. Обратите внимание на то, что теги, находящиеся вне тега `ui:decorate`, не удаляются (как было бы в случае применения тега `ui:composition`). В рассматриваемом примере автор страницы определил название страницы непосредственно на самой странице без применения шаблонов.

Шаблон определен следующим образом:

```
Необязательный заголовок XHTML
<ui:composition>
   <h:outputStylesheet library="css" name="styles.css" target="body"/>
   <div id="heading">
      <ui:insert name="heading">Default header</ui:insert>
   </div>

   <div id="sidebarLeft">
      <ui:insert name="sidebarLeft">Default sidebar</ui:insert>
   </div>
   <div id="content">
      <ui:insert/>
   </div>
</ui:composition>
Необязательный нижний колонтитул XHTML
```

Следует отметить, что тег `<ui:insert/>` не имеет атрибута `name`. Он вставляет все теги, дочерние по отношению к тегу `ui:decorate`.

Заслуживает также внимания тег `ui:composition`, который окружает команды макета в шаблоне. В данном случае не требуется, чтобы теги заголовка и нижнего колонтитула XHTML становились частью шаблона – страницы, которая уже оформляется с применением собственных тегов XHTML. При использовании декораторов, как и при использовании композиции, можно переопределять значения по умолчанию с помощью тегов `ui:define`, как в следующем примере:

```
<ui:decorate template="/templates/masterDecorator.xhtml">
   <ui:define name="heading">Special Header</ui:define>
   Текст
</ui:decorate>
```

Различие между тегами `ui:composition` и `ui:decorator` является главным образом концептуальным. Одни и те же результаты могут быть получены с помощью того и

другого тега. В технологии Facelets эти теги просто рассматриваются как взаимно дополняющие конструкции: тег ui:composition удаляет все окружающее содержимое, тогда как применение тега ui:decorator к этому не приводит (и поэтому требует наличия тега ui:composition в шаблоне).

Параметры

При вызове шаблона можно передавать параметры двумя способами: с помощью тега ui:define и тега ui:param. Как уже было сказано, тег ui:define используется для предоставления разметки, вставляемой в шаблон. В отличие от этого тег ui:param задает переменную языка выражений для использования в шаблоне:

```
<ui:composition template="templates/masterTemplate.xhtml">
  <ui:param name="currentDate" value="#{someBean.currentDate}"/>
</ui:composition>
```

В соответствующем шаблоне можно обращаться к параметру с помощью выражения языка выражений, как в следующем примере:

```
...
<body>
  Today's date: #{currentDate}"/>
</body>
...
```

Тег ui:param может также использоваться как дочерний по отношению к тегу ui:include.

Специализированные теги

Выше в данной главе было показано, как расположить элементы пользовательского интерфейса с помощью шаблонов. В дополнение к этому технология Facelets позволяет определять специализированные теги. Специализированный тег внешне напоминает обычный тег JSF, но в нем используется механизм композиции Facelets для вставки содержимого на конкретную страницу.

Например, ссылки на страницы со сведениями о планетах в боковом меню приложения planets создаются с помощью специализированного тега, который показан в листинге 5.7 на стр. 175:

```
<corejsf:planet name="#{mercury}"/>
```

Тег corejsf:planet создает ссылку с изображением значка соответствующей планеты, как показано в меню на рис. 5.2, стр. 168. Когда пользователь щелкает на ссылке, приложение показывает сведения о выбранной планете. Реализация специализированного тега Facelets с помощью платформы JSF 2.0 представляет собой двухэтапный процесс.

1. Реализация специализированного тега (или компонента) в файле XHTML.
2. Объявление специализированного тега в теге библиотеки тегов.

В листинге 5.9 показана реализация тега corejsf:planet.

Листинг 5.9. Файл planets/web/WEB-INF/tags/corejsf/planet.xhtml

```
1. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3.
4. <html xmlns="http://www.w3.org/1999/xhtml"
```

```
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
   <h:head><title>IGNORED</title></h:head>
   <h:body>
      <ui:composition>
         <div class='#{name == planetarium.selectedPlanet ?
                     "planetImageSelected" : "planetImage"}'>
            <h:commandLink action="#{planetarium.changePlanet(name)}">
               <h:graphicImage value="#{image}"/>
            </h:commandLink>
            <ui:insert name="content1"/>
         </div>
      </ui:composition>
   </h:body>
</html>
```

После того как пользователь щелкнет на ссылке, созданной с помощью специализированного тега, вызывается метод changePlanet класса Planetarium. Этот метод просто осуществляет переход к странице выбранной планеты. Класс Planetarium показан в листинге 5.10.

Листинг 5.10. Файл planets/src/com/corejsf/Planetarium.java

```
package com.corejsf;

import java.io.Serializable;
import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.RequestScoped;
   // или import javax.faces.bean.RequestScoped;

@Named // или @ManagedBean
@RequestScoped
public class Planetarium implements Serializable {
private String selectedPlanet;

public String getSelectedPlanet() { return selectedPlanet; }

public String changePlanet(String newValue) {
   selectedPlanet = newValue;
   return selectedPlanet;
}
}
```

Чтобы можно было использовать тег corejsf:planet, его необходимо объявить в файле библиотеки тегов. Этот файл определяет следующее.

- Пространство имен для тегов в этой библиотеке (такое как http://corejsf.com/facelets, которое отображается на префикс наподобие corejsf: на странице с использованием тегов).
- Имя каждого тега (такое как planet).
- Местонахождение шаблона (в данном случае tags/corejsf/planet.xhtml).

В листинге 5.11 показан код, предназначенный для файла библиотеки тегов.

Листинг 5.11. Файл planets/web/WEB-INF/corejsf.taglib.xml

```
<?xml version="1.0"?>
<!DOCTYPE facelet-taglib PUBLIC
   "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
```

```
4.      "http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
5. <facelet-taglib>
6. <namespace>http://corejsf.com/facelets</namespace>
7. <tag>
8.      <tag-name>planet</tag-name>
9.      <source>tags/corejsf/planet.xhtml</source>
10. </tag>
11. </facelet-taglib>
```

Затем необходимо задать местоположение файла библиотеки тегов в файле web.xml:

```
<context-param>
   <param-name>facelets.LIBRARIES</param-name>
   <param-value>/WEB-INF/corejsf.taglib.xml</param-value>
</context-param>
```

Если должно быть задано несколько файлов библиотек тегов, их необходимо разделить точками с запятой.

Можно также упаковать набор тегов Facelets в виде файла JAR. Разместите файлы шаблона и все необходимые ресурсы в файле JAR. Затем перенесите файл библиотеки тегов в каталог META-INF. Выбор имени файла библиотеки тегов не имеет значения, при условии, что это имя заканчивается суффиксом taglib.xml.

Задача реализации специализированных тегов Facelets в технологии JSF 2.0 является несложной, поэтому применение такого подхода настоятельно рекомендуется для исключения повторяющейся разметки.

Однако следует учитывать, что специализированные теги Facelets не являются столь же мощными, как всесторонне развитые компоненты JSF. В частности, за специализированными тегами Facelets нельзя закрепить дополнительные функции, например, выполняемые средствами проверки или прослушивателями. (Дополнительная информация о средствах проверки и прослушивателях событий приведена в главах 7 и 8.) Например, невозможно добавить прослушиватель action к тегу corejsf:planet. В версии JSF 2.0 этот недостаток устранен благодаря применению более развитого механизма компонентов, основанного на так называемых составных компонентах. Составные компоненты будут рассматриваться в главе 9.

Компоненты и фрагменты

В шаблоне для специализированного тега planet определен тег ui:composition. Если используется этот тег, он заменяется дочерними элементами композиции. Если тег ui:composition в листинге 5.9 на стр. 178 будет заменен тегом ui:component, это приведет к размещению дочерних элементов в компоненте JSF. Затем этот компонент добавляется к представлению.

В сочетании с тегом ui:component можно задавать атрибуты id, binding и rendered. Необходимость в этом может быть обусловлена двумя причинами. Если для привязки компонента к бину используется атрибут binding, то появляется возможность манипулировать компонентом программным путем. Кроме того, появляется возможность подготавливать компонент к отображению по условию, присваивая атрибуту rendered результат выражения значения. Аналогичным образом тег ui:fragment является аналогом тега ui:decorate, создающего компонент. Фрагмент можете использовать в теге ui:composition или ui:component для включения по условию дочерних тегов:

```
<ui:fragment rendered="#{name == planetarium.selectedPlanet}">
  Включаемый по условию дочерний элемент
</ui:fragment>
```

Нерассмотренные вопросы

В настоящем разделе будут рассматриваться остальные теги из числа приведенных в табл. 5.1 на стр. 166, за исключением тега `ui:repeat`, описание которого приведено в следующей главе. В завершение этого раздела будет сказано несколько слов об обработке пробельных символов в технологии Facelets.

Тег <ui:debug>

Если на странице Facelets помещается тег `ui:debug`, то к дереву компонентов для этой страницы добавляется отладочный компонент. После того как пользователь нажмет комбинацию клавиш, которыми по умолчанию являются <Ctrl+Shift+d>, реализация JSF откроет окно и отобразит состояние дерева компонентов наряду с переменными, областью действия которых является приложение. Это окно отладки показано на рис. 5.7.

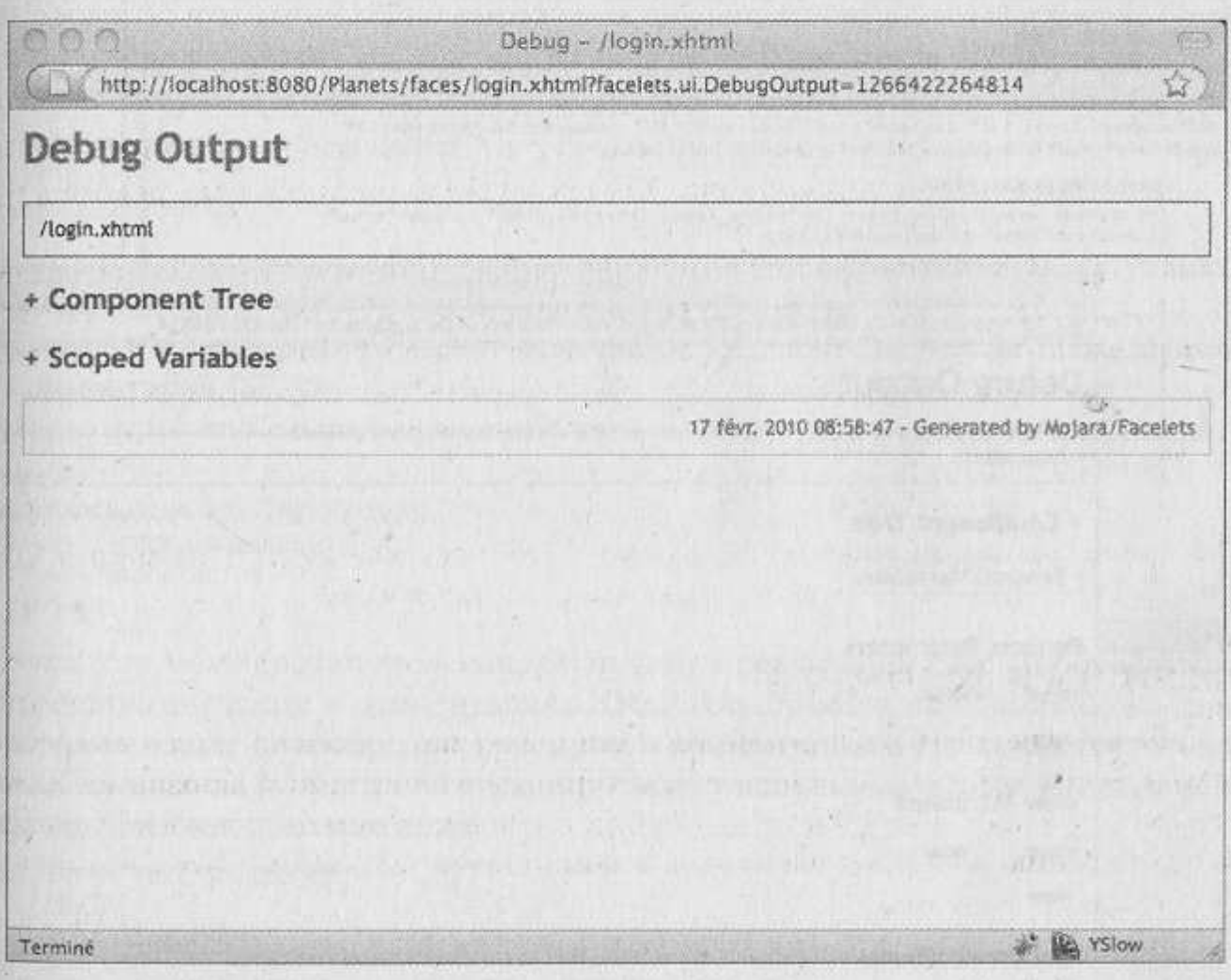


Рис. 5.7. Вывод отладочной информации в технологии Facelets

Можно щелкать на элементе Component Tree (Дерево компонентов) или Scoped Variables (Переменные с заданной областью), чтобы ознакомиться с деревом компонентов или переменными, областью действия которых является приложение (рис. 5.8). Тег `ui:debug` позволяет также переопределять комбинацию клавиш, применение которой приводит к открытию окна Debug Output (Вывод отладки), с помощью атрибута `hotkey`:

```
<ui:debug hotkey="i"/>
```

В предыдущем примере использования тега `ui:debug` показано, как переопределить комбинацию клавиш на <Ctrl+Shift+i>.

Тег `ui:debug` является полезным во время разработки, поскольку позволяет разработчикам непосредственно просматривать дерево компонентов текущей страницы и переменные с областью действия приложения; однако перед передачей приложения в производство, по-видимому, потребуется удалить этот тег. По указанной причине рекомендуется помещать тег `ui:debug` в шаблон, где он будет определяться в одном месте и совместно использоваться во многих представлениях, что исключает необходимость копировать тег на страницу XHTML каждого представления.

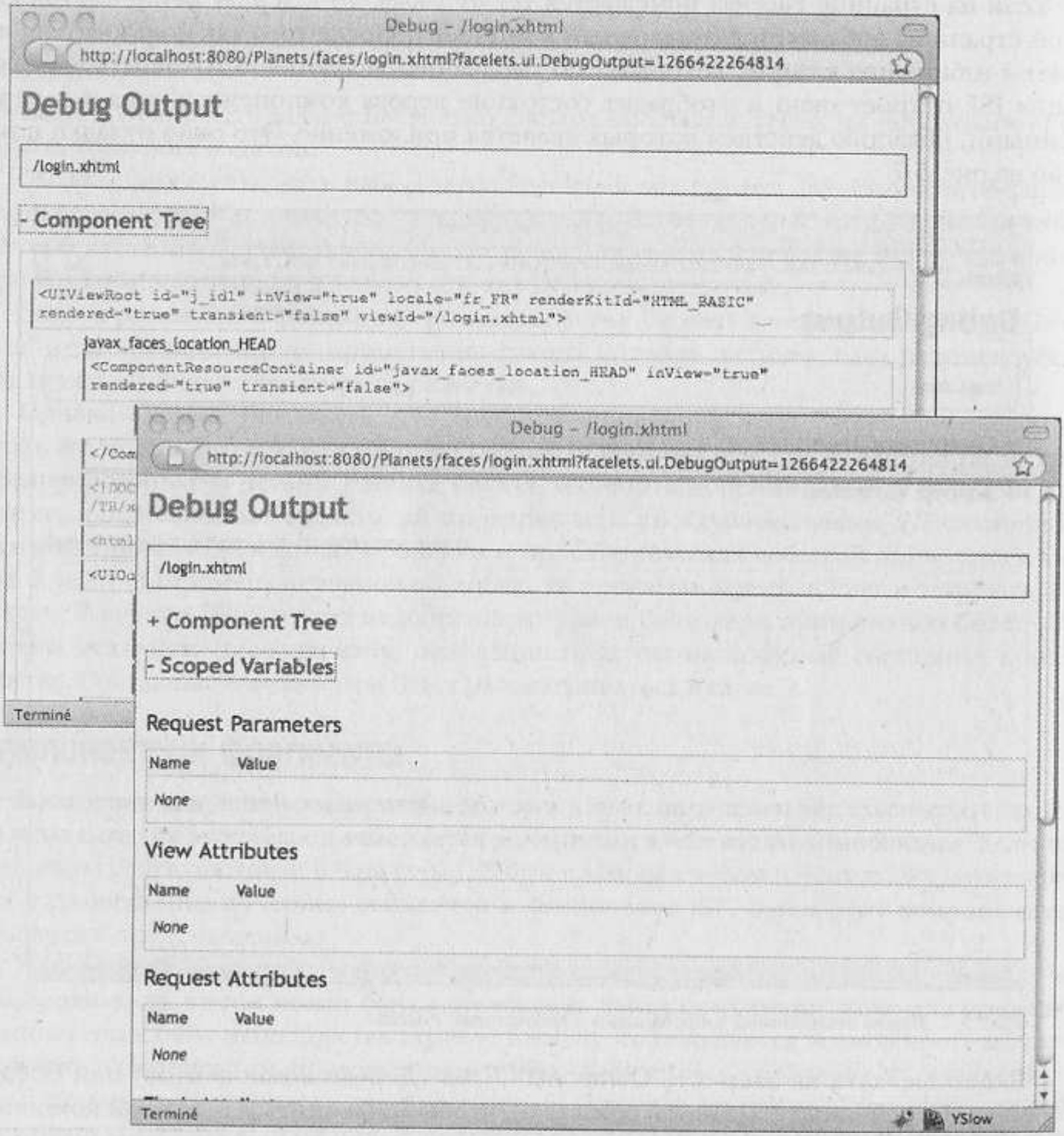


Рис. 5.8. Исследование дерева компонентов и переменных с заданной областью

Тег <ui:remove>

Иногда для выявления того, какая часть страницы JSF вызывает формирование трассировки стека, может потребоваться использовать проверенную временем стратегию "разделяй и властвуй", заключающуюся в том, что отдельные части страницы обозначаются комментариями так, чтобы оставался действующим только тот компонент, который вызывает нарушение в работе.

Однако заслуживает удивления то, что комментарии XML <!-- ... --> не могут применяться с этой целью. Например, если бы код формирования кнопки на странице XHTML был обозначен комментариями примерно так:

```
<!-- <h:commandButton id="loginButton"
  value="#{msgs.loginButtonText}"
  action="planetarium"/> -->
```

то реализация JSF обработала бы выражение значения #{msgs.loginButtonText} и поместила результат в виде комментария в сформированную страницу HTML. Если предположить, что вычисление выражения значения #{msgs.loginButtonText} приводит к получению строки "Log In", то на полученной странице HTML появилось бы следующее:

```
<!-- <h:commandButton id="loginButton"
  value="Log In"
  action="planetarium"/> -->
```

Если же метод getLoginButtonText активизирует исключительную ситуацию, то комментарии XML в таком случае вообще не могут помочь.

Кроме того, технология Facelets отлична от JSP, поэтому нельзя также использовать комментарии JSP, <%-- ... --%>.

Вместо этого необходимо применять теги ui:remove, как в следующем примере:

```
<ui:remove>
  <h:commandButton id="loginButton"
    value="#{msgs.loginButtonText}"
    action="planetarium"/>
</ui:remove>
```

У читателя может возникнуть вопрос: почему в реализации Facelets обрабатываются выражения значения в комментариях XML? Эта функция предназначалась для использования в коде JavaScript, заключенном в комментариях. Но в действительности для включения кода JavaScript на страницу следует использовать теги script, а не комментарии XML, как показано ниже.

```
<script type="text/javascript">
  //<![CDATA[
  Код Javascript
  //]]>
</script>
```

Это решение лучше еще и потому, что позволяет для поиска и обработки сценариев использовать инструментальные средства XML.

На заметку! Если в файле web.xml параметру контекста javax.faces.FACELETS_SKIP_COMMENTS присвоено значение true, то комментарии XML пропускаются. Это — важный параметр, наличие которого следует учитывать в своих проектах.

Обработка пробельных символов

То, как организована обработка пробельных символов на страницах Facelets, может вызвать удивление. По умолчанию пробельные символы вокруг компонентов удаляются. Например, рассмотрим следующие теги:

```
<h:outputText value="#{msgs.name}"/>
  <h:inputText value="#{user.name}"/>
```

Они отделены друг от друга пробельными символами (символом новой строки после тега `h:outputText` и пробелами перед тегом `h:inputText`). Но реализация Facelets не преобразует эти пробельные символы в текстовый компонент. И это на самом деле правильно, поскольку в противном случае последовательность тегов не действовала бы должным образом в теге `h:panelGrid`.

Но если на странице заданы подряд две ссылки, то такой способ обработки пробельных символов приводит к получению непонятных результатов. Обработка тегов

```
<h:commandLink value="Previous" .../> <h:commandLink value="Next" .../>
```

приводит к получению ссылок `PreviousNext` без пробела между ними. Выход из этой ситуации состоит в добавлении пробела с помощью выражения значения `#{' '}`.

Резюме

Facelets представляет собой намного более мощную технологию подготовки к отображению по сравнению с JSP. Платформа Facelets была специально предназначена для использования в сочетании с JSF, поэтому при совместной работе этих платформ не обнаруживаются неприятные несовместимости с JSF, как в некоторых известных случаях применения JSP вместе с JSF.

Как и Tiles, технология Facelets предоставляет возможность реализовать модульные пользовательские интерфейсы, простые для понимания, модификации и расширения, в которых используются встроенные возможности применения шаблонов. Кроме того, технология Facelets обеспечивает возможность оформления разделов страницы, что способствует еще лучшему разделению труда между специалистами по созданию основного содержимого и художественному оформлению. Наконец, технология Facelets предоставляет возможность использовать целый ряд сервисных тегов, включая `ui:debug`, благодаря которым реализация страниц JSF становится намного проще по сравнению с JSP.

Таблицы данных

В этой главе...

- Тег таблицы данных — `h:dataTable`
- Простая таблица
- Заголовки, нижние колонтитулы и надписи
- Стили
- Компоненты JSF в таблицах
- Редактирование таблиц
- Таблицы базы данных
- Модели таблиц
- Способы прокрутки

Глава 6

В классических веб-приложениях широко используются табличные данные. В прежние времена предпочитаемым средством решения этой задачи считались таблицы HTML, которые к тому же применялись для создания макетов страниц. Впоследствии вторая функция была в основном возложена на таблицы CSS, но обеспечение качественного отображения табличных данных до сих пор остается трудоемким. В настоящей главе обсуждается тег `h:dataTable` — компонент с широкими возможностями, хотя и небезграничными, который позволяет манипулировать табличными данными.

На заметку! Вообще говоря, тег `h:dataTable` представляет собой многофункциональную пару "компонент–модуль подготовки к отображению". Например, с его помощью можно легко отобразить компоненты JSF в ячейках таблиц, добавить заголовки и нижние колонтитулы к таблицам, а также усовершенствовать внешний вид таблиц с помощью классов CSS. Однако тег `h:dataTable` не поддерживает некоторые высокоуровневые средства, на применение которых можно было бы рассчитывать. Например, если потребуется отсортировать содержимое столбцов таблицы, придется написать определенный объем кода для выполнения этого. Прочитайте раздел "Сортировка и фильтрация" на стр. 210, чтобы ознакомиться с дополнительными сведениями о том, как это сделать.

Тег таблицы данных — `h:dataTable`

Тег `h:dataTable` выполняет итерации по данным для создания таблицы HTML. Ниже приведен пример того, как можно этим воспользоваться.

```
<h:dataTable value="#{items}" var="item">
   <h:column>
      <!-- Компоненты из левого столбца -->
      #{item.aPropertyName}
   </h:column>

   <h:column>
      <!-- Компоненты из следующего столбца -->
      <h:commandLink value="#{item.anotherPropertyName}" action="..."/>
   </h:column>

   <!-- При желании могут быть добавлены дополнительные столбцы -->
</h:dataTable>
```

Атрибут `value` представляет данные, по которым выполняет итерации тег `h:dataTable`. Эти данные должны представлять собой одно из следующих:

- объект Java;
- массив;
- экземпляр `java.util.List`;
- экземпляр `java.sql.ResultSet`;
- экземпляр `javax.servlet.jsp.jstl.sql.Result`;
- экземпляр `javax.faces.model.DataModel`.

По мере того как тег `h:dataTable` производит итерации, он делает доступным в пределах текста тега каждый элемент массива, списка, результирующего набора и т.д. Имя этого элемента задается с помощью атрибута `var` тега `h:dataTable`. В приведенном выше фрагменте кода каждый элемент (`item`) коллекции (`items`) становится доступным в ходе того, как тег `h:dataTable` выполняет итерации в коллекции. Свойства текущего элемента применяются для заполнения значений столбцов в текущей строке.

В качестве значения атрибута `value` тега `h:dataTable` можно также задать любой объект Java, хотя польза от такого подхода весьма сомнительна. Если рассматриваемый объект является скалярным (под этим подразумевается, что он не представляет собой коллекцию того или иного рода), то тег `h:dataTable` выполняет итерацию единожды, делая объект доступным в тексте тега.

Текст тега `h:dataTable` может содержать только теги `h:column`; тег `h:dataTable` игнорирует все прочие компоненты тегов. Каждый столбец может содержать неограниченное количество компонентов (а также необязательных аспектов заголовка и нижнего колонтитула, которые будут рассматриваться в следующем разделе).

Тег `h:dataTable` соединяет компонент `UIData` с модулем подготовки к отображению `Table`. Благодаря такому сочетанию обеспечивается надежное создание таблиц, в ходе которого осуществляются поддержка стилей CSS, доступ к базе данных, применение пользовательских моделей таблиц и т.д. Начнем изучение тега `h:dataTable` с простой таблицы.

Простая таблица

На рис. 6.1 показана таблица с именами.

Структура каталогов для приложения, показанного на рис. 6.1, приведена на рис. 6.2. Страница приложения JSF содержится в листинге 6.1.

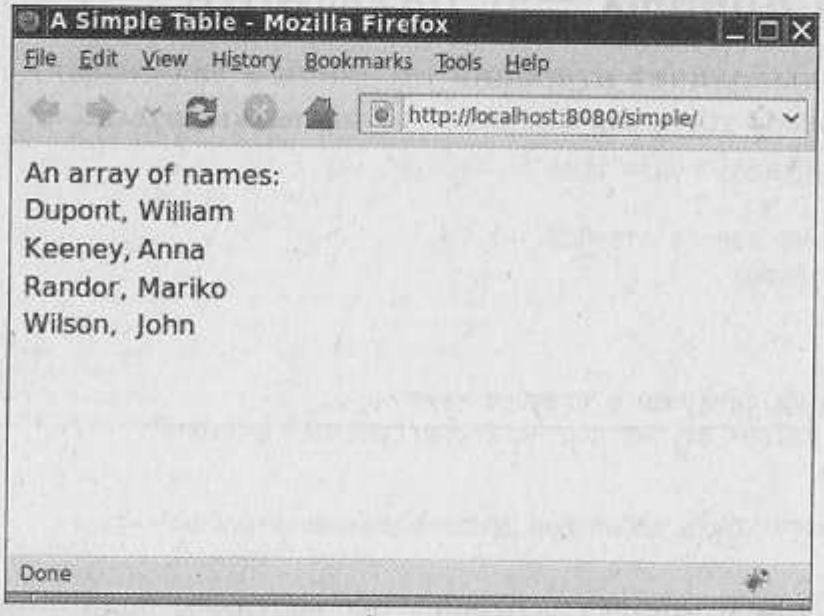


Рис. 6.1. Простая таблица

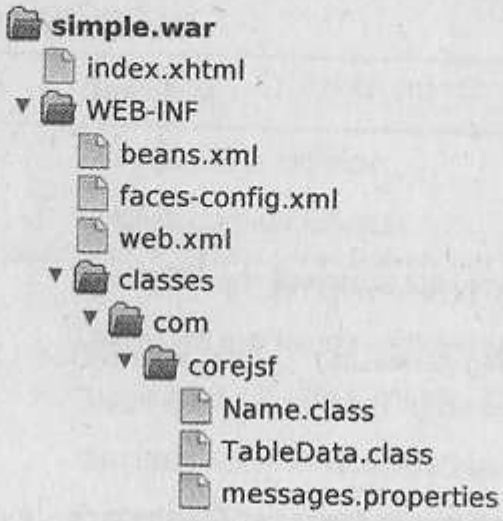


Рис. 6.2. Структура каталогов для простой таблицы

В листинге 6.1 показано, как используется тег `h:dataTable` для выполнения итерации по массиву имен. Фамилия, сопровождаемая запятой, размещается в левом столбце, а имя – в правом.

В листинге 6.2 показан класс `Name`. В этом примере для создания экземпляра массива имен применяется управляемый бин, который показан в листинге 6.3.

Листинг 6.1. Файл `simple/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.windowTitle}</title>
   </h:head>
   <h:body>
      #{msgs.pageTitle}
      <h:form>
         <h:dataTable value="#{tableData.names}" var="name">
            <h:column>
               #{name.last},
            </h:column>

            <h:column>
               #{name.first}
            </h:column>
         </h:dataTable>
      </h:form>
   </h:body>
</html>
```

Листинг 6.2. Файл `simple/src/java/com/corejsf/Name.java`

```
package com.corejsf;

import java.io.Serializable;

public class Name implements Serializable {
   private String first;
```

```
7.     private String last;
8.
9.     public Name(String first, String last) {
10.       this.first = first;
11.       this.last = last;
12.    }
13.
14.     public void setFirst(String newValue) { first = newValue; }
15.     public String getFirst() { return first; }
16.
17.     public void setLast(String newValue) { last = newValue; }
18.     public String getLast() { return last; }
19. }
```

Листинг 6.3. Файл simple/src/java/com/corejsf/TableData.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. import javax.inject.Named;
6.    // или import javax.faces.bean.ManagedBean;
7. import javax.enterprise.context.SessionScoped;
8.    // или import javax.faces.bean.SessionScoped;
9. import javax.faces.model.ArrayDataModel;
10. import javax.faces.model.DataModel;
11.
12. @Named // или @ManagedBean
13. @SessionScoped
14. public class TableData implements Serializable {
15.    private static final Name[] names = new Name[] {
16.        new Name("William", "Dupont"),
17.        new Name("Anna", "Keeney"),
18.        new Name("Mariko", "Randor"),
19.        new Name("John", "Wilson")
20.    };
21.
22.    public Name[] getNames() { return names;}
23. }
```

Таблица, которая приведена на рис. 6.1, преднамеренно сделана самой несложной. В дальнейшем изложении в этой главе будет показано, как можно добавить к таблицам дополнительные средства, такие как стили CSS и заголовки столбцов.

Внимание! Данные тега h:dataTable организованы по строкам, например, строкам таблицы соответствуют имена в листинге 6.3, но эти имена ничего не говорят о том, что хранится в каждом столбце. Иными словами, ответственность за определение содержимого столбцов лежит на авторе страницы. Данные, организованные построчно, могут также иметь другую структуру; например, модели таблиц Swing позволяют следить за тем, что находится в каждой строке и каждом столбце.

Атрибуты тега h:dataTable

Атрибуты тега h:dataTable перечислены в табл. 6.1.

Атрибуты binding и id обсуждаются в разделе "Идентификаторы и средства связывания" главы 4 на стр. 108, а атрибуты rendered – в разделе "Общие сведения о тегах HTML на платформе JSF" на стр. 106.

Таблица 6.1. Атрибуты тега `h:dataTable`

Атрибут	Описание
`bgcolor`	Цвет фона для таблицы
`border`	Ширина границы таблицы
`captionClass` JSF 1.2	Класс CSS для заголовка таблицы
`captionStyle` JSF 1.2	Стиль CSS для заголовка таблицы
`cellpadding`	Заполнение вокруг ячеек таблицы
`cellspacing`	Интервал между ячейками таблицы
`columnClasses`	Разделенный запятыми список классов CSS для столбцов
`dir`	Ориентация текста для того случая, когда текст не наследует направленность; допустимые значения: `LTR` (left to right — слева направо) и `RTL` (right to left — справа налево)
`first`	Индекс первой строки, показанной в таблице, с отсчетом от нуля
`footerClass`	Класс CSS для нижнего колонтитула таблицы
`frame`	Спецификация для сторон рамки, окружающей таблицу; допустимые значения: `none`, `above`, `below`, `hsides`, `vsides`, `lhs`, `rhs`, `box`, `border`
`headerClass`	Класс CSS для заголовка таблицы
`rowClasses`	Разделенный запятыми список классов CSS для строк
`rows`	Количество строк, отображенных в таблице, начиная со строки, указанной атрибутом `first`; если это значение задано равным нулю, то отображаются все строки таблицы
`rules`	Спецификация для линий, проведенных между ячейками; допустимые значения: `groups`, `rows`, `columns`, `all`
`summary`	Итоговые сведения о назначении и структуре таблицы, используемые для невизуальной обратной связи, такой как речь
`var`	Имя переменной, созданной таблицей данных, которая представляет текущий элемент в значении
`binding`, `id`, `rendered`, `styleClass`, `value`	Базовый
`lang`, `style`, `title`, `width`	HTML 4.0
`onclick`, `ondblclick`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`	События DHTML

Кроме того, в теге `h:dataTable` предусмотрено полное дополнение событий DHTML и передаваемых атрибутов HTML 4.0. Дополнительные сведения об этих атрибутах см. в главе 4.

Атрибут `first` определяет индекс с отсчетом от нуля первой видимой строки в таблице. Атрибут `value` указывает данные, по которым выполняет итерации тег `h:dataTable`. В начале каждой итерации тег `h:dataTable` создает переменную с областью действия запроса, название которой присваивается с помощью атрибута `var` тега

h:dataTable. Предусмотрена возможность ссылаться на текущий элемент с этим именем в тексте тега h:dataTable.

Атрибуты тега h:column

Атрибуты тега h:column перечислены в табл. 6.2.

Таблица 6.2. Атрибуты тега h:column

Атрибут	Описание
footerClass JSF 1.2	Класс CSS для нижнего колонтитула столбца
headerClass JSF 1.2	Класс CSS для заголовка столбца
binding, id, rendered, styleClass, value	Базовый

Заголовки, нижние колонтитулы и надписи

Если отображается список имен, как в разделе "Простая таблица" на стр. 188, то возникает необходимость проводить различия между фамилиями и именами. Это можно сделать с помощью заголовков столбцов, как показано на рис. 6.3.

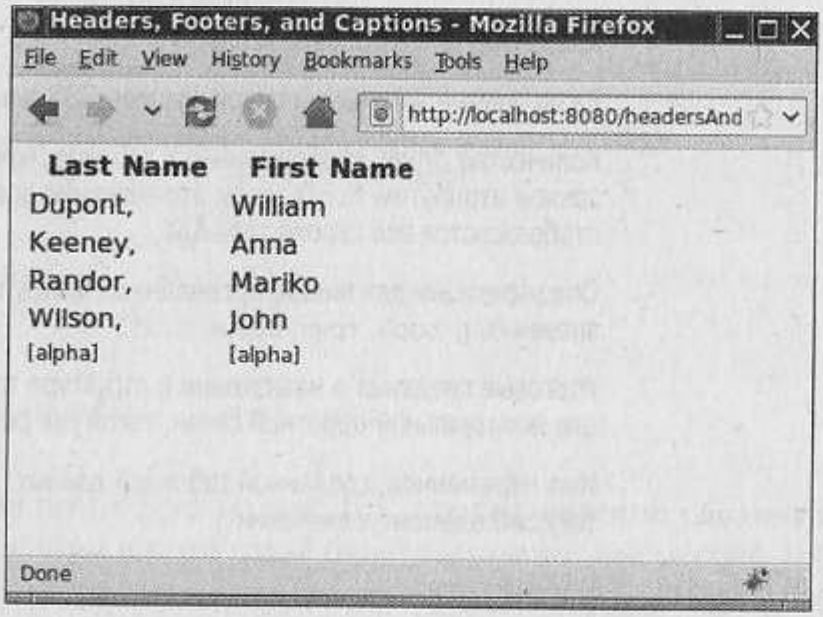


Рис. 6.3. Определение заголовков и нижних колонтитулов столбцов

Кроме заголовков, столбцы таблицы на рис. 6.3 содержат также нижние колонтитулы, которые указывают тип данных соответствующих им столбцов; в этом случае оба столбца обозначены как [alpha], т.е. как алфавитно-цифровые.

Для задания заголовков и нижних колонтитулов столбцов применяются аспекты, как показано в следующем примере:

```
<h:dataTable>
  ...
  <h:column headerClass="columnHeader"
            footerClass="columnFooter">
    <f:facet name="header">
      <!-- Здесь должны находиться компоненты заголовка -->
    </f:facet>

    <!-- Здесь должны находиться компоненты столбца -->
```

```
    <f:facet name="footer">
      <!-- Здесь должны находиться компоненты нижнего колонтитула -->
    </f:facet>
  </h:column>
  ...
</h:dataTable>
```

Тег `h:dataTable` размещает компоненты, заданные для аспектов заголовка и нижнего колонтитула, соответственно в заголовке и нижнем колонтитуле таблицы HTML. Обратите внимание на то, что в целях задания стилей CSS для заголовков и нижних колонтитулов столбцов используются атрибуты `headerClass` и `footerClass` тега `h:column`.

Чтобы снабдить таблицу заголовком, можно добавить аспект `caption` следующим образом:

```
<h:dataTable ...>
  <f:facet name="caption">An Array of Names:</f:facet>
  ...
</h:dataTable>
```

Если этот аспект будет добавлен к таблице, показанной на рис. 6.3, то будет получено изображение, которое приведено на рис. 6.4.

Для определения стиля или класса CSS для этой надписи можно использовать атрибуты `captionStyle` и `captionClass`:

```
<h:dataTable ... captionClass="caption">
  <f:facet name="caption">An Array of Names:</f:facet>
  ...
</h:dataTable>
```

В предыдущем фрагменте кода для аспекта применялся простой текст, но, как и в отношении любого аспекта, вместо этого можно определить компонент JSF.

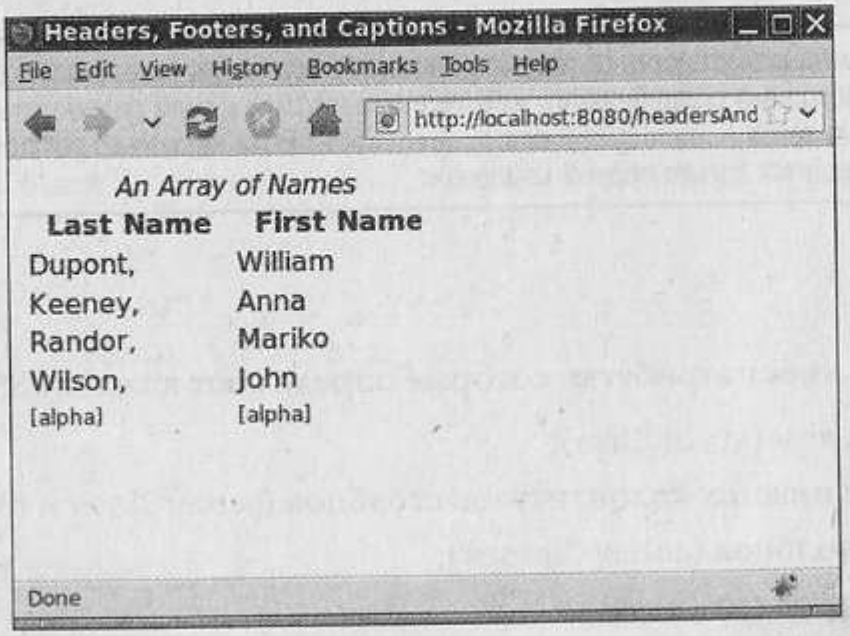


Рис. 6.4. Заголовок таблицы

Код для страницы JSF, которая показана на рис. 6.4, приведен в листинге 6.4.

Листинг 6.4. Файл `headersAndFooters/web/index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.windowTitle}</title>
   </h:head>
   <h:body>
      <h:form>
         <h:dataTable value="#{tableData.names}" var="name"
                      captionStyle="font-size: 0.95em; font-style:italic"
                      style="width: 250px;">

            <f:facet name="caption">An Array of Names</f:facet>

            <h:column headerClass="columnHeader" footerClass="columnFooter">
               <f:facet name="header">#{msgs.lastnameColumn}</f:facet>

               #{name.last},

               <f:facet name="footer">#{msgs.alphanumeric}</f:facet>
            </h:column>

            <h:column headerClass="columnHeader" footerClass="columnFooter">
               <f:facet name="header">#{msgs.firstnameColumn}</f:facet>

               #{name.first}

               <f:facet name="footer">#{msgs.alphanumeric}</f:facet>
            </h:column>
         </h:dataTable>
      </h:form>
   </h:body>
</html>
```

Совет. Чтобы иметь возможность разместить несколько компонентов в заголовке или нижнем колонтитуле таблицы, необходимо сгруппировать их в теге `h:panelGroup` или разместить в контейнерном компоненте с помощью тегов `h:panelGrid` или `h:dataTable`. Если несколько компонентов размещено в аспекте, то отображается только первый компонент.

Стили

Тег `h:dataTable` имеет атрибуты, которые определяют классы CSS для

- таблицы в целом (`styleClass`);
- заголовков и нижних колонтитулов столбцов (`headerClass` и `footerClass`);
- отдельных столбцов (`columnClasses`);
- отдельных строк (`rowClasses`).

В таблице, показанной на рис. 6.5, используются атрибуты `styleClass`, `headerClass` и `columnClasses`.

На заметку! Атрибуты `rowClasses` и `columnClasses` тега `h:dataTable` являются взаимоисключающими. Если определены и тот и другой, то атрибут `columnClasses` имеет приоритет.

Order Number	Order Date	Customer ID	Amount	Description
1	2002-05-20	1	129.99	Wristwatch
2	2002-05-21	1	19.95	Coffee grinder
3	2002-05-24	1	29.76	Bath towel
4	2002-05-23	1	39.34	Deluxe cheese grater
5	2002-05-22	2	56.75	Champagne glass set
6	2002-05-20	2	28.11	Instamatic camera
7	2002-05-22	2	38.77	Walkman
8	2002-05-21	2	56.76	Coffee maker
9	2002-05-23	2	21.47	Car wax
10	2002-05-21	2	16.8	Tape recorder
11	2002-05-24	2	25.44	Art brush set
12	2002-05-22	3	47.63	Game software

Рис. 6.5. Применение стилей к столбцам и заголовкам

Применение стилей к отдельным столбцам

Ниже показано, как задаются классы CSS, приведенные на рис. 6.5.

```
<h:dataTable value="#{order.all}" var="order"
   styleClass="orders"
   headerClass="ordersHeader"
   columnClasses="oddColumn,evenColumn">
```

Эти классы CSS перечислены ниже.

```
.orders {
   border: thin solid black;
}
.ordersHeader {
   text-align: center;
   font-style: italic;
   color: Snow;
   background: Teal;
}
.oddColumn {
   height: 25px;
   text-align: center;
   background: MediumTurquoise;
}
.evenColumn {
   text-align: center;
   background: PowderBlue;
}
```

Внимание! В качестве иллюстрации в применяемых здесь классах стилей используются названия цветов, такие как `PowderBlue` и `MediumTurquoise`. Но предпочтительны эквивалентные шестнадцатеричные константы, поскольку они являются переносимыми, а названия цветов — нет.

Мы определили классы только для двух столбцов, но необходимо отметить, что имеется пять столбцов. В этом случае в теге `h:dataTable` повторно используются классы столбцов, начиная с первого. Задавая классы только для первых двух столбцов, можно указать классы CSS для четных и нечетных столбцов. (В этом изложении под словами "четный" и "нечетный" подразумевается, что первый столбец имеет номер 1.)

Применение стилей к отдельным строкам

Атрибут `rowClasses` можно использовать для задания классов CSS применительно к строкам вместо столбцов, как показано на рис. 6.6. Эта таблица данных реализована следующим образом:

```
<h:dataTable value="#{order.all}" var="order"
   styleClass="orders"
   headerClass="ordersHeader"
   rowClasses="oddRow,evenRow">
```

Подобно классам столбцов, тег `h:dataTable` повторно использует классы строк, если количество классов меньше количества строк. В предыдущем фрагменте кода эта особенность использовалась в целях задания классов CSS для четных и нечетных строк.

Order Number	Order Date	Customer ID	Amount	Description
1	2002-05-20	1	129.99	Wristwatch
2	2002-05-21	1	19.95	Coffee grinder
3	2002-05-24	1	29.76	Bath towel
4	2002-05-23	1	39.34	Deluxe cheese grater
5	2002-05-22	2	56.75	Champagne glass set
6	2002-05-20	2	28.11	Instamatic camera
7	2002-05-22	2	38.77	Walkman
8	2002-05-21	2	56.76	Coffee maker
9	2002-05-23	2	21.47	Car wax
10	2002-05-21	2	16.8	Tape recorder
11	2002-05-24	2	25.44	Art brush set
12	2002-05-22	3	47.63	Game software

Рис. 6.6. Применение стилей к строкам

Тег `ui:repeat` JSF 2.0

Вместо тега `h:dataTable` можно использовать тег `ui:repeat`. Тег `ui:repeat` повторно вставляет свой текст на странице. Иногда приходится подготавливать к отображению разметку таблицы вручную, как в следующем примере:

```
<table>
   <ui:repeat value="#{tableData.names}" var="name">
   <tr>
      <td>#{name.last},</td>
      <td>#{name.first}</td>
   </tr>
   </ui:repeat>
</table>
```

Но мы находим это довольно затруднительным, особенно если приходится постоянно заботиться о вставке заголовков, нижних колонтитулов, надписей и стилей. Но для тех, кто знаком с организацией таблиц HTML, эта задача значительно упрощается, если вместо тега `h:dataTable` используется тег `ui:repeat`.

Тег `ui:repeat` имеет несколько атрибутов, благодаря которым в определенных ситуациях он становится более удобным по сравнению с тегом `h:dataTable`.

В частности, следующие атрибуты позволяют выполнять итерации по подмножеству коллекции:

- `offset` — индекс, с которого начинается итерация (значение по умолчанию — 0);
- `step` — разность между подряд идущими значениями индекса (значение по умолчанию — 1);
- `size` — число итераций (значение по умолчанию — (размер коллекции `size` — `offset`) / `step`).

Например, если необходимо отобразить элементы 10, 12, 14, 16, 18 из коллекции, воспользуйтесь следующим:

```
<ui:repeat ... offset="10" step="2" size="5">
```

Атрибут `varStatus` задает переменную, с помощью которой можно узнать состояние итерации. Свойства состояния итерации приведены ниже.

- Логические свойства `even`, `odd`, `first` и `last`, которые могут применяться для выбора стилей.
- Целочисленные свойства `index`, `begin`, `step` и `end`, которые предоставляют индекс текущей итерации, а также начальное смещение, размер шага и конечное смещение. Обратите внимание на то, что соблюдаются соотношения `begin = offset` и `end = offset + step X size`, где `offset` и `size` — значения атрибутов тега `ui:repeat`.

Свойство `index` может использоваться для получения номеров строк:

```
<table>
  <ui:repeat value="#{tableData.names}" var="name" varStatus="status">
  <tr>
    <td>#{status.index + 1}</td>
    <td>#{name.last},</td>
    <td>#{name.first}</td>
  </tr>
  </ui:repeat>
</table>
```

Компоненты JSF в таблицах

До сих пор в данной главе использовались только компоненты вывода в столбцах таблицы, но предусмотрена возможность размещать в ячейках таблицы любые компоненты JSF. На рис. 6.7 показано приложение, в котором используются разнообразные компоненты в одной таблице.

Тег `h:dataTable` выполняет итерации по данным, поэтому таблица, показанная на рис. 6.7, предоставляет с этой целью список целых чисел. Текущее целое число используется для настройки конфигурации компонентов в столбцах `"Number"`, `"Textfields"`, `"Buttons"` и `"Menu"`.

Компоненты в таблице не отличаются от компонентов вне ее; ими можно манипулировать с применением любого желаемого способа, включая подготовку к отображению по условию с помощью атрибута `rendered`, обработчики событий

Структура каталогов для приложения, показанного на рис. 6.7, приведена на рис. 6.8. Код страницы JSF содержится в листинге 6.5. В листинге 6.6 показан управляемый бин, содержащий данные модели: числа от 1 до 5.

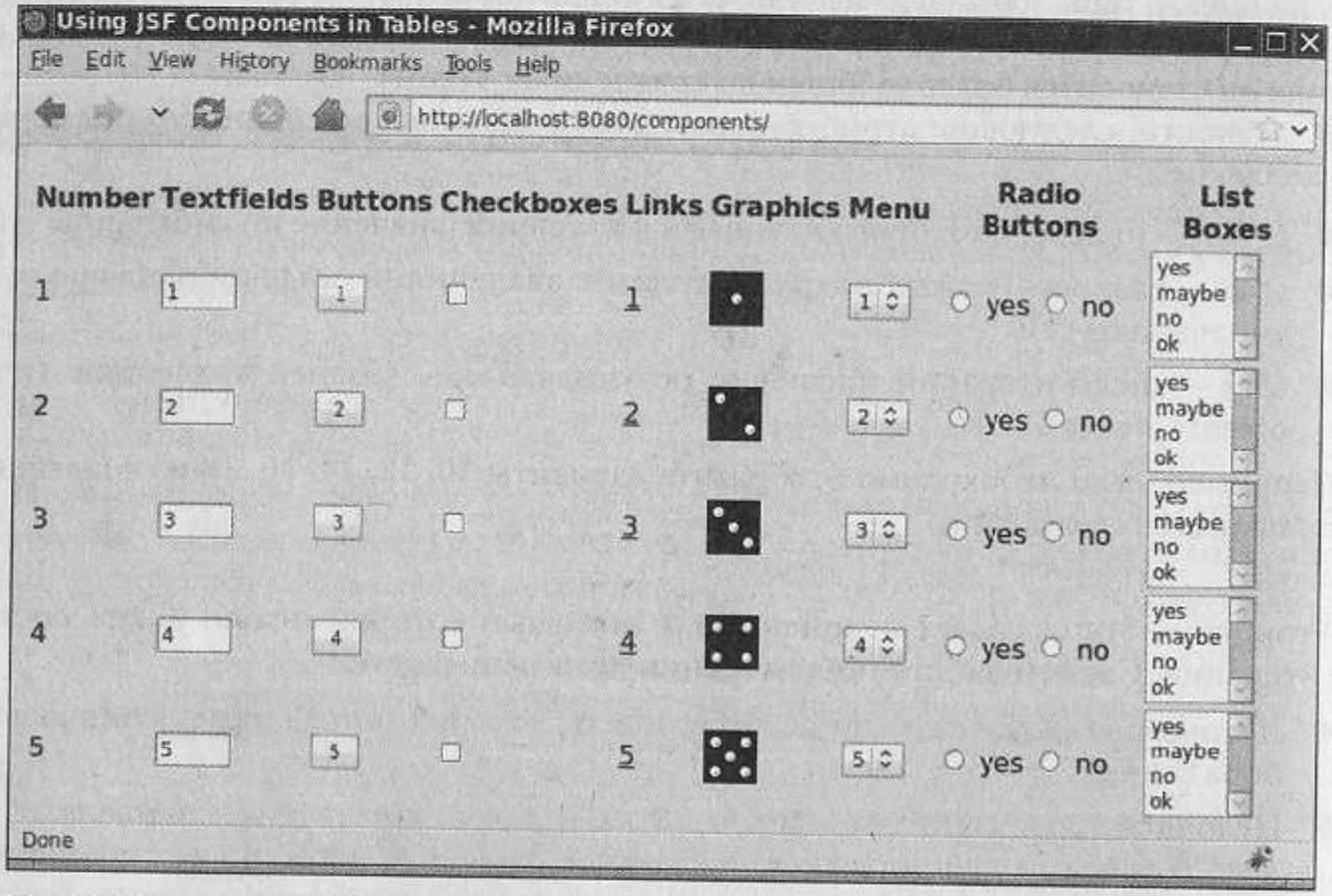


Рис. 6.7. Компоненты JSF в ячейках таблицы

Рис. 6.8. Структура каталогов для примера компонентов

Листинг 6.5. Файл `components/web/index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
```

```
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.windowTitle}</title>
   </h:head>
   <h:body style="background: #eee">
      <h:form>
         <h:dataTable value="#{numberList}" var="number">
            <h:column>
               <f:facet name="header">#{msgs.numberHeader}</f:facet>
               #{number}
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.textfieldHeader}</f:facet>
               <h:inputText value="#{number}" size="3"/>
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.buttonHeader}</f:facet>
               <h:commandButton value="#{number}"/>
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.checkboxHeader}</f:facet>
               <h:selectBooleanCheckbox value="false"/>
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.linkHeader}</f:facet>
               <h:commandLink>#{number}</h:commandLink>
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.graphicHeader}</f:facet>
               <h:graphicImage library="images" name="dice#{number}.gif"
                               style="border: 0px"/>
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.menuHeader}</f:facet>
               <h:selectOneMenu>
                  <f:selectItem itemLabel="#{number}" itemValue="#{number}"/>
               </h:selectOneMenu>
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.radioHeader}</f:facet>
               <h:selectOneRadio layout="lineDirection" value="nextMonth">
                  <f:selectItem itemValue="yes" itemLabel="yes"/>
                  <f:selectItem itemValue="no" itemLabel="no"/>
               </h:selectOneRadio>
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.listboxHeader}</f:facet>
               <h:selectOneListbox size="4">
                  <f:selectItem itemValue="yes" itemLabel="yes"/>
                  <f:selectItem itemValue="maybe" itemLabel="maybe"/>
                  <f:selectItem itemValue="no" itemLabel="no"/>
                  <f:selectItem itemValue="ok" itemLabel="ok"/>
               </h:selectOneListbox>
            </h:column>
         </h:dataTable>
      </h:form>
   </h:body>
</html>
```

Листинг 6.6. Файл components/web/WEB-INF/faces-config.xml

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
   version="2.0">
   <application>
      <resource-bundle>
         <base-name>com.corejsf.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
   </application>

   <managed-bean>
      <managed-bean-name>numberList</managed-bean-name>
      <managed-bean-class>java.util.ArrayList</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
      <list-entries>
         <value>1</value>
         <value>2</value>
         <value>3</value>
         <value>4</value>
         <value>5</value>
      </list-entries>
   </managed-bean>
</faces-config>
```

Редактирование таблиц

Ниже приведены два примера программ, в которых показано, как редактировать таблицы. Вначале будет показано, как редактировать отдельные ячейки таблицы. После этого приведен пример удаления строк таблицы. Аналогичный способ приемлем и для добавления строк. Если в приложении реализованы команды, которые затрагивают отдельные строки, то в нем должен быть предусмотрен способ определения того, какие строки выбрал пользователь. В каждом из примеров показаны разные подходы к выявлению выбора, сделанного пользователем.

Редактирование ячеек таблицы

Чтобы отредактировать ячейки таблицы, необходимо предоставить компонент ввода для каждой ячейки, которая будет редактироваться. В приложении, показанном на рис. 6.9, предусмотрена возможность редактировать все ячейки таблицы. Пользователь устанавливает флажок, чтобы отредактировать строку, а затем щелкает на кнопке Save Changes (Сохранить изменения) для сохранения внесенных изменений. Редактируемые ячейки показаны на рис. 6.9 сверху вниз.

В ячейках таблицы на рис. 6.9 используется компонент ввода, когда редактируется ячейка, и компонент вывода, когда этого не происходит. В следующем коде показано, как осуществляется такой подход:

```
<h:dataTable value="#{tableData.names}" var="name">
   <!-- Столбец с флажком -->
   <h:column>
      <f:facet name="header">
         <h:outputText value="#{msgs.editColumn}" style="font-weight: bold"/>
```

```
    </f:facet>
    <h:selectBooleanCheckbox value="#{name.editable}" onclick="submit()"/>
  </h:column>
  <!-- Столбец фамилий -->
  <h:column>
    ...
    <h:inputText value="#{name.last}" rendered="#{name.editable}" size="10"/>
    <h:outputText value="#{name.last}" rendered="#{not name.editable}"/>
  </h:column>
  ...
</h:dataTable>
```

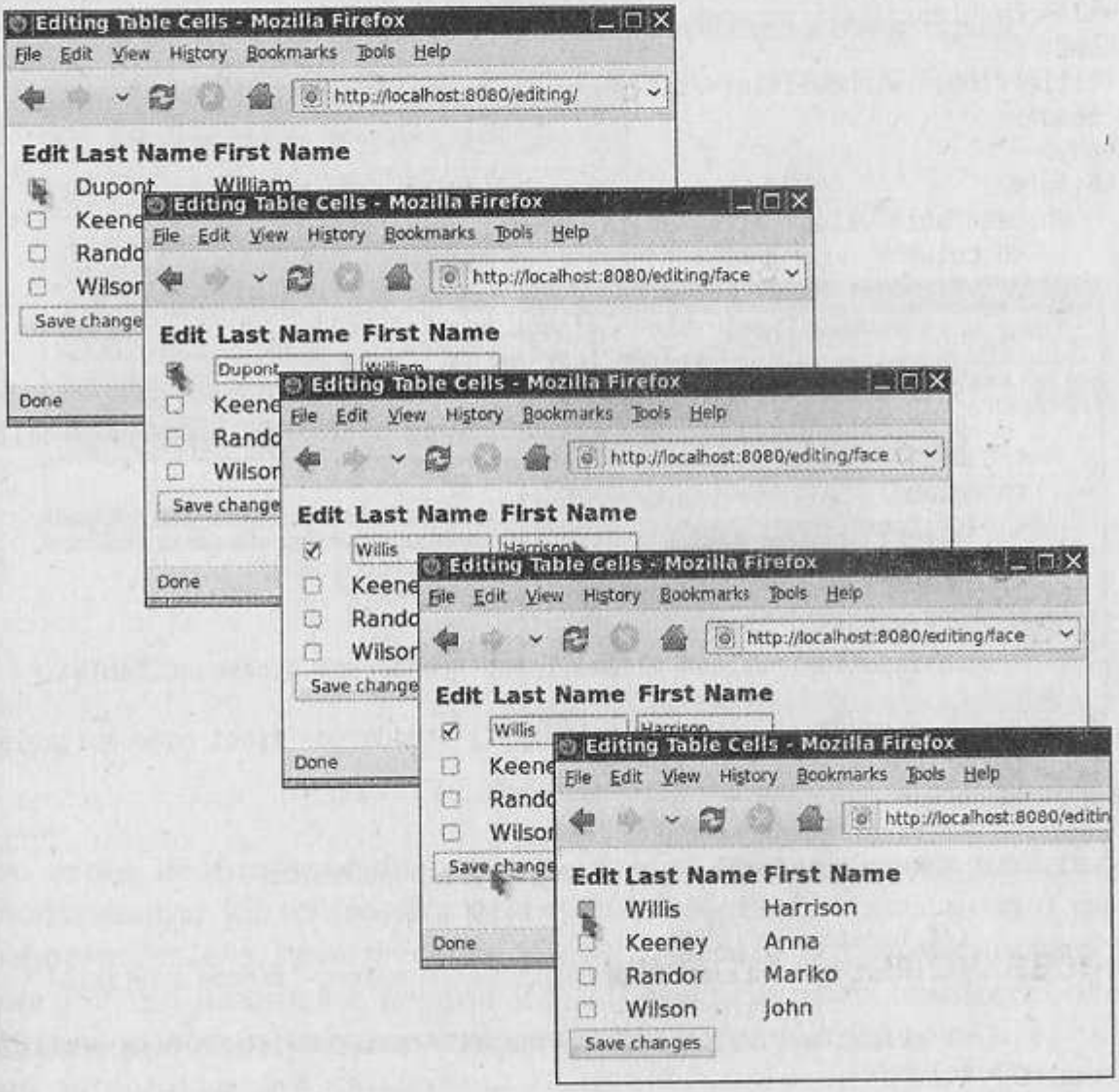


Рис. 6.9. Редактирование ячеек таблицы

В предыдущем фрагменте кода приведен только код для флажков и столбцов с фамилиями. Значение флажка показывает, доступно ли текущее значение для редактирования; если оно доступно, флажок установлен. Для столбца с фамилиями заданы два компонента: `h:inputText` и `h:outputText`. Если значение столбца доступно для редактирования, то к отображению подготовлен компонент ввода. Если же это значение не доступно для редактирования, то к отображению подготовлен компонент вывода.

Для добавления свойства `editable` к классу `Name` применяется следующее:

```
public class Name {
  private String first;
```

```
   private boolean editable;
   ...
   public boolean isEditable() { return editable; }
   public void setEditable(boolean newValue) { editable = newValue; }
}
```

Весь объем кода страницы JSF, показанной на рис. 6.9, приведен в листинге 6.7.

Листинг 6.7. Файл editing/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.windowTitle}</title>
   </h:head>
   <h:body>
      <h:form>
         <h:dataTable value="#{tableData.names}" var="name">
            <h:column>
               <f:facet name="header">
                  <h:outputText value="#{msgs.editColumn}"
                                style="font-weight: bold"/>
               </f:facet>
               <h:selectBooleanCheckbox value="#{name.editable}" onclick="submit()"/>
            </h:column>
            <h:column>
               <f:facet name="header">
                  <h:outputText value="#{msgs.lastnameColumn}"
                                style="font-weight: bold"/>
               </f:facet>
               <h:inputText value="#{name.last}" rendered="#{name.editable}"
                            size="10"/>
               <h:outputText value="#{name.last}" rendered="#{not name.editable}"/>
            </h:column>
            <h:column>
               <f:facet name="header">
                  <h:outputText value="#{msgs.firstnameColumn}"
                                style="font-weight: bold"/>
               </f:facet>
               <h:inputText value="#{name.first}" rendered="#{name.editable}"
                            size="10"/>
               <h:outputText value="#{name.first}" rendered="#{not name.editable}"/>
            </h:column>
         </h:dataTable>
         <h:commandButton value="#{msgs.saveChangesButtonText}"
                          action="#{tableData.save}"/>
      </h:form>
   </h:body>
</html>
```

На заметку! Способ редактирования ячеек таблицы, показанный в предыдущем разделе, может применяться для всех допустимых типов данных таблицы: объектов Java, массивов, списков, результирующих наборов и результатов вычисления. Но реализация JSF может применяться для обновления базы данных, только если результирующий набор базы данных, связанный с таблицей, является обновляемым.

Удаление строк JSF 2.0

Если происходит удаление или добавление строк, то использование массива для сбора данных строк становится затруднительным. Вместо этого следует использовать списки. В рассматриваемом примере можно просто применить конструкцию `ArrayList<Name>`.

В этом примере приложения рядом с каждой строкой приведена ссылка с обозначением `Delete` (Удалить), как показано на рис. 6.10. По аналогии с примером редактирования должен быть предусмотрен некоторый способ определения того, какую строку выбрал пользователь. В данном примере к классу `Name` добавлено свойство `editable`. Благодаря этому достигнуто весьма значительное упрощение кода, но такой подход не всегда осуществим. Если класс `Name` определен в бизнес-логике приложения, то не всегда есть возможность просто добавить свойства к этому классу.

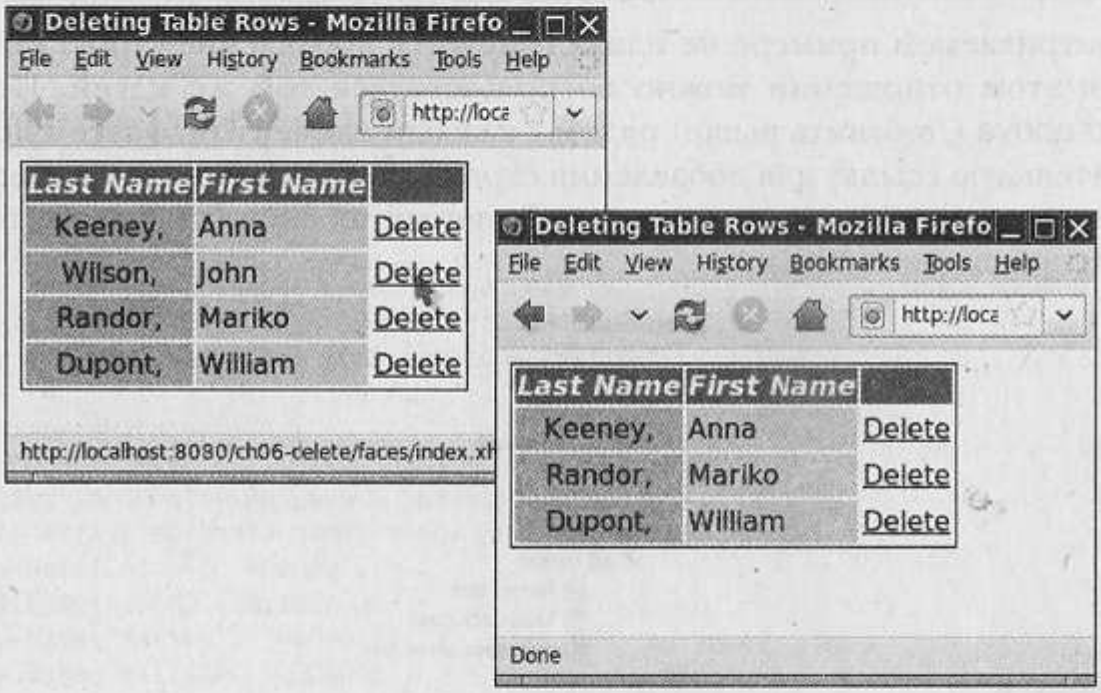


Рис. 6.10. Удаление строк таблицы

Вместо этого необходимо обеспечить, чтобы каждая ссылка идентифицировала строку, подлежащую удалению. На некоторых веб-платформах можно формировать ссылки в форме `/delete/rowNumber` или `delete?id=someId`. Это можно осуществлять и на платформе JSF, но начиная с версии JSF 2.0 предусмотрен намного более удобный способ. Можно просто передать элемент строки методу `action`:

```
<h:dataTable value="#{tableData.names}" var="name" ...>
   ...
   <h:commandLink value="Delete" action="#{tableData.deleteRow(name)}">
   ...
</h:dataTable>
```

После щелчка на этой ссылке вызывается метод `deleteRow` с текущим значением имени и это значение удаляется:

```
public String deleteRow(Name nameToDelete) {
   names.remove(nameToDelete);
   return null;
}
```

Следует отметить, что не требуется, чтобы объекты `Name` имели идентификаторы. Используется просто сам объект. Безусловно, этот объект не передается в прямом

и обратном направлениях между браузером и сервером. Вместо этого каждый компонент в таблице данных имеет идентификатор, содержащий его номер строки. Эти идентификаторы вырабатываются автоматически при подготовке таблицы данных к отображению. При активизации ссылки ее идентификатор передается на сервер. При форматировании ответа в таблице данных осуществляется цикл по значениям путем задания переменной `name` и повторного формирования идентификаторов ссылок. При обнаружении соответствующей ссылки активизируется ее метод `decode` и вычисляется выражение `#{tableData.deleteRow(name)}` с текущим значением.

Внимание! Если значение таблицы данных имеет область действия запроса, необходимо проследить за тем, чтобы данные таблицы не изменялись в интервале между подготовкой таблицы к отображению и декодированием ответа. Если новый набор данных будет отличаться от предыдущего, то на обработку поступит неправильно выбранная строка. Если новый набор данных будет пуст, то никакие действия вообще не производятся, поскольку соответствующие ссылки не обнаруживаются.

В рассматриваемом примере не иллюстрируется добавление строк, но для его дополнения в этом отношении можно воспользоваться той же идеей. Предоставьте ссылки Add above (Добавить выше) рядом с каждым элементом, а затем предусмотрите дополнительную ссылку для добавления строки ниже последней. На рис. 6.11 показана структура каталогов для приложения. В листингах 6.8 и 6.9 содержатся код страницы JSF и управляемого бина.

Рис. 6.11. Структура каталогов для примера удаления

Листинг 6.8. Файл `delete/web/index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:ui="http://java.sun.com/jsf/facelets"
6.       xmlns:f="http://java.sun.com/jsf/core"
7.       xmlns:h="http://java.sun.com/jsf/html">
8.    <h:head>
9.       <h:outputStylesheet library="css" name="styles.css"/>
10.      <title>#{msgs.windowTitle}</title>
11.   </h:head>
12.   <h:body>
13.      <h:form>
14.         <h:dataTable value="#{tableData.names}" var="name" styleClass="names"
15.                      headerClass="namesHeader" columnClasses="last,first">
16.            <h:column>
17.               <f:facet name="header">#{msgs.lastColumnHeader}</f:facet>
```

```
            #{name.last}
         </h:column>
         <h:column>
            <f:facet name="header">#{msgs.firstColumnHeader}</f:facet>
            #{name.first}
         </h:column>
         <h:column>
            <h:commandLink value="Delete"
                           action="#{tableData.deleteRow(name)}"/>
         </h:column>
      </h:dataTable>
   </h:form>
</h:body>
</html>
```

Листинг 6.9. Файл delete/src/java/com/corejsf/TableData.java

```
package com.corejsf;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;

@Named // или @ManagedBean
@SessionScoped
public class TableData implements Serializable {
   private ArrayList<Name> names = new ArrayList<Name>(Arrays.asList(
      new Name("Anna", "Keeney"),
      new Name("John", "Wilson"),
      new Name("Mariko", "Randor"),
      new Name("William", "Dupont")
   ));

   public ArrayList<Name> getNames() {
      return names;
   }

   public String deleteRow(Name nameToDelete) {
      names.remove(nameToDelete);
      return null;
   }
}
```

Таблицы базы данных

Базы данных хранят информацию в таблицах, поэтому компонент таблицы данных JSF хорошо подходит для отображения данных, хранящихся в базе данных. В этом разделе будет показано, как отобразить результаты запроса к базе данных.

На рис. 6.12 показано приложение JSF, которое отображает таблицу базы данных.

Customer ID	Name	Phone Number	Address	City	State
1	William Dupont	(652)488-9931	801 Oak Street	Eugene	Nebraska
2	Anna Keeney	(716)834-8772	86 East Amherst Street	Buffalo	New York
3	Mariko Randor	(451)842-8933	923 Maple Street	Springfield	Tennessee
4	John Wilson	(758)955-5934	8122 Genessee Street	El Reno	Oklahoma
5	Lynn Seckinger	(552)767-1935	712 Kehr Street	Kent	Washington
6	Richard Tattersall	(455)282-2936	21 South Park Drive	Dallas	Texas
7	Gabriella Sarintia	(819)152-8937	81123 West Seneca Street	Denver	Colorado
8	Lisa Hartwig	(818)852-1937	6652 Sheridan Drive	Sheridan	Wyoming
9	Shirley Jones	(992)488-3931	2831 Main Street	Butte	Montana
10	Bill Sprague	(316)962-0632	1043 Cherry Street	Cheektowaga	New York
11	Greg Doench	(136)692-6023	99 Oak Street	Upper Saddle River	New Jersey
12	Solange Nadeau	(255)767-0935	177 Rue St. Catherine	Montreal	Quebec
13	Heather McGann	(554)282-0936	7192 913 West Park	Buloxie	Mississippi
14	Roy Martin	(918)888-0937	5571 North Olean Avenue	White River	Arkansas
15	Claude Loubier	(857)955-0934	1003 Rue de la Montagne	St. Marguerite de Lingwick	Quebec
16	Dan Woodard	(703)555-1212	2993 Tonawonda Street	Springfield	Missouri
17	Ron Dunlap	(761)678-4251	5579 East Seneca Street	Kansas City	Kansas
18	Keith Frankart	(602)152-6723	88124 Milpidas Lane	Springfield	Maryland
19	Andre Nadeau	(541)842-0933	94219 Rue Florence	St. Marguerite de Lingwick	Quebec
20	Horace Celestin	(914)843-6553	99423 Spruce Street	Ann Arbor	Michigan

Рис. 6.12. Отображение таблиц базы данных

На странице JSF, показанной на рис. 6.12, используется тег `h:dataTable`:

```
<h:dataTable value="#{customerBean.all}" var="customer"
   styleClass="customers"
   headerClass="customersHeader"
   columnClasses="custid,name">
      <h:column>
      <f:facet name="header">#{msgs.customerIdHeader}</f:facet>
       #{customer.Cust_ID}
   </h:column>
   <h:column>
      <f:facet name="header">#{msgs.nameHeader}</f:facet>
       #{customer.Name}
   </h:column>
   ...
</h:dataTable>
```

где `customerBean` — управляемый бин, который позволяет подключаться к базе данных и выполнять запрос для получения данных о всех заказчиках из базы данных. Для выполнения этого запроса применяется метод `CustomerBean.all`.

Во время работы с базой данных значением, заданным для тега `h:dataTable`, является экземпляр `java.sql.ResultSet` или `javax.servlet.jsp.jstl.Result`. Однако не следует использовать результирующий набор, возвращенный методом `Statement.executeQuery`. Чтобы можно было подготовить этот результирующий набор к отображению, основополагающее соединение с базой данных должно оставаться открытым. Но затем шансы закрыть его больше не появятся. Лучший путь состоит в использовании оболочки, которая сохраняет результаты запроса, такой как `javax.sql.CachedRowSet` или `javax.servlet.jsp.jstl.Result` (эти оболочки были придуманы еще до того, как

объект CachedRowSet стал частью спецификации Java 5). В рассматриваемом примере используется объект CachedRowSet.

На предыдущей странице JSF доступ к данным столбцов осуществлялся с помощью ссылок на имена столбцов, например, выражение #{customer.Cust_ID} ссылается на столбец Cust_ID. Структура каталогов для примера с базой данных показана на рис. 6.13. Код веб-страницы и управляемого бина для приложения содержится в листингах 6.10-6.11.

На заметку! В этих примерах предполагается, что читатель знает, как задавать источник данных с помощью сервера приложений. В противном случае за подробными сведениями обратитесь к главе 12.

Рис. 6.13. Структура каталогов для примера базы данных

Листинг 6.10. Файл database/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.pageTitle}</title>
   </h:head>
   <h:body>
      <h:form>
         <h:dataTable value="#{customerBean.all}" var="customer"
                      styleClass="customers" headerClass="customersHeader"
                      columnClasses="custid,name">
            <h:column>
               <f:facet name="header">#{msgs.customerIdHeader}</f:facet>
               #{customer.Cust_ID}
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.nameHeader}</f:facet>
               #{customer.Name}
            </h:column>
```

```
            <h:column>
               <f:facet name="header">#{msgs.phoneHeader}</f:facet>
               #{customer.Phone_Number}
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.addressHeader}</f:facet>
               #{customer.Street_Address}
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.cityHeader}</f:facet>
               #{customer.City}
            </h:column>
            <h:column>
               <f:facet name="header">#{msgs.stateHeader}</f:facet>
               #{customer.State}
            </h:column>
         </h:dataTable>
      </h:form>
   </h:body>
</html>
```

Листинг 6.11. Файл database/src/java/com/corejsf/CustomerBean.java

```
package com.corejsf;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.annotation.Resource;
import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.RequestScoped;
   // или import javax.faces.bean.RequestScoped;
import javax.sql.DataSource;
import javax.sql.rowset.CachedRowSet;

@Named // или @ManagedBean
@RequestScoped
public class CustomerBean {
   @Resource(name="jdbc/mydb")
   private DataSource ds;

   public ResultSet getAll() throws SQLException {
      Connection conn = ds.getConnection();
      try {
         Statement stmt = conn.createStatement();
         ResultSet result = stmt.executeQuery("SELECT * FROM Customers");
         // return ResultSupport.toResult(result);
         CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
            // или использовать реализацию от поставщика базы данных
         crs.populate(result);
         return crs;
      } finally {
         conn.close();
      }
   }
}
```

Модели таблиц

Если для представления данных таблицы используется объект Java, массив, список, результирующий набор или результирующий объект JSTL, то тег `h:dataTable` заключает эти объекты в оболочку в виде модели, которая расширяет класс `javax.faces.model.DataModel`. Все применимые при этом классы модели, перечисленные ниже, находятся в пакете `javax.faces.model`.

- `ArrayDataModel`.
- `ListDataModel`.
- `ResultDataModel`.
- `ResultSetDataModel`.
- `ScalarDataModel`.

Обычно разработчику не требуется иметь сведения о модели, если он использует тег `h:dataTable`, но в следующих разделах рассматриваются три задачи, которые приходится решать путем обращения к модели: подготовка к отображению номеров строк, поиск выбранной пользователем строки и сортировка строк таблицы.

Подготовка к отображению номеров строк

Предположим, необходимо подготовить к отображению номера строк, как показано на рис. 6.14.

	Last Name	First Name
1	Keeney,	Anna
2	Wilson,	John
3	Randor,	Mariko
4	Dupont,	William

Рис. 6.14. Подготовка к отображению номеров строк

В таблице данных JSF не предусмотрен простой механизм для выработки этих номеров. Отображение различного содержимого в каждой строке осуществляется с использованием переменной, объявленной с атрибутом `var`. Эта переменная содержит данные строки, но не номер строки текста (разумеется, если только каждое значение строки само не содержит номер строки текста).

Но класс `DataModel` имеет метод `getRowIndex`, применение которого приводит к получению текущего номера строки. К этому методу можно обратиться со страницы JSF, при условии, что в приложении применяется модель таблицы, а не коллекция. Например, класс `TableData` можно изменить следующим образом:

```
public class TableData implements Serializable {
   private static final Name[] names = new Name[] {
       new Name("William", "Dupont"),
       new Name("Anna", "Keeney"),
       new Name("Mariko", "Randor"),
       new Name("John", "Wilson")
   };
   private DataModel<Name> model = new ArrayDataModel<Name>(names);
   public DataModel<Name> getNames() { return model; }
}
```

Следует отметить, что метод `getNames` возвращает значение `DataModel<Name>` вместо массива `Name[]`.

После этого необходимо добавить столбец:

```
<h:dataTable value="#{tableData.names}" var="name">
   <h:column>#{tableData.names.rowIndex + 1}</h:column>
   <h:column>#{name.last}</h:column>
```

```
    <h:column>#{name.first}</h:column>
</h:dataTable>
```

Этот способ имеет один существенный недостаток: он вынуждает учитывать в управляемых бинах методы API-интерфейса JSF. См. раздел "Тег ui:repeat" на стр. 196 для ознакомления с альтернативным подходом.

Поиск выбранной строки

Еще одна причина, по которой целесообразнее предоставить доступ к объекту DataModel вместо основополагающей коллекции, состоит в том, что это позволяет найти выбранную строку с помощью действия или прослушивателя действий. Еще раз вернемся к примеру "Удаление строк" на стр. 203, в котором применяется ссылка на таблицу данных:

```
<h:dataTable value="#{tableData.names}" var="name">
    ...
    <h:commandLink value="Delete" action="{tableData.deleteRow}"/>
    ...
</h:dataTable>
```

При работе с методом deleteRow необходимо знать, какая строка содержит ссылку, выбранную пользователем. Раньше мы решали эту задачу, передавая значение строки в качестве параметра: action="{tableData.deleteRow(name)}".

Но до появления версии JSF 2.0 не было возможности передавать параметры для метода action. Вместо этого можно осуществить выборку текущего элемента путем вызова метода getRowData класса DataModel. Например:

```
public String deleteRow() {
    Name nameToDelete = model.getRowData();
    names.remove(nameToDelete);
    return null;
}
```

Сортировка и фильтрация

Чтобы иметь возможность сортировать или фильтровать данные таблиц с помощью тега h:dataTable, необходимо реализовать модель таблицы, предназначенную для оформления одной из моделей таблиц, приведенных на стр. 209. На рис. 6.15 показано, что подразумевается под оформлением модели таблицы.

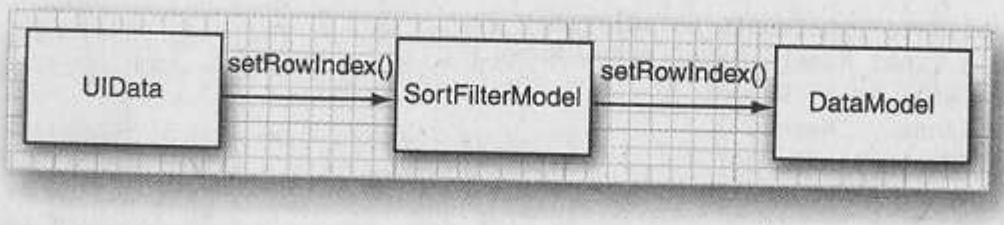


Рис. 6.15. Фильтр модели данных

В то время как реализация JSF подготавливает к отображению или декодирует данные таблицы, происходит вызов методов применительно к модели таблицы. При оформлении модели рассматриваемая модель перехватывает эти вызовы методов, создавая иллюзию, что данные отсортированы.

На рис. 6.16 показана применяемая нами базовая таблица, которая перезаписана в целях поддержки сортируемых столбцов таблицы.

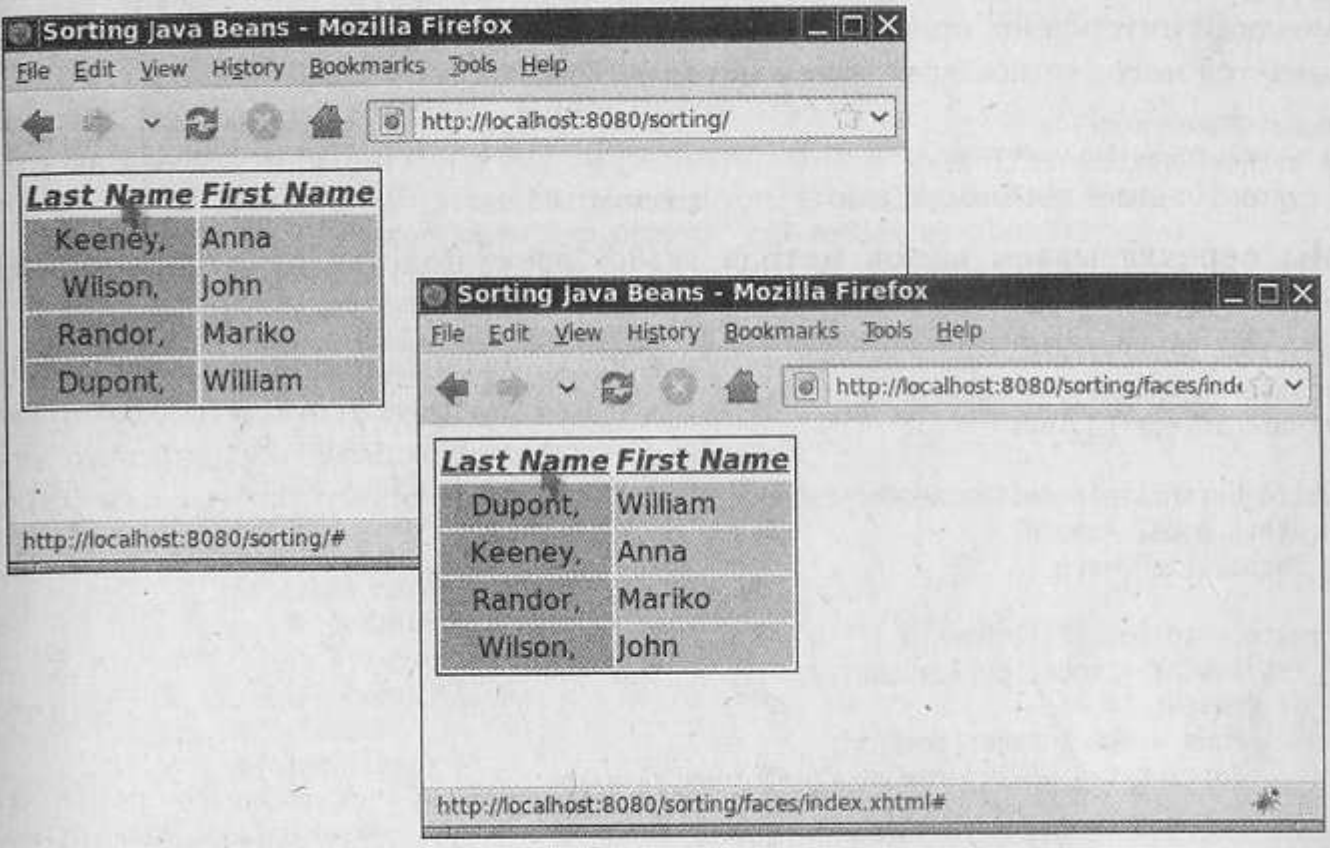


Рис. 6.16. Сортировка столбцов таблицы

Приложение сортирует столбцы таблицы путем оформления модели данных таблицы. Прежде всего необходимо определить атрибут `value` тега `h:dataTable`:

```
<h:dataTable value="#{tableData.names}" var="name" ...>
```

Метод `TableData.names` возвращает модель данных:

```
public class TableData {
   private SortFilterModel<Name> filterModel;
   private static final Name[] names = {
      new Name("Anna", "Keeney"),
      new Name("John", "Wilson"),
      new Name("Mariko", "Randor"),
      new Name("William", "Dupont"),
   };

    public TableData() {
       filterModel = new SortFilterModel<Name>(new ArrayDataModel(names));
    }
    public DataModel<Name> getNames() {
        return filterModel;
    }
}
```

При создании объекта `tableData` создается экземпляр `ArrayDataModel`, и ему передается массив имен. Это — исходная модель. Затем конструктор `TableData` заключает эту модель в оболочку в виде модели сортировки. При последующем вызове метода `getNames` для заполнения таблицы данных этот метод возвращает модель сортировки. Модель сортировки содержит обычную модель и массив строк целых чисел, где `rows[i]` указывает индекс данных модели, которые должны быть отображены в i-й строке. Чтобы отсортировать массив разными способами, достаточно выполнить сортировку индексов строк.

Чтобы лучше понять детали этой реализации, необходимо приобрести определенные знания об API-интерфейсе `DataModel`. Модель `DataModel` имеет довольно-таки

громоздкий интерфейс, предназначенный для получения элементов данных. Вначале вызывается метод setRowIndex, затем метод getRowData:

```
DataModel<Name> model = ...;
model.setRowIndex(currentIndex);
Name current = model.getRowData();
```

Мы перехватываем вызов метода setRowIndex, подставляя отсортированный индекс:

```
public class SortFilterModel<E> extends DataModel<E> {
   private DataModel<E> model;
   private Integer[] rows;
   ...
   public SortFilterModel(DataModel<E> model) {
      this.model = model;
      initializeRows();
   }
   private void initializeRows() {
      int rowCnt = model.getRowCount();
      if (rowCnt != -1) {
         rows = new Integer[rowCnt];
         for(int i = 0; i < rowCnt; i++) rows[i] = i;
      }
   }
   public void setRowIndex(int rowIndex) {
      if (0 <= rowIndex && rowIndex < model.getRowCount())
         model.setRowIndex(rows[i]);
      else
         model.setRowIndex(rowIndex);
   }
   ...
}
```

Для обеспечения сортировки вызывающий метод должен предоставить функцию сравнения данных. Модель фильтра сортировки использует эту функцию сравнения данных для переупорядочения массива индексов строк. С дополнительными сведениями можно ознакомиться в листинге 6.13 на стр. 213. (У читателя может возникнуть вопрос: почему мы использовали тип Integer вместо int для строк? Причина этого состоит в том, что метод Arrays.sort можно использовать в сочетании с пользовательской функцией сравнения только для массива Integer[], но не для массива int[].)

Структура каталогов для примера сортировки показана на рис. 6.17. В листингах 6.12–6.14 приведены страница JSF, модель сортировки и управляемый бин.

На заметку! В спецификации JSF рекомендуется, чтобы конкретные классы DataModel предоставляли по крайней мере два конструктора: конструктор без параметров, который вызывает метод setWrappedData(null), и конструктор, передающий заключенные в оболочку данные методу setWrappedData(). См. листинг 6.13 на стр. 213 для ознакомления с примерами применения этих конструкторов.

Листинг 6.12. Файл sorting/web/index.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:f="http://java.sun.com/jsf/core"
6.       xmlns:h="http://java.sun.com/jsf/html">
7.    <h:head>
```

```
8.        <h:outputStylesheet library="css" name="styles.css"/>
9.        <title>#{msgs.windowTitle}</title>
10.    </h:head>
11.    <h:body>
12.       <h:form>
13.          <h:dataTable value="#{tableData.names}" var="name" styleClass="names"
14.                       headerClass="namesHeader" columnClasses="last,first">
15.             <h:column>
16.                <f:facet name="header">
17.                   <h:commandLink action="#{tableData.sortByLast}">
18.                      #{msgs.lastColumnHeader}
19.                   </h:commandLink>
20.                </f:facet>
21.                #{name.last},
22.             </h:column>
23.             <h:column>
24.                <f:facet name="header">
25.                   <h:commandLink action="#{tableData.sortByFirst}">
26.                      #{msgs.firstColumnHeader}
27.                   </h:commandLink>
28.                </f:facet>
29.                #{name.first}
30.             </h:column>
31.          </h:dataTable>
32.       </h:form>
33.    </h:body>
34. </html>
```

Рис. 6.17. Структура каталогов для примера сортировки

Листинг 6.13. Файл `sorting/src/java/com/corejsf/SortFilterModel.java`

```
1. package com.corejsf;
2.
3. import java.util.Arrays;
4. import java.util.Comparator;
5.
6. import javax.faces.model.DataModel;
7.
8. public class SortFilterModel<E> extends DataModel<E> {
```

```
10.    private Integer[] rows;
11.
12.    public SortFilterModel() { // Требуется по спецификации JSF
13.       setWrappedData(null);
14.    }
15.    public SortFilterModel(E[] names) { // Рекомендуется в спецификации JSF
16.       setWrappedData(names);
17.    }
18.    public SortFilterModel(DataModel<E> model) {
19.       this.model = model;
20.       initializeRows();
21.    }
22.
23.    private E getData(int row) {
24.       int originalIndex = model.getRowIndex();
25.       model.setRowIndex(row);
26.       E thisRowData = model.getRowData();
27.       model.setRowIndex(originalIndex);
28.       return thisRowData;
29.    }
30.
31.    public void sortBy(final Comparator<E> dataComp) {
32.       Comparator<Integer> rowComp = new
33.       Comparator<Integer>() {
34.          public int compare(Integer r1, Integer r2) {
35.             E e1 = getData(r1);
36.             E e2 = getData(r2);
37.             return dataComp.compare(e1, e2);
38.          }
39.       };
40.       Arrays.sort(rows, rowComp);
41.    }
42.
43.    public void setRowIndex(int rowIndex) {
44.       if (0 <= rowIndex && rowIndex < rows.length)
45.          model.setRowIndex(rows[rowIndex]);
46.       else
47.          model.setRowIndex(rowIndex);
48.    }
49.
50.    // Следующие методы являются делегатами для оформленной модели
51.
52.    public boolean isRowAvailable() {
53.       return model.isRowAvailable();
54.    }
55.    public int getRowCount() {
56.       return model.getRowCount();
57.    }
58.    public E getRowData() {
59.       return model.getRowData();
60.    }
61.    public int getRowIndex() {
62.       return model.getRowIndex();
63.    }
64.    public Object getWrappedData() {
65.       return model.getWrappedData();
66.    }
67.    public void setWrappedData(Object data) {
68.       model.setWrappedData(data);
69.       initializeRows();
70.    }
71.    private void initializeRows() {
```

```
72.         int rowCnt = model.getRowCount();
73.         if (rowCnt != -1) {
74.            rows = new Integer[rowCnt];
75.            for(int i = 0; i < rowCnt; ++i) rows[i] = i;
76.         }
77.      }
78. }
```

Листинг 6.14. Файл sorting/src/java/com/corejsf/TableData.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.util.Comparator;
5.
6. import javax.inject.Named;
7.    // или import javax.faces.bean.ManagedBean;
8. import javax.enterprise.context.SessionScoped;
9.    // или import javax.faces.bean.SessionScoped;
10. import javax.faces.model.DataModel;
11. import javax.faces.model.ArrayDataModel;
12.
13. @Named // или @ManagedBean
14. @SessionScoped
15. public class TableData implements Serializable {
16.    private SortFilterModel<Name> filterModel;
17.    private static final Name[] names = {
18.       new Name("Anna", "Keeney"),
19.       new Name("John", "Wilson"),
20.          new Name("Mariko", "Randor"),
21.          new Name("William", "Dupont"),
22.    };
23.
24.    public TableData() {
25.       filterModel = new SortFilterModel<Name>(new ArrayDataModel<Name>(names));
26.    }
27.    public DataModel<Name> getNames() {
28.       return filterModel;
29.    }
30.
31.    public String sortByFirst() {
32.       filterModel.sortBy(new Comparator<Name>() {
33.          public int compare(Name n1, Name n2) {
34.             return n1.getFirst().compareTo(n2.getFirst());
35.          }
36.       });
37.       return null;
38.    }
39.
40.    public String sortByLast() {
41.       filterModel.sortBy(new Comparator<Name>() {
42.          public int compare(Name n1, Name n2) {
43.             return n1.getLast().compareTo(n2.getLast());
44.          }
45.       });
46.       return null;
47.    }
48. }
```

javax.faces.model.DataModel<E>

- `int getRowCount()`

  Возвращает общее количество строк, если оно известно; в противном случае этот метод возвращает –1. Метод `ResultSetDataModel` всегда возвращает –1 из этого метода.

- `int getRowIndex()`

  Возвращает индекс текущей строки.

- `void setRowIndex(int index)`

  Задает индекс текущей строки и обновляет переменную с заданной областью, представляющую текущий элемент в коллекции (эта переменная задается с помощью атрибута `var` тега `h:dataTable`).

- `E getRowData()`

  Возвращает данные, связанные с текущей строкой.

- `boolean isRowAvailable()`

  Возвращает `true`, если по индексу текущей строки имеются действительные данные.

- `Iterator<E> iterator()` JSF 2.0

  Возвращает итератор, применимый для перебора всех строк.

- `void addDataModelListener(DataModelListener listener)`

  Добавляет прослушиватель модели данных, получающий извещение при изменении индекса строки.

- `void removeDataModelListener(DataModelListener listener)`

  Удаляет прослушиватель модели данных.

- `void setWrappedData(Object obj)`

  Задает объект, заключаемый моделью данных в оболочку.

- `Object getWrappedData()`

  Возвращает заключенные в оболочку данные модели данных.

Способы прокрутки

Предусмотрены два способа прокрутки таблиц с большим количеством строк: с помощью полосы прокрутки или элемента управления какого-то другого типа, который может перемещаться по строкам. В настоящем разделе будут продемонстрированы оба способа.

Прокрутка с помощью полосы прокрутки

Прокрутка с помощью полосы прокрутки представляет собой самое простое решение. Для этого достаточно заключить применяемый тег `h:dataTable` в оболочку в виде элемента `div` языка HTML, как в следующем примере:

```
<div style="overflow:auto; width:100%; height:200px;">
   <h:dataTable...>
      <h:column>
         ...
      </h:column>
      ...
   </h:dataTable>
</div>
```

Приложение, показанное на рис. 6.18, идентично приложению, обсуждаемому в разделе "Таблицы базы данных" на стр. 205, за исключением того, что таблица данных размещена в прокручиваемом элементе `div`, как показано выше.

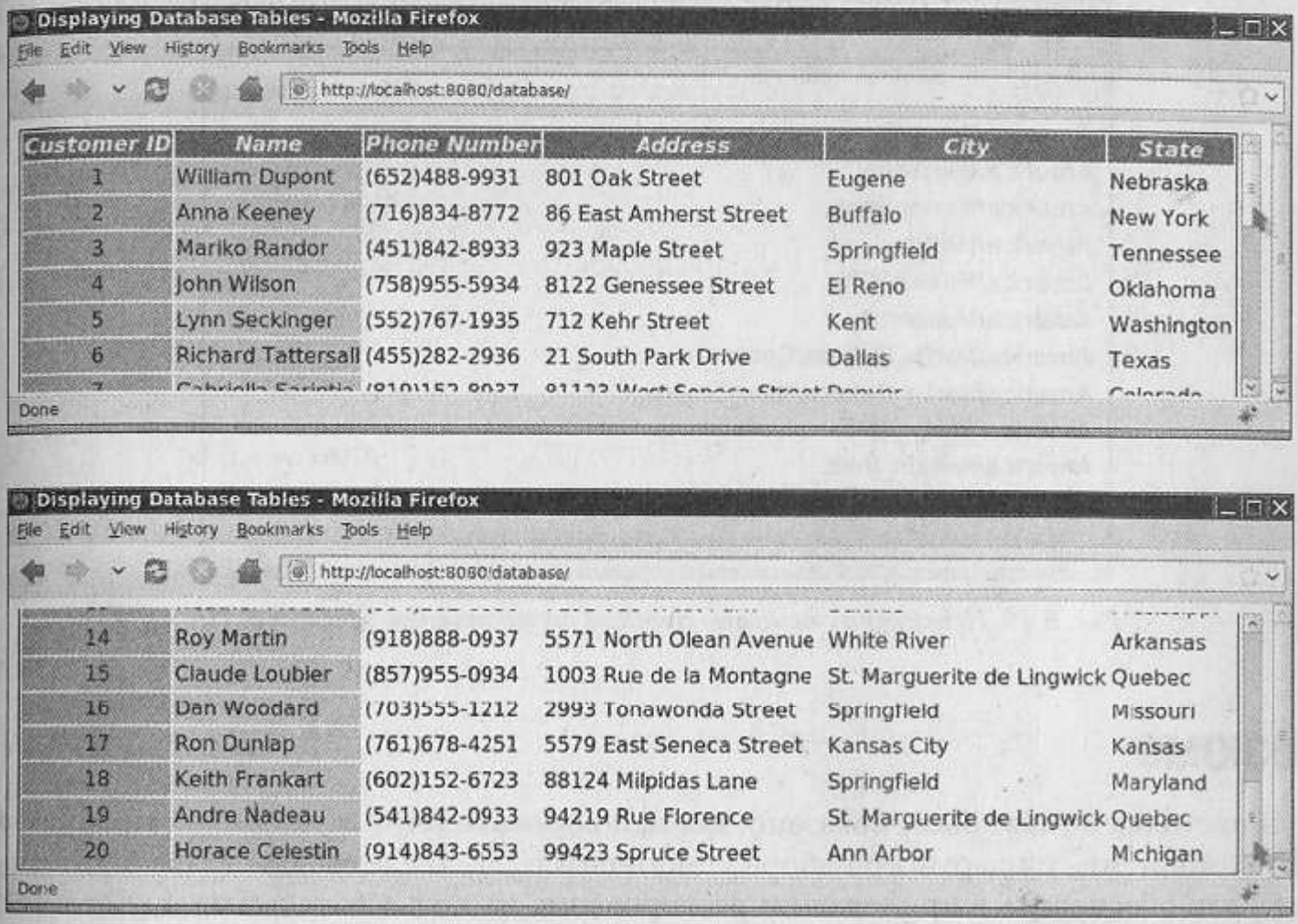


Рис. 6.18. Прокрутка таблицы с применением прокручиваемого элемента `div`

Полосы прокрутки вполне приемлемы с точки зрения удобства и простоты использования, но они могут оказаться дорогостоящими при обработке крупных таблиц, поскольку при их применении загружаются сразу все данные таблицы. Вариант, требующий меньшего расхода ресурсов, состоит в осуществлении прокрутки таблиц с помощью мини-приложений постраничного просмотра, поскольку при этом подходе требуется считывание только одной страницы данных одновременно.

Прокрутка с помощью мини-приложений постраничного просмотра

Прокрутка с помощью мини-приложений постраничного просмотра является более эффективной по сравнению с прокруткой на основе прокручиваемого элемента `div`, но вместе с тем является также значительно более сложной. В главе 13 будет показано, как реализовать мини-приложение постраничного просмотра, которое можно использовать с любой таблицей, созданной с помощью тега `h:dataTable` (см. раздел "Как показывать крупный набор данных по одной странице одновременно" главы 13 на стр. 488). На рис. 6.19 показан пример применения такого средства постраничного просмотра.

В приложении, приведенном на рис. 6.19, используется таблица данных, которая отображает коды стран по стандарту ISO для локалей. Этот список получен путем вы-

зова `java.util.Locale.getISOCountries()`, статического метода, который возвращает массив строк.

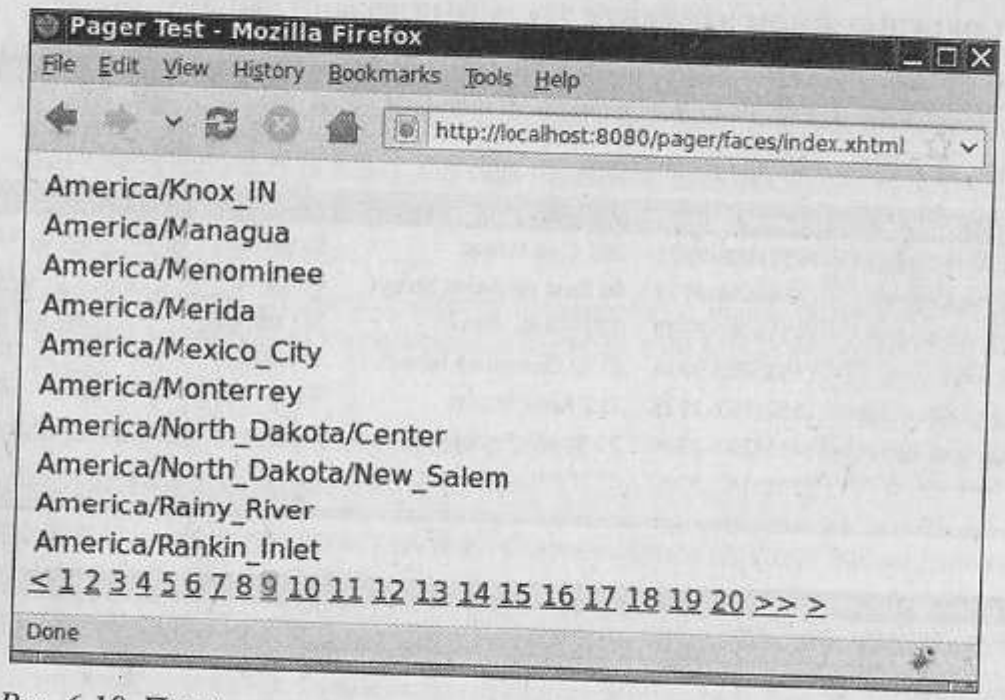


Рис. 6.19. Прокрутка с помощью средства постраничного просмотра JSF

Резюме

В настоящей главе было показано, как использовать тег `h:dataTable` — самый сложный компонент стандартного набора компонентов JSF. В следующей главе будет описано, как обеспечить в приложении реагирование на пользовательские и системные события.

Преобразование и проверка правильности

В этой главе...

- Общие сведения о процессе преобразования и проверки правильности
- Использование стандартных преобразователей
- Использование стандартных средств проверки
- Проверка правильности бина JSF 2.0
- Программирование с применением пользовательских преобразователей и средств проверки
- Реализация пользовательских тегов преобразователей и средств проверки

Глава 7

В настоящей главе рассматривается вопрос о том, как происходят преобразование данных формы в объекты Java и проверка правильности преобразования. Контейнер JSF выполняет эти шаги перед обновлением модели, поэтому разработчик может быть уверен в том, что недопустимые входные данные никогда не появятся в бизнес-логике.

Вначале рассматривается то, какие понятия лежат в основе процессов преобразования и проверки правильности. Затем приведено описание стандартных тегов, применяемых в технологии JSF для преобразования и проверки правильности. Эти теги позволяют выполнять наиболее распространенные действия. После этого показано, как предоставить собственный код преобразования и проверки правильности в более сложных сценариях.

Предусмотрена также возможность реализовывать специализированные теги — повторно используемые преобразователи и средства проверки, настройка которых может осуществляться авторами страниц. Однако для реализации специализированных тегов требуется значительно больший объем программирования. Применяемые для этого способы рассматриваются в последней части этой главы.

Общие сведения о процессе преобразования и проверке правильности

Прежде всего разберем по частям, как происходит обработка данных, введенных пользователем, по мере перехода из формы в браузере в бины, которые составляют бизнес-логику.

Сначала пользователь заполняет поле веб-формы. После того как он щелкнет на кнопке передачи формы, браузер отправляет введенное пользователем значение на сервер с использованием HTTP-запроса. Это значение принято называть *значением запроса*.

На этапе применения значений запроса реализация JSF сохраняет значения запроса в объектах компонентов. (Напомним, что с каждым входным тегом страницы JSF связан соответствующий объект компонента.) Значение, сохраняемое в объекте компонента, называется *переданным значением*.

Безусловно, все значения запроса являются строковыми, ведь в конечном итоге клиентский браузер передает именно строки, введенные пользователем. С другой

стороны, само веб-приложение рассчитано на работу с произвольными типами, такими как `int` или `Date`, и даже более сложными типами. Процесс преобразования предназначен для приведения входящих строк к этим типам. В следующем разделе преобразование будет рассматриваться более подробно.

Преобразованные значения не передаются немедленно в бины, которые составляют бизнес-логику. Вместо этого они вначале сохраняются в объектах компонентов как локальные значения. А после преобразования происходит проверка всех локальных значений. Разработчики приложений могут определять условия проверки правильности, например, указывая, что определенные поля должны иметь минимальную или максимальную длину. Обсуждение проверки правильности начинается с раздела "Использование стандартных средств проверки". После проверки всех локальных значений начинается этап обновления значений модели, и локальные значения сохраняются в бинах в соответствии с тем, что указано в их ссылках на конкретные значения.

Может возникнуть вопрос, почему в технологии JSF уделяется столько внимания локальным значениям. Нельзя ли просто сохранить значения запроса непосредственно в модели?

В технологии JSF используется двухэтапный подход, позволяющий проще обеспечивать целостность модели. Все программисты слишком хорошо знают, что пользователи вводят неправильные данные с неумолимым постоянством. Учитывая это, предположим, что обновление некоторых из значений модели происходит до обнаружения даже первой пользовательской ошибки. В таком случае модель может оказаться в несогласованном состоянии, и будет нелегко снова возвратить ее в прежнее, нормальное состояние.

По этой причине в технологии JSF вначале осуществляются преобразование и проверка правильности всего пользовательского ввода. При обнаружении ошибки снова отображается предыдущая страница со значениями, введенными пользователем, предоставляющая пользователю возможность сделать еще одну попытку. Этап обновления значений модели начинается только после того, как все проверки правильности окажутся успешными.

На рис. 7.1 показано, какой путь проходит значение поля от браузера до серверного объекта компонента, а затем к бину модели.

Рис. 7.1. Путь перемещения значения от браузера к модели

Использование стандартных преобразователей

В первой части этой главы рассматриваются преобразователи и средства проверки, которые входят в состав библиотеки JSF.

Преобразование чисел и дат

В веб-приложении сохраняются данные многих типов, но пользовательский веб-интерфейс работает исключительно со строками. Например, предположим, что пользователю необходимо отредактировать содержимое объекта `Date`, который хранится в бизнес-логике. Вначале объект `Date` преобразуется в строку, которая передается в клиентский браузер для отображения в текстовом поле. Затем пользователь редактирует текстовое поле. Сформированная в конечном итоге строка возвращается на сервер и должна быть снова преобразована в объект `Date`.

Та же процедура, разумеется, выполняется применительно к примитивным типам, таким как `int`, `double` или `boolean`. Пользователь веб-приложения редактирует строки, после чего контейнер JSF должен преобразовать каждую строку в тип, необходимый для приложения.

Чтобы ознакомиться с типичным использованием встроенного преобразователя, достаточно представить себе веб-приложение, которое обрабатывает платежи (рис. 7.2). Данные, касающиеся оплаты, включают следующее.

- Сумма, подлежащая оплате.
- Номер кредитной карточки.
- Дата истечения срока годности кредитной карточки.

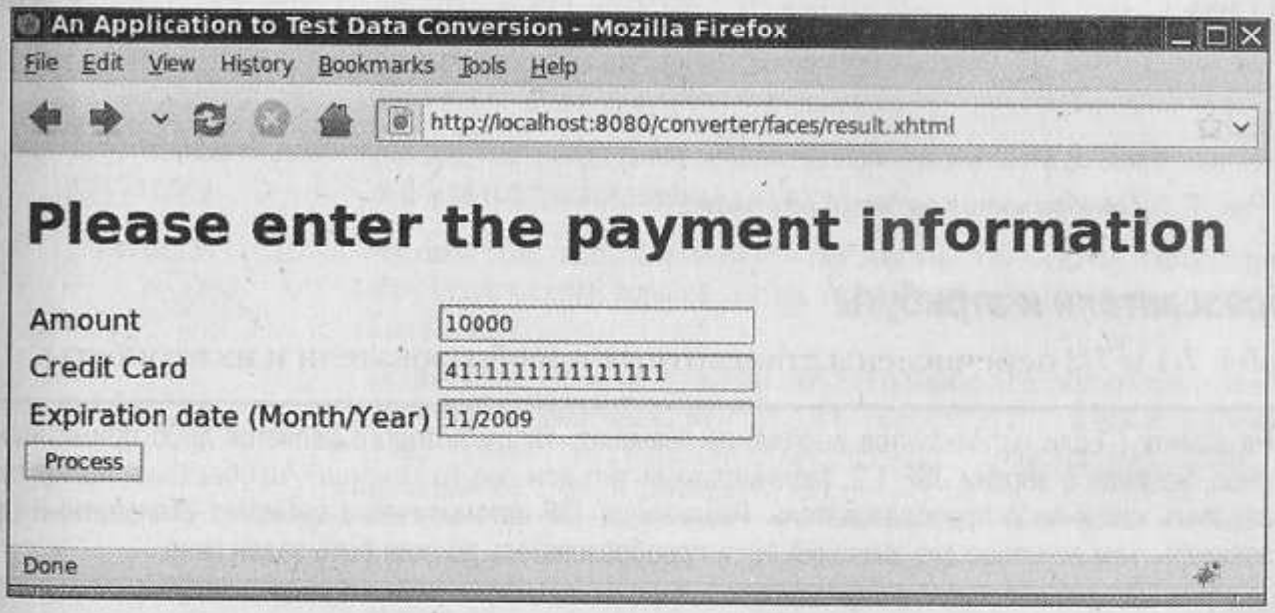


Рис. 7.2. Обработка платежей

За текстовым полем закрепляется преобразователь, который получает указание отформатировать текущее значение по крайней мере с двумя цифрами после десятичной точки:

```
<h:inputText value="#{payment.amount}">
   <f:convertNumber minFractionDigits="2"/>
</h:inputText>
```

Преобразователь `f:convertNumber` является одним из стандартных преобразователей, предоставляемых реализацией JSF.

Для второго поля на этом экране преобразователь не используется. (Ниже будет показано, как закрепить за полем пользовательский преобразователь.) Для третьего поля применяется преобразователь `f:convertDateTime`, в качестве атрибута `pattern` (шаблон) которого задана строка `MM/yyyy`. (Формат строки шаблона описан в той части документации API, которая касается класса `java.text.SimpleDateFormat`.)

```
<h:inputText value="#{payment.date}">
   <f:convertDateTime pattern="MM/yyyy"/>
</h:inputText>
```

На странице result.xhtml показаны входные данные, введенные пользователем, при том условии, что для суммы к оплате используется другой преобразователь:

```
<h:outputText value="#{payment.amount}">
   <f:convertNumber type="currency"/>
</h:outputText>
```

Этот преобразователь автоматически подставляет обозначение денежной единицы и десятичный разделитель (рис. 7.3).

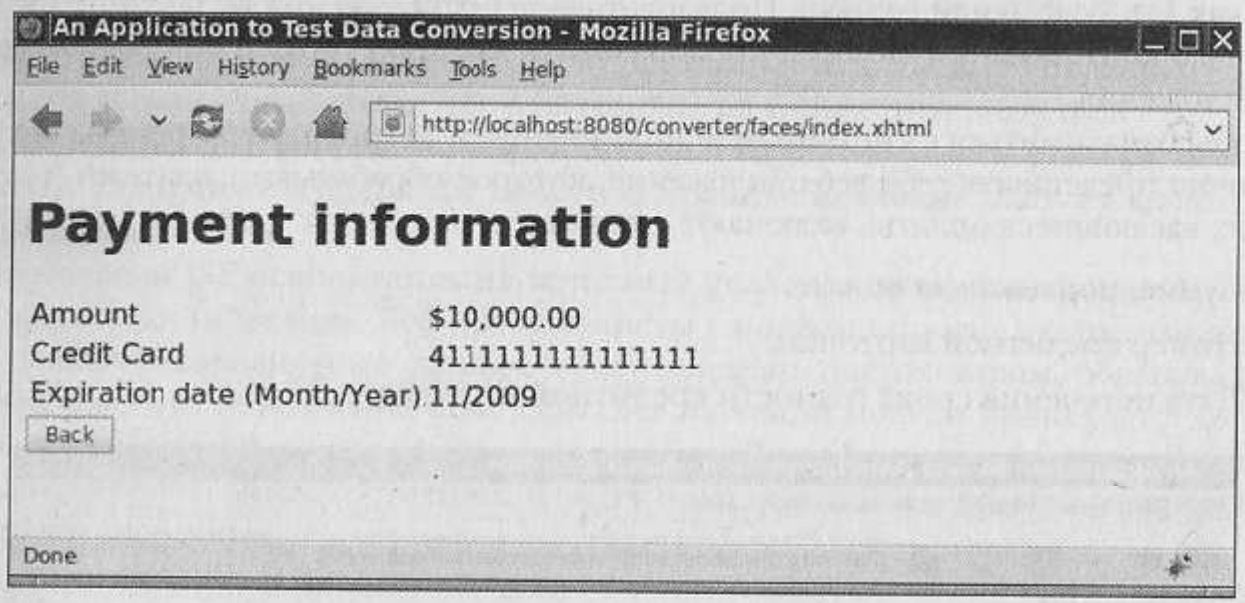


Рис. 7.3. Отображение сведений об оплате

Преобразователи и атрибуты

В табл. 7.1 и 7.2 перечислены стандартные преобразователи и их атрибуты.

На заметку! Если используется выражение значения, типом которого является либо примитивный тип, либо, начиная с версии JSF 1.2, перечислимый тип или тип BigInteger/BigDecimal, то не требуется задавать какой-либо преобразователь. Реализация JSF автоматически выбирает стандартный преобразователь. Тем не менее для значений Date преобразователь должен быть задан явно.

Таблица 7.1. Атрибуты тега f:convertNumber

Атрибут	Тип	Значение
type	String	number (по умолчанию), currency или percent
pattern	String	Форматирование шаблона, как определено в java.text.DecimalFormat
maxFractionDigits	int	Максимальное количество разрядов в дробной части
minFractionDigits	int	Минимальное количество разрядов в дробной части
maxIntegerDigits	int	Максимальное количество разрядов в целой части
minIntegerDigits	int	Минимальное количество разрядов в целой части
integerOnly	boolean	Значение true, только если проводится синтаксический анализ целой части (значение по умолчанию: false)

Окончание табл. 7.1

Атрибут	Тип	Значение
groupingUsed	boolean	Значение true, если используются группирующие разделители (значение по умолчанию: true)
locale	java.util.Locale или String	Локаль, настройки которой должны использоваться для синтаксического анализа и форматирования
currencyCode	String	Код валюты по стандарту ISO 4217, такой как USD или EUR, предназначенный для выбора преобразователя валюты
currencySymbol	String	Эта строка передается в DecimalFormat.setDecimalFormatSymbols, отменяя символ на основе локали. Не рекомендуется; вместо этого следует использовать currencyCode

Таблица 7.2. Атрибуты тега f:convertDateTime

Атрибут	Тип	Значение
type	String	date (по умолчанию), time или both
dateStyle	String	default, short, medium, long или full
timeStyle	String	default, short, medium, long или full
pattern	String	Форматирование шаблона, как определено в java.text.SimpleDateFormat
locale	java.util.Locale или String	Локаль, настройки которой должны использоваться для синтаксического анализа и форматирования
timeZone	java.util.TimeZone	Часовой пояс, используемый при синтаксическом анализе и форматировании; если часовой пояс не указан, то по умолчанию принимается время по Гринвичу. Примечание. Начиная с версии JSF 2.0 появилась возможность изменять значение по умолчанию на TimeZone.getDefault(), задавая значение javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE, равное true, в файле web.xml

Атрибут converter

Альтернативный синтаксис закрепления преобразователя за компонентом состоит в добавлении атрибута converter к тегу компонента. Идентификатор преобразователя определяется следующим образом:

```
<h:outputText value="#{payment.date}" converter="javax.faces.DateTime"/>
```

Это эквивалентно использованию тега f:convertDateTime без атрибута:

```
<h:outputText value="#{payment.date}">
   <f:convertDateTime/>
</h:outputText>
```

Третий способ определения преобразователя состоит в следующем:

```
<h:outputText value="#{payment.date}">
   <f:converter converterId="javax.faces.DateTime"/>
</h:outputText>
```

Все реализации JSF должны определять набор преобразователей с предопределенными идентификаторами:

- javax.faces.DateTime (используется тегом f:convertDateTime).
- javax.faces.Number (используется тегом f:convertNumber).
- javax.faces.Boolean, javax.faces.Byte, javax.faces.Character, javax.faces.Double, javax.faces.Float, javax.faces.Integer, javax.faces.Long, javax.faces.Short (автоматически используется для примитивных типов и их классов-оболочек).
- javax.faces.BigDecimal, javax.faces.BigInteger (автоматически используется для BigDecimal/BigInteger).

Дополнительные идентификаторы преобразователей можно задавать в файле конфигурации приложения (подробные сведения изложены в разделе "Определение преобразователей").

Внимание! Если значением атрибута converter является строка, то это значение указывает идентификатор преобразователя. Если же это — выражение значения, то результирующее значение должно представлять собой объект преобразователя — объект класса, который реализует интерфейс Converter. Этот интерфейс представлен ниже, в разделе "Реализация пользовательских классов преобразователей".

На заметку! Начиная с версии JSF 1.2 теги f:convertNumber, f:convertDateTime и f:converter имеют необязательный атрибут binding. Это позволяет связывать экземпляр преобразователя со свойством вспомогательного бина типа javax.faces.convert.Converter.

Ошибки преобразования

При возникновении ошибки преобразования реализация JSF выполняет следующие действия.

- Компонент, преобразование в котором окончилось неудачей, отправляет сообщение и объявляет себя находящемся в недопустимом состоянии. (Сведения о том, как отобразить это сообщение, будут приведены в следующих разделах.)
- Реализация JSF вновь отображает текущую страницу непосредственно после завершения этапа проверки правильности процесса. Вновь отображенная страница содержит все значения, предоставленные пользователем, поэтому не происходит потеря каких-либо данных, введенных пользователем.

Вообще говоря, именно таким и должно быть поведение приложения в случае ошибки. Если пользователь предоставляет недопустимые входные данные, скажем, для поля, которое требует ввода целого числа, то в веб-приложении не должна предприниматься попытка использовать этот недопустимый ввод. Реализация JSF автоматически повторно отображает текущую страницу, предоставляя пользователю еще один шанс ввести значение правильно.

Однако следует предотвращать применение чрезмерно ограничительных вариантов преобразования для полей ввода. В частности, рассмотрим в данном примере поле Amount (Сумма). Если используется формат валюты, то текущее значение будет привлекательно отформатировано с указанием символа валюты. Но допустим, что пользователь введет 100 (без ведущего знака $). Средство форматирования значения в валюте сообщит об ошибке, указывая, что введен недопустимый символ валюты. Но с точки зрения человека соблюдение такого требования является слишком ограничительным.

Чтобы преодолеть этот недостаток, можно предусмотреть в программе пользовательский преобразователь. Пользовательский преобразователь может форматировать введенное значение по всем правилам и вместе с тем оставаться дружественным, интерпретируя данные, введенные пользователем. Пользовательские преобразователи описаны ниже, в разделе "Реализация пользовательских классов преобразователей".

Совет. Собирая данные, введенные пользователем, необходимо либо использовать дружественные преобразователи, либо перепроектировать форму так, чтобы она была более удобной в работе. Например, вместо того, чтобы вынуждать пользователей вводить дату окончания срока годности в формате `MM/yyyy`, можно предусмотреть два поля ввода: одно для месяца, другое для года.

Отображение сообщений об ошибках

Необходимо обеспечить, чтобы пользователи видели сообщения, вызванные ошибками преобразования и проверки правильности. Добавляйте теги `h:message` каждый раз, когда используются преобразователи и средства проверки.

Как правило, сообщения об ошибках следует показывать рядом с компонентами, которые активизировали эти сообщения (рис. 7.4). Компонентам необходимо присвоить идентификаторы и ссылаться на них в теге `h:message`. Начиная с версии JSF 1.2 предусмотрена также возможность предоставлять метку компонента, которая отображается в сообщении об ошибке:

```
<h:inputText id="amount" label="#{msgs.amount}" value="#{payment.amount}"/>
<h:message for="amount"/>
```

При использовании версии JSF 1.1 следует опускать атрибут `label`.

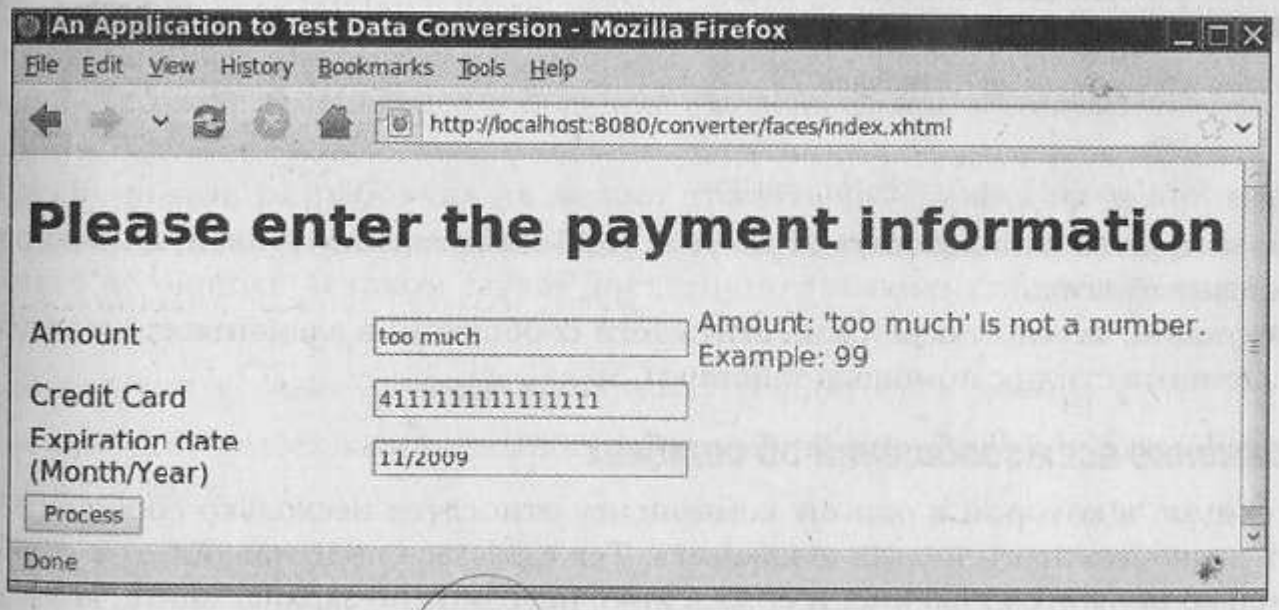


Рис. 7.4. Отображение сообщения об ошибке преобразования

В теге `h:message` применяется целый ряд атрибутов для описания внешнего вида сообщения (см. раздел "Сообщения" главы 4). В настоящей главе обсуждаются только атрибуты, которые представляют особый интерес с точки зрения передачи сообщений об ошибках.

Каждое сообщение имеет две версии: резюме и подробные сведения.

Что касается преобразователя чисел, то сообщение об ошибках с подробными сведениями показывает метку компонента, недопустимое значение и образец правильного значения, как в следующем примере:

```
Amount: 'too much' is not a number. Example: 99
```

В сообщении в виде резюме пример не приведен.

На заметку! В версии JSF 1.1 преобразователи отображали универсальное сообщение "Conversion error occurred (Произошла ошибка преобразования)".

По умолчанию тег h:message показывает подробные сведения и скрывает резюме. Если вместо этого необходимо показать сообщение в виде резюме, используйте следующие атрибуты:

```
<h:message for="amount" showSummary="true" showDetail="false"/>
```

Внимание! Если применяется стандартный преобразователь, используйте для отображения либо сообщение в виде резюме, либо сообщение с подробными сведениями, но не то и другое сразу, поскольку эти сообщения почти идентичны. Вряд ли какому-то разработчику захочется, чтобы пользователи его приложения ломали голову над сообщением об ошибке, которое выглядит примерно так: "...is not a number ... is not a number. Пример: 99".

Совет. Если разработчик не использует явно заданный преобразователь f:convertNumber, а вместо этого полагается на стандартные преобразователи для числовых типов, то ему следует применять сообщение в виде резюме, а не сообщение с подробными сведениями. Сообщения с подробными сведениями содержат слишком много пояснений. Например, стандартный преобразователь для значений с плавающей точкой двойной точности выдает следующее сообщение с подробными сведениями: ".. must be a number between 4.9E-324 and. 1.7976931348623157E308. Пример: 1999999".

Обычно возникает необходимость показывать сообщения об ошибках в другом цвете. Для изменения внешнего вида сообщения об ошибке используется атрибут styleClass или style:

```
<h:messages styleClass="errorMessage"/>
```

или

```
<h:message for="amount" style="color:red"/>
```

Рекомендуется использовать атрибут styleClass и таблицу стилей, а не жестко закодированный стиль.

Безусловно, можно также разместить теги сообщений в элементе div и задать для этого элемента стили с помощью таблицы CSS.

Отображение всех сообщений об ошибках

Ситуация, в которой к одному компоненту относятся несколько сообщений, возникает редко, но этого нельзя исключить. Тег h:message подготавливает к отображению только первое сообщение. К сожалению, невозможно заранее знать, является ли первое сообщение самым полезным для пользователя. Разумеется, ни один тег не позволяет показать все сообщения, относящиеся к конкретному компоненту, но можно подготовить листинг всех сообщений от всех компонентов с помощью тега h:messages.

По умолчанию тег h:messages показывает сообщения в виде резюме, а не сообщения с подробными сведениями. Тем самым он по своему поведению противоположен тегу h:message.

Значением по умолчанию атрибута layout тега h:messages является "list", что приводит к выводу ненумерованного списка, внешним видом которого можно управлять с помощью таблицы стилей. Иным образом, можно расположить сообщения по вертикали с использованием следующего:

```
<h:messages layout="table"/>
```

Совет. Тег `h:messages` является полезным средством для отладки. Каждый раз, когда приложение JSF застывает на конкретной странице и переход к следующим страницам становится невозможным, следует добавить тег `<h:messages/>`, чтобы узнать, является ли причиной этого нарушения в работе неудачное завершение преобразования или проверки правильности. Начиная с версии JSF 2.0 к представлению автоматически добавляется дочерний тег `<h:messages/>`, если в качестве стадии проекта задана разработка.

Внимание! В версии JSF 1.1 поведение средств вывода сообщений об ошибках состояло в том, что происходила конкатенация всех сообщений. Более того, сообщения об ошибках не включали метки сообщений. Тем самым удобство применения тега `h:messages` сводилось практически на нет, поскольку пользователям приходилось ломать голову, пытаясь догадаться о том, какие из введенных ими данных вызвали ошибку.

Изменение текста стандартных сообщений об ошибках

Иногда может потребоваться изменить стандартные сообщения преобразований для всего веб-приложения. В табл. 7.3 показаны самые полезные стандартные сообщения. Следует отметить, что все ключи сообщений с подробными сведениями оканчиваются суффиксом `_detail`. В целях экономии места в этой таблице не перечислены отдельно строки резюме и подробных сведений, если строка резюме является подстрокой строки с подробными сведениями. Вместо этого дополнительная часть, относящаяся к подробным сведениям, показана как продолжение. В большинстве сообщений `{0}` указывает недопустимое значение, `{1}` — образец допустимого значения и `{2}` — метку компонента; однако для преобразователя логического значения меткой компонента является `{1}`.

Чтобы заменить стандартное сообщение, задайте связку сообщений, как описано в главе 2. Добавьте заменяющее сообщение с использованием соответствующего ключа из табл. 7.3.

Предположим, разработчик не желает тратить свое время на разбирательство с входными метками или примерами значений, когда преобразователь `f:convertNumber` сообщает об ошибке. В таком случае достаточно добавить следующее определение к связке сообщений:

```
javax.faces.converter.NumberConverter.NUMBER_detail="{0}" is not a number.
```

После этого следует задать базовое имя связки в файле конфигурации (таком как `faces-config.xml`):

```
<faces-config>
   <application>
      <message-bundle>com.corejsf.messages</message-bundle>
   </application>
   ...
</faces-config>
```

Необходимо лишь указать сообщения, которые разработчик желает переопределить.

На заметку! Не следует путать связку сообщений со связкой ресурсов, доступ к которой осуществляется с помощью тега `resource-bundle` в файле `faces-config.xml`. Эти связки отображаются на переменную, которую можно использовать в выражениях значения. Связка, на которую ссылается тег `message-bundle`, используется для сообщений, вырабатываемых приложением.

Таблица 7.3. Стандартные сообщения об ошибках преобразования

Идентификатор ресурса	Текст, применяемый по умолчанию
javax.faces.converter.IntegerConverter.INTEGER	{2}: "{0}" must be a number consisting of one or more digits ({2}: "{0}" должен быть числом, состоящим из одной или более цифры)
javax.faces.converter.IntegerConverter.INTEGER_detail	{2}: "{0}" must be a number between -2147483648 and 2147483647. Example: {1} ({2}: "{0}" должен быть числом от -2147483648 до 2147483647. Пример: {1})
javax.faces.converter.DoubleConverter.DOUBLE	{2}: "{0}" must be a number consisting of one or more digits ({2}: "{0}" должен быть числом, состоящим из одной или более цифры)
javax.faces.converter.DoubleConverter.DOUBLE_detail	{2}: "{0}" must be a number between 4.9E-324 and 1.7976931348623157E308. Example: {1} ({2}: "{0}" должен быть числом от 4.9E-324 до 1.7976931348623157E308. Пример: {1})
javax.faces.converter.BooleanConverter.BOOLEAN_detail	{1}: "{0}" must be 'true' or 'false' ({1}: ("{0}" должен быть равен true или false). Любое значение, отличное от true, рассматривается как false.
Javax.faces.converter.BigDecimalConverter.BIGINTEGER_detail	"{0}" must be a number consisting of one or more digits. Example: {1} ("{0}" должен быть числом, состоящим из одной или более цифры. Пример: {1})
javax.faces.converter.BigDecimalConverter.BIGDECIMAL_detail	"{0}" must be a signed decimal number consisting of zero or more digits, that may be followed by a decimal point and fraction. Example: {1} ("{0}" должен быть десятичным числом со знаком, содержащим от нуля и более цифр, за которыми могут следовать десятичная точка и дробная часть. Пример: {1})
javax.faces.converter.NumberConverter.NUMBER_detail	{2}: "{0}" is not a number. Example: {1} ({2}: "{0}" — не число. Пример: {1})
javax.faces.converter.NumberConverter.CURRENCY_detail	{2}: "{0}" could not be understood as a currency value. Example: {1} ({2}: "{0}" не может интерпретироваться как значение валюты. Пример: {1})
javax.faces.converter.NumberConverter.PERCENT_detail	{2}: "{0}" could not be understood as a percentage. Example: {1} ({2}: "{0}" не может интерпретироваться как процентная доля. Пример: {1})
javax.faces.converter.DateTimeConverter.DATE_detail	{2}: "{0}" could not be understood as a date. Example: {1} ({2}: "{0}" не может интерпретироваться как дата. Пример: {1})
javax.faces.converter.DateTimeConverter.TIME_detail	{2}: "{0}" could not be understood as a time. Example: {1} ({2}: "{0}" не может интерпретироваться как время). Пример: {1}
javax.faces.converter.DateTimeConverter.PATTERN_TYPE	{1}: A 'pattern' or 'type' attribute must be specified to convert the value "{0}" (Должен быть указан атрибут pattern или type для преобразования значения "{0}").
javax.faces.converter.EnumConverter.ENUM	{2}: "{0}" must be convertible to an enum ({2}: "{0}" должен быть преобразуемым в перечисление).
javax.faces.converter.EnumConverter.ENUM_detail	{2}: "{0}" must be convertible to an enum from the enum that contains the constant "{1}" ({2}: "{0}" должен быть преобразуемым в перечисление из перечисления, которое содержит константу "{1}").
javax.faces.converter.EnumConverter.ENUM_NO_CLASS_detail	{1}: "{0}" must be convertible to an enum from the enum, but no enum class provided ({1}: "{0}" должен быть преобразуемым в перечисление из перечисления, но класс перечисления не предоставлен)

На заметку! В версии JSF 1.1 универсальное сообщение "Conversion error occurred" (Произошла ошибка преобразования) имеет ключ javax.faces.component.UIInput.Conversion.

Использование специализированного сообщения об ошибке

Начиная с версии JSF 1.2 предусмотрена возможность предоставлять пользовательские сообщения об ошибках преобразователя для компонента. Задайте атрибут converterMessage для компонента, значение которого преобразовывается. Например:

```
<h:inputText ... converterMessage="Not a valid number."/>
```

Внимание! В отличие от строк сообщений, приведенных в предыдущем разделе, эти атрибуты сообщения воспринимаются буквально. Метки-заполнители, такие как {0}, не заменяются.

Полный пример преобразователя

Теперь мы можем перейти к рассмотрению первого законченного примера. На рис. 7.5 показана структура каталогов приложения. Это веб-приложение обращается к пользователю, чтобы он предоставил информацию об оплате (листинг 7.1), а затем отображает отформатированные сведения на экране подтверждения (листинг 7.2). Эти сообщения приведены в листинге 7.3, а класс бина показан в листинге 7.4.

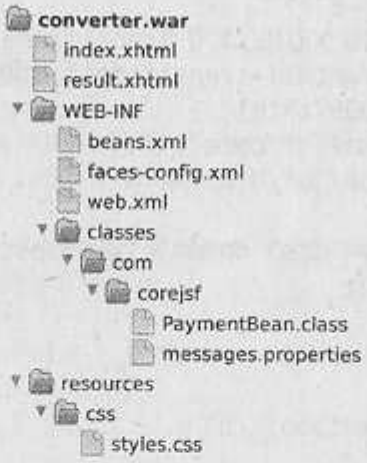


Рис. 7.5. Структура каталогов для образца преобразователя

Листинг 7.1. Файл converter/web/index.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:f="http://java.sun.com/jsf/core"
6.       xmlns:h="http://java.sun.com/jsf/html">
7.    <h:head>
8.       <h:outputStylesheet library="css" name="styles.css"/>
9.       <title>#{msgs.title}</title>
10.   </h:head>
11.   <h:body>
12.      <h:form>
13.         <h1>#{msgs.enterPayment}</h1>
14.         <h:panelGrid columns="3">
```

```
            #{msgs.amount}
            <h:inputText id="amount" label="#{msgs.amount}"
                         value="#{payment.amount}">
                <f:convertNumber minFractionDigits="2"/>
            </h:inputText>
            <h:message for="amount" styleClass="errorMessage"/>

            #{msgs.creditCard}
            <h:inputText id="card" label="#{msgs.creditCard}"
                         value="#{payment.card}"/>
            <h:message for="card" styleClass="errorMessage" />

            #{msgs.expirationDate}
            <h:inputText id="date" label="#{msgs.expirationDate}"
                         value="#{payment.date}">
              <f:convertDateTime pattern="MM/yyyy"/>
            </h:inputText>
            <h:message for="date" styleClass="errorMessage"/>
        </h:panelGrid>
        <h:commandButton value="#{msgs.process}" action="result"/>
    </h:form>
  </h:body>
</html>
```

Листинг 7.2. Файл converter/web/result.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <h:outputStylesheet library="css" name="styles.css"/>
    <title>#{msgs.title}</title>
  </h:head>
  <h:body>
    <h:form>
      <h1>#{msgs.paymentInformation}</h1>
      <h:panelGrid columns="2">
        #{msgs.amount}
        <h:outputText value="#{payment.amount}">
            <f:convertNumber type="currency"/>
        </h:outputText>

        #{msgs.creditCard}
        <h:outputText value="#{payment.card}"/>

        #{msgs.expirationDate}
        <h:outputText value="#{payment.date}">
            <f:convertDateTime pattern="MM/yyyy"/>
        </h:outputText>
      </h:panelGrid>
      <h:commandButton value="#{msgs.back}" action="index"/>
    </h:form>
  </h:body>
</html>
```

Листинг 7.3. Файл converter/src/java/com/corejsf/messages.properties

```
title=An Application to Test Data Conversion
enterPayment=Please enter the payment information
amount=Amount
creditCard=Credit Card
expirationDate=Expiration date (Month/Year)
process=Process
back=Back
paymentInformation=Payment information
```

Листинг 7.4. Файл converter/src/java/com/corejsf/PaymentBean.java

```
package com.corejsf;

import java.io.Serializable;
import java.util.Date;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;

@Named("payment") // или @ManagedBean(name="payment")
@SessionScoped
public class PaymentBean implements Serializable {
   private double amount;
   private String card = "";
   private Date date = new Date();

   public void setAmount(double newValue) { amount = newValue; }
   public double getAmount() { return amount; }

   public void setCard(String newValue) { card = newValue; }
   public String getCard() { return card; }

   public void setDate(Date newValue) { date = newValue; }
   public Date getDate() { return date; }
}
```

Использование стандартных средств проверки

В следующих разделах будут рассматриваться стандартные средства проверки JSF. Теперь мы можем также реализовать пользовательские средства проверки (подробнее об этом — ниже, в разделе "Реализация классов пользовательского средства проверки").

Проверка длин строк и числовых диапазонов

Задача использования средств проверки JSF на страницах JSF является несложной: достаточно добавить теги средств проверки к тексту тега компонента:

```
<h:inputText id="card" value="#{payment.card}">
   <f:validateLength minimum="13"/>
</h:inputText>
```

В предыдущем фрагменте кода показано добавление средства проверки к текстовому полю; при передаче формы с этим текстовым полем это средство проверки позволяет удостовериться, что строка, содержащаяся в поле, включает по меньшей мере

13 символов. Если проверка правильности оканчивается неудачей (в данном случае это происходит, если строка имеет длину 12 или меньше символов), то средства проверки вырабатывают сообщения об ошибках, относящиеся к компоненту, который вызвал нарушение в работе. Затем эти сообщения могут быть отображены на странице JSF с помощью тега `h:message` или `h:messages`.

В технологии JavaServer Faces предусмотрены встроенные механизмы, позволяющие выполнять проверки правильности.

- Проверка длины строки.
- Проверка ограничений для числового значения (например, находится ли число в пределах > 0 и <= 100).
- Проверка с помощью регулярного выражения (начиная с версии JSF 2.0).
- Проверка того, задано ли значение.

В табл. 7.4 перечислены стандартные средства проверки, которые предусмотрены в технологии JSF. Пример применения средства проверки длины строки был приведен в предыдущем разделе. Чтобы проверить правильность числового ввода, можно использовать средство проверки диапазона. Например:

```
<h:inputText id="amount" value="#{payment.amount}">
   <f:validateLongRange minimum="10" maximum="10000"/>
</h:inputText>
```

Средство проверки позволяет убедиться в том, что предоставленное значение находится в пределах >= 10 и <= 10000.

Все стандартные теги средств проверки диапазона имеют атрибуты `maximum` и `minimum`. При их использовании необходимо задать один или оба эти атрибута.

Таблица 7.4. Стандартные средства проверки

Тег JSP	Класс средства проверки	Атрибут[1]	Проверяет
`f:validateDoubleRange`	`DoubleRangeValidator`	`minimum`, `maximum`	Значение `double` в пределах необязательного диапазона
`f:validateLongRange`	`LongRangeValidator`	`minimum`, `maximum`	Значение `long` в пределах необязательного диапазона
`f:validateLength`	`LengthValidator`	`minimum`, `maximum`	Значение типа `String` с минимальным и максимальным числом символов
`f:validateRequired` JSF 2.0	`RequiredValidator`		Присутствие значения
`f:validateRegex` JSF 2.0	`RegexValidator`	`pattern`	Соответствие значения типа `String` регулярному выражению
`f:validateBean` JSF 2.0	`BeanValidator`	`validationGroups`	Определяет группы проверки правильности для средств проверки бина (см. спецификацию JSR 303 для ознакомления с подробными сведениями)

[1] Предусмотрена возможность отключить любое средство проверки, задав значение `true` логического атрибута `disabled`.

Проверка обязательных значений

Для проверки того, задано ли некоторое значение, можно вложить средство проверки в тег компонента ввода:

```
<h:inputText id="date" value="#{payment.date}">
   <f:validateRequired/>
</h:inputText>
```

Еще один вариант состоит в том, что можно просто использовать атрибут `required="true"` в компоненте ввода:

```
<h:inputText id="date" value="#{payment.date}" required="true"/>
```

Тег `f:validateRequired` был впервые введен в версии JSF 2.0. Он просто задает значение атрибута `required` компонента равным `true`.

Внимание! Если атрибут `required` не задан и пользователь оставляет незаполненным одно из полей ввода, то проверка правильности вообще не происходит! Вместо этого пустое поле ввода интерпретируется как запрос, согласно которому существующее значение должно остаться неизменным.

Начиная с версии JSF 2.0 можно изменить такое поведение путем задания значения контекстного параметра `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` равным `true` в файле `web.xml`.

Альтернативный синтаксис закрепления средства проверки за компонентом состоит в использовании тега `f:validator`. При этом идентификатор средства проверки и параметры средства проверки задаются таким образом:

```
<h:inputText id="card" value="#{payment.card}">
   <f:validator validatorId="javax.faces.validator.LengthValidator">
      <f:attribute name="minimum" value="13"/>
   </f:validator>
</h:inputText>
```

Еще одним способом определения средства проверки является применение атрибута `validator` к тегу компонента (подробнее об этом речь пойдет в разделе "Проверка с помощью методов бина").

На заметку! Начиная с версии JSF 1.2 теги `f:validateLength`, `f:validateLongRange`, `f:validateDoubleRange` и `f:validator` имеют необязательный атрибут `binding`. Это позволяет связать экземпляр средства проверки со свойством вспомогательного бина типа `javax.faces.validator.Validator`.

Отображение ошибок проверки правильности

Ошибки проверки правильности обрабатываются так же, как и ошибки преобразования. К компоненту, проверка правильности которого окончилась неудачей, добавляется сообщение, и текущая страница повторно отображается непосредственно после завершения этапа проверки правильности процесса.

Для отображения ошибок проверки правильности используется тег `h:message` или `h:messages`. (Подробные сведения можно получить в разделе "Отображение сообщений об ошибках".)

Начиная с версии JSF 1.2 предусмотрена возможность задавать пользовательское сообщение для компонента, устанавливая значение атрибута `requiredMessage` или `validatorMessage`, как в следующем примере:

```
<h:inputText id="card" value="#{payment.card}" required="true"
      requiredMessage="#{msgs.cardRequired}"
      validatorMessage="#{msgs.cardInvalid}">
   <f:validateLength minimum="13"/>
</h:inputText>
```

Можно также переопределить сообщения средства проверки по умолчанию, приведенные в табл. 7.5, глобально. Определите связку сообщений для своего приложения и предоставьте сообщения с соответствующими ключами, как было показано в разделе "Изменение текста стандартных сообщений об ошибках".

На заметку! В версии JSF 1.1 метка компонента ввода не была включена в сообщения проверки правильности. Ключом для сообщения "`not in range` (не находится в диапазоне)" было `javax.faces.validator.NOT_IN_RANGE`.

Внимание! Чаще всего стандартное сообщение, относящееся к средству проверки `LengthValidator`, вызывает у пользователей недоумение. Например, если для почтового индекса задана минимальная длина 5, а пользователь вводит `"9410"`, то сообщение об ошибке выглядит следующим образом: "`Value is less than allowable, minimum of 5` (Значение короче допустимого; минимальная длина — 5)".

Таблица 7.5. Стандартные сообщения об ошибках проверки правильности

Идентификатор ресурса	Текст, применяемый по умолчанию	Источник или причина сообщения
`javax.faces.component.UIInput.REQUIRED`	`{0}: Validation error: Value is required` ({0}. Ошибка проверки правильности. Значение является обязательным.)	Значение `UIInput` с атрибутом `required`, если значение отсутствует
`javax.faces.validator.DoubleRangeValidator.NOT_IN_RANGE`, `javax.faces.validator.LongRangeValidator.NOT_IN_RANGE`	`{2}: Validation error: Specified attribute is not between the expected values of {0} and {1}` ({2}. Ошибка проверки правильности. Заданный атрибут не находится в пределах между ожидаемыми значениями {0} и {1})	Заданы атрибуты `DoubleRangeValidator` и `LongRangeValidator`, притом что значение находится вне диапазона и указаны минимальное и максимальное значения
`javax.faces.validator.DoubleRangeValidator.MAXIMUM`, `javax.faces.validator.LongRangeValidator.MAXIMUM`	`{1}: Validation error: Value is greater than allowable maximum of "{0}"` ({1}. Ошибка проверки правильности. Значение больше допустимого максимума, равного "{0}")	Заданы атрибуты `DoubleRangeValidator` или `LongRangeValidator`, притом что значение находится вне диапазона и указано только максимальное значение
`javax.faces.validator.DoubleRangeValidator.MINIMUM`, `javax.faces.validator.LongRangeValidator.MINIMUM`	`{1}: Validation error: Value is less than allowable minimum of "{0}"` ({1}. Ошибка проверки правильности. Значение меньше допустимого минимума, равного "{0}")	Заданы атрибуты `DoubleRangeValidator` или `LongRangeValidator`, притом что значение находится вне диапазона и указано только минимальное значение
`javax.faces.validator.DoubleRangeValidator.TYPE`, `javax.faces.validator.LongRangeValidator.TYPE`	`{0}: Validation error: Value is not of the correct type` ({0}. Ошибка проверки правильности. Значение не имеет правильного типа.)	Заданы атрибуты `DoubleRangeValidator` или `LongRangeValidator`, притом что значение не может быть преобразовано к типу `double` или `long`

Окончание табл. 7.5

Идентификатор ресурса	Текст, применяемый по умолчанию	Источник или причина сообщения
javax.faces.validator.LengthValidator.MAXIMUM	{1}: Validation error: Value is greater than allowable maximum of "{0}" ({1}. Ошибка проверки правильности. Значение больше допустимого максимума, равного "{0}".)	Задан атрибут LengthValidator, притом что длина строки больше максимальной
javax.faces.validator.LengthValidator.MINIMUM	{1}: Validation error: Value is less than allowable minimum of "{0}" ({1}. Ошибка проверки правильности. Значение меньше допустимого минимума, равного "{0}".)	Задан атрибут LengthValidator, притом что длина строки меньше минимальной
javax.faces.validator.BeanValidator.MESSAGE	{0}	Средство проверки, относящееся к платформе проверки правильности бинов. См. спецификацию JSR 303, чтобы ознакомиться с подробными сведениями о настройке сообщений

Исключение процедуры проверки правильности

Как явствует из предыдущих примеров, ошибки проверки правильности (как и ошибки преобразования) вынуждают приложение повторно отображать текущую страницу. В сочетании с некоторыми действиями по определению навигации такое поведение может стать причиной нарушений в работе. Предположим, например, что на странице, содержащей обязательные поля, присутствует кнопка Cancel (Отмена). Если пользователь щелкнет на кнопке Cancel, оставив обязательное поле незаполненным, то сработает механизм проверки правильности и принудительно обеспечит повторное отображение текущей страницы.

Вряд ли можно рассчитывать на то, что пользователи заполнят обязательные поля и только после этого воспользуются предоставленной им возможностью отменить свой ввод. К счастью, предусмотрен механизм исключения проверки. Если для команды задан атрибут immediate, то команда выполняется во время этапа применения значений запроса.

Поэтому кнопка Cancel должна быть реализована примерно так:

```
<h:commandButton value="Cancel" action="cancel" immediate="true"/>
```

Полный пример проверки правильности

В следующем примере приложения показана форма, в которой используются все стандартные проверки правильности JSF: обязательные поля, длина строки и числовые ограничения. Это приложение позволяет удостовериться, что во все поля введены значения, сумма находится в пределах от $10 до $10000, номер кредитной карточки имеет длину по меньшей мере 13 символов, а также задана дата истечения срока действия кредитной карточки. На рис. 7.6 показаны типичные сообщения об ошибках проверки правильности. Предусмотрена также кнопка Cancel, позволяющая продемонстрировать, как можно исключить проверку правильности.

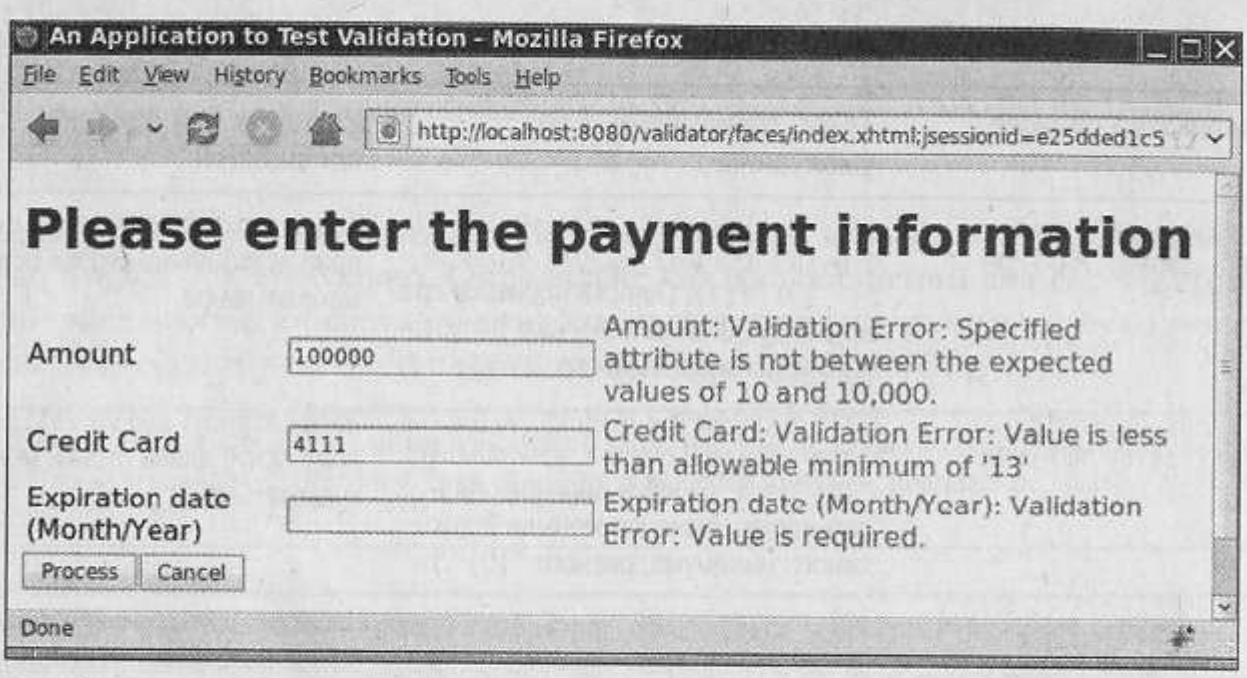


Рис. 7.6. Типичные сообщения об ошибках проверки правильности

На рис. 7.7 показана структура каталогов приложения. В листингах 7.5-7.6 содержатся страница JSF со средствами проверки и страница, отображаемая при отмене запроса. (Необходимо учитывать, что проверка правильности не выполняется, если пользователь щелкает на кнопке Cancel.)

Рис. 7.7. Структура каталогов для примера проверки правильности

Листинг 7.5. Файл `validator/web/index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:f="http://java.sun.com/jsf/core"
6.       xmlns:h="http://java.sun.com/jsf/html">
7.    <h:head>
8.       <h:outputStylesheet library="css" name="styles.css"/>
9.       <title>#{msgs.title}</title>
10.   </h:head>
11.   <h:body>
12.      <h:form>
13.         <h1>#{msgs.enterPayment}</h1>
```

```
14.          <h:panelGrid columns="3">
15.             #{msgs.amount}
16.             <h:inputText id="amount" label="#{msgs.amount}"
17.                          value="#{payment.amount}" required="true">
18.                <f:convertNumber minFractionDigits="2"/>
19.                <f:validateDoubleRange minimum="10" maximum="10000"/>
20.             </h:inputText>
21.             <h:message for="amount" styleClass="errorMessage"/>
22.
23.             #{msgs.creditCard}
24.             <h:inputText id="card" label="#{msgs.creditCard}"
25.                          value="#{payment.card}" required="true"
26.                           requiredMessage="#{msgs.cardRequired}">
27.                <f:validateLength minimum="13"/>
28.             </h:inputText>
29.             <h:message for="card" styleClass="errorMessage"/>
30.
31.             #{msgs.expirationDate}
32.             <h:inputText id="date" label="#{msgs.expirationDate}"
33.                           value="#{payment.date}" required="true">
34.                <f:convertDateTime pattern="MM/yyyy"/>
35.             </h:inputText>
36.             <h:message for="date" styleClass="errorMessage"/>
37.          </h:panelGrid>
38.          <h:commandButton value="#{msgs.process}" action="result"/>
39.          <h:commandButton value="#{msgs.cancel}" action="canceled"
40.                           immediate="true"/>
41.       </h:form>
42.    </h:body>
43. </html>
```

Листинг 7.6. Файл validator/web/canceled.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:f="http://java.sun.com/jsf/core"
6.       xmlns:h="http://java.sun.com/jsf/html">
7.    <h:head>
8.       <h:outputStylesheet library="css" name="styles.css"/>
9.       <title>#{msgs.title}</title>
10.   </h:head>
11.   <h:body>
12.      <h:form>
13.         #{msgs.canceled}
14.         <br/>
15.         <h:commandButton value="#{msgs.back}" action="index"/>
16.      </h:form>
17.   </h:body>
18. </html>
```

Проверка правильности бина JSF 2.0

В версии JSF 2.0 предусмотрена интеграция с платформой проверки правильности бинов (JSR 303), которая представляет собой общую платформу для определения ограничений проверки правильности. Средства проверки правильности закрепляются за полями или методами получения свойств класса Java, как в следующем примере:

```
public class PaymentBean {
   @Size(min=13) private String card;
   @Future public Date getDate() { ... }
   ...
}
```

Доступные аннотации показаны в табл. 7.6.

Таблица 7.6. Аннотации, применяемые на платформе проверки правильности бинов

Аннотация	Атрибут[1]	Назначение
`@Null`, `@NotNull`	Отсутствует	Проверка того, является ли значение пустым (`null`) или не пустым
`@Min`, `@Max`	Граница, заданная с помощью типа данных `long`	Проверка того, соответствует ли значение, самое меньшее или самое большее, заданной границе. Тип должен представлять собой один из типов `int`, `long`, `short`, `byte`, либо принадлежать к одной из оболочек `BigInteger` или `BigDecimal`. Примечание. Типы данных `double` и `float` не поддерживаются из-за потенциальных ошибок округления
`@DecimalMin`, `@DecimalMax`	Граница, заданная с помощью типа данных `String`, как указано выше	Может также применяться к данным типа `String`
`@Digits`	`integer`, `fraction`	Проверка того, имеет ли самое большее значение указанное количество цифр в целой или дробной части. Применяется к типам данных `int`, `long`, `short`, `byte` и их оболочкам `BigInteger`, `BigDecimal`, `String`
`@AssertTrue`, `@AssertFalse`	Отсутствует	Проверка того, равно ли логическое значение `true` или `false`
`@Past`, `@Future`	Отсутствует	Проверка того, находится ли дата в прошлом или будущем
`@Size`	`min`, `max`	Проверка того, находится ли размер строки, массива, коллекции или карты, самое меньшее или самое большее, в пределах указанной границы
`@Pattern`	`regexp`, `flags`	Регулярное выражение и необязательные флаги компиляции

[1] Все аннотации проверки правильности имеют атрибуты `message` и `groups`. В настоящей главе группы проверки правильности не рассматриваются.

Платформа проверки правильности бинов имеет существенные преимущества перед средствами проверки правильности на уровне страницы. Предположим, что обновление бина, применяемого в веб-приложении, происходит на нескольких страницах. Если применяется указанная платформа, то отпадает необходимость добавлять правила проверки к каждой странице, а также можно быть уверенным, что процедуры проверки правильности всегда будут осуществляться единым образом.

Чтобы переопределить сообщения, предусмотренные по умолчанию, необходимо включить файл `ValidationMessages.properties` в применяемый по умолчанию (корневой) пакет конкретного приложения. Стандартные сообщения можно переопределить, например, таким образом:

```
javax.validation.constraints.Min.message=Must be at least {value}
```

что является более доступным для восприятия по сравнению с применяемым по умолчанию текстом: "`Must be greater than or equal to ...`" (Должно быть больше или

кретной проверке правильности, необходимо сослаться на ключ связки в атрибуте сообщения:

```
@Size(min=13, message="{com.corejsf.creditCardLength}")
private String card = "";
```

Затем определите этот ключ в файле ValidationMessages.properties:

```
com.corejsf.creditCardLength=The credit card number must have at least 13 digits
```

На заметку! Если используется сервер приложений, совместимый с Java EE 6, то автоматически предоставляется доступ к реализации JSR 303. В противном случае поместите JAR-файлы средства проверки Hibernate Validator (http://validator.hibernate.org/) в каталог WEB-INF/lib.

Мы предоставляем пример приложения, которое демонстрирует проверку правильности бина (рис. 7.8). Замечательной особенностью этого примера является его простота. На страницах JSF не предусмотрены специальные средства проверки правильности. Класс модели, показанный в листинге 7.7, содержит аннотации проверки правильности.

Заслуживает внимания аннотация @LuhnCheck, которая проверяет цифры кредитной карточки с использованием формулы Луна (Luhn).

На заметку! Формула Луна, разработанная группой математиков в конце 1960-х годов, применяется для проверки и выработки номеров кредитных карточек, а также номеров карточек социального страхования для канадского правительства. Эта формула позволяет обнаружить, не введена ли какая-то цифра неправильно и не произошла ли перестановка местами двух цифр. (Посетите веб-сайт http://www.merriampark.com/anatomycc.htm, чтобы получить дополнительную информацию о формуле Луна.) Один из удобных способов отладки состоит в определении того, проходит ли номер 4111 1111 1111 1111 проверку по формуле Луна.

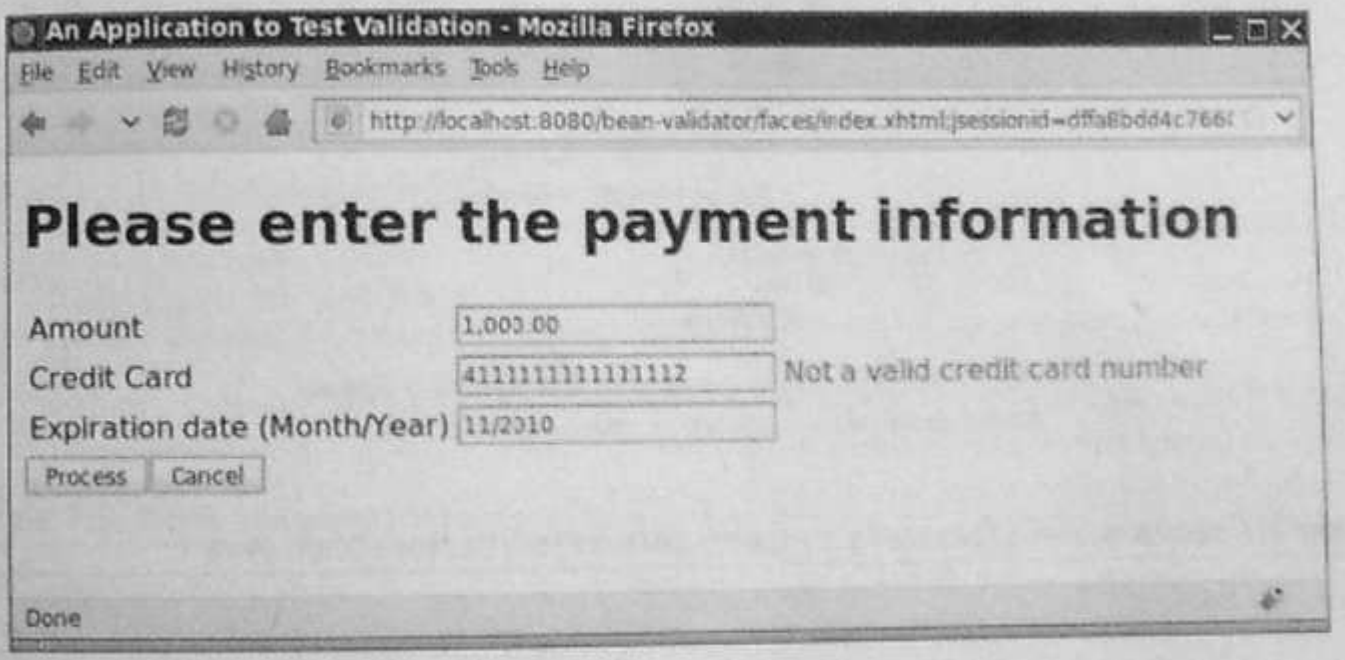


Рис. 7.8. Пример неудачной проверки по формуле Луна

Аннотация @LuhnCheck представляет собой пользовательское средство проверки, которое мы написали для этой книги, чтобы продемонстрировать расширяемость платформы проверки правильности бинов. Для написания пользовательского средства проверки необходимо предусмотреть следующее.

- Аннотация (листинг 7.8).
- Класс, который реализует интерфейс ConstraintValidator (листинг 7.9).

- По желанию, применяемые по умолчанию сообщения в файле ValidationMessages.properties (листинг 7.10).

Заслуживает внимания наличие циклической зависимости между аннотацией и классом средства проверки. Аннотация ссылается на класс средства проверки:

```
@Constraint(validatedBy=LuhnCheckValidator.class)
public @interface LuhnCheck
```

Класс средства проверки ссылается на тип аннотации:

```
public class LuhnCheckValidator implements ConstraintValidator<LuhnCheck, String>
```

Второй параметр типа интерфейса ConstraintValidator — это тип проверяемого объекта; в данном случае таковым является String. Поэтому метод isValid имеет параметр String:

```
public boolean isValid(String value, ConstraintValidatorContext context) {
   return luhnCheck(value.replaceAll("\\D", "")); // удаление знаков, отличных от цифр
}
```

В настоящем разделе не приводятся подробные сведения о платформе проверки правильности бинов. Читатель, желающий реализовать собственные пользовательские правила проверки, может просто следовать этому примеру.

На рис. 7.9 показана структура каталогов рассматриваемого приложения.

Рис. 7.9. Структура каталогов для примера применения платформы проверки правильности бинов

Листинг 7.7. Файл bean-validator/src/java/com/corejsf/PaymentBean.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.util.Date;
5.
6. import javax.inject.Named;
7.    // или import javax.faces.bean.ManagedBean;
8. import javax.enterprise.context.SessionScoped;
9.    // или import javax.faces.bean.SessionScoped;
10. import javax.validation.constraints.Future;
11. import javax.validation.constraints.Max;
12. import javax.validation.constraints.Min;
```

```
13. import javax.validation.constraints.Size;
14.
15. @Named("payment") // или @ManagedBean(name="payment")
16. @SessionScoped
17. public class PaymentBean implements Serializable {
18.    @Min(10) @Max(10000)
19.    private double amount;
20.    @Size(min=13,message="{com.corejsf.creditCardLength}") @LuhnCheck
21.     private String card = "";
22.    @Future
23.     private Date date = new Date();
24.
25.     public void setAmount(double newValue) { amount = newValue; }
26.     public double getAmount() { return amount; }
27.
28.     public void setCard(String newValue) { card = newValue; }
29.     public String getCard() { return card; }
30.
31.     public void setDate(Date newValue) { date = newValue; }
32.     public Date getDate() { return date; }
33. }
```

Листинг 7.8. Файл bean-validator/src/java/com/corejsf/LuhnCheck.java

```
1. package com.corejsf;
2.
3. import java.lang.annotation.Documented;
4. import java.lang.annotation.Retention;
5. import java.lang.annotation.Target;
6. import javax.validation.Constraint;
7. import javax.validation.Payload;
8. import static java.lang.annotation.ElementType.*;
9. import static java.lang.annotation.RetentionPolicy.*;
10.
11. @Target({METHOD, FIELD})
12. @Retention(RUNTIME)
13. @Documented
14. @Constraint(validatedBy=LuhnCheckValidator.class)
15. public @interface LuhnCheck {
16.    String message() default "{com.corejsf.LuhnCheck.message}";
17.    Class[] groups() default {};
18.    Class<? extends Payload>[] payload() default {};
19. }
```

Листинг 7.9. Файл bean-validator/src/java/com/corejsf/LuhnCheckValidator.java

```
1. package com.corejsf;
2. import javax.validation.ConstraintValidator;
3. import javax.validation.ConstraintValidatorContext;
4.
5. public class LuhnCheckValidator implements ConstraintValidator<LuhnCheck, String> {
6.    public void initialize(LuhnCheck constraintAnnotation) {
7.    }
8.
9.     public boolean isValid(String value, ConstraintValidatorContext context) {
10.       return luhnCheck(value.replaceAll("\\D", "")); // Удаление нецифровых знаков
11.    }
12.
13.    private static boolean luhnCheck(String cardNumber) {
```

```
14.         int sum = 0;
15.
16.          for(int i = cardNumber.length() - 1; i >= 0; i -= 2) {
17.             sum += Integer.parseInt(cardNumber.substring(i, i + 1));
18.             if(i > 0) {
19.                int d = 2 * Integer.parseInt(cardNumber.substring(i - 1, i));
20.                if(d > 9) d -= 9;
21.                sum += d;
22.             }
23.          }
24.
25.          return sum % 10 == 0;
26.       }
27. }
```

Листинг 7.10. Файл bean-validator/src/java/ValidationMessages.properties

```
1. javax.validation.constraints.Min.message=Must be at least {value}
2. com.corejsf.creditCardLength=The credit card number must have at least 13 digits
3. com.corejsf.LuhnCheck.message=Not a valid credit card number
```

Программирование с применением пользовательских преобразователей и средств проверки

Стандартные преобразователи и средства проверки JSF охватывают целый ряд базовых типов, но для многих веб-приложений этого недостаточно. Например, может потребоваться провести преобразование в типы, отличные от чисел и дат, или выполнить проверку правильности, характерную для конкретного приложения, такую как проверка номера кредитной карточки.

В следующих разделах будет показано, как реализовать специализированные для приложения преобразователи и средства проверки. Осуществление этих реализаций требует определенного объема работы по программированию.

Реализация пользовательских классов преобразователей

Преобразователь — это класс, который осуществляет преобразования между строками и объектами. Преобразователь должен реализовывать интерфейс `Converter`, который имеет два метода:

```
Object getAsObject(FacesContext context, UIComponent component, String newValue)
String getAsString(FacesContext context, UIComponent component, Object value)
```

Первый метод преобразовывает строку в объект требуемого типа, активизируя исключение `ConverterException`, если преобразование не может быть выполнено. Этот метод вызывается при передаче строки от клиента, как правило, в текстовом поле. Второй метод преобразовывает объект в строковое представление, которое должно быть отображено в клиентском интерфейсе.

Чтобы проиллюстрировать эти методы, разработаем пользовательский преобразователь для номеров кредитных карточек. Этот преобразователь должен предоставлять пользователям возможность ввести номер кредитной карточки с пробелами или

без пробелов. Таким образом, допустимыми являются входные данные в следующих форматах:

```
1234567890123456
1234 5678 9012 3456
```

Код пользовательского преобразователя показан в листинге 7.11. Метод `getAsObject` преобразователя удаляет все символы, не являющиеся цифровыми. Затем этот метод создает объект типа `CreditCard`. При обнаружении ошибки формируется объект `FacesMessage` и активизируется исключение `ConverterException`. Эти шаги будут обсуждаться в следующем разделе, "Сообщения об ошибках преобразования", на стр. 248.

В методе `getAsString` рассматриваемого преобразователя выполняются действия по форматированию номера кредитной карточки в виде, удобном для восприятия пользователем. Цифры разделяются на привычные группы с учетом типа кредитной карточки. В табл. 7.7 приведены наиболее распространенные форматы номеров кредитных карточек.

Таблица 7.7. Форматы номеров кредитных карточек

Тип карточки	Цифры	Формат
MasterCard	16	5xxx xxxx xxxx xxxx
Visa	16	4xxx xxxx xxxx xxxx
Visa	13	4xxx xxx xxx xxx
Discover	16	6xxx xxxx xxxx xxxx
American Express	15	37xx xxxxxx xxxxx
American Express	22	3xxxxx xxxxxxxx xxxxxxxx
Diners Club, Carte Blanche	14	3xxxx xxxx xxxxx

В этом примере класс `CreditCard` содержит предельно малый объем кода; в него включен лишь номер кредитной карточки (листинг 7.12). Мы могли бы сохранить номер кредитной карточки в виде объекта типа `String`, по существу заменив преобразователь средством форматирования. Но большинство преобразователей имеет целевой тип, отличный от `String`. Чтобы упростить для читателя повторное использование этого примера, было решено применить другой целевой тип.

Листинг 7.11. Файл `converter2/src/java/com/coresjf/CreditCardConverter.java`

```
1. package com.corejsf;
2.
3. import javax.faces.application.FacesMessage;
4. import javax.faces.component.UIComponent;
5. import javax.faces.context.FacesContext;
6. import javax.faces.convert.Converter;
7. import javax.faces.convert.ConverterException;
8. import javax.faces.convert.FacesConverter;
9.
10. @FacesConverter(forClass=CreditCard.class)
11. public class CreditCardConverter implements Converter {
12.    public Object getAsObject(FacesContext context, UIComponent component,
13.          String newValue) throws ConverterException {
14.       StringBuilder builder = new StringBuilder(newValue);
15.       boolean foundInvalidCharacter = false;
```

```
16.       char invalidCharacter = '\0';
17.       int i = 0;
18.       while (i < builder.length() && !foundInvalidCharacter) {
19.          char ch = builder.charAt(i);
20.          if (Character.isDigit(ch))
21.             i++;
22.          else if (Character.isWhitespace(ch))
23.             builder.deleteCharAt(i);
24.          else {
25.             foundInvalidCharacter = true;
26.             invalidCharacter = ch;
27.          }
28.       }
29.
30.       if (foundInvalidCharacter) {
31.          FacesMessage message = com.corejsf.util.Messages.getMessage(
32.                "com.corejsf.messages", "badCreditCardCharacter",
33.                new Object[]{ new Character(invalidCharacter) });
34.          message.setSeverity(FacesMessage.SEVERITY_ERROR);
35.          throw new ConverterException(message);
36.       }
37.
38.       return new CreditCard(builder.toString());
39.    }
40.
41.    public String getAsString(FacesContext context, UIComponent component,
42.          Object value) throws ConverterException {
43.       // длина 13: xxxx xxx xxx xxx
44.       // длина 14: xxxxx xxxx xxxxx
45.       // длина 15: xxxx xxxxxx xxxxx
46.       // длина 16: xxxx xxxx xxxx xxxx
47.       // длина 22: xxxxxx xxxxxxxx xxxxxxxx
48.       String v = value.toString();
49.       int[] boundaries = null;
50.       int length = v.length();
51.       if (length == 13)
52.          boundaries = new int[]{ 4, 7, 10 };
53.       else if (length == 14)
54.          boundaries = new int[]{ 5, 9 };
55.       else if (length == 15)
56.          boundaries = new int[]{ 4, 10 };
57.       else if (length == 16)
58.          boundaries = new int[]{ 4, 8, 12 };
59.       else if (length == 22)
60.          boundaries = new int[]{ 6, 14 };
61.       else
62.          return v;
63.       StringBuilder result = new StringBuilder();
64.       int start = 0;
65.       for (int i = 0; i < boundaries.length; i++) {
66.          int end = boundaries[i];
67.          result.append(v.substring(start, end));
68.          result.append(" ");
69.          start = end;
70.       }
71.       result.append(v.substring(start));
72.       return result.toString();
73.    }
74. }
```

Листинг 7.12. Файл converter2/src/java/com/corejsf/CreditCard.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. public class CreditCard implements Serializable {
6.    private String number;
7.
8.    public CreditCard(String number) { this.number = number; }
9.    public String toString() { return number; }
10. }
```

Определение преобразователей JSF 2.0

Один из механизмов определения преобразователей предусматривает задание символического идентификатора, регистрируемого для приложения JSF. Для рассматриваемого преобразователя номеров кредитных карточек будет использоваться идентификатор com.corejsf.Card.

Идентификатор можно связать с преобразователем одним из двух способов. В версии JSF 2.0 появилась возможность использовать аннотацию @FacesConverter:

```
@FacesConverter("com.corejsf.Card")
public class CreditCardConverter implements Converter
```

До выхода версии JSF 2.0 приходилось вводить в файл faces-config.xml запись, которая связывает идентификатор преобразователя с классом, реализующим преобразователь:

```
<converter>
   <converter-id>com.corejsf.Card</converter-id>
   <converter-class>com.corejsf.CreditCardConverter</converter-class>
</converter>
```

В следующих примерах предполагается, что свойство card бина PaymentBean имеет тип CreditCard, как показано в листинге 7.18 на стр. 255. После этого можно воспользоваться тегом f:converter и определить идентификатор преобразователя:

```
<h:inputText value="#{payment.card}">
   <f:converter converterId="com.corejsf.Card"/>
</h:inputText>
```

Или, что более кратко, можно применить атрибут converter:

```
<h:inputText value="#{payment.card}" converter="com.corejsf.Card"/>
```

В ином случае, если можно быть уверенным в том, что применяемый преобразователь приемлем для всех преобразований между данными типа String и объектами CreditCard, то есть возможность зарегистрировать его как преобразователь по умолчанию для класса CreditCard.

Для этого можно использовать аннотацию

```
@FacesConverter(forClass=CreditCard.class)
```

или ввести запись faces-config:

```
<converter>
   <converter-for-class>com.corejsf.CreditCard</converter-for-class>
   <converter-class>com.corejsf.CreditCardConverter</converter-class>
</converter>
```

После этого уже не требуются вообще какие-либо явные упоминания о наличии преобразователя. Он автоматически используется каждый раз, когда встречается ссылка на значение типа CreditCard. В качестве примера рассмотрим следующий тег:

```
<h:inputText value="#{payment.card}"/>
```

Реализация JSF, приступая к преобразованию значения запроса, обнаруживает, что целевым типом является CreditCard, и определяет местонахождение преобразователя для этого класса. Для автора страницы невозможно предложить более удобный преобразователь!

Внимание! Если в аннотации FacesConverter определены оба атрибута, value и forClass, то последний игнорируется.

javax.faces.convert.Converter

- Object getAsObject(FacesContext context, UIComponent component, String value)
  Преобразовывает заданное строковое значение в объект, подходящий для хранения указанного компонента.
- String getAsString(FacesContext context, UIComponent component, Object value)
  Преобразовывает заданный объект, который хранится в указанном компоненте, в строковое представление.

@javax.faces.convert.FacesConverter

- String value (Default: "")
  Идентификатор преобразователя.
- Class forClass (Default: Object.class)
  Класс, для которого предусмотрен определенный преобразователь.

Сообщения об ошибках преобразования

Преобразователь, обнаруживая ошибку, должен активизировать исключение ConverterException. Например, метод getAsObject рассматриваемого преобразователя номеров кредитных карточек проверяет, содержит ли номер кредитной карточки иные символы, кроме цифр и разделителей. При обнаружении недопустимых симво-лов этот метод сообщает об ошибке:

```
if (foundInvalidCharacter) {
   FacesMessage message = new FacesMessage(
      "Conversion error occurred. ", "Invalid card number. ");
   message.setSeverity(FacesMessage.SEVERITY_ERROR);
   throw new ConverterException(message);
```

Объект FacesMessage содержит сообщения в виде резюме и сообщения с подроб-ными сведениями, которые могут быть отображены с помощью тегов сообщений.

javax.faces.application.FacesMessage

- FacesMessage(FacesMessage.Severity severity, String summary, String detail)
  Создает сообщение с указанными степенью серьезности, резюме и подробными сведениями. Степень серьезности — это одна из констант SEVERITY_ERROR, SEVERITY_FATAL, SEVERITY_INFO или SEVERITY_WARN в классе FacesMessage.

- `FacesMessage(String summary, String detail)`

  Создает сообщение со степенью серьезности `SEVERITY_INFO`, а также с указанными резюме и подробными сведениями.

- `void setSeverity(FacesMessage.Severity severity)`

  Задает степень серьезности с указанным уровнем. Степень серьезности — это одна из констант `SEVERITY_ERROR`, `SEVERITY_FATAL`, `SEVERITY_INFO` или `SEVERITY_WARN` в классе `FacesMessage`.

`javax.faces.convert.ConverterException`

- `ConverterException(FacesMessage message)`
- `ConverterException(FacesMessage message, Throwable cause)`

  Эти конструкторы создают исключительные ситуации, метод `getMessage` которых возвращает резюме указанного сообщения, а метод `getFacesMessage` возвращает это сообщение.

- `ConverterException()`
- `ConverterException(String detailMessage)`
- `ConverterException(Throwable cause)`
- `ConverterException(String detailMessage, Throwable cause)`

  Эти конструкторы создают исключительные ситуации, метод `getMessage` которых возвращает указанное сообщение с подробными сведениями, а метод `getFacesMessage` возвращает значение `null`.

- `FacesMessage getFacesMessage()`

  Возвращает сообщение `FacesMessage`, с которым был создан этот объект исключительной ситуации, или возвращает значение `null`, если таковой не был задан.

Получение сообщений об ошибках из связок ресурсов

Безусловно, полноценная локализация невозможна без выборки сообщений об ошибках из связки сообщений.

Для обеспечения этого необходимо затратить определенные усилия, проводя работу с локалями и загрузчиками классов.

1. Получение текущей локали.

```
FacesContext context = FacesContext.getCurrentInstance();
UIViewRoot viewRoot = context.getViewRoot();
Locale locale = viewRoot.getLocale();
```

2. Получение текущего загрузчика классов. Это необходимо для определения местонахождения связки ресурсов.

```
ClassLoader loader = Thread.currentThread().getContextClassLoader();
```

3. Получение связки ресурсов с указанными именем, локалью и загрузчиком классов.

```
ResourceBundle bundle = ResourceBundle.getBundle(bundleName, locale, loader);
```

4. Выборка строки ресурсов с заданным идентификатором из связки.

```
String resource = bundle.getString(resourceId);
```

Но в ходе этого процесса возникают определенные затруднения. Фактически требуются две строки сообщений: одна — для сообщений в виде резюме, другая — для сообщений с подробными сведениями. В соответствии с принятым соглашением получе-

ние идентификатора ресурса для сообщения с подробными сведениями осуществляется путем добавления суффикса _detail к ключу сообщения в виде резюме. Например:

```
badCreditCardCharacter=Invalid card number.
badCreditCardCharacter_detail=The card number contains invalid characters.
```

Кроме того, преобразователи обычно входят в состав повторно используемой библиотеки. Целесообразно предусмотреть в конкретном приложении возможность переопределять сообщения. (Сведения о том, как переопределять стандартные сообщения преобразователя, см. в разделе "Изменение текста стандартных сообщений об ошибках" на стр. 229.) Поэтому необходимо вначале попытаться найти нужные сообщения в связке сообщений, относящейся к приложению, и только после этого заниматься выборкой сообщений, предусмотренных по умолчанию.

Напомним, что имя связки приложения можно указать в файле конфигурации, как в следующем примере:

```
<faces-config>
   <application>
      <message-bundle>com.mycompany.myapp.messages</message-bundle>
   </application>
   ...
</faces-config>
```

Ниже приведен фрагмент кода, в котором осуществляется выборка этого имени связки.

```
Application app = context.getApplication();
String appBundleName = app.getResourceBundle();
```

Вначале следует осуществлять поиск требуемых ресурсов в этой связке и только после этого использовать библиотечное значение по умолчанию.

Наконец, может потребоваться, чтобы некоторые сообщения предоставляли подробные сведения о характере ошибки. Например, иногда возникает необходимость сообщить пользователю, какой символ в номере кредитной карточки оказался недопустимым. Строки сообщений могут содержать такие метки-заполнители, как {0}, {1} и так далее, например:

```
The card number contains the invalid character {0}.
```

Класс java.text.MessageFormat позволяет подставлять значения вместо меток-заполнителей:

```
Object[] params = ...;
MessageFormat formatter = new MessageFormat(resource, locale);
String message = formatter.format(params);
```

В данном случае значения, которые применяются для подстановки, содержатся в массиве params. (Дополнительные сведения о классе MessageFormat приведены в книге Кея Хорстмена (Cay Horstmann) и Гэри Корнелла (Gary Cornell) *Core Java™*, 8th ed., Santa Clara, CA: Sun Microsystems Press/Prentice Hall, 2008, том 2, глава 5.)

В идеальном случае основная часть этой трудоемкой работы должна была быть выполнена платформой JSF. Безусловно, можно было бы найти применимый для этого код в недрах справочной реализации, но разработчики платформы решили не предоставлять к нему доступ программистам на JSF.

Мы предоставляем пакет com.corejsf.util со вспомогательными классами, которые реализуют эти недостающие части. Читатель вправе использовать эти классы в своем собственном коде.

Класс com.corejsf.util.Messages имеет статический метод getMessage, который возвращает сообщение FacesMessage с заданными именем связки, идентификатором ресурса и параметрами:

```
FacesMessage message
   = com.corejsf.util.Messages.getMessage(
      "com.corejsf.messages", "badCreditCardCharacter",
      new Object[] { new Character(invalidCharacter) });
```

Если сообщение не содержит меток-заполнителей, то в качестве массива параметров можно передать значение null.

В применяемой нами реализации соблюдается соглашение JSF, в соответствии с которым отсутствующие ресурсы обозначаются как *???resourceId???*. Исходный код приведен в листинге 7.13.

На заметку! Программисты, предпочитающие повторно использовать стандартное сообщение JSF для указания на ошибки преобразования, могут применить следующий вызов:

```
FacesMessage message = com.corejsf.util.Messages.getMessage(
            "javax.faces.Messages", "javax.faces.component.UIInput.CONVERSION", null);
```

javax.faces.context.FacesContext

- static FacesContext getCurrentInstance()

  Возвращает контекст для запроса, обрабатываемого текущим потоком, или значение null, если текущий поток не обрабатывает запрос.

- UIViewRoot getViewRoot()

  Возвращает компонент корневого каталога для запроса, описанного этим контекстом.

javax.faces.component.UIViewRoot

- Locale getLocale()

  Возвращает локаль, применяемую для подготовки этого представления к отображению.

Листинг 7.13. Файл converter2/src/java/com/corejsf/util/Messages.java

```
1. package com.corejsf.util;
2.
3. import java.text.MessageFormat;
4. import java.util.Locale;
5. import java.util.MissingResourceException;
6. import java.util.ResourceBundle;
7. import javax.faces.application.Application;
8. import javax.faces.application.FacesMessage;
9. import javax.faces.component.UIViewRoot;
10. import javax.faces.context.FacesContext;
11.
12. public class Messages {
13.    public static FacesMessage getMessage(String bundleName, String resourceId,
14.       Object[] params) {
15.       FacesContext context = FacesContext.getCurrentInstance();
16.       Application app = context.getApplication();
17.       String appBundle = app.getMessageBundle();
18.       Locale locale = getLocale(context);
19.       ClassLoader loader = getClassLoader();
20.       String summary = getString(appBundle, bundleName, resourceId,
21.          locale, loader, params);
```

```
22.       if (summary == null) summary = "???" + resourceId + "???";
23.       String detail = getString(appBundle, bundleName, resourceId + "_detail",
24.          locale, loader, params);
25.       return new FacesMessage(summary, detail);
26.    }
27.
28.    public static String getString(String bundle, String resourceId,
29.          Object[] params) {
30.       FacesContext context = FacesContext.getCurrentInstance();
31.       Application app = context.getApplication();
32.       String appBundle = app.getMessageBundle();
33.       Locale locale = getLocale(context);
34.       ClassLoader loader = getClassLoader();
35.       return getString(appBundle, bundle, resourceId, locale, loader, params);
36.    }
37.
38.    public static String getString(String bundle1, String bundle2,
39.           String resourceId, Locale locale, ClassLoader loader,
40.           Object[] params) {
41.        String resource = null;
42.        ResourceBundle bundle;
43.
44.        if (bundle1 != null) {
45.           bundle = ResourceBundle.getBundle(bundle1, locale, loader);
46.           if (bundle != null)
47.              try {
48.                 resource = bundle.getString(resourceId);
49.              } catch (MissingResourceException ex) {
50.              }
51.        }
52.
53.        if (resource == null) {
54.           bundle = ResourceBundle.getBundle(bundle2, locale, loader);
55.           if (bundle != null)
56.              try {
57.                 resource = bundle.getString(resourceId);
58.              } catch (MissingResourceException ex) {
59.              }
60.        }
61.
62.        if (resource == null) return null; // Отсутствие соответствия
63.        if (params == null) return resource;
64.
65.        MessageFormat formatter = new MessageFormat(resource, locale);
66.        return formatter.format(params);
67.    }
68.
69.    public static Locale getLocale(FacesContext context) {
70.       Locale locale = null;
71.       UIViewRoot viewRoot = context.getViewRoot();
72.       if (viewRoot != null) locale = viewRoot.getLocale();
73.       if (locale == null) locale = Locale.getDefault();
74.       return locale;
75.    }
76.
77.    public static ClassLoader getClassLoader() {
78.       ClassLoader loader = Thread.currentThread().getContextClassLoader();
79.       if (loader == null) loader = ClassLoader.getSystemClassLoader();
80.       return loader;
81.    }
82. }
```

Пример приложения с пользовательским преобразователем

В настоящем разделе рассматриваются последние части очередного примера приложения. На рис. 7.10 показана структура каталогов. В листингах 7.14 и 7.15 приведены страницы ввода и результатов. Чтобы ознакомиться с двумя стилями определения пользовательских преобразователей, можно рассмотреть теги `inputText` и `outputText` для номеров кредитных карточек. (Обе спецификации преобразователей можно было бы опустить, если бы для типа `CreditCard` был зарегистрирован преобразователь, применяемый по умолчанию.)

Пользовательский преобразователь определен в файле `faces-config.xml` (листинг 7.16). Файл `messages.properties`, показанный в листинге 7.17, содержит сообщение об ошибке для преобразователя номеров кредитных карточек. Наконец, в листинге 7.18 показан бин средства оплаты с тремя свойствами типа `double`, `Date` и `CreditCard`.

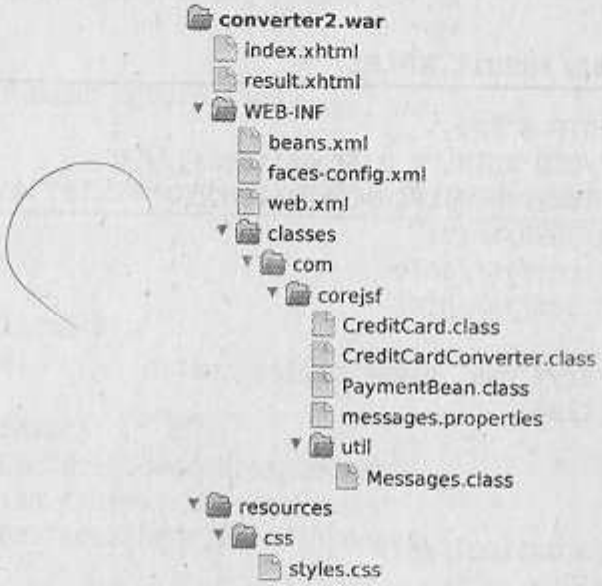


Рис. 7.10. Структура каталогов для примера пользовательского преобразователя

Листинг 7.14. Файл `converter2/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <h1>#{msgs.enterPayment}</h1>
         <h:panelGrid columns="3">
            #{msgs.amount}
            <h:inputText id="amount" label="#{msgs.amount}"
                         value="#{payment.amount}">
               <f:convertNumber minFractionDigits="2"/>
            </h:inputText>
            <h:message for="amount" styleClass="errorMessage"/>

```

```
            #{msgs.creditCard}
            <h:inputText id="card" label="#{msgs.creditCard}"
                         value="#{payment.card}">
            </h:inputText>
            <h:message for="card" styleClass="errorMessage"/>

            #{msgs.expirationDate}
            <h:inputText id="date" label="#{msgs.expirationDate}"
                         value="#{payment.date}">
               <f:convertDateTime pattern="MM/yyyy"/>
            </h:inputText>
            <h:message for="date" styleClass="errorMessage"/>
         </h:panelGrid>
         <h:commandButton value="#{msgs.process}" action="result"/>
      </h:form>
   </h:body>
</html>
```

Листинг 7.15. Файл converter2/web/result.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <h1>#{msgs.paymentInformation}</h1>
         <h:panelGrid columns="2">
            #{msgs.amount}
            <h:outputText value="#{payment.amount}">
               <f:convertNumber type="currency"/>
            </h:outputText>

            #{msgs.creditCard}
            <h:outputText value="#{payment.card}"/>
            #{msgs.expirationDate}
            <h:outputText value="#{payment.date}">
               <f:convertDateTime pattern="MM/yyyy"/>
            </h:outputText>
         </h:panelGrid>
         <h:commandButton value="#{msgs.back}" action="index"/>
      </h:form>
   </h:body>
</html>
```

Листинг 7.16. Файл converter2/web/WEB-INF/faces-config.xml

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
   version="2.0">
   <application>
```

```
        <message-bundle>com.corejsf.messages</message-bundle>
        <resource-bundle>
           <base-name>com.corejsf.messages</base-name>
           <var>msgs</var>
        </resource-bundle>
     </application>
</faces-config>
```

Листинг 7.17. Файл converter2/src/java/com/corejsf/messages.properties

```
badCreditCardCharacter=Invalid card number.
badCreditCardCharacter_detail=The card number contains the invalid character {0}.
title=An Application to Test Data Conversion
enterPayment=Please enter the payment information
amount=Amount
creditCard=Credit Card
expirationDate=Expiration date (Month/Year)
process=Process
back=Back
paymentInformation=Payment information
```

Листинг 7.18. Файл converter2/src/java/com/corejsf/PaymentBean.java

```
package com.corejsf;

import java.io.Serializable;
import java.util.Date;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;

@Named("payment") // или @ManagedBean(name="payment")
@SessionScoped
public class PaymentBean implements Serializable {
   private double amount;
   private CreditCard card = new CreditCard("");
   private Date date = new Date();

   public void setAmount(double newValue) { amount = newValue; }
   public double getAmount() { return amount; }

   public void setCard(CreditCard newValue) { card = newValue; }
   public CreditCard getCard() { return card; }

   public void setDate(Date newValue) { date = newValue; }
   public Date getDate() { return date; }
}
```

Предоставление атрибутов для преобразователей

Каждый компонент JSF может хранить произвольные атрибуты. Можно задать атрибут и для компонента, за которым закрепляется преобразователь; для этого служит тег `f:attribute`. В применяемом преобразователе можно затем осуществлять выборку атрибута из его компонента. Ниже показано, как можно применить этот метод для задания разделительной строки, используемой в преобразователе номеров кредитных карточек.

При закреплении преобразователя необходимо также вложить тег f:attribute в компонент:

```
<h:outputText value="#{payment.card}">
   <f:converter converterId="CreditCard"/>
   <f:attribute name="separator" value="-"/>
</h:outputText>
```

В преобразователе выборка атрибута выполняется следующим образом:

```
separator = (String) component.getAttributes().get("separator");
```

Ниже в этой главе будет показан более изящный механизм передачи атрибутов преобразователю, основанный на применении собственного тега преобразователя.

javax.faces.component.UIComponent

- Map getAttributes()

  Возвращает изменяемую карту всех атрибутов и свойств данного компонента.

Реализация классов пользовательского средства проверки

Процесс реализации классов пользовательского средства проверки аналогичен процессу реализации пользовательских преобразователей. Применяемый класс средства проверки должен реализовывать интерфейс javax.faces.validator.Validator.

Этот интерфейс средства проверки определяет только один метод:

```
void validate(FacesContext context, UIComponent component, Object value)
```

Если проверка правильности оканчивается неудачей, вырабатывается сообщение FacesMessage с описанием ошибки и на основе сообщения создается и активизируется исключение ValidatorException:

```
if (validation fails) {
   FacesMessage message = ...;
   message.setSeverity(FacesMessage.SEVERITY_ERROR);
   throw new ValidatorException(message);
}
```

Этот процесс аналогичен формированию сообщений об ошибках преобразования, за исключением того, что активизируется исключение ValidatorException вместо ConverterException.

Например, в листинге 7.19 на стр. 258 показано средство проверки, которое проверяет цифры кредитной карточки с использованием формулы Луна. Как описано в разделе "Получение сообщений об ошибках из связок ресурсов" на стр. 249, используется вспомогательный класс com.corejsf.util.Messages для поиска строк сообщений в связке ресурсов.

javax.faces.validator.Validator

- void validate(FacesContext context, UIComponent component, Object value)

  Проверяет компонент, за которым закреплено это средство проверки. При возникновении ошибки проверки правильности активизируется исключение ValidatorException.

Регистрация пользовательских средств проверки

После создания средства проверки (как в приведенном выше примере) необходимо присвоить ему идентификатор. Как и применительно к идентификаторам преоб-

разователя, предусмотрены две возможности. В версии JSF 2.0 можно использовать аннотацию:

```
@FacesValidator("com.corejsf.Card")
public class CreditCardValidator implements Validator
```

В качестве другого способа можно зарегистрировать средство проверки в файле конфигурации (таком как `faces-config.xml`) следующим образом:

```
<validator>
   <validator-id>com.corejsf.Card</validator-id>
   <validator-class>com.corejsf.CreditCardValidator</validator-class>
</validator>
```

Идентификатор средства проверки можно определить в теге `f:validator`, в частности, в следующем фрагменте кода используется средство проверки номеров кредитных карточек, о котором речь шла выше:

```
<h:inputText id="card" value="#{payment.card}" required="true">
   <f:converter converterId="com.corejsf.Card"/>
   <f:validator validatorId="com.corejsf.Card"/>
</h:inputText>
```

В теге `f:validator` используется идентификатор средства проверки для поиска соответствующего класса, в случае необходимости создается экземпляр этого класса и вызывается его метод проверки.

На заметку! В технологии JSF используются отдельные пространства имен для идентификаторов средств проверки и преобразователей. Поэтому вполне допустимо иметь и преобразователь, и средство проверки с одинаковым идентификатором `com.corejsf.Card`.

На заметку! В реализации JSF применяемые стандартные средства проверки регистрируются с идентификаторами `javax.faces.LongRange`, `javax.faces.DoubleRange`, `javax.faces.Length`, `javax.faces.RegularExpression`, `javax.faces.Required` и `javax.faces.Bean`.

Остальные части этого примера приложения являются несложными. На рис. 7.11 показана структура каталогов, а листинг 7.20 содержит страницу JSF.

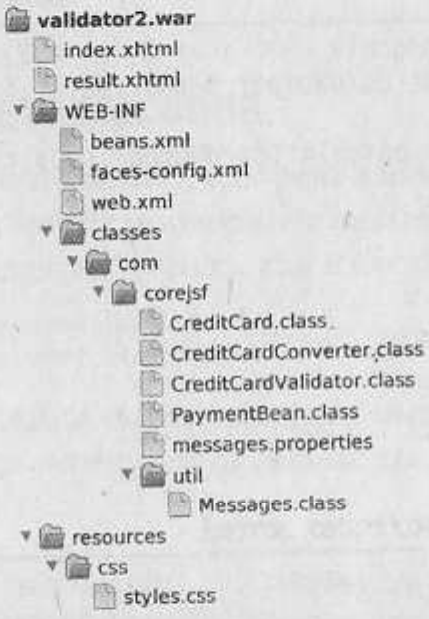


Рис. 7.11. Структура каталогов для примера пользовательского средства проверки

Тег f:validator может применяться для создания простых средств проверки, не имеющих параметров, таких как средство проверки номеров кредитных карточек, описанное выше. Если же требуется средство проверки, свойства которого могут быть заданы на странице JSF, то следует реализовать специализированный тег для этого средства проверки. О том, как это сделать, см. в разделе "Реализация классов пользовательского средства проверки" на стр. 256.

Листинг 7.19. Файл validator2/src/java/com/corejsf/CreditCardValidator.java

```
1. package com.corejsf;
2.
3. import javax.faces.application.FacesMessage;
4. import javax.faces.component.UIComponent;
5. import javax.faces.context.FacesContext;
6. import javax.faces.validator.FacesValidator;
7. import javax.faces.validator.Validator;
8. import javax.faces.validator.ValidatorException;
9.
10. @FacesValidator("com.corejsf.Card")
11. public class CreditCardValidator implements Validator {
12.    public void validate(FacesContext context, UIComponent component,
13.          Object value) {
14.       if(value == null) return;
15.       String cardNumber;
16.       if (value instanceof CreditCard)
17.          cardNumber = value.toString();
18.       else
19.        cardNumber = value.toString().replaceAll("\\D", ""); // Удаление нецифр. знаков
20.       if(!luhnCheck(cardNumber)) {
21.          FacesMessage message
22.            = com.corejsf.util.Messages.getMessage(
23.                "com.corejsf.messages", "badLuhnCheck", null);
24.          message.setSeverity(FacesMessage.SEVERITY_ERROR);
25.          throw new ValidatorException(message);
26.       }
27.    }
28.
29.    private static boolean luhnCheck(String cardNumber) {
30.       int sum = 0;
31.
32.       for(int i = cardNumber.length() - 1; i >= 0; i -= 2) {
33.          sum += Integer.parseInt(cardNumber.substring(i, i + 1));
34.          if(i > 0) {
35.             int d = 2 * Integer.parseInt(cardNumber.substring(i - 1, i));
36.             if(d > 9) d -= 9;
37.             sum += d;
38.          }
39.       }
40.
41.       return sum % 10 == 0;
42.    }
43. }
```

Листинг 7.20. Файл validator2/web/index.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
```

```
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <h1>#{msgs.enterPayment}</h1>
         <h:panelGrid columns="3">
            #{msgs.amount}
            <h:inputText id="amount" label="#{msgs.amount}"
                         value="#{payment.amount}">
               <f:convertNumber minFractionDigits="2"/>
            </h:inputText>
            <h:message for="amount" styleClass="errorMessage"/>

            #{msgs.creditCard}
            <h:inputText id="card" label="#{msgs.creditCard}"
                         value="#{payment.card}" required="true">
               <f:converter converterId="com.corejsf.Card"/>
               <f:validator validatorId="com.corejsf.Card"/>
            </h:inputText>
            <h:message for="card" styleClass="errorMessage"/>

            #{msgs.expirationDate}
            <h:inputText id="date" label="#{msgs.expirationDate}"
                         value="#{payment.date}">
               <f:convertDateTime pattern="MM/yyyy"/>
            </h:inputText>
            <h:message for="date" styleClass="errorMessage"/>
         </h:panelGrid>
         <h:commandButton value="#{msgs.process}" action="result"/>
      </h:form>
   </h:body>
</html>
```

`@javax.faces.validator.FacesValidator`

- `String value`

  Идентификатор преобразователя.

Проверка с помощью методов бина

В предыдущем разделе было показано, как реализовать класс проверки правильности. Но можно также добавить метод проверки правильности к существующему классу и вызвать его с помощью выражения метода, как в следующем примере:

```
<h:inputText id="card" value="#{payment.card}"
   required="true" validator="#{payment.luhnCheck}"/>
```

В таком случае бин обеспечения оплаты должен иметь метод точно с такой же сигнатурой, как и у метода `validate` интерфейса `Validator`:

```
public class PaymentBean {
   ...
   public void luhnCheck(FacesContext context, UIComponent component, Object value) {
      ... // Тот же код, что и в предыдущем примере
   }
}
```

С чем может быть связана необходимость в применении такого способа? Он имеет одно важное преимущество. Метод проверки правильности может обращаться к другим переменным экземпляра класса. Соответствующий пример см. в разделе "Предоставление атрибутов для преобразователей" на стр. 255.

Данный способ имеет также недостаток, заключающийся в том, что становится более затруднительным перемещение средства проверки в новое веб-приложение, поэтому, скорее всего, его применение будет ограничиваться только сценариями, характерными для приложения.

Внимание! Значение атрибута `validator` представляет собой выражение метода, тогда как с виду подобный атрибут `converter` задает идентификатор преобразователя (если он представляет собой строку) или объект преобразователя (если он является выражением значения). Эмерсон недаром сказал, что "тупое единообразие — это инструмент запугивания в руках недалеких людей".

Проверка связей между несколькими компонентами

Механизм проверки правильности в технологии JSF был разработан в целях проверки отдельных компонентов. Но на практике часто возникает необходимость убедиться в том, что все взаимосвязанные компоненты имеют приемлемые значения, и только после этого дать разрешение распространить эти значения на модель. Например, как было отмечено ранее, пользователям не очень удобно вводить многокомпонентную дату в единственном текстовом поле. Вместо этого следует использовать три разных текстовых поля: для числа, месяца и года (рис. 7.12).

Если пользователь введет недопустимую дату, скажем, 30 февраля, то потребуется отобразить сообщение об ошибке при проверке правильности и предотвратить попадание недопустимых данных в модель.

Эту задачу можно решить с помощью следующего подхода. Закрепите средство проверки за последним из компонентов. Ко времени вызова этого средства проверки предыдущие компоненты пройдут проверку правильности и для них будут заданы локальные значения. Затем выполняется преобразование для последнего компонента и преобразованное значение передается в качестве параметра `Object` метода проверки правильности.

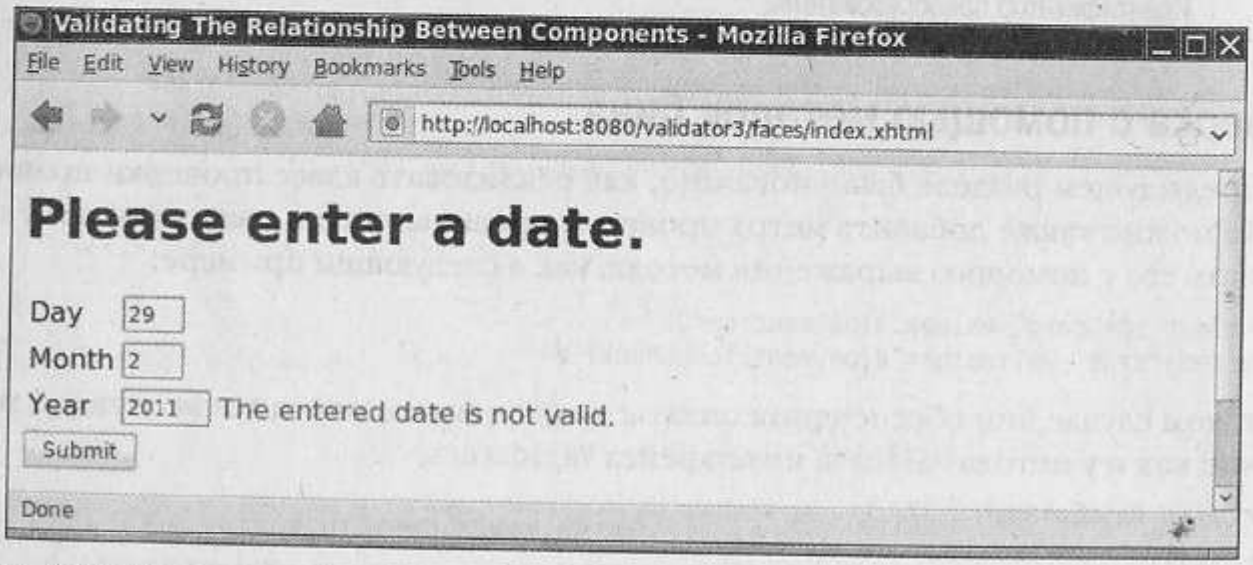


Рис. 7.12. Проверка связей, включающих в себя три компонента

Чтобы можно было осуществить этот подход, средство проверки последнего компонента должно иметь доступ к другим компонентам. Такой доступ можно обеспе-

чить, назначив значения идентификаторов другим компонентам. После этого можно воспользоваться методом findComponent класса UIComponent для определения местонахождения этих компонентов:

```
public class BackingBean {
   ...
   public void validateDate(FacesContext context, UIComponent component,
         Object value) {
      UIInput dayInput = (UIInput) component.findComponent("day");
      UIInput monthInput = (UIInput) component.findComponent("month");
      int d = ((Integer) dayInput.getLocalValue()).intValue();
      int m = ((Integer) monthInput.getLocalValue()).intValue();
      int y = ((Integer) value).intValue();
       if (!isValidDate(d, m, y)) {
          FacesMessage message = ...;
          throw new ValidatorException(message);
       }
   }
   ...
}
```

Заслуживает внимания то, что этот поиск значения характеризуется некоторой асимметрией. Дело в том, что для последнего компонента еще не задано локальное значение, поскольку он пока не прошел проверку правильности.

Альтернативный подход состоит в том, что средство проверки можно закрепить за скрытым полем ввода, расположенным вслед за всеми другими полями в форме:

```
<h:inputHidden id="datecheck" validator="#{bb.validateDate}"
   value="needed"/>
```

Это скрытое поле подготавливается к отображению как скрытое поле ввода HTML. При обратной отправке значения этого поля в действие вступает средство проверки. (Важно, чтобы было задано хотя бы какое-то значение поля. В противном случае значение компонента так и не будет обновлено.) При осуществлении такого подхода функция проверки правильности действует более симметрично, поскольку ко времени ее использования для всех прочих компонентов формы уже заданы локальные значения.

На заметку! В главе 8 будет продемонстрирован еще один подход: применение прослушивателя событий PostValidateEvent, который проверяет три указанных компонента.

На заметку! А в действительности имело бы смысл разработать пользовательский компонент для даты, который подготавливал бы к отображению три поля ввода и возвращал единственное значение типа Date. Для этого единственного компонента было бы совсем несложно предусмотреть проверку. Однако подход, описанный в настоящем разделе, применим для любых форм, которые нуждаются в перекрестной проверке правильности заданных в них полей.

Реализация пользовательских тегов преобразователей и средств проверки

Пользовательские преобразователи и классы средств проверки, описанные в предыдущих разделах, имеют определенный недостаток: они не допускают использование атрибутов. Например, может потребоваться задать символ разделителя для преобразователя номеров кредитных карточек, чтобы проектировщик страниц мог вы-

бирать, должны ли использоваться для разделения групп цифр номера тире или пробелы. В частности, было бы желательно, чтобы проектировщики могли применять теги, подобные следующим:

```
<h:outputText value="#{payment.card}">
   <corejsf:convertCreditcard separator="-"/>
</h:outputText>
```

Иными словами, пользовательские преобразователи должны предоставлять такие же возможности, как и стандартные теги `f:convertNumber` и `f:convertDateTime`. Для достижения этого необходимо реализовать пользовательский тег преобразователя. В настоящем разделе будет описано, как реализуются собственные теги преобразователей и средств проверки.

Прежде всего пользовательские теги преобразователей и средств проверки должны быть определены в файле описания тегов. С такими файлами описания читатель уже встречался в главе 5. Местоположение этого файла можно задать в файле `web.xml`:

```
<context-param>
  <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
  <param-value>/WEB-INF/corejsf.taglib.xml</param-value>
</context-param>
```

Еще один вариант состоит в том, что если потребуется упаковать создаваемые теги, чтобы они стали применимыми в других проектах, то можно разместить их в JAR-файле, зарегистрировать и добавить файл описания тегов в каталог `META-INF`.

В листинге 7.21 показан файл описания, который представляет пользовательский преобразователь и средство проверки.

Листинг 7.21. Файл `custom-tags/web/WEB-INF/corejsf.taglib.xml`

```
 1. <?xml version="1.0" encoding="UTF-8"?>
 2. <facelet-taglib version="2.0"
 3.    xmlns="http://java.sun.com/xml/ns/javaee"
 4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 5.    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
 6.    http://java.sun.com/xml/ns/javaee/web-facelettaglibary_2_0.xsd">
 7.    <namespace>http://corejsf.com</namespace>
 8.    <tag>
 9.       <tag-name>convertCreditCard</tag-name>
10.       <converter>
11.          <converter-id>com.corejsf.CreditCard</converter-id>
12.       </converter>
13.    </tag>
14.    <tag>
15.       <tag-name>validateCreditCard</tag-name>
16.       <validator>
17.          <validator-id>com.corejsf.CreditCard</validator-id>
18.       </validator>
19.    </tag>
20. </facelet-taglib>
```

Затем можно просто задать идентификатор преобразователя или средства проверки для указанного имени тега.

Полный код класса преобразователя показан в листинге 7.22 на стр. 263. Заслуживает внимания метод `setSeparator`, который вызывается при указании разделителя в теге.

Реализуя преобразователи или средства проверки, сохраняющие состояние, необходимо убедиться в том, что состояние может быть сохранено. Простейший способ

осуществления этого состоит в том, чтобы реализовать интерфейс Serializable и придерживаться обычных правил для сериализации в коде Java. (Дополнительная информация о сохранении состояния приведена в главе 11.)

В рассматриваемом примере приложения был также предусмотрен тег средства проверки для выполнения проверки по формуле Луна. Данное средство проверки используется следующим образом:

```
<h:inputText id="card" value="#{payment.card}" required="true">
   <corejsf:validateCreditCard errorDetail="#{msgs.creditCardError}"/>
</h:inputText>
```

По умолчанию средство проверки отображает сообщение об ошибке, которое указывает на неудачное завершение проверки по формуле Луна. Если какие-либо пользователи конкретного приложения незнакомы с этой терминологией, то может потребоваться сделать это сообщение более понятным. Для этой цели предусмотрены атрибуты errorSummmary и errorDetail.

Код указанного средства проверки приведен в листинге 7.23. В нем заслуживают внимания методы setErrorSummary и setErrorDetail, применяемые для задания атрибутов тегов.

На рис. 7.13 показана структура каталогов приложения, а в листинге 7.24 приведен код страницы JSF.

Листинг 7.22. Файл custom-tags/src/java/com/corejsf/CreditCardConverter.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import javax.faces.component.UIComponent;
5. import javax.faces.context.FacesContext;
6. import javax.faces.convert.Converter;
7. import javax.faces.convert.ConverterException;
8. import javax.faces.convert.FacesConverter;
9.
10. @FacesConverter("com.corejsf.CreditCard")
11. public class CreditCardConverter implements Converter, Serializable
12. {
13.    private String separator;
14.
15.    public void setSeparator(String newValue) { separator = newValue; }
16.
17.    public Object getAsObject(
18.       FacesContext context,
19.       UIComponent component,
20.       String newValue)
21.       throws ConverterException {
22.       StringBuilder builder = new StringBuilder(newValue);
23.       int i = 0;
24.       while (i < builder.length()) {
25.          if (Character.isDigit(builder.charAt(i)))
26.             i++;
27.          else
28.             builder.deleteCharAt(i);
29.       }
30.       return new CreditCard(builder.toString());
31.    }
32.
33.    public String getAsString(
34.       FacesContext context,
```

```
35.         UIComponent component,
36.         Object value)
37.         throws ConverterException {
38.         // длина 13: xxxx xxx xxx xxx
39.         // длина 14: xxxxx xxxx xxxxx
40.         // длина 15: xxxx xxxxxx xxxxx
41.         // длина 16: xxxx xxxx xxxx xxxx
42.         // длина 22: xxxxxx xxxxxxxx xxxxxxxx
43.         if (!(value instanceof CreditCard))
44.            throw new ConverterException();
45.         String v = ((CreditCard) value).toString();
46.         String sep = separator;
47.         if (sep == null) sep = " ";
48.         int[] boundaries = null;
49.         int length = v.length();
50.         if (length == 13)
51.            boundaries = new int[] { 4, 7, 10 };
52.         else if (length == 14)
53.            boundaries = new int[] { 5, 9 };
54.         else if (length == 15)
55.            boundaries = new int[] { 4, 10 };
56.         else if (length == 16)
57.            boundaries = new int[] { 4, 8, 12 };
58.         else if (length == 22)
59.            boundaries = new int[] { 6, 14 };
60.         else
61.            return v;
62.         StringBuilder result = new StringBuilder();
63.         int start = 0;
64.         for (int i = 0; i < boundaries.length; i++) {
65.            int end = boundaries[i];
66.            result.append(v.substring(start, end));
67.            result.append(sep);
68.            start = end;
69.         }
70.         result.append(v.substring(start));
71.         return result.toString();
72.      }
73. }
```

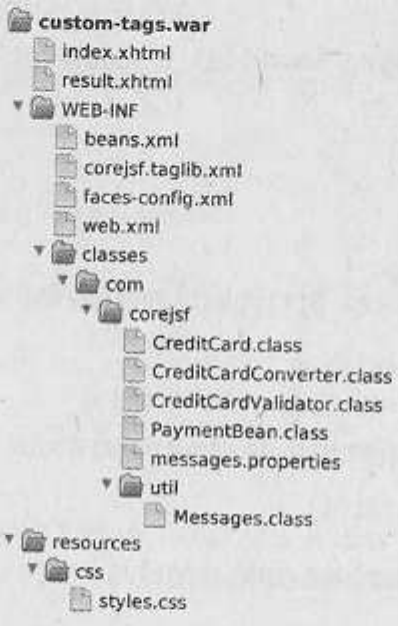


Рис. 7.13. Структура каталогов приложения проверки с широкими функциями

Листинг 7.23. Файл custom-tags/src/java/com/corejsf/CreditCardValidator.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.text.MessageFormat;
5. import java.util.Locale;
6.
7. import javax.faces.application.FacesMessage;
8. import javax.faces.component.UIComponent;
9. import javax.faces.context.FacesContext;
10. import javax.faces.validator.FacesValidator;
11. import javax.faces.validator.Validator;
12. import javax.faces.validator.ValidatorException;
13.
14. @FacesValidator("com.corejsf.CreditCard")
15. public class CreditCardValidator implements Validator, Serializable {
16.    private String errorSummary;
17.    private String errorDetail;
18.
19.    public void validate(FacesContext context, UIComponent component,
20.          Object value) {
21.       if(value == null) return;
22.       String cardNumber;
23.       if (value instanceof CreditCard)
24.          cardNumber = value.toString();
25.       else
26.          cardNumber = getDigitsOnly(value.toString());
27.       if(!luhnCheck(cardNumber)) {
28.          FacesMessage message
29.             = com.corejsf.util.Messages.getMessage(
30.                "com.corejsf.messages", "badLuhnCheck", null);
31.          message.setSeverity(FacesMessage.SEVERITY_ERROR);
32.          Locale locale = context.getViewRoot().getLocale();
33.          Object[] params = new Object[] { value };
34.          if (errorSummary != null)
35.             message.setSummary(
36.                   new MessageFormat(errorSummary, locale).format(params));
37.          if (errorDetail != null)
38.             message.setDetail(
39.                   new MessageFormat(errorDetail, locale).format(params));
40.          throw new ValidatorException(message);
41.       }
42.    }
43.
44.    public void setErrorSummary(String newValue) {
45.       errorSummary = newValue;
46.    }
47.
48.    public void setErrorDetail(String newValue) {
49.       errorDetail = newValue;
50.    }
51.
52.    private static boolean luhnCheck(String cardNumber) {
53.       int sum = 0;
54.
55.       for(int i = cardNumber.length() - 1; i >= 0; i -= 2) {
56.          sum += Integer.parseInt(cardNumber.substring(i, i + 1));
57.          if(i > 0) {
58.             int d = 2 * Integer.parseInt(cardNumber.substring(i - 1, i));
59.             if(d > 9) d -= 9;
60.             sum += d;
```

```
            }
         }

         return sum % 10 == 0;
      }

      private static String getDigitsOnly(String s) {
         StringBuilder digitsOnly = new StringBuilder ();
         char c;
         for(int i = 0; i < s.length (); i++) {
            c = s.charAt (i);
            if (Character.isDigit(c)) {
               digitsOnly.append(c);
            }
         }
         return digitsOnly.toString ();
      }
}
```

Листинг 7.24. Файл custom-tags/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:corejsf="http://corejsf.com">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <h1>#{msgs.enterPayment}</h1>
         <h:panelGrid columns="2">
            #{msgs.amount}
            <h:inputText id="amount" value="#{payment.amount}">
                <f:convertNumber minFractionDigits="2"/>
            </h:inputText>

            #{msgs.creditCard}
            <h:inputText id="card" value="#{payment.card}" required="true">
                <corejsf:validateCreditCard
                      errorDetail="#{msgs.creditCardError}"/>
            </h:inputText>

            #{msgs.expirationDate}
            <h:inputText id="date" value="#{payment.date}">
                <f:convertDateTime pattern="MM/yyyy"/>
            </h:inputText>
         </h:panelGrid>
         <h:messages styleClass="errorMessage"
                     showSummary="false" showDetail="true"/>
         <br/>
         <h:commandButton value="Process" action="result"/>
      </h:form>
   </h:body>
</html>
```

Резюме

Как уже было сказано, технология JSF предоставляет мощную и расширяемую поддержку для преобразования и проверки правильности. Предусмотрена возможность включать в состав страниц JSF стандартные преобразователи и средства проверки, для чего часто бывает достаточно одной строки кода, или предусматривать применение собственных алгоритмов, если потребуются более сложные преобразования или проверки правильности. Наконец, программисты могут определять собственные теги преобразования и проверки правильности.

Обработка событий

В этой главе...

- События и жизненный цикл JSF
- События изменения значения
- События действия
- Теги прослушивателей событий
- Немедленно активизируемые компоненты
- Передача данных из пользовательского интерфейса на сервер
- События фазы
- Системные события JSF 1.2
- Объединение рассматриваемых средств в одном приложении

Глава 8

В веб-приложениях часто возникает необходимость отвечать на события, активизированные пользователем, такие как выбор элементов в меню или щелчок на кнопке. Например, может потребоваться ответить на выбор страны в форме адреса путем изменения локали и перезагрузки текущей страницы, что способствует повышению удобства в работе пользователей.

Как правило, обработчики событий регистрируются путем закрепления за компонентами, например, прослушиватель изменений значений может быть зарегистрирован как закрепленный за меню на странице JSF примерно так:

```
<h:selectOneMenu valueChangeListener="#{form.countryChanged}"...>
  ...
</h:selectOneMenu>
```

В приведенном выше коде выражение связывания метода `#{form.countryChanged}` ссылается на метод `countryChanged` бина `form`. Этот метод вызывается реализацией JSF после того, как пользователь сделает выбор в меню. Одной из тем обсуждения настоящей главы является точное определение момента, в который происходит вызов этого метода.

Платформа JSF поддерживает четыре вида событий.

- События изменения значения.
- События действия.
- События фазы.
- Системные события (начиная с версии JSF 2.0).

События изменения значения активизируются тегами, содержащими доступные для редактирования значения, такими как `h:inputText`, `h:selectOneRadio` и `h:selectManyMenu`, при изменении значения компонента.

События действия активизируются источниками действий (например, `h:commandButton` и `h:commandLink`) при активизации кнопки или ссылки. События фазы обычно активизируются в ходе жизненного цикла JSF. В версии JSF 2.0 добавлено большое количество системных событий. Некоторые из системных событий представляют интерес для прикладных программистов. Например, теперь появилась воз-

можность выполнять требуемые действия перед подготовкой к отображению представления или компонента.

На заметку! Следует учитывать, что обработка всех событий JSF выполняется на сервере. Если на странице JSF предусмотрен обработчик событий, это служит для реализации JSF указанием, что рассматриваемые события должны быть обработаны в подходящем месте жизненного цикла, когда сервер обрабатывает пользовательский ввод, поступающий от этой страницы.

События и жизненный цикл JSF

Запросы в приложениях JSF обрабатываются реализацией JSF с помощью сервлета контроллера, который, в свою очередь, выполняет жизненный цикл JSF. Обработка событий в жизненном цикле JSF демонстрируется на рис. 8.1.

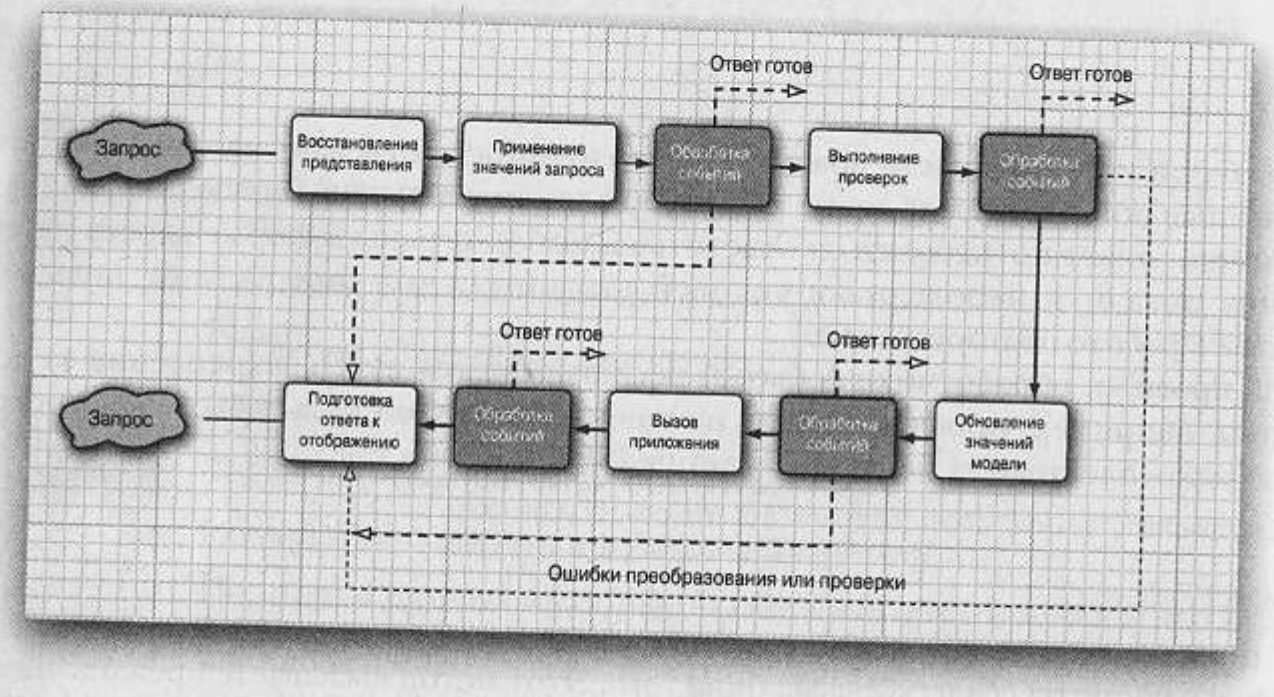


Рис. 8.1. Обработка событий в жизненном цикле JSF

Начиная с фазы применения значений запроса, реализация JSF может создавать события и добавлять их к очереди событий на протяжении каждой фазы жизненного цикла. После прохождения этих фаз реализация JSF выполняет рассылку поставленных в очередь событий по зарегистрированным прослушивателям. Эти события и связанные с ними прослушиватели являются основной темой настоящей главы.

На заметку! Прослушиватели событий могут затрагивать жизненный цикл JSF в одном из трех отношений.

1. Позволять жизненному циклу проходить обычным образом.
2. Вызывать метод `renderResponse` класса `FacesContext` для пропуска оставшейся части жизненного цикла вплоть до фазы подготовки ответа к отображению.
3. Вызывать метод `responseComplete` класса `FacesContext` для полного пропуска оставшейся части жизненного цикла.

С примером использования метода `renderResponse` можно ознакомиться в разделе "Немедленно активизируемые компоненты" на стр. 281.

События изменения значения

Компоненты в веб-приложении часто зависят друг от друга. Например, в приложении, показанном на рис. 8.2, значение приглашения State (Штат) зависит от значения элемента меню Country (Страна). Предусмотрена возможность поддерживать синхронизацию зависимых компонентов с помощью событий изменения значения, которые активизируются компонентами ввода после проверки их новых значений.

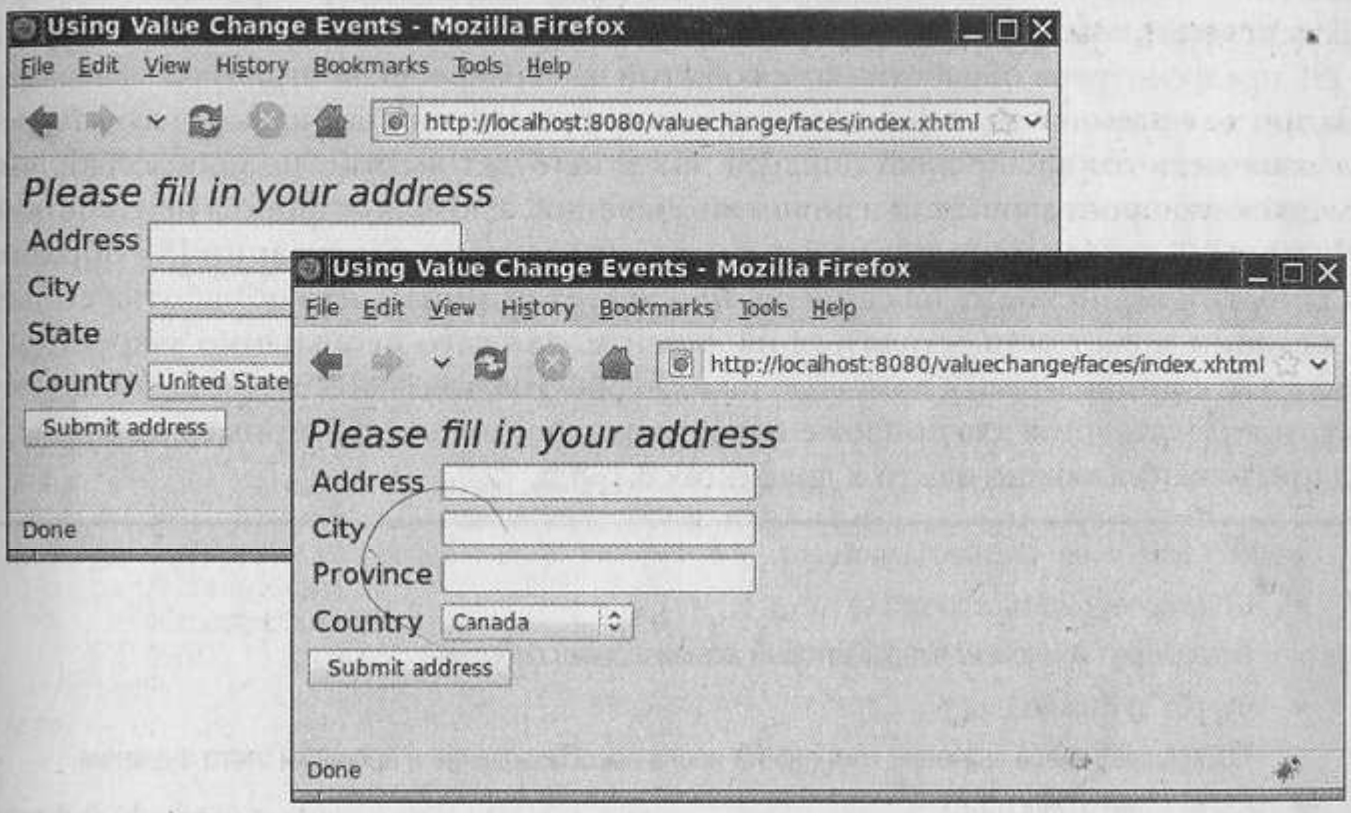


Рис. 8.2. Использование событий изменения значения

В приложении на рис. 8.2 прослушиватель изменений значений закреплен за меню `Country` и использует атрибут `onchange` для принудительной передачи формы после изменения значения меню:

```
<h:selectOneMenu value="#{form.country}" onchange="submit()"
      valueChangeListener="#{form.countryChanged}">
   <f:selectItems value="#{form.countries}" var="loc"
      itemLabel="#{loc.displayCountry}" itemValue="#{loc.country}"/>
</h:selectOneMenu>
```

В данном случае выражение значения `#{form.countries}` привязано к массиву объектов `Locale`.

После того как пользователь выбирает страну в меню, вызывается функция `submit` языка JavaScript для передачи формы меню, что впоследствии приводит к вызову жизненного цикла JSF. После прохождения фазы проверок правильности реализация JSF вызывает метод `countryChanged` бина `form`. Этот метод изменяет локаль корневого каталога представления в соответствии с новым значением страны:

```
public void countryChanged(ValueChangeEvent event) {
   for (Locale loc : countries)
      if (loc.getCountry().equals(event.getNewValue()))
         FacesContext.getCurrentInstance().getViewRoot().setLocale(loc);
}
```

Как и всем прослушивателям изменений значений, приведенному выше прослушивателю передается событие изменения значения. Прослушиватель использует это событие для доступа к новому значению компонента.

В этом примере заслуживает внимания еще одна особенность. Речь идет о том, что мы добавили атрибут `onchange` со значением `submit()` к применяемому тегу `h:selectOneMenu`. Применение такого значения атрибута указывает на то, что функция `submit` языка JavaScript будет вызываться каждый раз, когда кто-либо изменит выбранное в меню значение, а это станет причиной передачи на сервер окружающей формы.

Обеспечение такой передачи формы является крайне важным, поскольку в реализации JSF предусмотрена обработка всех событий на сервере. Если атрибут `onchange` не будет задан, то не произойдет передача формы при изменении выбранного пункта меню, а это означает, что жизненный цикл JSF так и не будет вызван, не произойдет вызов применяемого прослушивателя изменений значений, а локаль останется неизменной.

На первый взгляд может показаться странным, что в реализации JSF обработка всех событий происходит на сервере, но следует помнить, что обработка событий при желании может осуществляться на клиенте, для чего необходимо закрепить код JavaScript за компонентами с помощью таких атрибутов, как `onblur`, `onfocus`, `onclick` и т.д.

Структура каталогов для приложения, показанного на рис. 8.2, приведена на рис. 8.3, а код приложения можно найти в листингах 8.1–8.5.

`javax.faces.event.ValueChangeEvent`

- `UIComponent getComponent()`

  Возвращает компонент ввода, который активизировал событие.

- `Object getNewValue()`

  Возвращает новое значение компонента после преобразования и проверки этого значения.

- `Object getOldValue()`

  Возвращает предыдущее значение компонента.

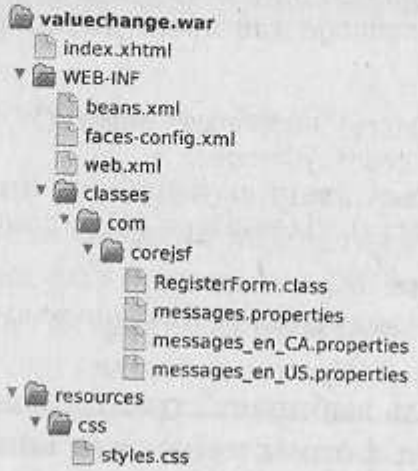


Рис. 8.3. Структура каталогов для примера изменения значения

Листинг 8.1. Файл `valuechange/web/index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
```

```
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.windowTitle}</title>
   </h:head>

   <h:body>
      <h:form>
         <span class="emphasis">#{msgs.pageTitle}</span>
         <h:panelGrid columns="2">
            #{msgs.streetAddressPrompt}
            <h:inputText value="#{form.streetAddress}"/>

            #{msgs.cityPrompt}
            <h:inputText value="#{form.city}"/>

            #{msgs.statePrompt}
            <h:inputText value="#{form.state}"/>

            #{msgs.countryPrompt}
            <h:selectOneMenu value="#{form.country}" onchange="submit()"
                  valueChangeListener="#{form.countryChanged}">
               <f:selectItems value="#{form.countries}" var="loc"
                  itemLabel="#{loc.displayCountry}" itemValue="#{loc.country}"/>
            </h:selectOneMenu>
         </h:panelGrid>
         <h:commandButton value="#{msgs.submit}"/>
      </h:form>
   </h:body>
</html>
```

Листинг 8.2. Файл valuechange/src/java/com/corejsf/RegisterForm.java

```
package com.corejsf;

import java.io.Serializable;
import java.util.LinkedHashMap;
import java.util.Locale;
import java.util.Map;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
import javax.faces.event.ValueChangeEvent;

@Named("form") // или @ManagedBean(name="form")
@SessionScoped
public class RegisterForm implements Serializable {
   private String streetAddress;
   private String city;
   private String state;
   private String country;

   private static final Locale[] countries = { Locale.US, Locale.CANADA };
    public Locale[] getCountries() { return countries; }

    public void setStreetAddress(String newValue) { streetAddress = newValue; }
    public String getStreetAddress() { return streetAddress; }
```

```
28.
29.     public void setCity(String newValue) { city = newValue; }
30.     public String getCity() { return city; }
31.
32.     public void setState(String newValue) { state = newValue; }
33.     public String getState() { return state; }
34.
35.     public void setCountry(String newValue) { country = newValue; }
36.     public String getCountry() { return country; }
37.
38.     public void countryChanged(ValueChangeEvent event) {
39.        for (Locale loc : countries)
40.           if (loc.getCountry().equals(event.getNewValue()))
41.              FacesContext.getCurrentInstance().getViewRoot().setLocale(loc);
42.     }
43. }
```

Листинг 8.3. Файл valuechange/web/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2. <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5.       http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
6.    version="2.0">
7.    <application>
8.       <resource-bundle>
9.          <base-name>com.corejsf.messages</base-name>
10.         <var>msgs</var>
11.      </resource-bundle>
12.   </application>
13. </faces-config>
```

Листинг 8.4. Файл valuechange/src/java/com/corejsf/messages_en_US.properties

```
1. windowTitle=Using Value Change Events
2. pageTitle=Please fill in your address
3.
4. streetAddressPrompt=Address
5. cityPrompt=City
6. statePrompt=State
7. countryPrompt=Country
8. submit=Submit address
```

Листинг 8.5. Файл valuechange/src/java/com/corejsf/messages_en_CA.properties

```
1. windowTitle=Using Value Change Events
2. pageTitle=Please fill in your address
3.
4. streetAddressPrompt=Address
5. cityPrompt=City
6. statePrompt=Province
7. countryPrompt=Country
8. submit=Submit address
```

События действия

События действия активизируются с помощью кнопок и ссылок. Как было показано в разделе "События и жизненный цикл JSF" на стр. 270, события действия активизируются в фазе вызова приложения, ближе к концу жизненного цикла.

Предусмотрена возможность добавить прослушиватель действий к источнику действий, как в следующем примере:

```
<h:commandLink actionListener="#{bean.linkActivated}">
  ...
</h:commandLink>
```

Командные компоненты передают запросы при их активизации, поэтому нет необходимости использовать атрибут onchange для обеспечения принудительной передачи формы, как было в случае применения событий изменения значения, описанном в разделе "События изменения значения" на стр. 271. При активизации кнопки или ссылки передается окружающая форма, а реализация JSF впоследствии активизирует события действия.

Важно провести четкое различие между прослушивателями действий и действиями. По существу, действия предназначены для поддержки бизнес-логики и участвуют в обработке элементов навигации, тогда как прослушиватели действий обычно выполняют логику пользовательского интерфейса и не участвуют в обработке элементов навигации.

Прослушиватели действий иногда работают совместно с действиями, если действие нуждается в информации о пользовательском интерфейсе. Например, в приложении, показанном на рис. 8.4, используются действие и прослушиватель действий в целях реагирования на щелчки кнопкой мыши путем перенаправления на определенную страницу JSF.

После щелчка на изображении лица президента приложение перенаправляет пользователя на страницу JSF со сведениями об этом президенте. Обратите внимание на то, что с помощью одного лишь действия нельзя было бы реализовать такое поведение, поскольку действие позволяет обеспечить переход на требуемую страницу, но не позволяет определить саму требуемую страницу, поскольку не имеет никаких данных о наличии кнопки с изображением в пользовательском интерфейсе или о выполнении щелчка мышью на этой кнопке.

В приложении, показанном на рис. 8.4, кнопка с изображением используется так:

```
<h:commandButton image="/resources/images/mountrushmore.jpg"
  actionListener="#{rushmore.handleMouseClick}"
  action="#{rushmore.act}"/>
```

После того как пользователь щелкнет на изображении лица президента, прослушиватель (имеющий доступ к координатам щелчка мышью) определяет, какой президент был выбран. Но этот прослушиватель не обеспечивает переход на другую страницу, поэтому сохраняет полученный им результат, соответствующий выбранному президенту, в поле экземпляра:

```
public class Rushmore {
  private String outcome;
  private Rectangle washingtonRect    = new Rectangle(70,30,40,40);
  private Rectangle jeffersonRect     = new Rectangle(115,45,40,40);
  private Rectangle rooseveltRect     = new Rectangle(135,65,40,40);
  private Rectangle lincolnRect       = new Rectangle(175,62,40,40);
```

```
    public void listen(ActionEvent e) {
        FacesContext context = FacesContext.getCurrentInstance();
        String clientId = e.getComponent().getClientId(context);
        Map requestParams = context.getExternalContext().
           getRequestParameterMap();

        int x = new Integer((String) requestParams.get(clientId + ".x")).intValue();
        int y = new Integer((String) requestParams.get(clientId + ".y")).intValue();

        outcome = null;

        if (washingtonRect.contains(new Point(x,y)))
           outcome = "washington";

        if (jeffersonRect.contains(new Point(x,y)))
           outcome = "jefferson";

        if (rooseveltRect.contains(new Point(x,y)))
           outcome = "roosevelt";

        if (lincolnRect.contains(new Point(x,y)))
           outcome = "lincoln";
    }
}
```

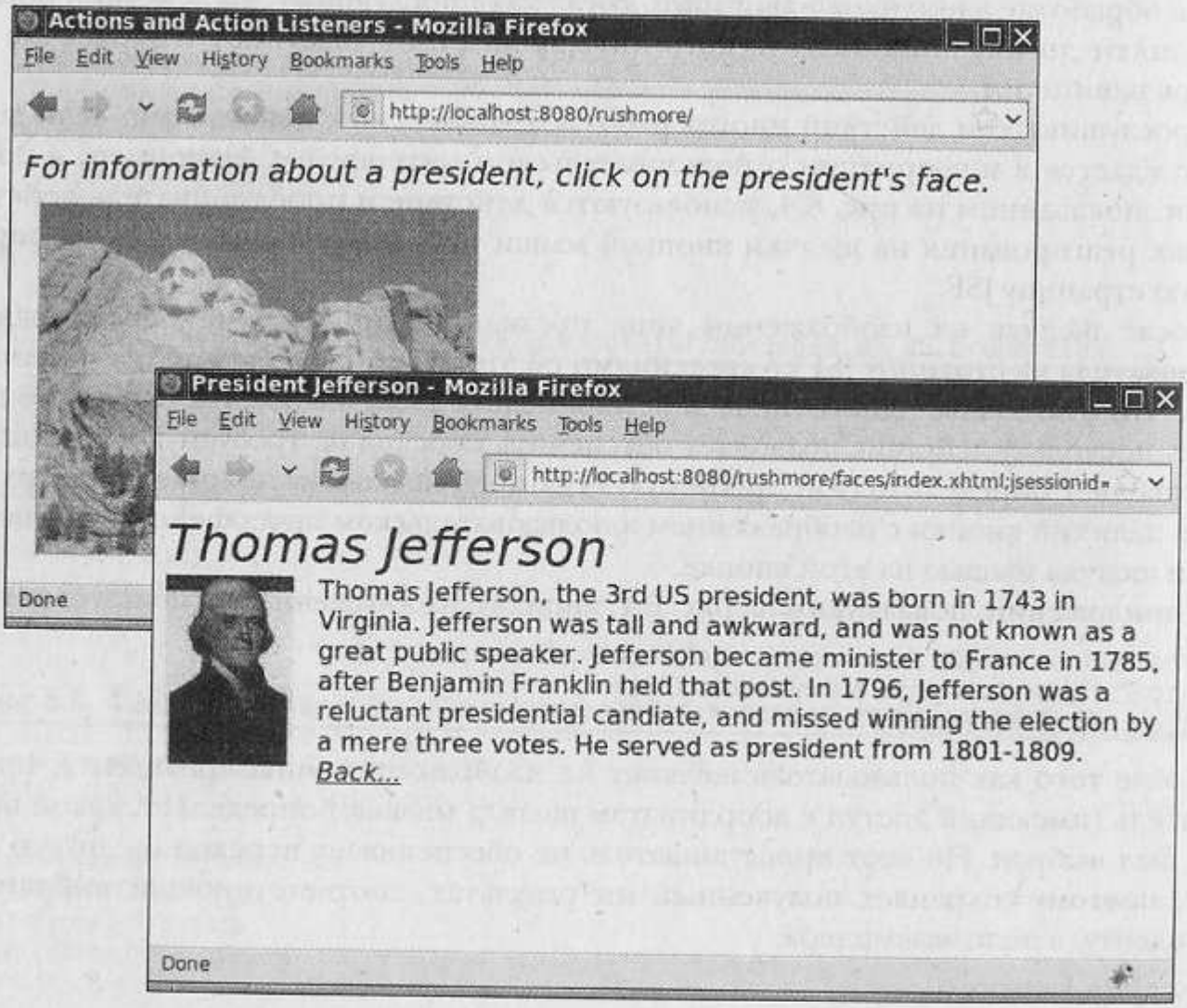


Рис. 8.4. Приложение Rushmore, посвященное соответствующему мемориалу

Действие, связанное с кнопкой, использует этот результат для осуществления требуемой навигации:

```
public String act() {
   return outcome;
}
```

Следует отметить, что в реализации JSF прослушиватели действий всегда вызываются перед действиями.

На заметку! Используя технологию JSF, нельзя не отделять логику пользовательского интерфейса и бизнес-логику, поскольку реализация JSF исключает возможность предоставления действиям доступа к событиям или компонентам, которые их активизируют. В предыдущем примере действие не может получить доступ к клиентскому идентификатору компонента, который активизировал событие, а этот идентификатор необходим для извлечения значений координат щелчка мышью из параметров запроса. Таким образом, в действии отсутствует какая-либо информация о пользовательском интерфейсе, поэтому мы обязаны дополнительно включить прослушиватель действий в данное приложение, чтобы реализовать требуемое поведение.

Структура каталогов для приложения, показанного на рис. 8.4, приведена на рис. 8.5. Код приложения содержится в листингах 8.6–8.9.

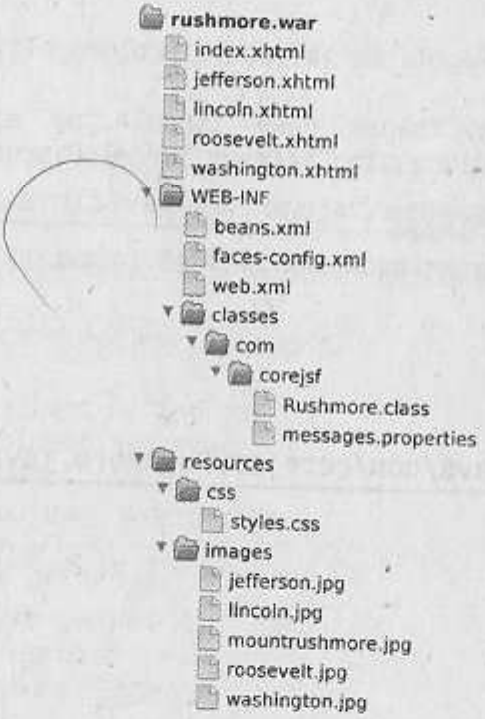


Рис. 8.5. Структура каталогов для примера приложения Rushmore

Листинг 8.6. Файл `rushmore/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.indexWindowTitle}</title>
   </h:head>

   <h:body>
      <span class="instructions">#{msgs.instructions}</span>
      <h:form>
         <h:commandButton image="/resources/images/mountrushmore.jpg"
```

```
                        styleClass="imageButton"
                        actionListener="#{rushmore.handleMouseClick}"
                        action="#{rushmore.navigate}"/>
        </h:form>
    </h:body>
</html>
```

Листинг 8.7. Файл rushmore/web/lincoln.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
       <h:outputStylesheet library="css" name="styles.css"/>
       <title>#{msgs.lincolnWindowTitle}</title>
    </h:head>

    <h:body>
       <h:form>
          <span class="presidentPageTitle">#{msgs.lincolnPageTitle}</span>
          <br/>
          <h:graphicImage library="images" name="lincoln.jpg" styleClass="leftImage"/>
          <span class="presidentDiscussion">#{msgs.lincolnDiscussion}</span>
          <br/>
          <h:commandLink action="index"
                         styleClass="backLink">${msgs.indexLinkText}</h:commandLink>
       </h:form>
    </h:body>
</html>
```

Листинг 8.8. Файл rushmore/src/java/com/corejsf/Rushmore.java

```
package com.corejsf;

import java.awt.Point;
import java.awt.Rectangle;
import java.util.Map;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.RequestScoped;
   // или import javax.faces.bean.RequestScoped;
import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;

@Named // или @ManagedBean
@RequestScoped
public class Rushmore {
   private String outcome = null;
   private Rectangle washingtonRect = new Rectangle(70, 30, 40, 40);
   private Rectangle jeffersonRect = new Rectangle(115, 45, 40, 40);
   private Rectangle rooseveltRect = new Rectangle(135, 65, 40, 40);
   private Rectangle lincolnRect = new Rectangle(175, 62, 40, 40);

   public void handleMouseClick(ActionEvent e) {
      FacesContext context = FacesContext.getCurrentInstance();
      String clientId = e.getComponent().getClientId(context);
      Map<String, String> requestParams
```

```
27.          = context.getExternalContext().getRequestParameterMap();
28.
29.       int x = new Integer((String) requestParams.get(clientId + ".x")).intValue();
30.       int y = new Integer((String) requestParams.get(clientId + ".y")).intValue();
31.
32.       outcome = null;
33.
34.       if (washingtonRect.contains(new Point(x, y)))
35.          outcome = "washington";
36.
37.       if (jeffersonRect.contains(new Point(x, y)))
38.          outcome = "jefferson";
39.
40.       if (rooseveltRect.contains(new Point(x, y)))
41.          outcome = "roosevelt";
42.
43.       if (lincolnRect.contains(new Point(x, y)))
44.          outcome = "lincoln";
45.    }
46.
47.    public String navigate() {
48.       return outcome;
49.    }
50. }
```

Листинг 8.9. Файл rushmore/src/java/com/corejsf/messages.properties

```
1. instructions=For information about a president, click on the president's face.
2.
3. indexWindowTitle=Actions and Action Listeners
4. indexLinkText=Back...
5. jeffersonWindowTitle=President Jefferson
6. rooseveltWindowTitle=President Roosevelt
7. lincolnWindowTitle=President Lincoln
8. washingtonWindowTitle=President Washington
9.
10. jeffersonPageTitle=Thomas Jefferson
11. rooseveltPageTitle=Theodore Roosevelt
12. lincolnPageTitle=Abraham Lincoln
13. washingtonPageTitle=George Washington
14.
15. lincolnDiscussion=President Lincoln was known as the Great Emancipator because \
16. he was instrumental in abolishing slavery in the United States. He was born \
17. into a poor family in Kentucky in 1809, elected president in 1860 and \
18. assassinated by John Wilkes Booth in 1865.
19.
20. washingtonDiscussion=George Washington was the first president of the United \
21. States. He was born in 1732 in Virginia and was elected Commander in Chief of \
22. the Continental Army in 1775 and forced the surrender of Cornwallis at Yorktown \
23. in 1781. He was inaugurated on April 30, 1789.
24.
25. rooseveltDiscussion=Theodore Roosevelt was the 26th president of the United \
26. States. In 1901 he became president after the assassination of President \
27. McKinley. At only 42 years of age, he was the youngest president in US history.
28.
29. jeffersonDiscussion=Thomas Jefferson, the 3rd US president, was born in \
30. 1743 in Virginia. Jefferson was tall and awkward, and was not known as a \
31. great public speaker. Jefferson became minister to France in 1785, after \
32. Benjamin Franklin held that post. In 1796, Jefferson was a reluctant \
33. presidential candiate, and missed winning the election by a mere three votes. \
34. He served as president from 1801-1809.
```

Теги прослушивателей событий

Выше в настоящей главе было показано, как добавлять действия и прослушиватели изменений значений к компонентам с помощью атрибутов actionListener и valueChangeListener соответственно. Но действия и прослушиватели изменений значений можно добавлять к компонентам также с помощью следующих тегов:

- f:actionListener.
- f:valueChangeListener.

Теги f:actionListener и f:valueChangeListener

Теги f:actionListener и f:valueChangeListener по своему назначению аналогичны атрибутам valueChangeListener и actionListener. В частности, в листинге 8.1 на стр. 272 меню было определено следующим образом:

```
<h:selectOneMenu value="#{form.country}" onchange="submit()"
   valueChangeListener="#{form.countryChanged}">
   <f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

Вместо этого можно было бы так использовать тег f:valueChangeListener:

```
<h:selectOneMenu value="#{form.country}" onchange="submit()">
   <f:valueChangeListener type="com.corejsf.CountryListener"/>
   <f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

Теги имеют одно преимущество перед атрибутами: они позволяют закреплять несколько прослушивателей за одним компонентом.

Обратите внимание на различие между значениями, указанными для атрибута valueChangeListener и тега f:valueChangeListener в приведенном выше коде. Первый определяет связывание метода, а последний задает класс Java. Например, класс, упомянутый в предыдущем фрагменте кода, выглядит следующим образом:

```
public class CountryListener implements ValueChangeListener {
   public void processValueChange(ValueChangeEvent event) {
      FacesContext context = FacesContext.getCurrentInstance();
      if ("US".equals(event.getNewValue()))
         context.getViewRoot().setLocale(Locale.US);
      else
         context.getViewRoot().setLocale(Locale.CANADA);
   }
}
```

Как и все прослушиватели, заданные с помощью тега f:valueChangeListener, указанный класс реализует интерфейс ValueChangeListener. Этот класс определяет единственный метод void processValueChange(ValueChangeEvent).

Тег f:actionListener аналогичен тегу f:valueChangeListener, но первый имеет также атрибут type, который задает имя класса; класс должен реализовывать интерфейс ActionListener. Например, в листинге 8.6 на стр. 277 кнопка была определена следующим образом:

```
<h:commandButton image="mountrushmore.jpg"
   styleClass="imageButton"
   actionListener="#{rushmore.handleMouseClick}"
   action="#{rushmore.navigate}"/>
```

Вместо использования атрибута actionListener для определения конкретного прослушивателя можно было бы применить тег f:actionListener:

```
<h:commandButton image="mountrushmore.jpg" action="#{rushmore.navigate}">
   <f:actionListener type="com.corejsf.RushmoreListener"/>
</h:commandButton>
```

Классы прослушивателя действий должны реализовывать интерфейс ActionListener, который определяет метод processAction, поэтому в предыдущем фрагменте кода реализация JSF вызывает метод RushmoreListener.processAction после активизации кнопки с изображением.

Применение нескольких тегов f:actionListener или f:valueChangeListener для одного компонента позволяет задать несколько прослушивателей. В частности, в предыдущем примере можно было бы добавить еще один прослушиватель действий, допустим, таким образом:

```
<h:commandButton image="mountrushmore.jpg" action="#{rushmore.navigate}">
   <f:actionListener type="com.corejsf.RushmoreListener"/>
   <f:actionListener type="com.corejsf.ActionLogger"/>
</h:commandButton>
```

В приведенном выше фрагменте кода класс ActionLogger представляет собой простой прослушиватель действий, который вносит в журнал события действия.

Если для компонента определено несколько прослушивателей, как в предыдущем фрагменте кода, то прослушиватели вызываются в следующем порядке.

1. Прослушиватель, определенный атрибутом listener.
2. Прослушиватели, определенные тегами прослушивателей, в том порядке, в котором они объявлены.

На заметку! У читателя может возникнуть вопрос, с чем связана необходимость задавать связывание методов для прослушивателей, если используются атрибуты actionListener и valueChangeListener, а также почему необходимо применять имя класса для прослушивателей, которые определены с помощью тегов f:valueChangeListener и f:actionListener. Это рассогласование между атрибутами listener и тегами явилось следствием недосмотра со стороны экспертной группы JSF.

Немедленно активизируемые компоненты

В разделе "События и жизненный цикл JSF" на стр. 270 было показано, что события изменения значения обычно активизируются после фазы проверок правильности, а события действия, как правило, активизируются после фазы вызова приложения. Вообще говоря, в этом состоит предпочтительное поведение. При обычных обстоятельствах желательно получение уведомлений об изменениях значений только тогда, когда они являются действительными, а действия должны вызываться после того, как все переданные значения были перенесены в модель.

Но иногда возникает необходимость в том, чтобы события изменения значений или события действий активизировались в начале жизненного цикла, поскольку нужно обойти проверку правильности для одного или нескольких компонентов. В разделе "Использование немедленно активизируемых компонентов ввода" текущей главы на стр. 282 и в разделе "Исключение процедуры проверки правильности" главы 7 на стр. 237 приведены убедительные аргументы в пользу подобного поведения. Но в данный момент рассмотрим суть того, как происходит доставка немедленно активизируемых событий, с помощью схемы, показанной на стр. 8.6

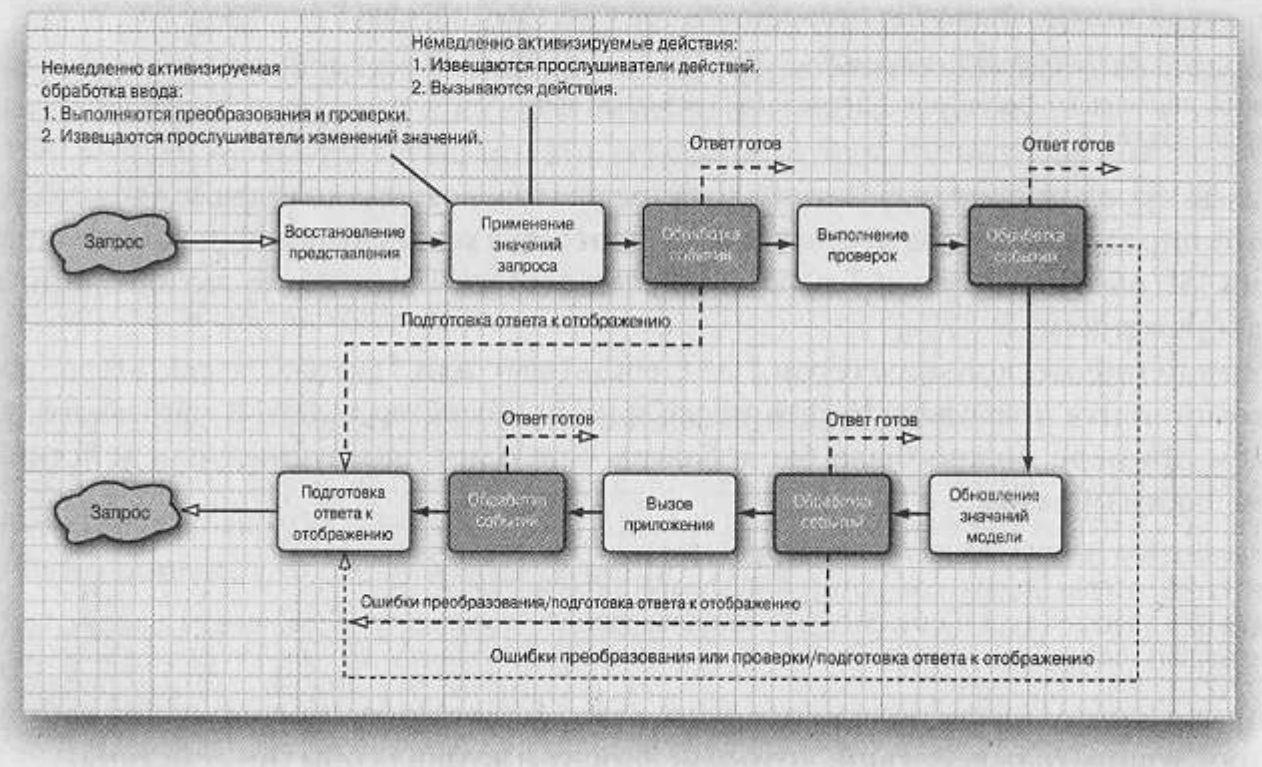


Рис. 8.6. Обработка событий для немедленно активизируемых компонентов

Компонент, имеющий заданный атрибут `immediate`, активизирует события после фазы применения значений запроса. Немедленно активизируемый компонент ввода выполняет преобразование и проверку правильности раньше, чем обычно, после завершения фазы применения значений запроса. Затем этот компонент активизирует событие изменения значения. Кроме того, немедленно активизируемый командный компонент вызывает прослушиватели действий и действия раньше, чем обычно, после фазы применения значений запроса. Этот процесс приводит к запуску обработчика навигации и исключению остальной части жизненного цикла, что влечет за собой переход непосредственно к фазе подготовки ответа к отображению.

Использование немедленно активизируемых компонентов ввода

На рис. 8.7 показан пример изменения значения, обсуждаемый в разделе "События изменения значения" на стр. 271. Напомним, что в рассматриваемом приложении прослушиватель изменений значений используется для изменения локали представления, что, в свою очередь, приводит к изменению локализованного приглашения указать штат в соответствии с выбранной локалью.

В данный момент в приложение было внесено безвредное с виду изменение: добавлен атрибут `required` к полю `Address`:

```
<h:inputText value="#{form.streetAddress}" required="true"/>
```

Но это приводит к ошибке при выборе страны без заполнения поля `Address` (напомним, что меню со списком стран организовано так, что при изменении выбранного в нем значения происходит передача формы).

Проблема состоит в следующем: указанная выше корректировка была сделана с целью запуска проверки правильности при активизации кнопки передачи формы, а не при выборе другой страны. Но как задать проверку правильности для одного элемента, но исключить ее для другого?

Рис. 8.7. Нежелательная проверка правильности

Решение заключается в том, что меню со списком стран должно быть реализовано как немедленно активизируемый компонент. Немедленно активизируемые компоненты ввода выполняют преобразование и проверку правильности, а затем доставляют события изменения значений в начале жизненного цикла JSF после фазы применения значений запроса, а не фазы осуществления проверок правильности.

Немедленно активизируемые компоненты задаются с помощью атрибута `immediate`, который применением для всех компонентов ввода и командных компонентов:

```
<h:selectOneMenu value="#{form.country}" onchange="submit()" immediate="true"
   valueChangeListener="#{form.countryChanged}">
   <f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

Если для атрибута `immediate` задано значение `true`, то применяемое меню будет активизировать события изменения значений после фазы применения значений запроса, т.е. задолго до того, как будет происходить проверка каких-либо иных компонентов ввода. У читателя может возникнуть вопрос: какая разница, будут ли остальные проверки правильности происходить позже, чем обычно, если в конечном итоге все проверки правильности будут выполнены и по-прежнему будут показаны обнаруженные ошибки проверки правильности? Чтобы предотвратить проверки правильности для других компонентов в форме, необходимо вызвать метод `renderResponse` класса `FacesContext` в конце работы применяемого прослушивателя изменений значений:

```
private static final String US = "United States";
...
public void countryChanged(ValueChangeEvent event) {
   FacesContext context = FacesContext.getCurrentInstance();
   if (US.equals((String) event.getNewValue()))
      context.getViewRoot().setLocale(Locale.US);
   else
      context.getViewRoot().setLocale(Locale.CANADA);

   context.renderResponse();
}
```

Вызов метода `renderResponse` приводит к пропуску последней части жизненного цикла, включая проверку правильности остальных компонентов ввода в форме, вплоть до фазы подготовки ответа к отображению. Таким образом, все прочие про-

верки правильности пропускаются, и ответ подготавливается к отображению обычным образом (в данном случае повторно отображается текущая страница).

Подведя итог, отметим, что можно пропустить проверку правильности при активизации события изменения значения, выполнив следующее.

1. Добавить атрибут `immediate` к тегу ввода.
2. Вызвать метод `renderResponse` класса `FacesContext` в конце работы прослушивателя.

Использование немедленно активизируемых командных компонентов

В главе 4 рассматривалось приложение, показанное на рис. 8.8, в котором используются командные ссылки для изменения локали.

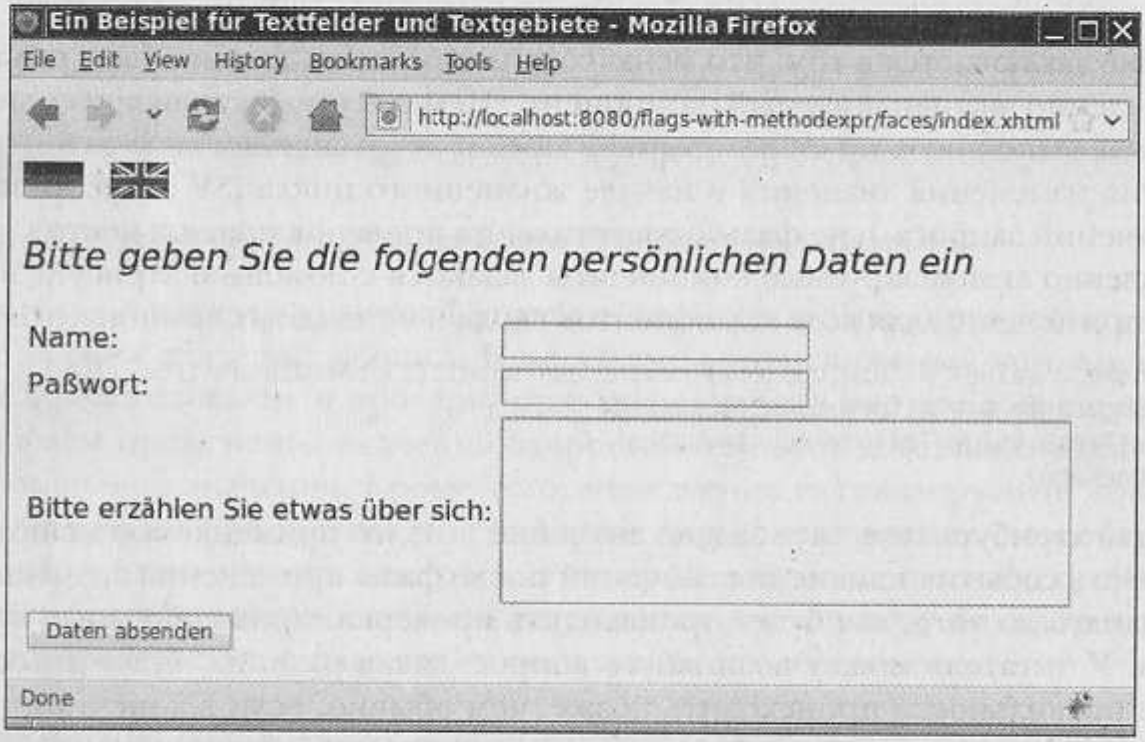


Рис. 8.8. Изменение локалей с помощью ссылок

Если бы мы добавили требуемое средство проверки к одному из полей ввода в форме, то столкнулись бы с той же проблемой, которая возникла в приложении, обсуждаемом в разделе "Использование немедленно активизируемых компонентов ввода" на стр. 282: ошибка проверки правильности появлялась бы даже при попытке изменить локаль щелчком на ссылке. Но на сей раз требуется немедленно активизируемый командный компонент, а не немедленно активизируемый компонент ввода. Для этого достаточно добавить атрибут `immediate` к применяемому тегу `h:commandLink` следующим образом:

```
<h:commandLink action="#{localeChanger.germanAction}" immediate="true">
   <h:graphicImage library="images" name="de_flag.gif" style="border: 0px"/>
</h:commandLink>
```

В отличие от той ситуации, когда речь идет о событиях изменения значений, мы не обязаны вносить корректировки в код прослушивателя, чтобы он вызывал метод `FacesContext.renderResponse()`, поскольку все действия, будь то немедленно активизи-

руемые или нет, передаются непосредственно в фазу подготовки ответа к отображению, независимо от того, когда они были активизированы.

Передача данных из пользовательского интерфейса на сервер

Два флага в приложении, показанном на рис. 8.8, реализованы с помощью ссылок. Ссылка для флага немецкого языка приведена в предыдущем разделе. Ниже показана ссылка для флага британского диалекта английского языка.

```
<h:commandLink action="#{localeChanger.englishAction}" immediate="true">
  <h:graphicImage library="images" name="en_flag.gif" style="border: 0px"/>
</h:commandLink>
```

Обратите внимание на то, что в каждой из ссылок имеются разные действия: действие `localeChanger.englishAction` для флага британского диалекта английского языка и действие `localeChanger.germanAction` для флага немецкого языка. Реализации этих действий являются весьма несложными:

```
public class LocaleChanger {
   public String germanAction() {
      FacesContext context = FacesContext.getCurrentInstance();
      context.getViewRoot().setLocale(Locale.GERMAN);
      return null;
   }

   public String englishAction() {
      FacesContext context = FacesContext.getCurrentInstance();
      context.getViewRoot().setLocale(Locale.ENGLISH);
      return null;
   }
}
```

Каждый метод действия задает локаль корневого каталога представления и возвращает значение `null` для указания на то, что реализация JSF должна повторно загрузить ту же страницу. Это довольно просто.

Но представьте себе необходимость поддерживать большое количество национальных языков; например, если бы приложение поддерживало 100 языков, то пришлось бы реализовать 100 действий, причем каждое действие было бы идентичным всем прочим за исключением заданной в нем локали. Это уже не так просто.

Чтобы уменьшить объем избыточного кода, который нужно было бы писать и сопровождать, лучше передавать код языка из пользовательского интерфейса на сервер. Таким образом, можно написать единственное действие или единственный прослушиватель действий для изменения локали корневого каталога представления. Технология JSF предоставляет четыре механизма передачи данных из пользовательского интерфейса на сервер.

- Параметры выражения метода (начиная с версии JSF 2.0).
- Тег `f:param`.
- Тег `f:attribute`.
- Тег `f:setPropertyActionListener` (начиная с версии JSF 1.2).

Теперь рассмотрим каждый тег по очереди, чтобы узнать, можно ли с его помощью устранить избыточный код.

Параметры выражения метода JSF 2.0

Начиная с версии JSF 2.0 выражения методов могут принимать параметры. Поэтому можно просто передать требуемую локаль в виде значения в метод действия, как в следующем примере:

```
<h:commandLink action="#{localeChanger.changeLocale('de')}">
   <h:graphicImage library="images" name="de_flag.gif" style="border: 0px"/>
</h:commandLink>
<h:commandLink action="#{localeChanger.changeLocale('en')}">
   <h:graphicImage library="images" name="en_flag.gif" style="border: 0px"/>
</h:commandLink>
```

На сервере метод `localeChanger` имеет параметр `languageCode`, применяемый для задания локали:

```
public class LocaleChanger {
   public String changeLocale(String languageChange) {
      FacesContext context = FacesContext.getCurrentInstance();
      context.getViewRoot().setLocale(new Locale(languageCode));
      return null;
   }
}
```

Независимо от того, сколько флагов будет размещено на странице JSF, код `LocaleChanger` по-прежнему остается применимым. Избыточного кода больше нет.

Тег `f:param`

Тег `f:param` позволяет закрепить параметр за компонентом. Способ интерпретации этого параметра зависит от типа компонента, за которым он закреплен. Например, если тег `f:param` закреплен за тегом `h:outputFormat`, то параметр определяет метки-заполнители, такие как {0}, {1} и т.д. Если же тег `f:param` закреплен за командным компонентом, таким как кнопка или ссылка, то параметр превращается в параметр запроса. Ниже показано, как можно было бы использовать тег `f:param` в рассматриваемом примере с флагами.

```
<h:commandLink immediate="true"
   action="#{localeChanger.changeLocale}">
   <f:param name="languageCode" value="de"/>
   <h:graphicImage library="images" name="de_flag.gif" style="border: 0px"/>
</h:commandLink>
<h:commandLink immediate="true"
   action="#{localeChanger.changeLocale}">
   <f:param name="languageCode" value="en"/>
   <h:graphicImage library="images" name="en_flag.gif" style="border: 0px"/>
</h:commandLink>
```

На сервере происходит доступ к параметру запроса `languageCode` для задания локали:

```
public class LocaleChanger {
   public String changeLocale() {
      FacesContext context = FacesContext.getCurrentInstance();
      String languageCode = getLanguageCode(context);
      context.getViewRoot().setLocale(new Locale(languageCode));
      return null;
   }
   private String getLanguageCode(FacesContext context) {
      Map<String, String> params = context.getExternalContext().
         getRequestParameterMap();
      return params.get("languageCode");
   }
}
```

Тег f:attribute

Еще один способ передачи информации из пользовательского интерфейса на сервер состоит в определении одного из атрибутов компонента с помощью тега f:attribute. Ниже показано, как это сделать в рассматриваемом примере с флагами.

```
<h:commandLink immediate="true"
   actionListener="#{localeChanger.changeLocale}">
   <f:attribute name="languageCode" value="de"/>
   <h:graphicImage library="images" name="de_flag.gif" style="border: 0px"/>
</h:commandLink>
<h:commandLink immediate="true"
   actionListener="#{localeChanger.changeLocale}">
   <f:attribute name="languageCode" value="en"/>
   <h:graphicImage library="images" name="en_flag.gif" style="border: 0px"/>
</h:commandLink>
```

Здесь заслуживают внимания две особенности. Во-первых, для задания атрибута на ссылке используется тег f:attribute. Имя этого атрибута — languageCode, а его значением является en или de.

Во-вторых, мы переключились с действия на прослушиватель действий. Это связано с тем, что прослушиватели действий передают объект события, предоставляющий доступ к компоненту, активизировавшему событие; разумеется, таковым является одна из применяемых ссылок. Этот компонент требуется для обеспечения доступа к его атрибуту languageCode. Ниже показано, как применить все описанные средства на сервере.

```
public class LocaleChanger {
   public void changeLocale(ActionEvent event) {
      UIComponent component = event.getComponent();
      String languageCode = getLanguageCode(component);
      FacesContext.getCurrentInstance()
          .getViewRoot().setLocale(new Locale(languageCode));
   }
   private String getLanguageCode(UIComponent component) {
      Map<String, Object> attrs = component.getAttributes();
      return (String) attrs.get("languageCode");
   }
}
```

На сей раз вместо выборки кода языка из параметра запроса происходит его выборка из атрибута компонента.

Тег f:setPropertyActionListener

Как уже было сказано, теги f:param и f:attribute являются удобным средством передачи информации из пользовательского интерфейса на сервер, но эти теги требуют от нас извлекать вручную сведения из параметра запроса или атрибута компонента.

Тег f:setPropertyActionListener, который был впервые введен в версии JSF 1.2, специально предназначен для того, чтобы положить конец этой ручной работе. Если используется тег f:setPropertyActionListener, то реализация JSF задает необходимое свойство в применяемом вспомогательном бине автоматически. Ниже показано, как воспользоваться этим средством в рассматриваемом примере с флагами.

```
<h:commandLink immediate="true" action="#{localeChanger.changeLocale}">
   <f:setPropertyActionListener target="#{localeChanger.languageCode}" value="de"/>
   <h:graphicImage library="images" name="de_flag.gif" style="border: 0px"/>
</h:commandLink>
<h:commandLink immediate="true" action="#{localeChanger.changeLocale}">
```

```
    <f:setPropertyActionListener target="#{localeChanger.languageCode}" value="en"/>
    <h:graphicImage library="images" name="en_flag.gif" style="border: 0px"/>
</h:commandLink>
```

В приведенном выше коде JSP было дано указание реализации JSF задавать свойство languageCode бина localeChanger равным de или en. Ниже приведена соответствующая реализация бина localeChanger.

```
public class LocaleChanger {
   private String languageCode;

    public String changeLocale() {
       FacesContext context = FacesContext.getCurrentInstance();
       context.getViewRoot().setLocale(new Locale(languageCode));
       return null;
    }
    public void setLanguageCode(String newValue) {
       languageCode = newValue;
    }
}
```

Для этой реализации LocaleChanger предусмотрено предназначенное только для чтения свойство languageCode, задаваемое реализацией JSF.

Очевидно, что в контексте рассматриваемого примера параметры метода являются наилучшим выбором для задания свойства languageCode бина localeChanger. Они являются простыми в реализации и доступными для понимания. А тег f:setPropertyActionListener, по-видимому, будет иметь ограниченное применение в приложениях JSF 2.0. Но теги f:param и f:attribute вполне применимы в других контекстах, когда необходимо задавать параметры запроса или атрибуты компонента.

События фазы

Реализация JSF активизирует события, называемые *событиями фазы*, до и после каждой фазы жизненного цикла. Эти события обрабатываются прослушивателями фаз. В отличие от прослушивателей изменения значений и прослушивателей действий, которые закрепляются за отдельными компонентами, прослушиватель фаз закрепляется за корневым каталогом представления. Прослушиватель фаз для отдельной страницы можно задать с помощью тега, размещенного в любом месте на этой странице:

```
<f:phaseListener type="com.corejsf.PhaseTracker"/>
```

Еще один вариант состоит в том, что можно определить глобальные прослушиватели фаз в файле конфигурации faces примерно так:

```
<faces-config>
   <lifecycle>
      <phase-listener>com.corejsf.PhaseTracker</phase-listener>
   </lifecycle>
</faces-config>
```

В предыдущем фрагменте кода определен только один прослушиватель, но предусмотрена возможность задавать любое необходимое количество прослушивателей. Прослушиватели вызываются в том порядке, в котором они указаны в файле конфигурации.

Прослушиватели фаз реализуются с помощью интерфейса PhaseListener из пакета javax.faces.event. Этот интерфейс определяет три метода.

- PhaseId getPhaseId().
- void afterPhase(PhaseEvent).
- void beforePhase(PhaseEvent).

Метод getPhaseId сообщает реализации JSF, когда события фазы должны передаваться прослушивателю; например, метод getPhaseId() может возвращать значения PhaseId.APPLY_REQUEST_VALUES. В этом случае методы beforePhase() и afterPhase() вызывались бы по одному разу за жизненный цикл: до и после фазы применения значений запроса. Можно также задать значение PhaseId.ANY_PHASE, которое фактически означает все фазы. Методы beforePhase и afterPhase применяемого прослушивателя фаз будут вызываться шесть раз на протяжении жизненного цикла: по одному разу для каждой фазы жизненного цикла.

Еще один вариант состоит в том, что страницу JSF можно заключить в тег f:view с атрибутами afterPhase или beforePhase. Эти атрибуты должны указывать на методы с сигнатурой void listener(javax.faces.event.PhaseEvent). Они вызываются перед каждой фазой, за исключением фазы восстановления представления. Например:

```
<f:view beforePhase="#{backingBean.beforeListener}">
   ...
</f:view>
```

Прослушиватели фаз применимы для средств отладки, а до выхода версии JSF 2.0 они представляли собой единственный механизм, позволяющий разрабатывать пользовательские компоненты, способные работать с фазами жизненного цикла JSF. Но мы предполагаем, что разработчики на JSF 2.0 предпочтут использовать системные события, которые обсуждаются в следующем разделе.

Системные события JSF 1.2

В версии JSF 2.0 впервые введена весьма детализированная система уведомления, в которой отдельные компоненты, а также реализация JSF передают извещения прослушивателям многих событий, потенциально представляющих интерес. Системные события JSF перечислены в табл. 8.1.

Таблица 8.1. Системные события

Класс события	Описание	Тип источника
PostConstructApplicationEvent, PreDestroyApplicationEvent	Непосредственно после запуска приложения; непосредственно перед тем, как приложение должно быть остановлено	Приложение
PostAddToViewEvent, PreRemoveFromViewEvent	После добавления компонента к дереву представлений; перед тем, как он должен быть удален	UIComponent
PostRestoreStateEvent	После восстановления состояния компонента	UIComponent
PreValidateEvent, PostValidateEvent	До и после проверки компонента	UIComponent
PreRenderViewEvent	Перед тем, как корневой каталог представления должен быть подготовлен к отображению	UIViewRoot

Окончание табл. 8.1

Класс события	Описание	Тип источника
PreRenderComponentEvent	Перед тем, как компонент должен быть подготовлен к отображению	UIComponent
PostConstructViewMapEvent, PreDestroyViewMapEvent	После создания компонентом корневого каталога карты области действия представления; после очистки карты представления[1]	UIViewRoot
PostConstructCustomScopeEvent, PreDestroyCustomScopeEvent	После создания пользовательской области действия; перед тем, как она должна быть уничтожена	ScopeContext
ExceptionQueuedEvent	После постановки исключительной ситуации в очередь	ExceptionQueuedEventContext

[1] Для контроля жизненного цикла приложения, сеанса и карты запросов используются прослушиватели ServletContextListener, ServletHttpSessionListener или ServletRequestListener.

Предусмотрены четыре способа, с помощью которых класс может получать системные события.

- С помощью тега f:event:

```
<inputText value="#{...}">
   <f:event name="postValidate" listener="#{bean.method}"/>
</inputText>
```

Метод должен иметь сигнатуру

```
public void listener(ComponentSystemEvent) throws AbortProcessingException
```

Это — самый удобный способ прослушивания событий компонента или представления.

- С применением аннотации для класса UIComponent или Renderer:

```
@ListenerFor(systemEventClass=PreRenderViewEvent.class)
```

Эти классы будут рассматриваться в главе 11. Указанный механизм может оказаться применимым для разработчиков компонентов.

- Будучи указанным как прослушиватель системных событий в файле faces-config.xml:

```
<application>
   <system-event-listener>
      <system-event-listener-class>listenerClass</system-event-listener-
class>
      <system-event-class>eventClass</system-event-class>
   </system-event-listener>
</application>
```

Этот механизм может применяться в целях установки прослушивателя для событий приложения.

- Путем вызова метода subscribeToEvent класса UIComponent или Application.

Данный метод предназначен для разработчиков платформ; подробные сведения см. в документации API JSF.

В следующих разделах будут рассматриваться два типичных примера использования системных событий.

Многокомпонентная проверка правильности

Как было указано в главе 7, в технологии JSF не предусмотрен механизм проверки группы компонентов. Например, если для ввода даты используются поля ввода дня, месяца и года, то невозможно воспользоваться каким-либо естественным способом проверки даты в целом. Чтобы преодолеть это ограничение, можно применить событие `PostValidateEvent`.

В настоящем разделе будет показано, как закрепить прослушиватель событий за панелью, которая содержит компоненты ввода:

```
<h:panelGrid id="date" columns="2">
   <f:event type="postValidate" listener="#{bb.validateDate}"/>
   #{msgs.day}
   <h:inputText id="day" value="#{bb.day}" size="2" required="true"/>

    #{msgs.month}
    <h:inputText id="month" value="#{bb.month}" size="2" required="true"/>

    #{msgs.year}
   <h:inputText id="year" value="#{bb.year}" size="4" required="true"/>
</h:panelGrid>
<h:message for="date" styleClass="errorMessage"/>
```

В прослушивателе событий происходит получение значений, введенных пользователем, и проверка того, образуют ли они допустимую дату. В случае отрицательного ответа к компоненту добавляется сообщение об ошибке и вызывается метод `renderResponse`:

```
public void validateDate(ComponentSystemEvent event) {
   UIComponent source = event.getComponent();
   UIInput dayInput = (UIInput) source.findComponent("day");
   UIInput monthInput = (UIInput) source.findComponent("month");
   UIInput yearInput = (UIInput) source.findComponent("year");
   int d = ((Integer) dayInput.getLocalValue()).intValue();
   int m = ((Integer) monthInput.getLocalValue()).intValue();
   int y = ((Integer) yearInput.getLocalValue()).intValue();
   if (!isValidDate(d, m, y)) {
      FacesMessage message = com.corejsf.util.Messages.getMessage(
         "com.corejsf.messages", "invalidDate", null);
      message.setSeverity(FacesMessage.SEVERITY_ERROR);
      FacesContext context = FacesContext.getCurrentInstance();
      context.addMessage(source.getClientId(), message);
      context.renderResponse();
   }
}
```

Обратите внимание на то, что применение метода `renderResponse` не приводит к немедленной подготовке ответа к отображению. Сначала осуществляется фаза проверок правильности; она включает обработку всех событий прослушивателей, которые следуют за проведением проверки правильности. Затем ответ подготавливается к отображению и снова выводится текущее представление с сообщением об ошибке (рис. 8.9).

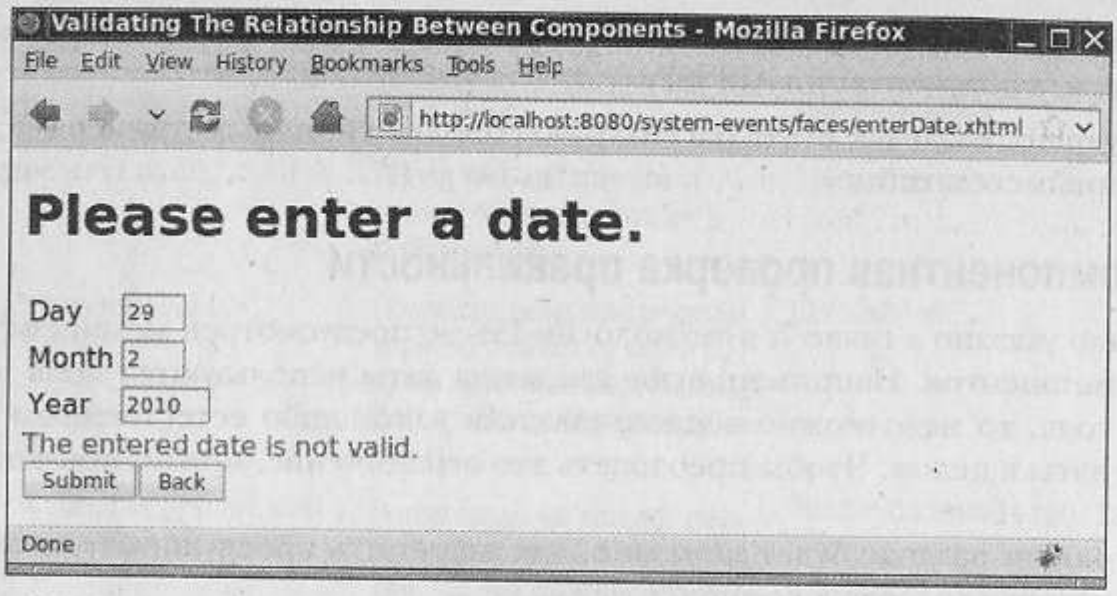


Рис. 8.9. Использование события `PostValidateEvent` для проверки группы компонентов

Принятие решений перед подготовкой представления к отображению

Иногда возникает необходимость получать определенные уведомления до того, как произойдет подготовка представления к отображению, например, в связи с необходимостью загрузить данные, внести изменения в компоненты на странице или по условию перейти на другую страницу.

Допустим, необходимо удостовериться, что пользователь действительно вошел в систему, и только после этого разворачивать перед ним какую-то конкретную страницу. В таком случае страницу следует заключить в тег `f:view` и закрепить за ней прослушиватель:

```
<f:view>
   <f:event type="preRenderView" listener="#{user.checkLogin}"/>
   <h:head>
      <title>...</title>
   </h:head>
   <h:body>
      ...
   </h:body>
</f:view>
```

В коде прослушивателя должна быть проведена проверка того, вошел ли пользователь в систему. В противном случае необходимо перейти к странице входа:

```
public void checkLogin(ComponentSystemEvent event) {
   if (!loggedIn) {
      FacesContext context = FacesContext.getCurrentInstance();
      ConfigurableNavigationHandler handler = (ConfigurableNavigationHandler)
         context.getApplication().getNavigationHandler();
      handler.performNavigation("login");
   }
}
```

В следующем примере приложения применяется сочетание проверки входа в систему и проверки правильности даты. Если пользователь в первую очередь загружает страницу `index.xhtml`, то вместо нее обработчик событий осуществляет переход на страницу `login.xhtml`. А после возврата пользователя на эту страницу после входа в

систему она отображается обычным образом. После входа в систему можно продолжить работу с представлением ввода даты.

На рис. 8.10 показана структура каталогов приложения. В листингах 8.10 и 8.11 показаны страницы с тегами `f:event`. В листингах 8.12 и 8.13 содержатся управляемые бины с обработчиками событий.

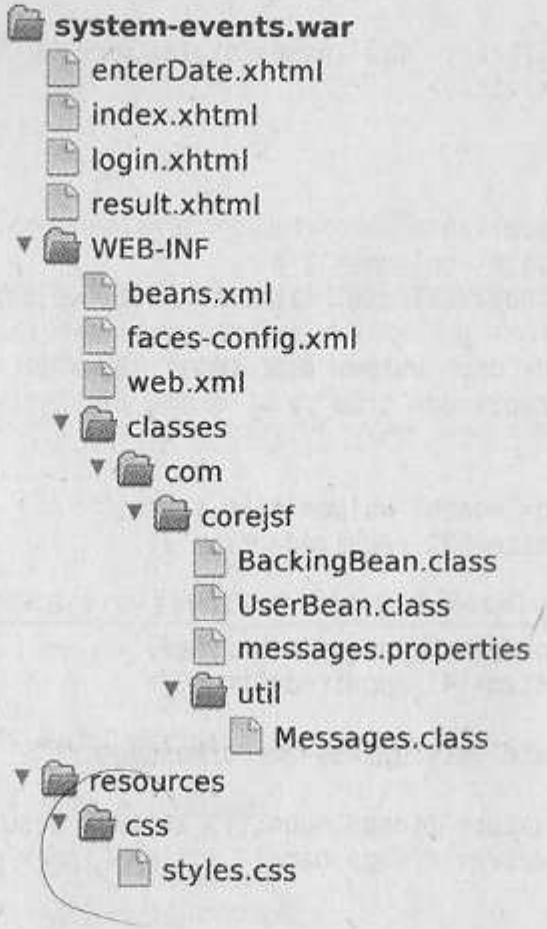


Рис. 8.10. Структура каталогов приложения с демонстрацией системных событий

Листинг 8.10. Файл `system-events/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
   <f:view>
      <f:event type="preRenderView" listener="#{user.checkLogin}"/>
      <h:head>
         <title>Welcome</title>
      </h:head>
      <h:body>
         <h3><h:outputText value="Welcome to JavaServer Faces, #{user.name}!" /></h3>
         <h:form>
            <h:commandButton value="Logout" action="#{user.logout}" />
            <h:commandButton value="Continue" action="enterDate" />
         </h:form>
      </h:body>
   </f:view>
</html>
```

Листинг 8.11. Файл system-events/web/enterDate.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <h1>#{msgs.enterDate}</h1>
         <h:panelGrid id="date" columns="2">
            <f:event type="postValidate" listener="#{bb.validateDate}"/>
            #{msgs.day}
            <h:inputText id="day" value="#{bb.day}" size="2"
                         required="true"/>

            #{msgs.month}
            <h:inputText id="month" value="#{bb.month}"
                         size="2" required="true"/>

            #{msgs.year}
            <h:inputText id="year" value="#{bb.year}"
                         size="4" required="true"/>
         </h:panelGrid>
         <h:message for="date" styleClass="errorMessage"/>
         <br/>
         <h:commandButton value="#{msgs.submit}" action="result"/>
         <h:commandButton value="#{msgs.back}" action="index" immediate="true"/>
      </h:form>
   </h:body>
</html>
```

Листинг 8.12. Файл system-events/src/java/com/corejsf/UserBean.java

```
package com.corejsf;

import javax.faces.application.ConfigurableNavigationHandler;
import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ComponentSystemEvent;

@Named("user") // или @ManagedBean(name="user")
@SessionScoped
public class UserBean {
   private String name = "";
   private String password;
   private boolean loggedIn;

   public String getName() { return name; }
   public void setName(String newValue) { name = newValue; }

   public String getPassword() { return password; }
```

```
23.    public void setPassword(String newValue) { password = newValue; }
24.
25.    public boolean isLoggedIn() { return loggedIn; }
26.
27.    public String login() {
28.       loggedIn = true;
29.       return "index";
30.    }
31.
32.    public String logout() {
33.       loggedIn = false;
34.       return "login";
35.    }
36.
37.    public void checkLogin(ComponentSystemEvent event) {
38.       if (!loggedIn) {
39.          FacesContext context = FacesContext.getCurrentInstance();
40.          ConfigurableNavigationHandler handler = (ConfigurableNavigationHandler)
41.             context.getApplication().getNavigationHandler();
42.          handler.performNavigation("login");
43.       }
44.    }
45. }
```

Листинг 8.13. Файл system-events/src/java/com/corejsf/BackingBean.java

```
1.  package com.corejsf;
2.
3.  import java.io.Serializable;
4.
5.  import javax.faces.application.FacesMessage;
6.  import javax.inject.Named;
7.     // или import javax.faces.bean.ManagedBean;
8.  import javax.enterprise.context.SessionScoped;
9.     // или import javax.faces.bean.SessionScoped;
10. import javax.faces.component.UIComponent;
11. import javax.faces.component.UIForm;
12. import javax.faces.component.UIInput;
13. import javax.faces.context.FacesContext;
14. import javax.faces.event.ComponentSystemEvent;
15. import javax.faces.validator.ValidatorException;
16.
17. @Named("bb") // или @ManagedBean(name="bb")
18. @SessionScoped
19. public class BackingBean implements Serializable {
20.    private int day;
21.    private int month;
22.    private int year;
23.
24.    public int getDay() { return day; }
25.    public void setDay(int newValue) { day = newValue; }
26.
27.    public int getMonth() { return month; }
28.    public void setMonth(int newValue) { month = newValue; }
29.
30.    public int getYear() { return year; }
31.    public void setYear(int newValue) { year = newValue; }
32.
33.    public void validateDate(ComponentSystemEvent event) {
34.       UIComponent source = event.getComponent();
35.       UIInput dayInput = (UIInput) source.findComponent("day");
```

```
36.       UIInput monthInput = (UIInput) source.findComponent("month");
37.       UIInput yearInput = (UIInput) source.findComponent("year");
38.       int d = ((Integer) dayInput.getLocalValue()).intValue();
39.       int m = ((Integer) monthInput.getLocalValue()).intValue();
40.       int y = ((Integer) yearInput.getLocalValue()).intValue();
41.       if (!isValidDate(d, m, y)) {
42.          FacesMessage message = com.corejsf.util.Messages.getMessage(
43.             "com.corejsf.messages", "invalidDate", null);
44.          message.setSeverity(FacesMessage.SEVERITY_ERROR);
45.             FacesContext context = FacesContext.getCurrentInstance();
46.             context.addMessage(source.getClientId(), message);
47.             context.renderResponse();
48.       }
49.    }
50.
51.    private static boolean isValidDate(int d, int m, int y) {
52.       if (d < 1 || m < 1 || m > 12) return false;
53.       if (m == 2) {
54.          if (isLeapYear(y)) return d <= 29;
55.          else return d <= 28;
56.       }
57.       else if (m == 4 || m == 6 || m == 9 || m == 11)
58.          return d <= 30;
59.       else
60.          return d <= 31;
61.    }
62.
63.    private static boolean isLeapYear(int y) {
64.       return y % 4 == 0 && (y % 400 == 0 || y % 100 != 0);
65.    }
66. }
```

Объединение рассматриваемых средств в одном приложении

В завершение главы будет приведен пример непритязательной реализации области окна с вкладками. Данный пример демонстрирует обработку событий и дополнительные аспекты использования тегов HTML JSF. К таким дополнительным средствам относятся следующие.

- Вложение тегов `h:panelGrid`.
- Использование аспектов.
- Определение индексации вкладок.
- Добавление подсказок к компонентам с атрибутом заголовка.
- Динамическое определение классов стиля.
- Использование прослушивателей действий.
- Необязательная подготовка к отображению.
- Включение страниц JSF.

В технологии JSF не предусмотрен компонент области окна с вкладками, поэтому, если в приложении требуется создать область окна с вкладками, придется воспользоваться одним из двух вариантов: реализовать пользовательский компонент или воспользоваться существующими тегами (со вспомогательным бином) для создания специальной области окна с вкладками. На рис. 8.11 показан последний вариант. Первый

из указанных вариантов обсуждается в разделе "Использование дочерних компонентов и аспектов" главы 11 на стр. 395.

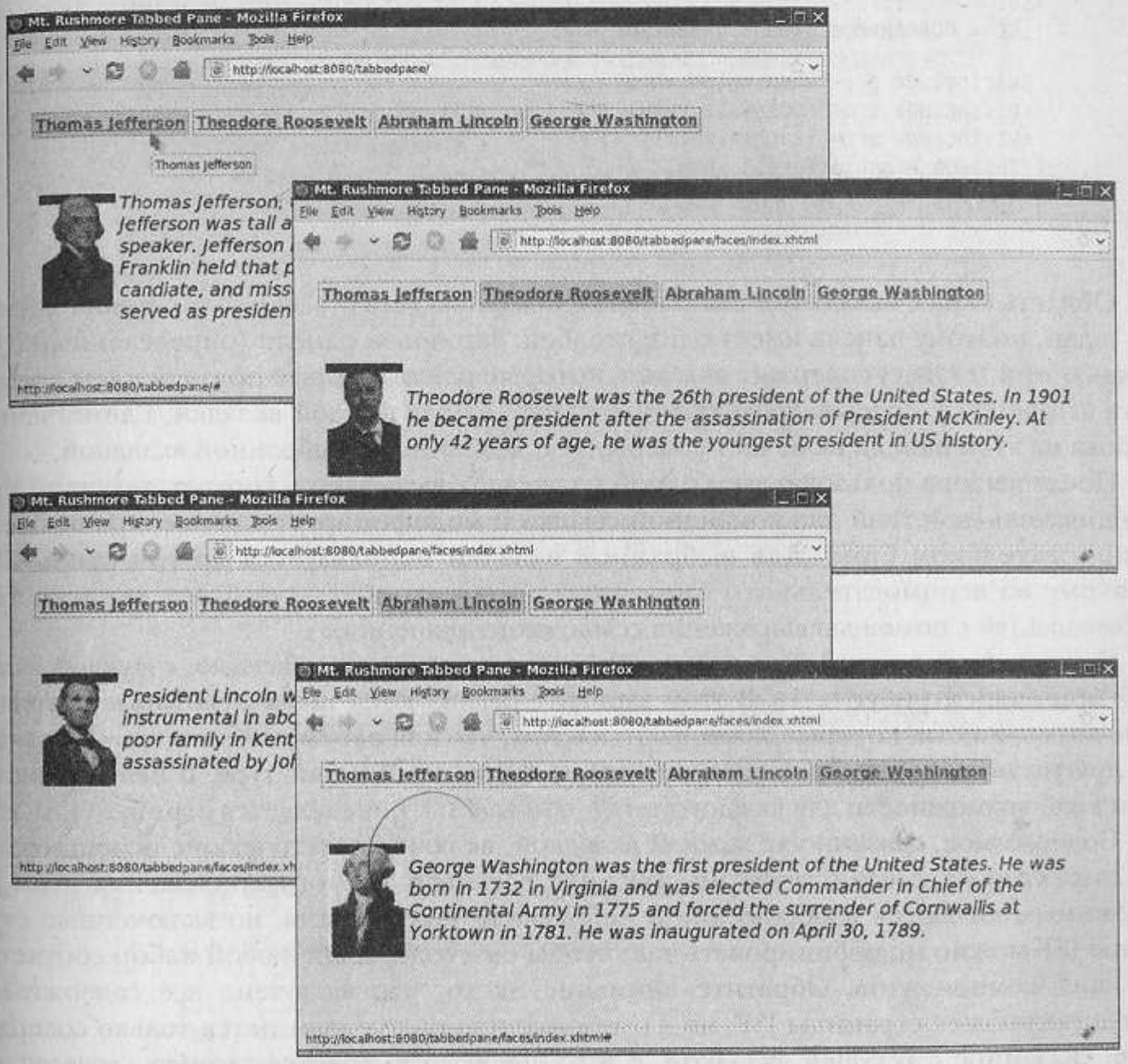


Рис. 8.11. Применение несложного способа реализации области окна с вкладками

Область окна с вкладками, показанная на рис. 8.11, реализована исключительно с использованием существующих тегов HTML JSF и вспомогательного бина; какие-либо пользовательские модули подготовки к отображению или компоненты не применяются. Страница JSF для области окна с вкладками выглядит следующим образом:

```
...
<h:form>
    <h:panelGrid styleClass="tabbedPane" columnClasses="displayPanel">
      <!-- Вкладки -->

      <f:facet name="header">
     <h:panelGrid columns="4" styleClass="tabbedPaneHeader">
       <h:commandLink tabindex="1"
           title="#{msgs.jeffersonTooltip}"
           styleClass="#{tp.jeffersonStyle}"
           actionListener="#{tp.jeffersonAction}">
           #{msgs.jeffersonTab}
       </h:commandLink>
```

```
            ...
        </h:panelGrid>
        </f:facet>

            <!-- Содержимое окна с вкладками -->

            <ui:include src="washington.xhtml" />
            <ui:include src="roosevelt.xhtml" />
            <ui:include src="lincoln.xhtml" />
          <ui:include src="jefferson.xhtml" />
      </h:panelGrid>
</h:form>
...
```

Область окна с вкладками выполнена с помощью тега `h:panelGrid`. Атрибут `columns` не задан, поэтому панель имеет один столбец. Заголовок панели (определенный с помощью тега `f:facet`) содержит вкладки, которые реализованы с применением другого тега `h:panelGrid`, содержащего теги `h:commandLink` для каждой вкладки. Единственная строка на этой панели включает содержимое, связанное с выбранной вкладкой.

После выбора пользователем одной из вкладок вызывается соответствующий прослушиватель действий для командной ссылки и модифицирует данные, хранимые во вспомогательном бине. Для выбранной вкладки используется другой стиль CSS, поэтому из вспомогательного бина извлекается атрибут `styleClass` каждого тега `h:commandLink` с помощью выражения ссылочного значения.

Как показано в верхней части рис. 8.11, для связывания подсказки с каждой вкладкой применен атрибут `title`. В этом варианте реализуются также некоторые специальные возможности, которые заключаются в том, что для перемещения с одной вкладки на другую может использоваться не только мышь, но и клавиатура. В целях реализации этой возможности для каждого тега `h:commandLink` определяется атрибут `tabindex`.

Содержимое, связанное с каждой вкладкой, включено статически с использованием директивы `include` платформы JSP. Для рассматриваемого приложения этим содержимым являются изображение и небольшой объем текста, но включенные страницы JSF можно модифицировать так, чтобы они содержали любой набор соответствующих компонентов. Обратите внимание на то, что включено все содержимое, представляющее страницы JSF, но к отображению подготавливается только содержимое, связанное с текущей вкладкой. В этих целях применяется атрибут `rendered`; например, файл `jefferson.xhtml` выглядит следующим образом:

```
<h:panelGrid columns="2" columnClasses="presidentDiscussionColumn"
   rendered="#{tp.jeffersonCurrent}">
   <h:graphicImage value="/images/jefferson.jpg"/>
   <span class="tabbedPaneContent">"#{msgs.jeffersonDiscussion}"</span>
</h:panelGrid>
```

На рис. 8.12 показана структура каталогов для приложения, в котором создается область окна с вкладками, а в листингах 8.14–8.17 приведены самые важные файлы.

Листинг 8.14. Файл `tabbedpane/web/index.xhtml`

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:ui="http://java.sun.com/jsf/facelets"
6.       xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
7.    <h:head>
8.       <h:outputStylesheet library="css" name="styles.css"/>
```

```
      <title>#{msgs.windowTitle}</title>
   </h:head>

   <h:body>
      <h:form>
         <h:panelGrid styleClass="tabbedPane" columnClasses="displayPanel">
            <!-- Вкладки -->

            <f:facet name="header">
               <h:panelGrid columns="4" styleClass="tabbedPaneHeader">
                  <h:commandLink tabindex="1" title="#{msgs.jeffersonTooltip}"
                                 styleClass="#{tp.jeffersonStyle}"
                                 actionListener="#{tp.jeffersonAction}">
                     #{msgs.jeffersonTabText}
                  </h:commandLink>

                  <h:commandLink tabindex="2" title="#{msgs.rooseveltTooltip}"
                                 styleClass="#{tp.rooseveltStyle}"
                                 actionListener="#{tp.rooseveltAction}">
                     #{msgs.rooseveltTabText}
                  </h:commandLink>

                  <h:commandLink tabindex="3" title="#{msgs.lincolnTooltip}"
                                 styleClass="#{tp.lincolnStyle}"
                                 actionListener="#{tp.lincolnAction}">
                     #{msgs.lincolnTabText}
                  </h:commandLink>

                  <h:commandLink tabindex="4" title="#{msgs.washingtonTooltip}"
                                 styleClass="#{tp.washingtonStyle}"
                                 actionListener="#{tp.washingtonAction}">
                     #{msgs.washingtonTabText}
                  </h:commandLink>
               </h:panelGrid>
            </f:facet>

            <!-- Содержимое окна с вкладками -->

            <ui:include src="washington.xhtml"/>
            <ui:include src="roosevelt.xhtml"/>
            <ui:include src="lincoln.xhtml"/>
            <ui:include src="jefferson.xhtml"/>
         </h:panelGrid>
      </h:form>
   </h:body>
</html>
```

Листинг 8.15. Файл tabbedpane/web/jefferson.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
   <ui:composition>
      <h:panelGrid columns="2" columnClasses="presidentDiscussionColumn"
                   rendered="#{tp.jeffersonCurrent}">

         <h:graphicImage library="images" name="jefferson.jpg"/>
         <span class="tabbedPaneContent">#{msgs.jeffersonDiscussion}</span>
```

```
13.
14.         </h:panelGrid>
15.     </ui:composition>
16. </html>
```

Рис. 8.12. Структура каталогов для примера области окна с вкладками

Листинг 8.16. Файл tabbedpane/src/java/com/corejsf/messages.properties

```
1. windowTitle=Mt. Rushmore Tabbed Pane
2. lincolnTooltip=Abraham Lincoln
3. lincolnTabText=Abraham Lincoln
4. lincolnDiscussion=President Lincoln was known as the Great Emancipator because \
5. he was instrumental in abolishing slavery in the United States. He was born \
6. into a poor family in Kentucky in 1809, elected president in 1860 and \
7. assassinated by John Wilkes Booth in 1865.
8.
9. washingtonTooltip=George Washington
10. washingtonTabText=George Washington
11. washingtonDiscussion=George Washington was the first president of the United \
12. States. He was born in 1732 in Virginia and was elected Commander in Chief of \
13. the Continental Army in 1775 and forced the surrender of Cornwallis at Yorktown \
14. in 1781. He was inaugurated on April 30, 1789.
15.
16. rooseveltTooltip=Theodore Roosevelt
17. rooseveltTabText=Theodore Roosevelt
18. rooseveltDiscussion=Theodore Roosevelt was the 26th president of the United \
19. States. In 1901 he became president after the assassination of President \
20. McKinley. At only 42 years of age, he was the youngest president in US history.
21.
22. jeffersonTooltip=Thomas Jefferson
23. jeffersonTabText=Thomas Jefferson
24. jeffersonDiscussion=Thomas Jefferson, the 3rd US president, was born in \
25. 1743 in Virginia. Jefferson was tall and awkward, and was not known as a \
26.   great public speaker. Jefferson became minister to France in 1785, after \
27.   Benjamin Franklin held that post. In 1796, Jefferson was a reluctant \
```

```
28.    presidential candiate, and missed winning the election by a mere three votes. \
29.    He served as president from 1801-1809.
```

Листинг 8.17. Файл tabbedpane/src/java/com/corejsf/TabbedPane.java

```
1.  package com.corejsf;
2.
3.  import java.io.Serializable;
4.
5.  import javax.inject.Named;
6.     // или import javax.faces.bean.ManagedBean;
7.  import javax.enterprise.context.SessionScoped;
8.     // или import javax.faces.bean.SessionScoped;
9.  import javax.faces.event.ActionEvent;
10.
11. @Named("tp") // или @ManagedBean(name="tp")
12. @SessionScoped
13. public class TabbedPane implements Serializable {
14.    private int index;
15.    private static final int JEFFERSON_INDEX = 0;
16.    private static final int ROOSEVELT_INDEX = 1;
17.    private static final int LINCOLN_INDEX = 2;
18.    private static final int WASHINGTON_INDEX = 3;
19.
20.    private String[] tabTooltips = { "jeffersonTooltip", "rooseveltTooltip",
21.          "lincolnTooltip", "washingtonTooltip" };
22.
23.    public TabbedPane() {
24.       index = JEFFERSON_INDEX;
25.    }
26.
27.    // Прослушиватели действий, которые задают текущую вкладку
28.
29.    public void jeffersonAction(ActionEvent e) { index = JEFFERSON_INDEX; }
30.    public void rooseveltAction(ActionEvent e) { index = ROOSEVELT_INDEX; }
31.    public void lincolnAction(ActionEvent e) { index = LINCOLN_INDEX;     }
32.    public void washingtonAction(ActionEvent e) { index = WASHINGTON_INDEX; }
33.
34.    // Стили CSS
35.
36.    public String getJeffersonStyle() { return getCSS(JEFFERSON_INDEX); }
37.    public String getRooseveltStyle() { return getCSS(ROOSEVELT_INDEX); }
38.    public String getLincolnStyle() { return getCSS(LINCOLN_INDEX);     }
39.    public String getWashingtonStyle() { return getCSS(WASHINGTON_INDEX); }
40.
41.     private String getCSS(int forIndex) {
42.        return forIndex == index ? "tabbedPaneTextSelected" : "tabbedPaneText";
43.     }
44.
45.     // Методы определения текущей вкладки
46.
47.     public   boolean   isJeffersonCurrent() { return index == JEFFERSON_INDEX; }
48.     public   boolean   isRooseveltCurrent() { return index == ROOSEVELT_INDEX; }
49.     public   boolean   isLincolnCurrent() { return index == LINCOLN_INDEX;     }
50.     public   boolean   isWashingtonCurrent() { return index == WASHINGTON_INDEX; }
51. }
```

Резюме

Данный пример завершает приведенное в настоящей главе введение в обработку событий. В следующей главе будет показано, как объединять стандартные компоненты JSF для создания собственных составных компонентов.

Составные компоненты JSF 2.0

В этой главе...

- Библиотека составных тегов
- Использование составных компонентов
- Реализация составных компонентов
- Настройка конфигурации составных компонентов
- Типы атрибутов
- Атрибуты required и значения атрибутов по умолчанию
- Манипулирование серверными данными
- Локализация составных компонентов
- Обеспечение доступа к отдельным компонентам составных компонентов
- Аспекты
- Дочерние теги
- Применение кода JavaScript
- Вспомогательные компоненты
- Упаковка составных компонентов в файлах JAR

Глава 9

В отличие от платформ, основанных на применении действий, таких как Struts или Ruby on Rails, платформа JSF является компонентно-ориентированной, а это означает, что с ее помощью можно реализовать компоненты, предназначенные для многократного использования самим разработчиком или другими программистами. Компоненты представляют собой мощный механизм, обеспечивающий многократное использование готового программного обеспечения.

Но при использовании версии JSF 1.0 добиться полноценного применения компонентов было почти невозможно по двум причинам. Во-первых, задача реализации компонентов была сложной. Для этого приходилось писать код Java и задавать настройку в формате XML. Необходимо было также хорошо знать все особенности жизненного цикла JSF.

Во-вторых, в версии JSF 1.0 не было предусмотрено никаких средств, которые позволяли бы легко составлять новые компоненты из существующих. Если программист реализовывал простой компонент поля, в который, допустим, входили приглашение (`h:outputText`) и поле ввода (`h:inputText`), то был вынужден создавать эти компоненты и манипулировать в коде Java.

В версии JSF 2.0 оба эти недостатка устранены благодаря тому, что упростилась возможность реализовывать пользовательские компоненты в коде Java, а также введены в действие новые средства составления новых компонентов из существующих. В настоящей главе подробно рассматривается последнее из указанных средств, которое принято называть созданием составных компонентов.

На заметку! В версии JSF 2.0 для обозначения компонентов, для реализации которых применяется библиотека составных компонентов, принято использовать название "составные компоненты". В технологии JSF используется термин "составной", поскольку новые компоненты составляются из существующих компонентов.

Задача реализации составных компонентов является несложной. Для этого не приходится писать какой-либо код Java или задавать конфигурацию в формате XML. Во многих сценариях создания составных компонентов бывает достаточно подготовить простой файл XHTML, который определяет компонент.

Библиотека составных тегов

В состав версии JSF 2.0 входит библиотека тегов, предназначенная для реализации составных компонентов. В соответствии с принятым соглашением разработчики используют префикс `composite:` для этой библиотеки. Чтобы воспользоваться библиотекой составных тегов, добавьте объявление пространства имен к файлу XHTML, как в следующем примере:

```
<html xmlns="http://www.w3.org/1999/xhtml"...
  xmlns:composite="http://java.sun.com/jsf/composite">
  ...
</html>
```

После определения пространства имен для библиотеки составных компонентов можно использовать любой из тегов, приведенных в табл. 9.1.

На заметку! Мы проводим различие между *автором компонента* — разработчиком, который проектирует и реализует составной компонент, и *авторами страниц* — разработчиками, использующими этот компонент на своих страницах.

Таблица 9.1. Теги составных компонентов

Тег	Описание	Где используется
`interface`	Содержит другие составные теги, которые предоставляют доступ к атрибутам составного компонента, источникам действий, хранителям значений, редактируемым хранителям значений и аспектам	Не определено
`implementation`	Содержит разметку XHTML, которая определяет компонент. В теге реализации автор компонента может обращаться к атрибутам с помощью выражения `#{cc.attrs.attributeName}`	Не определено
`attribute`	Предоставляет авторам страниц доступ к атрибуту компонента	Интерфейс
`valueHolder`	Предоставляет авторам страниц доступ к компоненту, который хранит значение	Интерфейс
`editableValueHolder`	Предоставляет авторам страниц доступ к компоненту, который хранит редактируемое значение	Интерфейс
`actionSource`	Предоставляет авторам страниц доступ к компоненту, который активизирует события действия, такие как кнопки или ссылки	Интерфейс
`facet`	Объявляет, что этот компонент поддерживает аспект с указанным именем	Интерфейс
`extension`	Автор компонента может поместить этот тег в любом элементе интерфейса. Тег `extension` может содержать произвольный код XML	Подэлемент интерфейса
`insertChildren`	Вставляет любые дочерние компоненты, определенные автором страницы	Реализация
`renderFacet`	Подготавливает к отображению аспект, который был указан автором страницы как дочерний компонент	Реализация
`insertFacet`	Вставляет аспект, определенный автором страницы, как аспект включающего компонента	Реализация

Как показывают первые два тега в табл. 9.1, составные компоненты имеют интерфейсы и реализации. Реализации представляют собой разметку Facelet, в которой используются стандартные теги JSF. Интерфейсы позволяют обеспечить доступ к настраиваемым характеристикам конкретного составного компонента.

В качестве примера можно указать составной компонент входа в систему, который реализован как форма с приглашениями к вводу имени и пароля и с кнопкой отправки формы. Для реализации этого компонента с именем `login` могут применяться теги `h:form`, `h:inputText`, `h:commandButton` и так далее, как и при создании любой формы в представлении Facelets.

Но для обеспечения повторного использования составные компоненты требуют чего-то большего, чем просто реализация, поскольку они должны также быть настраиваемыми. Например, что касается рассматриваемого компонента `login`, то, по всей вероятности, потребуется также обеспечить настройку меток, связанных с полями имени и пароля; закрепить средство проверки за одним или обоими полями, а также прослушиватель действий — за кнопкой отправки формы компонента `login`. Все эти задачи можно решить с помощью интерфейса разрабатываемого компонента `login`.

Наконец, программист приступает к реальному созданию компонентов, поэтому необходимо каким-то образом зарегистрировать их на платформе JSF. К счастью, в версии JSF 2.0 такая настройка конфигурации выполняется автоматически, как будет показано ниже.

Использование составных компонентов

Для удаления настройки в версии JSF 2.0 используется соглашение об именах для составных компонентов. Чтобы проиллюстрировать это соглашение об именах, воспользуемся составным компонентом, показанным на рис. 9.1, который отображает сведения о текущем запросе HTTP.

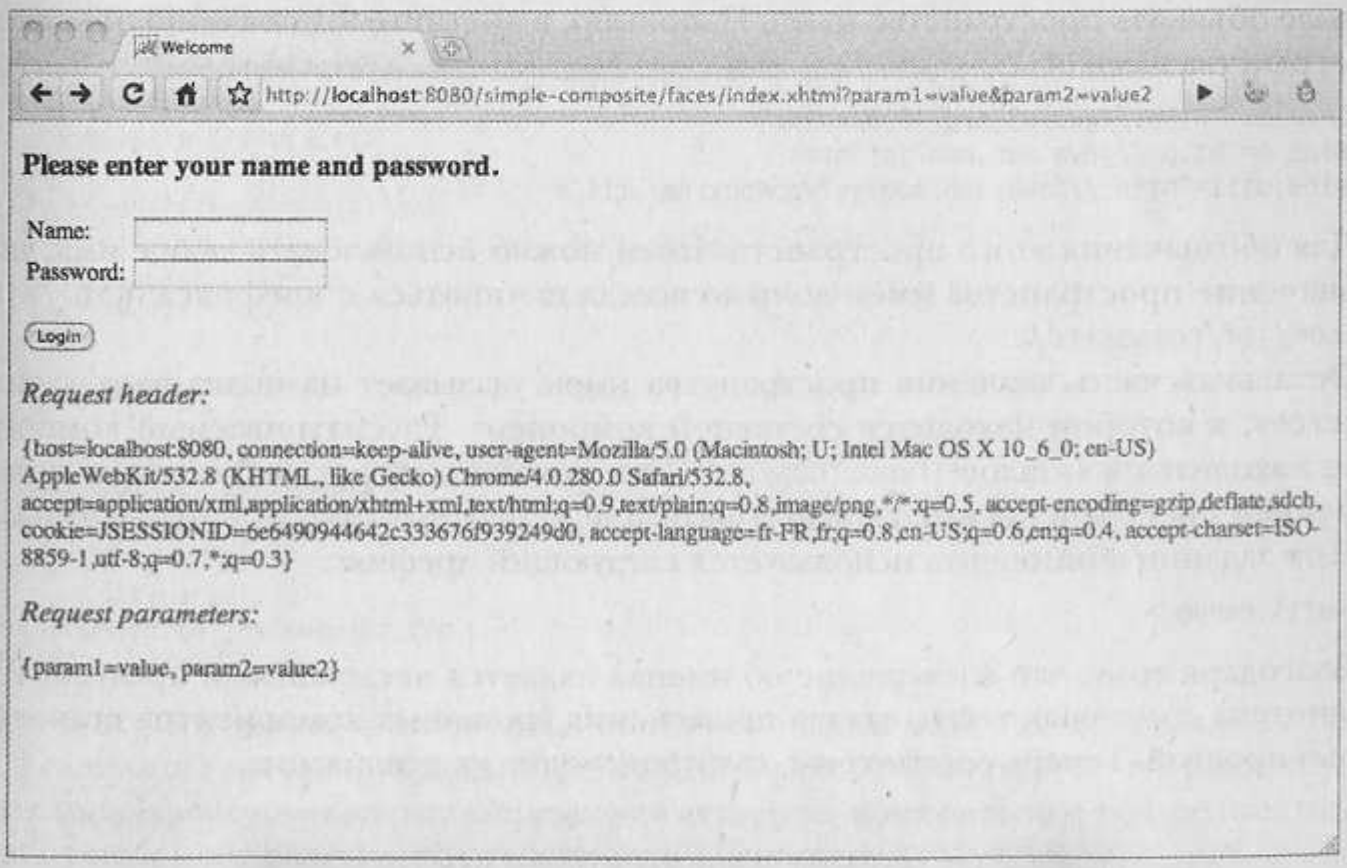


Рис. 9.1. Использование отладочного компонента debug

Тег пользовательского компонента <util:debug/> показан в конце листинга 9.1.

Листинг 9.1. Файл simple-composite/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:util="http://java.sun.com/jsf/composite/util">
   <h:head>
      <title>Welcome</title>
   </h:head>
   <h:body>
      <h:form>
         <h3>Please enter your name and password.</h3>
         <table>
            <tr>
               <td>Name:</td>
               <td><h:inputText value="#{user.name}"/></td>
            </tr>
            <tr>
               <td>Password:</td>
               <td><h:inputSecret value="#{user.password}"/></td>
            </tr>
         </table>
         <p><h:commandButton value="Login" action="welcome"/></p>
      </h:form>
      <div style="color: red;">
         <util:debug />
      </div>
   </h:body>
</html>
```

Чтобы иметь возможность воспользоваться составным компонентом, необходимо вначале объявить пространство имен. Например, в листинге 9.1 показано объявление пространства имен util:

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:util="http://java.sun.com/jsf/composite/util">
```

Для обозначения этого пространства имен можно использовать любое имя, но само значение пространства имен должно всегда начинаться с префикса http://java.sun.com/jsf/composite/.

Остальная часть значения пространства имен указывает на подкаталог каталога resources, в котором находится составной компонент. Рассматриваемый компонент debug находится в каталоге resources/util, как показано на рис. 9.2, поэтому полным значением пространства имен должно быть http://java.sun.com/jsf/composite/util.

При задании компонента используется следующий префикс:

```
<util:debug/>
```

Благодаря тому, что соглашение об именах является несложным и предусмотрена библиотека составных тегов, задача применения составных компонентов становится весьма простой. Теперь рассмотрим, как происходит их реализация.

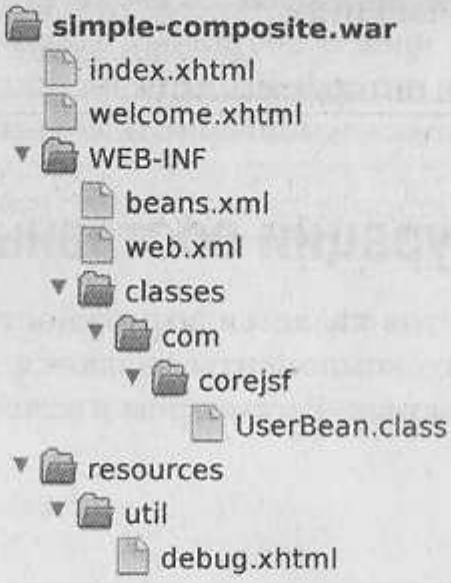


Рис. 9.2. Структура каталогов приложения с компонентом debug

Реализация составных компонентов

Реализация составных компонентов является почти такой же простой, как и их использование; доказательством этого может служить листинг 9.2, который показывает реализацию компонента debug, приведенного в листинге 9.1.

Листинг 9.2. Файл simple-composite/web/resources/util/debug.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:composite="http://java.sun.com/jsf/composite">

    <composite:interface/>

    <composite:implementation>
        <div style="font-size: 1.2em; font-style: italic">
        Request header:
        </div>

        <p>#{header}</p>

        <div style="font-size: 1.2em; font-style: italic">
        Request parameters:
        </div>

       <p>#{param}</p>
    </composite:implementation>
</html>
```

Как и все составные компоненты, компонент debug имеет интерфейс и реализацию. Реализация составного компонента представляет собой просто его разметку, тогда как интерфейс компонента определяет атрибуты компонента с той целью, чтобы разработчики могли настраивать составные компоненты.

Компонент debug ничего не определяет в своем интерфейсе, хотя такая ситуация является редкой, когда речь идет о составных компонентах. Почти все составные компоненты с помощью своих интерфейсов предоставляют доступ к своим атрибутам, чтобы можно было их настраивать и повторно использовать в различных контекстах.

Настройка конфигурации составных компонентов

Преимуществом компонентов является возможность их многократного использования, обусловленная тем, что компоненты являются настраиваемыми в целях применения в разных обстоятельствах. Рассмотрим в качестве примера значки, показанные на рис. 9.3.

Рис. 9.3. Значки

Значки имеют два атрибута: изображение image и действие action, вызываемое при щелчке на изображении. Авторам страниц может быть предоставлена возможность задавать эти два атрибута примерно так:

```
<util:icon image="#{resource['images:back.jpg']}"
   actionMethod="#{user.logout}" />
```

В следующем файле icon.xhtml определен простой компонент icon:

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:composite="http://java.sun.com/jsf/composite">

   <composite:interface>
      <composite:attribute name="image"/>
      <composite:attribute name="actionMethod"
         method-signature="java.lang.String action()" />
   </composite:interface>

   <composite:implementation>
      <h:form>
         <h:commandLink action="#{cc.attrs.actionMethod}">
         <h:graphicImage url="#{cc.attrs.image}" styleClass="icon" />
         </h:commandLink>
      </h:form>
   </composite:implementation>
</html>
```

В этой реализации значка осуществляется доступ к атрибутам image и actionMethod с помощью выражений #{cc.attrs.image} и #{cc.attrs.actionMethod} соответственно. Здесь cc представляет составной компонент, а выражение cc.attrs проходит специальную обработку в языке выражений, чтобы можно было искать атрибуты компонента. Таким образом, выражение #{cc.attrs.attributeName} позволяет получить доступ к атрибуту attributeName составного компонента.

Следует учитывать, что эти два атрибута компонента icon весьма значительно отличаются друг от друга: атрибут image "#{resource['images:back.jpg']}" представляет собой выражение значения, а атрибут actionMethod "#{user.logout}" является выражением метода. При вычислении выражения #{cc.attrs.attributeName} реализация JSF проверяет, является ли значение с ключом attributeName выражением значения.

В случае положительного ответа выражение вычисляется. В противном случае просто возвращается значение, связанное с ключом.

На заметку! Если атрибут составного компонента ссылается на выражение метода, то необходимо задать сигнатуру метода, чтобы реализации JSF было известно, что применяется ссылка на имя метода, а не имя свойства. В рассматриваемом случае это правило относится к атрибуту значка `actionMethod`.

Типы атрибутов

Как и в примере со значками, атрибуты можно определять с помощью выражений значения таким образом:

```
<util:icon image="#{resource['images:back.jpg']}"
  actionMethod="#{user.logout}" />
```

По умолчанию в реализации JSF предполагается, что значения атрибута имеют тип `java.lang.Object`. Например, в приведенном выше коде вычисление выражения значения `#{resource['images:back.jpg']}` приводит к получению URL, представленного как строка: `/context-root/faces/javax.faces.resource/back.jpg?ln=images`. В действительности эту строку можно также определить непосредственно:

```
<util:icon image="/composite-login/faces/javax.faces.resource/back.jpg?ln=images"
  actionMethod="#{user.logout}" />
```

Если необходимо, чтобы составной атрибут представлял подкласс типа `java.lang.Object`, то следует указать реализации JSF, каковым является тип атрибута. Один из способов осуществления этого состоит в использовании атрибута `method-signature` тега `composite:attribute`, как было показано в листинге 9.1. Если для атрибута определена сигнатура метода, то реализация JSF преобразует выражение значения атрибута в выражение метода вместо объекта.

Еще один способ определения типа атрибута состоит в использовании атрибута типа `composite:attribute`. Атрибут типа `type` должен быть указан с полным именем класса Java. Так, например, если бы потребовалось задать атрибут `date`, значением которого является объект `Date`, то можно было бы применить следующий тег: `<composite:attribute name ="date" type="java.util.Date" />`.

На заметку! И тип тега `composite:attribute`, и атрибуты `method-signature` сообщают реализации JSF, что значение атрибута представляет собой нечто иное по сравнению со строкой. С помощью атрибута типа `type` можно задать любой тип, тогда как атрибут `method-signature` всегда указывает, что тип — это объект выражения метода JSF.

Атрибуты `type` и `method-expression` для тега `composite:attribute` являются взаимоисключающими, причем атрибут `type` рассматривается как более приоритетный по сравнению с атрибутом `method-signature`, если по недосмотру программист определит оба атрибута.

Атрибуты `required` и значения атрибутов по умолчанию

Компонент `icon`, реализованный в листинге 9.1, является довольно удобным. С его помощью можно добиться того, чтобы значки выглядели и действовали в полном со-

ответствии с предъявленными к ним требованиями. Но существует возможность еще больше повысить удобство работы со значками следующим образом.

- Сделать атрибут `image` обязательным.
- Разрешить автору страницы определять класс CSS для изображения.
- Разрешить автору страницы активизировать проверку правильности при вызове действия значка.

В листинге 9.3 показан обновленный код реализации значка, в котором воплощены указанные выше возможности.

Листинг 9.3. Файл `/composite-login/web/resources/util/icon.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:composite="http://java.sun.com/jsf/composite">

    <composite:interface>
      <composite:attribute name="image" required="true" />
      <composite:attribute name="doValidation" default="false" />
      <composite:attribute name="styleClass" default="icon" />
      <composite:attribute name="actionMethod"
         method-signature="java.lang.String action()" />
    </composite:interface>

    <composite:implementation>
    <h:form>
      <h:commandLink action="#{cc.attrs.actionMethod}"
         immediate="#{not cc.attrs.doValidation}">

         <h:graphicImage url="#{cc.attrs.image}"
             styleClass="#{cc.attrs.styleClass}" />

      </h:commandLink>
    </h:form>
    </composite:implementation>

</html>
```

Безусловно, встречаются крайние случаи, в которых приходится применять невидимые значки, но приведенный пример рассчитан на подавляющее большинство случаев применения значков, поэтому к авторам страниц предъявляется требование предоставить для значка изображение с помощью атрибута `required` тега изображения `composite:attribute`.

Мы также предоставляем авторам страницы задавать стиль CSS для изображения значка с помощью атрибута `styleClass`. Мы определяем значение значка по умолчанию для этого атрибута.

Наконец, автору страницы предоставляется контроль над тем, должен ли щелчок на значке активизировать проверку правильности ввода на сервере. По умолчанию в ссылке на значок атрибут `immediate` задан равным `true`; при этом значении атрибут `immediate` указывает, что проверка правильности должна быть пропущена.

Если же необходимо, чтобы при щелчке на значке происходила проверка правильности, то следует указать это явно: `<util:icon doValidation="true".../>`.

Манипулирование серверными данными

Компонент icon, показанный в листинге 9.1, представляет больший интерес по сравнению с компонентом debug, с описания которого началась эта глава, поскольку он является настраиваемым и поэтому в большей степени подходит для повторного использования, чем компонент debug. Но в компоненте icon недостает нечто из того, что предусмотрено во многих нетривиальных составных компонентах: взаимодействие с серверными данными.

На рис. 9.4 показан компонент login, который взаимодействует с управляемым бином.

Компонент login, который показан на рис. 9.4, может использоваться следующим образом:

```
<util:login namePrompt="#{msgs.namePrompt}"
       passwordPrompt="#{msgs.passwordPrompt}"
          loginAction="#{user.login}"
      loginButtonText="#{msgs.loginButtonText}"
                 user="#{user}"/>
```

Автор страницы передает ссылку на пользовательский управляемый бин в дополнение к другим атрибутам, с помощью которых настраиваются приглашения и кнопка компонента login.

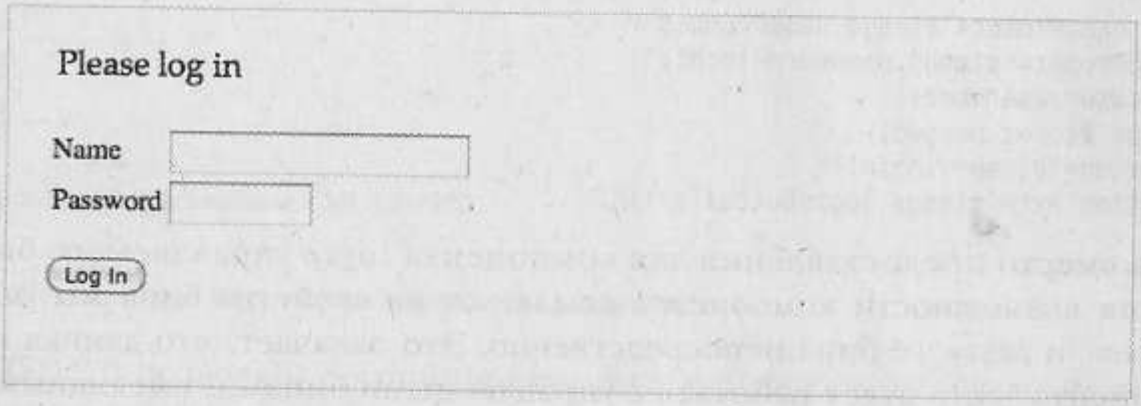


Рис. 9.4. Компонент login

Простой компонент login можно реализовать следующим образом:

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:composite="http://java.sun.com/jsf/composite">

    <composite:interface>
       <composite:attribute user="user"/>

       <composite:attribute name="namePrompt"/>
       <composite:attribute name="passwordPrompt"/>

       <composite:attribute name="loginButtonText"/>
       <composite:attribute name="loginAction"
          method-signature="java.lang.String action()"/>
    </composite:interface>

    <composite:implementation>
       <h:form id="form">
          <h:panelGrid columns="2">
             #{cc.attrs.namePrompt}
             <h:inputText id="name" value="#{cc.attrs.user.name}"/>
```

```
            #{cc.attrs.passwordPrompt}
            <h:inputSecret id="password" value="#{cc.attrs.user.password}"/>
        </h:panelGrid>
        <p>
            <h:commandButton id="loginButton" value="#{cc.attrs.loginButtonText}"
               action="#{cc.attrs.loginAction}"/>
        </p>
    </h:form>
  </composite:implementation>
</html>
```

В компоненте `login` объявляется пользовательский атрибут, затем этот атрибут используется для входных значений имени, `#{cc.attrs.user.name}`, и пароля, `#{cc.attrs.user.password}`.

Одной из стратегий обеспечения взаимодействия с серверными данными является передача управляемого бина составному компоненту. Однако применение этой стратегии по отношению к компоненту `login` из листинга 9.3 приводит к созданию слишком тесной связи: компонент `login` может работать только с управляемыми бинами, имеющими свойства `name` и `password`.

Чтобы компонент `login` стал доступнее для более широкого диапазона управляемых бинов, допустим, для тех, которые имеют свойства `username` и `passwd`, можно, в частности, применить более детализированный подход, как в следующем примере:

```
<util:login namePrompt="#{msgs.namePrompt}"
    passwordPrompt="#{msgs.passwordPrompt}"
    name="#{user.username}"
    password="#{user.passwd}"
    loginAction="#{user.login}"
    loginButtonText="#{msgs.loginButtonText}"/>
```

Теперь вместо предоставления для компонента `login` управляемого бина `user` и применения возможности компонента ссылаться на свойства бина мы определяем свойства `name` и `password` бина непосредственно. Это означает, что данная новая версия компонента `login` будет работать с управляемыми бинами, имеющими свойства с любыми именами.

Необходимо внести следующие изменения в определение компонента:

```
<composite:interface>
  <composite:attribute name="name"/>
  <composite:attribute name="password"/>
  ...
</composite:interface>

<composite:implementation>
  ...
  #{cc.attrs.namePrompt}
  <h:inputText id="name" value="#{cc.attrs.name}"/>

  #{cc.attrs.passwordPrompt}
  <h:inputSecret id="password" value="#{cc.attrs.password}"/>
  ...
</composite:implementation>
```

После этого авторы страниц имеют возможность связывать отдельные свойства управляемого бина, в рассматриваемом случае `#{user.username}` и `#{user.passwd}`, с полями ввода, созданными компонентом `login`.

Итак, в нашем распоряжении теперь имеется вспомогательный бин (`user`) со свойствами (`name` и `password`), привязанными к полям ввода в компоненте `login`. Компонент

login начинает действительно приобретать черты полезного компонента. Однако к этому моменту он все еще имеет один серьезный недостаток, из-за которого становится менее привлекательным, чем можно было бы ожидать: пока что отсутствует возможность закреплять средства проверки за полями ввода имени и пароля. В разделе "Предоставление доступа к компонентам составного компонента" на стр. 316 будет показано, как именно это можно сделать.

Локализация составных компонентов

В большинстве случаев для этого достаточно предоставить авторам страниц возможность настраивать текст, отображаемый составными компонентами, но иногда это становится нецелесообразным. Например, если бы было решено продавать часть свободной площади в компоненте login для размещения рекламы и в связи с этим заняться поиском жаждущих получения такой возможности рекламных агентов, то можно было бы отобразить некоторый текст в нижней части компонента, как показано на рис. 9.5.

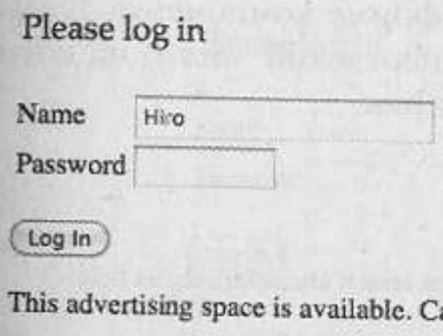


Рис. 9.5. Локализация текста в составных компонентах

Версия JSF 2.0 позволяет ассоциировать связку ресурсов с составным компонентом, чтобы можно было локализовать текст, отображаемый в составном компоненте. В каталоге составного компонента создается файл properties с именем component_name.properties, где component_name — имя составного компонента. Для компонента login был создан файл login.properties, как показано на рис. 9.6.

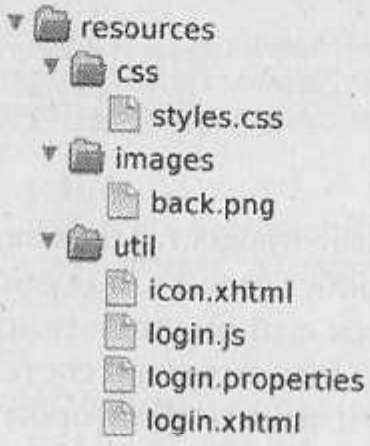


Рис. 9.6. Файл properties компонента login

Рассматриваемый файл login.properties содержит только одну пару "ключ–значение", как показано в листинге 9.4.

Листинг 9.4. Файл `composite-login/web/resources/util/login.properties`

```
1. footer=This advertising space is available. Call 555-1212.
```

Со времени создания файла `properties` появляется возможность обращаться к его содержимому с помощью выражения `#{cc.resourceBundleMap.key}`, где `key` — ключ из файла `properties`. В рассматриваемом компоненте `login` была применена следующая конструкция:

```
<composite:implementation>
   ...
   <p>#{cc.resourceBundleMap.footer}</p>
   ...
</composite:implementation>
```

Обеспечение доступа к отдельным компонентам составных компонентов

Теперь внесем в компонент `login` такие изменения, чтобы можно было закреплять средства проверки за полями ввода имени и пароля в форме компонента `login`. На рис. 9.7 показан результат намеренного допущения ошибки в той части, на которую распространяется действие одного из этих средств проверки.

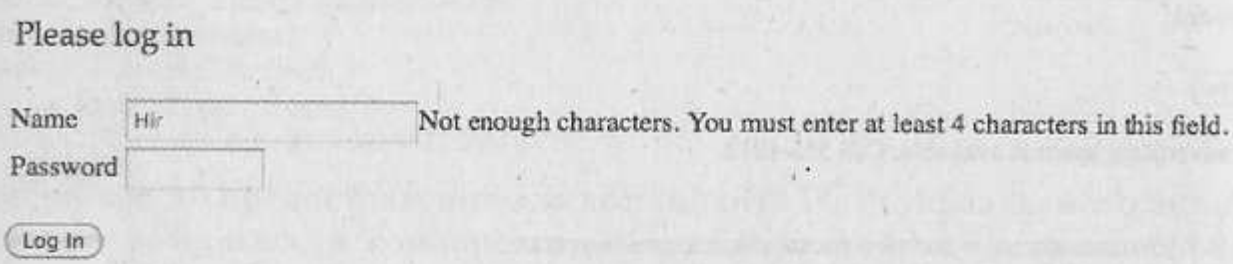


Рис. 9.7. Результат добавления средства проверки к одному из полей ввода компонента `login`

Мы добавляем три тега `composite:editableValueHolder` к интерфейсу рассматриваемого компонента `login`. Первые два ссылаются на поля ввода имени и пароля, а третий ссылается на оба поля ввода:

```
<composite:interface>
   <composite:editableValueHolder name="nameInput" targets="form:name"/>
   <composite:editableValueHolder name="passwordInput" targets="form:password"/>
   <composite:editableValueHolder name="inputs" targets="form:name form:password"/>
   ...
</composite:interface>
```

Тег `composite:editableValueHolder` предоставляет автору страницы доступ к компонентам, на которые ссылается атрибут `targets`, под именем, указанным с помощью атрибута `name`. Компоненты задаются с применением идентификаторов компонентов. Эти идентификаторы определены относительно составного компонента, а поскольку оба поля ввода находятся в форме с идентификатором `form`, необходимо сопроводить оба идентификатора компонента префиксом `form:`.

Теперь, после того как мы предоставили авторам страниц доступ к компонентам ввода имени и пароля, появилась возможность использовать эти компоненты примерно так:

```
<util:login namePrompt="#{msgs.namePrompt}"
      passwordPrompt="#{msgs.passwordPrompt}"
```

```
      name="#{user.name}"
      password="#{user.password}"
      loginAction="#{user.login}"
      loginButtonText="#{msgs.loginButtonText}">

    <f:validateLength maximum="10" for="inputs"/>
    <f:validateLength minimum="4" for="nameInput"/>
    <f:validator id="com.corejsf.Password" for="passwordInput"/>

</util:login>
```

В предыдущем фрагменте кода были добавлены три средства проверки к полям ввода в компоненте формы `form`. Первые два средства проверки задают максимальное количество в 10 символов для полей ввода имени и пароля и минимальное количество в 4 символа для поля ввода имени. Третьим средством проверки является пользовательское средство проверки, которое закрепляется за полем ввода пароля и обеспечивает проверку пароля на наличие запрещенных символов. На рис. 9.8 показан результат неудачного завершения этой проверки правильности.

Код данного средства проверки, закрепленного за полем ввода пароля, показан в листинге 9.5.

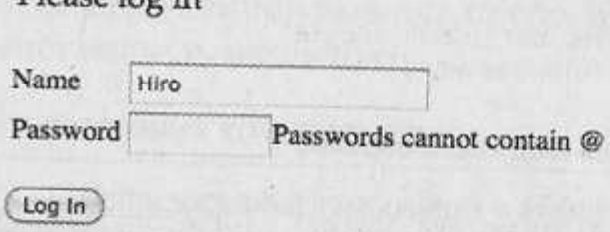


Рис. 9.8. Проверка пароля на наличие запрещенных символов (следует отметить, что реализация JSF очищает поле ввода пароля перед отображением этого сообщения об ошибке)

Листинг 9.5. Файл `composite-login/src/java/com/corejsf/PasswordValidator.java`

```
1. package com.corejsf;
2.
3. import javax.faces.application.FacesMessage;
4. import javax.faces.component.UIComponent;
5. import javax.faces.context.FacesContext;
6. import javax.faces.validator.FacesValidator;
7. import javax.faces.validator.Validator;
8. import javax.faces.validator.ValidatorException;
9.
10. @FacesValidator("com.corejsf.Password")
11. public class PasswordValidator implements Validator {
12.    public void validate(FacesContext context, UIComponent component, Object value)
13.          throws ValidatorException {
14.       String pwd = (String) value;
15.       if (pwd.contains("@")) {
16.          throw new ValidatorException(new FacesMessage("Passwords cannot contain @"));
17.       }
18.    }
19. }
```

Замысел средства проверки, приведенного в предыдущем листинге, является довольно несложным: в нем осуществляется лишь проверка на наличие в пароле символов @. Но можно довольно легко дополнить эту логику, чтобы ввести другие проверки

На заметку! В предыдущем примере идентификаторы компонентов по существу были заменены псевдонимами:

```
<composite:interface>
        ...
        <composite:editableValueHolder name="passwordInput"
            targets="form:password"/>
        <composite:editableValueHolder name="inputs"
            targets="form:name form:password"/>
        ...
    </composite:interface>
```

Атрибут `name` служит псевдонимом для атрибута `targets`, поэтому авторы страниц могут ссылаться на имя, а не применять более громоздкую конструкцию на основе компонента `id`, указанного для атрибута `targets`:

```
<util:login ...>
        <f:validator binding="#{passwordValidator}" for="passwordInput"/>
        <f:validateLength maximum="10" for="inputs"/>
    </util:login>
```

Если адресаты `targets` не заданы, то автор страницы должен определить полное имя компонента. Например, если интерфейс является таковым:

```
<composite:interface>
        <composite:editableValueHolder name="form:password"/> <!--
     Целевое значение отсутствует -->
        <composite:editableValueHolder name="inputs"
            targets="form:name form:password"/>
        ...
```

то ссылка на поле пароля осуществляется следующим образом:

```
<util:login...>
        <f:validator binding="#{passwordValidator}" for="form:password"/>
    </util:login>
```

Предоставление доступа к источникам действий

Кроме тега `composite:editableValueHolder`, в состав библиотеки составных тегов входят еще два тега, которые можно использовать для предоставления авторам страниц доступа к компонентам, содержащимся в составных компонентах: `composite:valueHolder` и `composite:actionSource`.

Тег `composite:valueHolder` обеспечивает доступ к компонентам, подобным компонентам вывода, которые имеют нередактируемое значение. Тег `composite:actionSource` обеспечивает доступ к таким компонентам, как кнопки и ссылки, которые активизируют события действия. Например, мы имеем возможность обращаться к кнопке отправки формы компонента `login`:

```
<composite:interface>
   <composite:actionSource name="loginButton" targets="form:loginButton"/>
   ...
</composite:interface>"

<composite:implementation>
   ...
   <h:form id="form"... >
      ...
      <h:commandButton id="loginButton"
         value="#{cc.attrs.loginButtonText}"
         action="#{cc.attrs.loginAction}"/>
      ...
   </h:form>
</composite:implementation>
```

Со времени предоставления доступа к кнопке отправки формы авторы страниц могут использовать эту кнопку, как показано ниже.

```
<util:login ...>
  <f:actionListener for="loginButton" type="com.corejsf.LoginActionListener"/>
</util:login>
```

В версии JSF 2.0 к тегу `f:actionListener` добавлен атрибут `for`. Значение этого атрибута ссылается на источник действий, доступ к которому предоставляется во включающем его компоненте `login`.

Результатом закрепления прослушивателя действий за кнопкой отправки формы компонента `login` становится то, что после щелчка на кнопке отправки формы компонента `login` реализация JSF создает экземпляр объекта `com.corejsf.LoginActionListener` и вызывает его метод `processAction`.

В рассматриваемом примере приложения предусмотрено применение объекта `LoginActionListener` (показанного в листинге 9.6), который проверяет, допустимы ли заданные значения имени и пароля пользователя. (На практике было бы проще обрабатывать нарушения процесса входа в систему с помощью метода `action`; в данном случае авторы намереваются лишь показать всю организацию работы.)

В табл. 9.2 перечислены теги, которые можно использовать в разделе интерфейса конкретного составного компонента для предоставления доступа к компонентам, содержащимся в составном компоненте.

Таблица 9.2. Теги составных компонентов

Тег интерфейса	Использование компонента с этими тегами на применяемой странице
`actionSource`	`f:actionListener`
`valueHolder`	`f:converter`, `f:convertDateTime`, `f:setPropertyActionListener`, `f:validate...`
`editableValueHolder`	`f:converter`, `f:convertDateTime`, `f:setPropertyActionListener`, `f:validate...`, `f:valueChangeListener`

В правом столбце табл. 9.2 перечислены теги, имеющие, подобно `f:actionListener`, атрибут `for`, применимый для ссылки на компонент в составном компоненте, доступ к которому предоставляется через интерфейс компонента с тегами в левом столбце табл. 9.2.

Листинг 9.6. Файл `composite-login/src/java/com/corejsf/LoginActionListener.java`

```
1. package com.corejsf;
2.
3. import javax.faces.application.FacesMessage;
4. import javax.faces.component.UIComponent;
5. import javax.faces.component.UIInput;
6. import javax.faces.context.FacesContext;
7. import javax.faces.event.AbortProcessingException;
8. import javax.faces.event.ActionEvent;
9. import javax.faces.event.ActionListener;
10.
11. public class LoginActionListener implements ActionListener {
12.   public void processAction(ActionEvent event) throws AbortProcessingException {
13.     UIComponent container = event.getComponent().getNamingContainer();
14.     String name = (String) ((UIInput)
15.       container.findComponent("form:name")).getValue();
16.     String pwd = (String) ((UIInput)
```

```
17.          container.findComponent("form:password")).getValue();
18.       if (Registrar.isRegistered(name, pwd)) return;
19.
20.       FacesContext context = FacesContext.getCurrentInstance();
21.       context.addMessage(container.getClientId(),
22.          new FacesMessage("Name and password are invalid. Please try again."));
23.       throw new AbortProcessingException("Invalid credentials");
24.    }
25. }
```

Аспекты

Одним из способов расширения функциональных средств компонента является добавление объектов, таких как прослушиватели, преобразователи и средства проверки, к компонентам, содержащимся в составном компоненте. Еще одним из способов усовершенствования функциональности компонента является предоставление авторам страниц возможности определять аспекты составного компонента.

В компоненте используются аспекты, если пользователь компонента нуждается в том, чтобы задавать содержимое в дополнение к дочерним компонентам. Например, таблицы данных имеют аспекты `header` и `footer`, как в следующем примере:

```
<h:dataTable ...>
   <f:facet name="header">#{msgs.tableHeader}</f:facet>
   ...
   <f:facet name="footer">#{msgs.tableFooter}</f:facet>
</h:dataTable>
```

Разработчику в любое время может потребоваться предоставлять аспекты в своих собственных составных компонентах. Например, с помощью всего лишь нескольких строк разметки можно добавить аспекты `header` и `error` к компоненту `login`, чтобы автор страницы мог применить, допустим, такой код:

```
<util:login ...>
   <f:facet name="header" styleClass="header">
      <div class="prompt">#{msgs.loginPrompt}</div>
   </f:facet>
   <f:facet name="error" styleClass="error">
      <h:messages layout="table" styleClass="error"/>
   </f:facet>
   ...
</util:login>
```

Ниже показано, как реализованы аспекты `header` и `error` на странице определения компонента `login`.

```
<composite:interface>
   ...
   <composite:facet name="header"/>
   <composite:facet name="error"/>
</composite:interface>

<composite:implementation>
   ...
   <composite:renderFacet name="header"/>
   <h:form ...>
   ...
   </h:form>
   <composite:renderFacet name="error"/>
</composite:implementation>
```

В приведенном выше коде объявлены аспекты в интерфейсе компонента с тегом composite:facet и применен тег composite:renderFacet в реализации компонента для подготовки аспектов к отображению.

Тег composite:renderFacet подготавливает предоставленный аспект к отображению как дочерний компонент. Если вместо этого потребуется вставить его как аспект, то можно воспользоваться тегом composite:insertFacet, как в следующем примере:

```
<composite:implementation>
   ...
   <h:dataTable>
      <composite:insertFacet name="header"/>
      ...
      <composite:insertFacet name="footer"/>
   </h:dataTable>
   ...
</composite:implementation>
```

Здесь заголовок и нижний колонтитул становятся аспектами header и footer таблицы данных.

Дочерние теги

Составные компоненты представлены тегами. Иногда имеет смысл предусматривать возможность размещения содержимого в тексте этих тегов. По умолчанию, если что-либо помещено в текст тега составного компонента, реализация JSF просто пропускает это содержимое, но в реализации составного компонента можно использовать тег <composite:insertChildren/> для подготовки к отображению компонентов в тексте тега составного компонента.

Например, может оказаться удобным с помощью рассматриваемого компонента login предоставить авторам страниц возможность добавления каких-либо необходимых для них объектов в нижней части компонента. В качестве обычного примера использования такой возможности может служить добавление регистрационной ссылки к компоненту login, как показано на рис. 9.9.

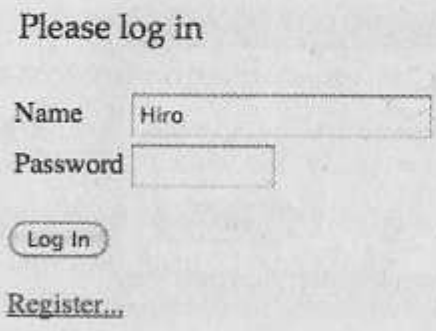


Рис. 9.9. Компонент login, в котором дополнительно предусмотрена регистрационная ссылка

Регистрационная ссылка, показанная на рис. 9.9, может быть добавлена так:

```
<util:login...>
   ...
   <f:facet name="header" styleClass="header">...</f:facet>
   <f:facet name="error" styleClass="error">...</f:facet>

   <!-- Дочерний компонент -->
```

```
    <h:link>#{msgs.registerLinkText}</h:link>
</util:login>
```

В реализации компонента `login` мы использовали тег `composite:insertChildren` для подготовки к отображению дочерних тегов составного компонента, т.е. любых компонентов, отличных от аспектов, в тексте тега составного компонента. В качестве примера ниже показано, как подготовить к отображению дочерние компоненты после аспекта `error` в рассматриваемом компоненте `login`.

```
<composite:implementation>
    <composite:renderFacet name="header"/>
    <h:form ...>
    ...
    </h:form>
    <composite:renderFacet name="error"/>

    <composite:insertChildren/>
</composite:implementation>
```

Применение кода JavaScript

Часто бывает удобно закрепить за компонентом код JavaScript, например, для проверки правильности на стороне клиента. В реализации JSF упрощено применение кода JavaScript благодаря наличию встроенных тегов JSF.

Код JavaScript можно также использовать в разрабатываемых составных компонентах. Но для эффективного решения этой задачи необходимо знать, как следует обращаться к клиентским идентификаторам элементов HTML, сформированных конкретным компонентом.

На рис. 9.10 показано, как действуют средства проверки правильности на стороне клиента применительно к полям компонента `login`. При этой проверке правильности используется определенный объем кода JavaScript.

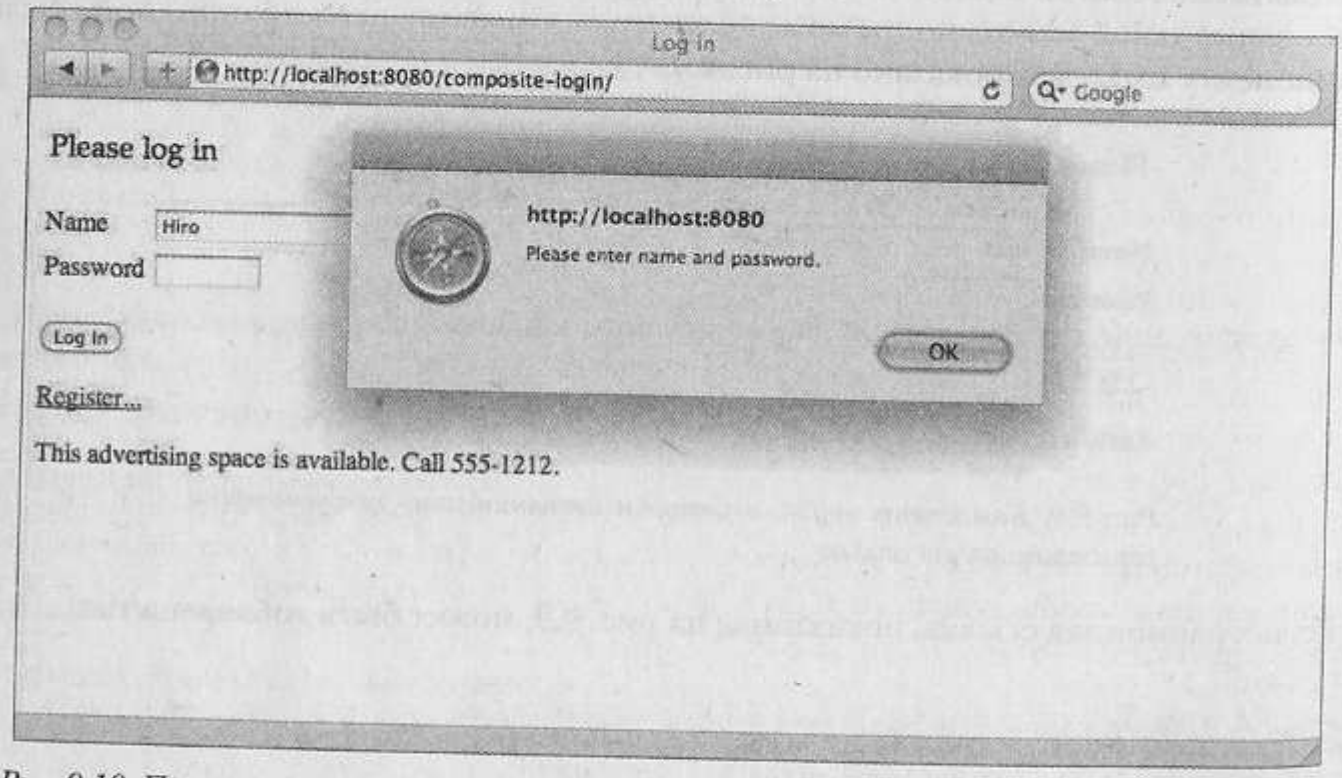


Рис. 9.10. Применение средств проверки правильности на стороне клиента для компонента `login`

Можно поместить код JavaScript непосредственно в раздел реализации составного компонента, но, чтобы программа в большей степени соответствовала принципу модульной организации, мы поместили код JavaScript для компонента `login` в отдельном файле. Этот код JavaScript загружается с помощью тега `h:outputScript`, после чего происходит вызов одной из функций из этого кода JavaScript при отправке пользователем регистрационной формы:

```
<composite:implementation>
  <h:outputScript library="util" name="login.js" target="head"/>
  <h:form id="form" onsubmit="return checkForm(this, '#{cc.clientId}')">
    ...
  </h:form>
  ...
</composite:implementation>
```

Функция `checkForm()` принимает два параметра: ссылку на форму и клиентский идентификатор составного компонента, в котором находится форма. В этой функции клиентский идентификатор используется для доступа к элементам формы составного компонента, как показано в листинге 9.7.

Листинг 9.7. Файл composite-login/web/resources/util/login.js

```
1. function checkForm(form, ccId) {
2.    var name = form[ccId + ':form:name'].value;
3.    var pwd = form[ccId + ':form:password'].value;
4.
5.    if (name == "" || pwd == "") {
6.       alert("Please enter name and password.");
7.       return false;
8.    }
9.    return true;
10. }
```

Следует отметить, что здесь пришлось проходить несколько уровней доступа к полям ввода, через составной компонент и форму ввода. От обозначения формы можно избавиться, дав указание реализации JSF не задавать префикс в виде идентификатора формы:

```
<h:form id="form" prependId="false"
  onsubmit="return checkForm(this, '#{cc.clientId}')">
```

Теперь мы можем обращаться к полям имени и пароля немного проще:

```
var name = form[ccId + 'name'].value;
var pwd = form[ccId + 'password'].value;
```

В рассматриваемом примере приложения применяется сочетание приемов, представленных в предыдущих разделах.

Окончательная версия определения компонента показана в листинге 9.8. В листинге 9.9 приведена веб-страница, в которой используется компонент `login`. Класс `User` в листинге 9.10 представляет пользователя, вошедшего в систему. В данном случае показана простейшая реализация службы регистрации исключительно в целях демонстрации; чтобы ознакомиться с реализацией, см. код на сайте, сопровождающем данную книгу (`http://corejsf.com`).

Листинг 9.8. Файл composite-login/web/resources/util/login.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3.    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
```

```
5.      xmlns:h="http://java.sun.com/jsf/html"
6.      xmlns:composite="http://java.sun.com/jsf/composite">
7.
8.    <composite:interface>
9.      <composite:editableValueHolder name="nameInput" targets="form:name"/>
10.     <composite:editableValueHolder name="passwordInput" targets="form:password"/>
11.     <composite:editableValueHolder name="inputs"
12.           targets="form:name form:password"/>
13.     <composite:actionSource name="loginButton" targets="form:loginButton"/>
14.
15.     <composite:attribute name="name"/>
16.     <composite:attribute name="password"/>
17.
18.     <composite:attribute name="namePrompt"/>
19.     <composite:attribute name="passwordPrompt"/>
20.
21.     <composite:attribute name="loginValidate"
22.       method-signature="void validateLogin(ComponentSystemEvent e)
23.         throws javax.faces.event.AbortProcessingException"/>
24.
25.     <composite:attribute name="loginAction"
26.       method-signature="java.lang.String action()"/>
27.
28.     <composite:facet name="heading"/>
29.     <composite:facet name="error"/>
30.   </composite:interface>
31.
32.   <composite:implementation>
33.     <h:outputScript library="components/util" name="login.js" target="head"/>
34.     <h:form id="form" onsubmit="return checkForm(this, '#{cc.clientId}')">
35.       <composite:renderFacet name="heading"/>
36.       <h:panelGrid columns="2">
37.         #{cc.attrs.namePrompt}
38.         <h:panelGroup>
39.           <h:inputText id="name" value="#{cc.attrs.name}"/>
40.           <h:message for="name"/>
41.         </h:panelGroup>
42.
43.         #{cc.attrs.passwordPrompt}
44.
45.         <h:panelGroup>
46.           <h:inputSecret id="password" value="#{cc.attrs.password}" size="8"/>
47.           <h:message for="password"/>
48.         </h:panelGroup>
49.       </h:panelGrid>
50.
51.       <p>
52.         <h:commandButton id="loginButton"
53.           value="#{cc.attrs.loginButtonText}"
54.           action="#{cc.attrs.loginAction}"/>
55.       </p>
56.
57.     </h:form>
58.
59.     <composite:renderFacet name="error"/>
60.
61.     <p><composite:insertChildren/></p>
62.
63.     <p>#{cc.resourceBundleMap.footer}</p>
64.   </composite:implementation>
65. </html>
```

Листинг 9.9. Файл composite-login/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:util="http://java.sun.com/jsf/composite/util"
      xmlns:ui="http://java.sun.com/jsf/facelets">
   <h:head>
      <title>#{msgs.loginHeading}</title>
      <h:outputStylesheet library="css" name="styles.css" />
   </h:head>
   <h:body>
      <util:login namePrompt="#{msgs.namePrompt}"
            passwordPrompt="#{msgs.passwordPrompt}"
            name="#{user.name}"
            password="#{user.password}"
            loginAction="#{user.login}"
            loginButtonText="#{msgs.loginButtonText}">

         <f:validateLength minimum="4" for="nameInput"/>
         <f:validator validatorId="com.corejsf.Password" for="passwordInput"/>
         <f:actionListener type="com.corejsf.LoginActionListener" for="loginButton"/>

         <f:facet name="heading" styleClass="header">
            <div class="prompt">#{msgs.loginPrompt}</div>
         </f:facet>

         <f:facet name="error" styleClass="error">
            <h:messages layout="table" styleClass="error"/>
         </f:facet>

         <!-- Дочерний компонент -->
         <h:link outcome="register">#{msgs.registerLinkText}</h:link>

      </util:login>
      <ui:debug/>
   </h:body>
</html>
```

Листинг 9.10. Файл composite-login/src/java/com/corejsf/User.java

```
package com.corejsf;

import java.io.Serializable;
import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;

@Named // или @ManagedBean
@SessionScoped
public class User implements Serializable {
   private String name;
   private String password;

   public User() { this("", ""); }
   public User(String name, String password) {
      this.name = name;
```

```
18.         this.password = password;
19.     }
20.
21.     public String getPassword() { return password; }
22.     public void setPassword(String newValue) { password = newValue; }
23.
24.     public String getName() { return name; }
25.     public void setName(String newValue) { name = newValue; }
26.
27.     public String register() {
28.         Registrar.register(name, password);
29.         return "welcome";
30.     }
31.
32.     public String login() {
33.         return "welcome";
34.     }
35.
36.     public String logout() {
37.         name = password = "";
38.         return "index";
39.     }
40. }
```

Вспомогательные компоненты

Иногда возникает необходимость в получении большего контроля над поведением составного компонента, чем может быть достигнут с помощью объявлений XML. Чтобы добавить код Java к составному компоненту, необходимо предусмотреть вспомогательный компонент. К вспомогательному компоненту предъявляются следующие требования.

- Он является подклассом класса `UIComponent`.
- Реализует интерфейс маркера `NamingContainer`.
- Его свойство `family` имеет значение `"javax.faces.NamingContainer"`.

Класс `UIComponent` и понятие семейства компонентов подробно рассматриваются в главе 11. А в данном разделе приведен простой, но типичный пример.

Рассмотрим компонент задания даты, в котором используются три компонента `h:selectOneMenu` для указания дня, месяца и года (рис. 9.11).

С использованием описанных выше приемов можно очень легко создать такой компонент и предоставить доступ к трем его дочерним компонентам, чтобы можно было закрепить ссылку на значение для каждого из них:

```
<util:date day="#{user.birthDate}" month="#{user.birthMonth}"
   year="#{user.birthYear}"/>
```

Однако предположим, что в применяемом классе `User` дата хранится в виде значения типа `java.util.Date`, а не в виде трех отдельных целых чисел. Вообще говоря, в объектно-ориентированном программировании следует при любой возможности использовать классы, а не хранить все, что попало, как числа и строки.

Было бы намного лучше, если бы рассматриваемый компонент содержал единственное значение типа `Date`:

```
<util:date value="#{user.birthDay}"/>
```

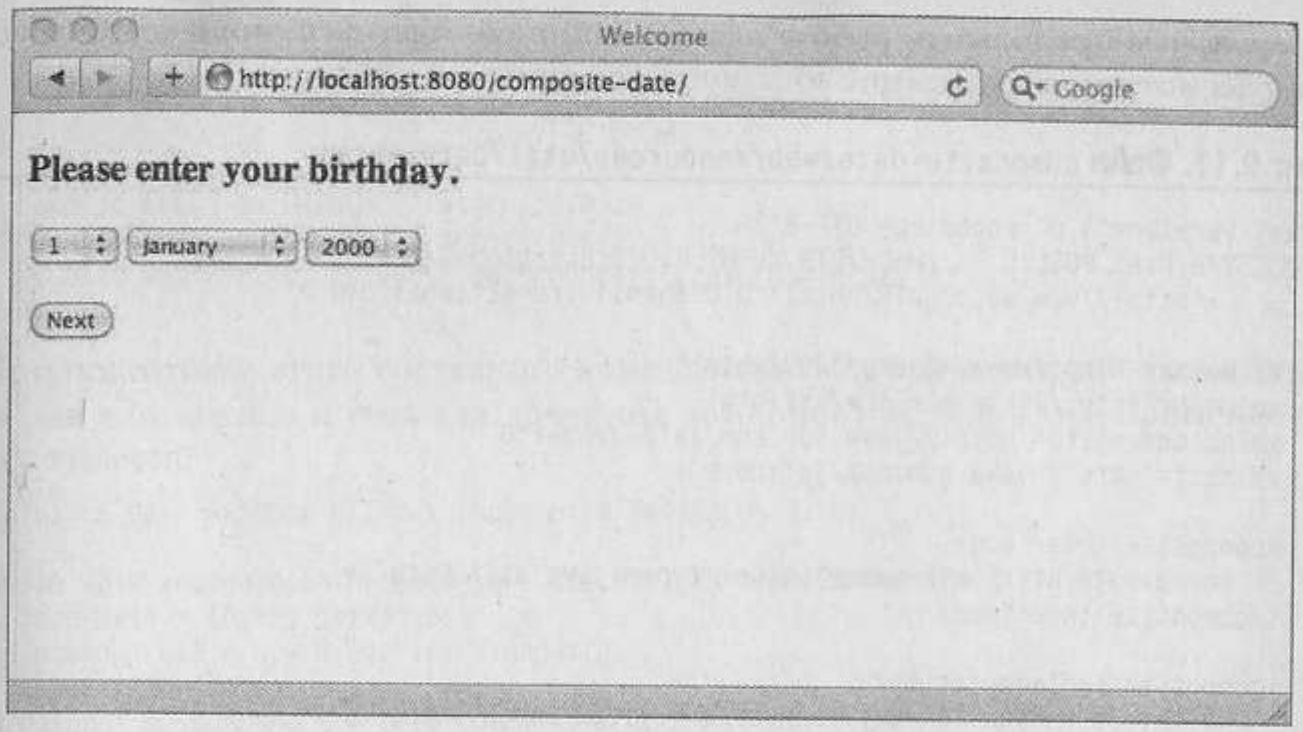


Рис. 9.11. Составной компонент задания даты

Но класс `Date` языка Java не имеет методов получения и задания свойств для дня, месяца и года. Поэтому мы не можем просто воспользоваться выражением значения наподобие `#{cc.attrs.value.day}` в рассматриваемом составном компоненте. Вместо этого необходимо применить определенный объем кода Java для сборки значения типа `Date` из его составляющих. В связи с этим предусмотрено применение вспомогательного компонента.

Существуют два способа определения вспомогательного компонента для составного компонента. Простейший из них состоит в использовании соглашения об именах, т.е. в использовании имени класса `libraryName.componentName`. В рассматриваемом примере именем класса является `util.date`; иными словами, речь идет о классе `date` из пакета `util`. (В данном случае немного непривычно то, что имя класса задано в нижнем регистре, но это – цена, которую приходится платить за соблюдение принципа "соглашение об именах имеет приоритет над конфигурацией".)

Еще один вариант состоит в том, что можно использовать аннотацию `@FacesComponent` для определения так называемого *типа компонента*, а затем задать этот тип компонента в объявлении `composite:interface`:

```
<composite:interface componentType="com.corejsf.Date">
```

Типы компонентов будут рассматриваться в главе 11. В данном примере мы следуем соглашению об именах и предусматриваем применение класса `util.date`.

Как будет показано в главе 11, классы компонентов, которые принимают ввод от пользователя, должны расширять класс `UIInput`. Таким образом, необходимый нам вспомогательный компонент имеет следующую форму:

```
package util;
...
public class date extends UIInput implements NamingContainer {
   public String getFamily() {
      return "javax.faces.NamingContainer";
   }
   ...
}
```

Прежде чем продолжить работу над реализацией этого вспомогательного компонента, рассмотрим определение составного компонента в листинге 9.11.

Листинг 9.11. Файл composite-date/web/resources/util/date.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:composite="http://java.sun.com/jsf/composite"
   xmlns:f="http://java.sun.com/jsf/core">

   <composite:interface>
      <composite:attribute name="value" type="java.util.Date"/>
   </composite:interface>

   <composite:implementation>
      <h:selectOneMenu id="day" converter="javax.faces.Integer">
         <f:selectItems value="#{dates.days}"/>
      </h:selectOneMenu>
      <h:selectOneMenu id="month" converter="javax.faces.Integer">
         <f:selectItems value="#{dates.months}"/>
      </h:selectOneMenu>
      <h:selectOneMenu id="year" converter="javax.faces.Integer">
         <f:selectItems value="#{dates.years}"/>
      </h:selectOneMenu>
   </composite:implementation>
</html>
```

Здесь бин dates просто формирует массивы значений 1...31 January, December, а также 1900...2100, как показано в листинге 9.12.

Листинг 9.12. Файл composite-date/src/java/com/corejsf/Dates.java

```
package com.corejsf;
import java.io.Serializable;
import java.text.DateFormatSymbols;
import java.util.LinkedHashMap;
import java.util.Map;
import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.ApplicationScoped;
   // или import javax.faces.bean.ApplicationScoped;

@Named // или @ManagedBean
@ApplicationScoped
public class Dates implements Serializable {
   private int[] days;
   private int[] years;
   private Map<String, Integer> months;

   private static int[] intArray(int from, int to) {
      int[] result = new int[to - from + 1];
      for (int i = from; i <= to; i++) result[i - from] = i;
      return result;
   }

   public Dates() {
      days = intArray(1, 31);
```

```
26.         years = intArray(1900, 2100);
27.         months = new LinkedHashMap<String, Integer>();
28.         String[] names = new DateFormatSymbols().getMonths();
29.         for (int i = 0; i < 12; i++) months.put(names[i], i + 1);
30.     }
31.
32.     public int[] getDays() { return days; }
33.     public int[] getYears() { return years; }
34.     public Map<String, Integer> getMonths() { return months; }
35. }
```

При подготовке этого составного компонента к отображению необходимо задать значения дня, месяца и года для дочерних компонентов. Для этого применяется метод `encodeBegin`:

```
public class date extends UIInput implements NamingContainer {
   ...
   public void encodeBegin(FacesContext context) throws IOException {
      Date date = (Date) getValue();
      Calendar cal = new GregorianCalendar();
      cal.setTime(date);
      UIInput dayComponent = (UIInput) findComponent("day");
      UIInput monthComponent = (UIInput) findComponent("month");
      UIInput yearComponent = (UIInput) findComponent("year");
      dayComponent.setValue(cal.get(Calendar.DATE));
      monthComponent.setValue(cal.get(Calendar.MONTH) + 1);
      yearComponent.setValue(cal.get(Calendar.YEAR));
      super.encodeBegin(context);
   }
   ...
}
```

После отправки формы необходимо воссоздать значение типа `Date` из значений дня, месяца и года. Чтобы правильно разместить код, в котором осуществляется это преобразование, рассмотрим, как начинается жизненный цикл JSF.

1. В запросе HTTP передаются пары "имя–значение".
2. В фазе применения значений запроса каждый из компонентов `h:selectOneMenu` задает свое переданное значение.
3. Во время проверки правильности переданное значение компонента преобразуется в требуемый тип и становится преобразованным значением. В данном случае каждый компонент `h:selectOneMenu` имеет целочисленный преобразователь, который преобразовывает входящую строку в значение типа `Integer`.
4. Если преобразованное значение проходит проверку правильности, то становится значением компонента.

Теперь рассмотрим составной компонент. Он не имеет переданного значения, поскольку в запросе HTTP нет ничего, что непосредственно соответствовало бы составному компоненту. Но нам требуется, чтобы этот компонент имел преобразованное значение, что позволяло бы закрепить средства проверки за этим составным компонентом. Поэтому мы вычисляем дату в методе `getConvertedValue`:

```
public class date extends UIInput implements NamingContainer {
   ...
   protected Object getConvertedValue(FacesContext context, Object newSubmittedValue)
         throws ConverterException {
      UIInput dayComponent = (UIInput) findComponent("day");
      UIInput monthComponent = (UIInput) findComponent("month");
      UIInput yearComponent = (UIInput) findComponent("year");
```

```
        int day = (Integer) dayComponent.getValue();
        int month = (Integer) monthComponent.getValue();
        int year = (Integer) yearComponent.getValue();
        if (isValidDate(day, month, year))
           return new GregorianCalendar(year, month - 1, day).getTime();
        else
           throw new ConverterException(new FacesMessage(FacesMessage.SEVERITY_ERROR,
                "Invalid date", "Invalid date"));
    }
    ...
}
```

Если пользователь предоставляет недопустимую дату, такую как 30 февраля, возникает исключительная ситуация преобразователя. Следует учитывать, что это — не ошибка проверки правильности. Что касается значений типа `Date`, то ошибкой проверки правильности должно быть появление даты, не находящейся в ожидаемом диапазоне дат. Но если пользователь предоставляет в качестве входных данных значение "30 февраля", то в нашем распоряжении еще нет значения типа `Date`, поскольку такой пользовательский ввод не может быть преобразован в это значение.

Как оказалось, фактически нам требуется переопределить метод `getSubmittedValue`, поскольку в нем нулевое переданное значение трактуется как частный случай. Мы просто возвращаем сам компонент, как показано в листинге 9.15.

Как можно видеть в листинге 9.13, задача использования этого компонента является очень простой. Для этого достаточно лишь закрепить свойство со значением типа `Date`; в данном случае речь идет о свойстве `birthday` класса `UserBean`, показанного в листинге 9.14. Заслуживает внимания также аннотация проверки правильности `@Past` свойства `birthday`. Эта аннотация безукоризненно взаимодействует с рассматриваемым компонентом задания даты, проверяя переданные значения типа `Date`. Если пользователь указывает дату, которая лежит в будущем, то приложение возвращается к старой дате и выводит сообщение об ошибке (рис. 9.12).

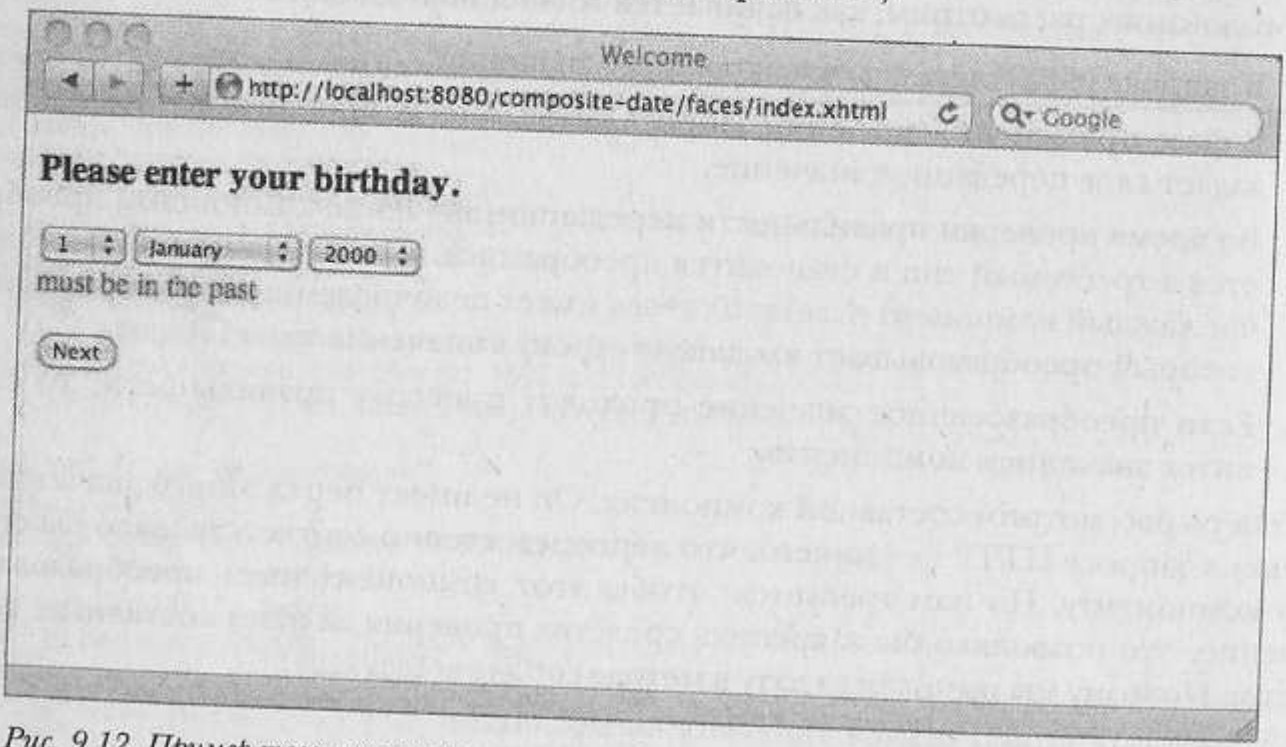


Рис. 9.12. Пример того, как попытка ввести дату, относящуюся к будущему, отвергается и активизируется ошибка проверки правильности

На рис. 9.13 показана структура каталогов приложения.

Очевидно, что для реализации вспомогательного компонента требуются определенные знания в части API-интерфейса компонентов JSF (эта тема рассматривается в главе 11). Однако ввод небольшого объема кода для создания вспомогательного компонента требует намного меньше работы, чем написание пользовательского компонента с нуля.

Рис. 9.13. Структура каталогов приложения задания даты с составным компонентом

Листинг 9.13. Файл `composite-date/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:util="http://java.sun.com/jsf/composite/util">
   <h:head>
      <title>Welcome</title>
      <h:outputStylesheet library="css" name="styles.css"/>
   </h:head>
   <h:body>
      <h:form>
         <h3>Please enter your birthday.</h3>
         <util:date id="date" value="#{user.birthday}"/>
         <br/><h:message for="date" styleClass="error"/>
         <p><h:commandButton value="Next" action="welcome"/></p>
      </h:form>

   </h:body>
</html>
```

Листинг 9.14. Файл `composite-date/src/java/com/corejsf/UserBean.java`

```
package com.corejsf;

import java.io.Serializable;
import java.util.Date;
import java.util.GregorianCalendar;
import javax.inject.Named;
```

```
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;
import javax.validation.constraints.Past;

@Named("user") // или @ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable {
   private String name;
   private String password;
   @Past private Date birthday = new GregorianCalendar(2000, 0, 1).getTime();

   public String getName() { return name; }
   public void setName(String newValue) { name = newValue; }

   public String getPassword() { return password; }
   public void setPassword(String newValue) { password = newValue; }

   public Date getBirthday() { return birthday; }
   public void setBirthday(Date newValue) { birthday = newValue; }
}
```

Листинг 9.15. Файл composite-date/src/java/util/date.java

```
package util;

import com.corejsf.util.Messages;

import java.io.IOException;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Locale;
import java.util.ResourceBundle;
import javax.faces.application.FacesMessage;
import javax.faces.component.NamingContainer;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.convert.ConverterException;

public class date extends UIInput implements NamingContainer {
   public String getFamily() {
      return "javax.faces.NamingContainer";
   }

   public void encodeBegin(FacesContext context) throws IOException {
      Date date = (Date) getValue();
      Calendar cal = new GregorianCalendar();
      cal.setTime(date);
      UIInput dayComponent = (UIInput) findComponent("day");
      UIInput monthComponent = (UIInput) findComponent("month");
      UIInput yearComponent = (UIInput) findComponent("year");
      dayComponent.setValue(cal.get(Calendar.DATE));
      monthComponent.setValue(cal.get(Calendar.MONTH) + 1);
      yearComponent.setValue(cal.get(Calendar.YEAR));
      super.encodeBegin(context);
   }

   public Object getSubmittedValue() {
      return this;
   }
```

```
38.
39.    protected Object getConvertedValue(FacesContext context, Object newSubmittedValue)
40.          throws ConverterException {
41.       UIInput dayComponent = (UIInput) findComponent("day");
42.       UIInput monthComponent = (UIInput) findComponent("month");
43.       UIInput yearComponent = (UIInput) findComponent("year");
44.       int day = (Integer) dayComponent.getValue();
45.       int month = (Integer) monthComponent.getValue();
46.       int year = (Integer) yearComponent.getValue();
47.       if (isValidDate(day, month, year))
48.          return new GregorianCalendar(year, month - 1, day).getTime();
49.       else {
50.          FacesMessage message
51.                = Messages.getMessage("util.messages", "invalidDate", null);
52.          message.setSeverity(FacesMessage.SEVERITY_ERROR);
53.          throw new ConverterException(message);
54.       }
55.    }
56.
57.    private static boolean isValidDate(int d, int m, int y) {
58.       if (d < 1 || m < 1 || m > 12) {
59.          return false;
60.       }
61.       if (m == 2) {
62.          if (isLeapYear(y)) {
63.             return d <= 29;
64.          } else {
65.             return d <= 28;
66.          }
67.       } else if (m == 4 || m == 6 || m == 9 || m == 11) {
68.          return d <= 30;
69.       } else {
70.          return d <= 31;
71.       }
72.    }
73.
74.    private static boolean isLeapYear(int y) {
75.       return y % 4 == 0 && (y % 400 == 0 || y % 100 != 0);
76.    }
77. }
```

Упаковка составных компонентов в файлах JAR

Разработчик может упаковать созданные им составные компоненты в виде файлов JAR, чтобы другие разработчики могли использовать эти компоненты. Для этого достаточно поместить составной компонент и все относящиеся к нему артефакты, такие как код JavaScript, таблицы стилей или файлы `properties`, в каталог `META-INF` файла JAR, как показано на рис. 9.14.

После создания файла JAR, включающего один или несколько составных компонентов, можно поместить файл JAR в каталог, указанный системной переменной `classpath`, чтобы использовать эти компоненты.

Рис. 9.14. Упаковка компонентов `icon` и `login` в файле JAR

Резюме

Наибольшим преимуществом JSF является то, что это — компонентно-ориентированная платформа, а в версии JSF 2.0 задача реализации пользовательских компонентов окончательно упрощена: если разработчик может реализовать представление Facelets, то имеет возможность реализовать и составной компонент.

В версии JSF 2.0 предусмотрена широкая поддержка составных компонентов, включая поддержку обработки аспектов и дочерних тегов.

Составные компоненты, применяемые в сочетании с новыми возможностями JSF 2.0 по поддержке Ajax, позволяют разработчикам на JSF легко реализовывать пользовательские компоненты, которые инкапсулируют средства Ajax.

Технология Ajax

В этой главе...

- Совместное применение Ajax и JSF
- Жизненный цикл JSF и технология Ajax
- Рекомендации по применению Ajax JSF
- Тег `f:ajax`
- Группы Ajax
- Проверка правильности полей в Ajax
- Мониторинг запросов Ajax
- Пространства имен JavaScript
- Обработка ошибок в Ajax
- Ответы Ajax
- Библиотека JavaScript в версии JSF 2.0
- Передача дополнительных параметров запроса Ajax
- Организация очередей событий
- Объединение событий
- Перехват значений функции `jsf.ajax.request()`
- Использование Ajax в составных компонентах

Глава 10

Технология Ajax (Asynchronous JavaScript and XML) всегда рассматривалась пользователями и разработчиками как принадлежащая к самому высокоуровневому классу программных средств, но в наши дни без этой технологии буквально невозможно обойтись, когда речь идет о создании привлекательных и конкурентоспособных веб-приложений.

В версии JSF 2.0 предусмотрена встроенная поддержка Ajax на основе стандартной библиотеки JavaScript. Разработчик может обращаться к этой библиотеке и в создаваемых представлениях, и в коде Java.

Большинство распространенных вариантов применения Ajax, таких как проверка правильности поля и создание индикаторов хода работы, можно реализовать с помощью одного из тегов основной библиотеки JSF: `f:ajax`. Как и другие теги из основной библиотеки JSF, такие как `f:validator` и `f:converter`, тег `f:ajax` закрепляет правило поведения за компонентом. В качестве примера ниже показано, как закрепить правило поведения Ajax за полем ввода текста.

```
<h:inputText value="#{someBean.someProperty}">
   <f:ajax event="keyup" render="someOtherComponentId"/>
</h:inputText>
```

При возникновении каждого события `keyup` в поле ввода текста реализация JSF отправляет вызов Ajax на сервер и обрабатывает значение из поля ввода. После возврата вызова Ajax реализация JSF подготавливает к отображению компонент с идентификатором `someOtherComponentId`.

Совместное применение Ajax и JSF

По своему принципу применение технологии Ajax является несложной задачей. Фактически запросы Ajax отличаются от обычных запросов HTTP только в двух отношениях.

1. В Ajax процессы *отчасти* формируются на сервере в течение вызова Ajax.
2. Реализация Ajax *частично* подготавливает к отображению элементы DOM (Document Object Model — объектная модель документа) на клиенте после возврата вызова Ajax с сервера.

Эта последовательность событий иллюстрируется на рис. 10.1, который показывает вызов Ajax, проверяющий единственное поле ввода, скорее всего, когда поле теряет фокус.

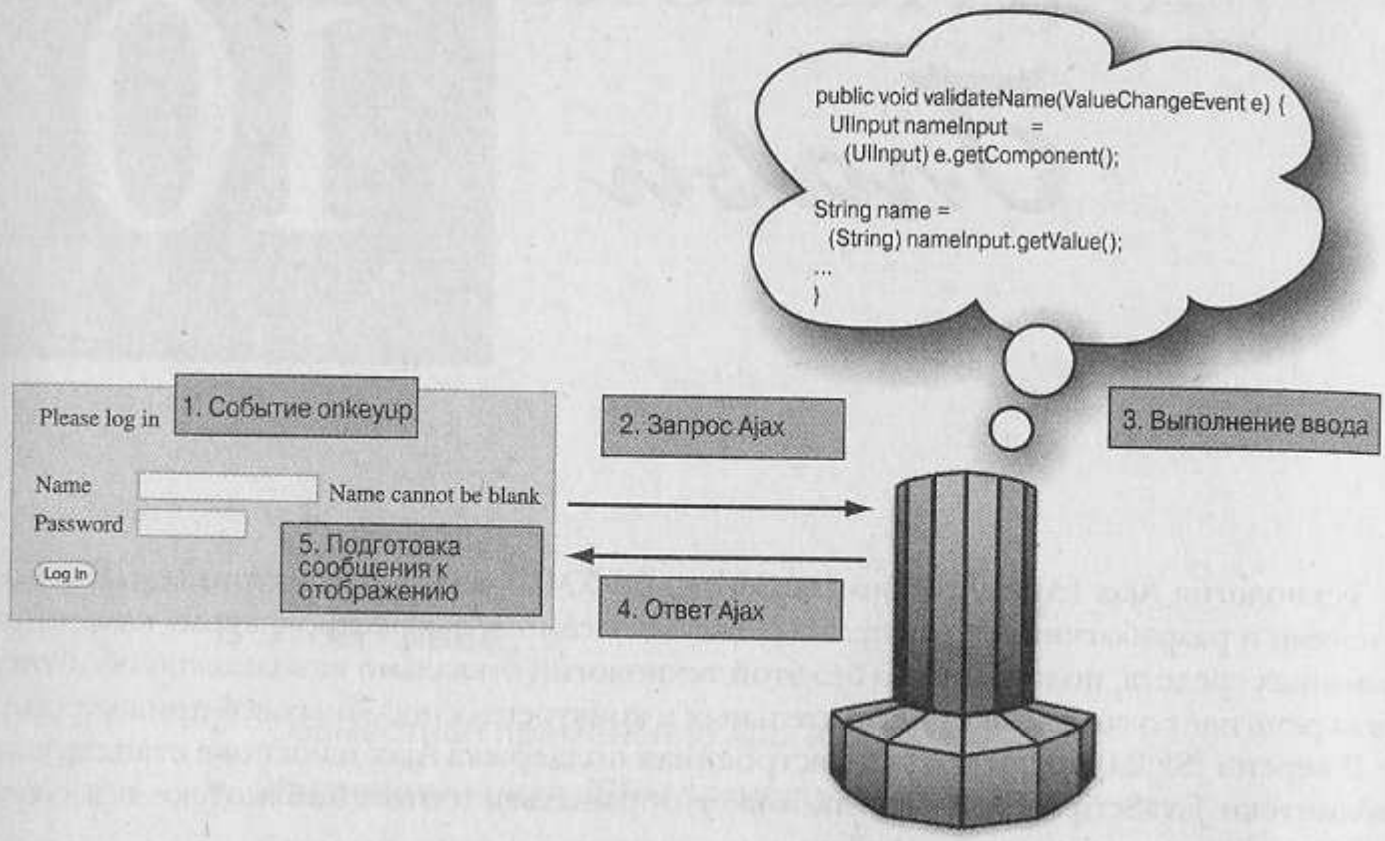


Рис. 10.1. Запрос Ajax на проверку поля ввода

В технологии JSF мы имеем дело с компонентами, поэтому можем определить Ajax JSF следующим образом.

> *Запросы Ajax JSF представляют собой отчасти компоненты процесса на сервере, а отчасти компоненты подготовки к отображению на клиенте при возврате запроса.*

Как будет показано в следующем разделе, реализация JSF встраивает Ajax в свой жизненный цикл, который является основой функционирования всех приложений JSF. Интеграция JSF и Ajax является очень тесной, что позволяет обрабатывать запросы Ajax таким же способом, который применяется при осуществлении других правил поведения компонента, таких как проверка правильности или преобразование.

Жизненный цикл JSF и технология Ajax

В версии JSF 2.0 жизненный цикл JSF разбит на две части: выполнение (в которой компоненты выполняются) и подготовка к отображению, как показано на рис. 10.2 и 10.3 соответственно. При любом конкретном запросе Ajax задается набор компонентов, которые выполняет реализация JSF, и еще один набор компонентов, которые она подготавливает к отображению.

В части выполнения жизненного цикла осуществляется обработка полей ввода на сервере, что представлено шагом 3 на рис. 10.1.

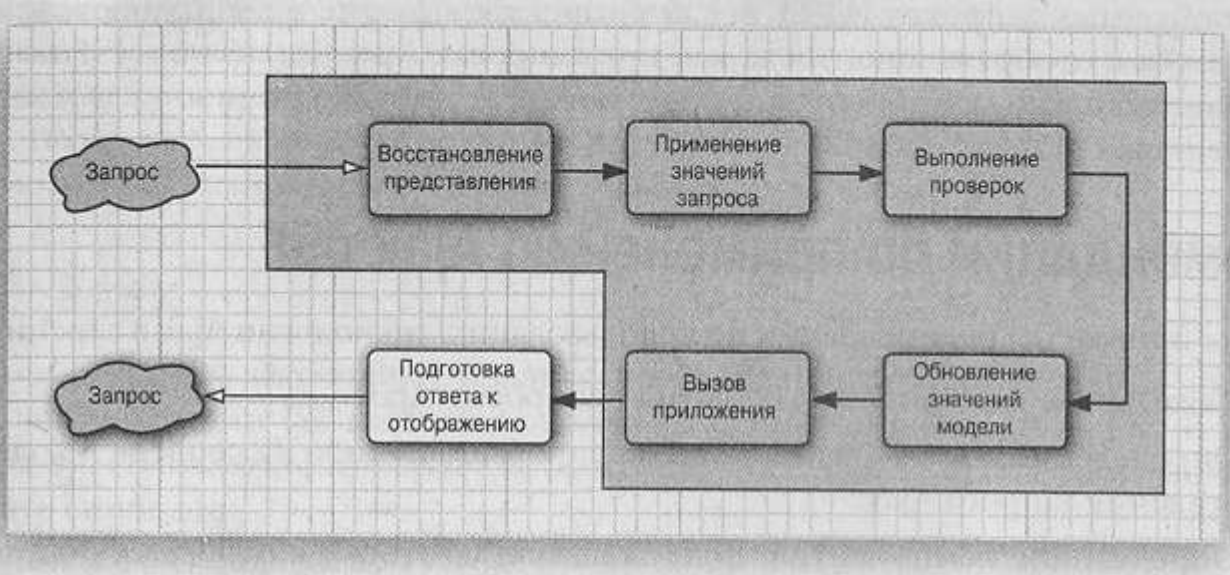


Рис. 10.2. Часть выполнения жизненного цикла JSF

В части подготовки к отображению жизненного цикла, который иллюстрируется шагом 5 на рис. 10.1, компоненты подготавливаются к отображению на клиенте. Часть подготовки к отображению жизненного цикла показана на рис. 10.3.

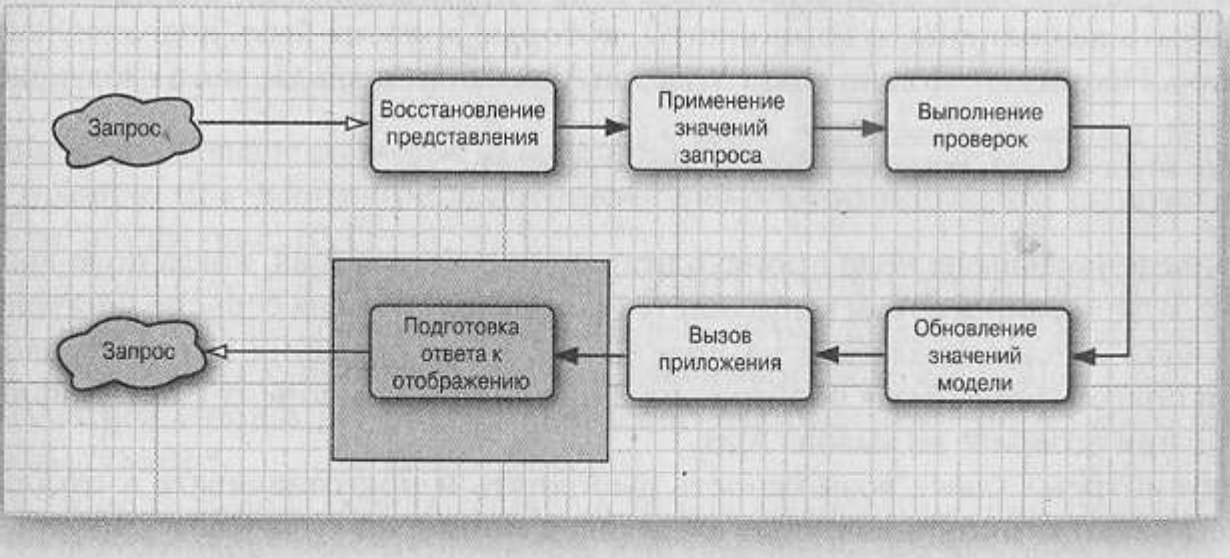


Рис. 10.3. Часть подготовки к отображению жизненного цикла

Как показано на рис. 10.2, при выполнении реализацией JSF компонента на сервере происходит следующее.

- Преобразование и проверка правильности значения компонента (если компонент представляет собой поле ввода).
- Передача допустимых входных значений в модель (если компонент привязан к свойству бина).
- Выполнение действий и прослушивателей действий (если компонент является действием).

Таким образом, в версии JSF 2.0, по существу, применяются два жизненных цикла: в одном из них происходит выполнение компонентов, а в другом — подготовка компонентов к отображению. Реализация JSF всегда вначале выполняет компоненты, а затем подготавливает их к отображению.

При обычных запросах HTTP все компоненты в форме и выполняются, и подготавливаются к отображению, тогда как при запросах Ajax реализация JSF выполняет один или несколько компонентов и подготавливает к отображению нуль или больше компонентов.

Рекомендации по применению Ajax JSF

Ниже приведены рекомендации по использованию технологии Ajax в версии JSF 2.0.

1. Связывание компонента и события с запросом Ajax.
2. Определение компонентов, подлежащих выполнению на сервере в ходе осуществления запроса Ajax.
3. Определение компонентов, подлежащих подготовке к отображению после запроса Ajax.

Для осуществления большинства вариантов использования технологии Ajax, как правило, достаточно ввести ряд строк кода XML в файл XHTML, а также, возможно, несколько строк кода Java в управляемом бине, включая проверку полей ввода и отображение индикаторов выполнения, как будет показано ниже в главе.

Вызов Ajax необходимо связать с событием, таким как `keyup` или `blur`, активизируемым определенным компонентом. Затем определяются компоненты, которые должны быть выполнены, и компоненты, которые должны быть подготовлены к отображению. Например, можно связать вызов Ajax с полем ввода, как в следующем коде:

```
<h:inputText id="name" value="#{user.name}">
   <f:ajax event="blur" execute="@this" render="nameError"/>
</h:inputText>
```

В приведенном выше коде происходит запуск события Ajax, когда поле ввода теряет фокус. В этом запросе Ajax выполняется компонент `name` на сервере (значение `@this` для атрибута `execute` ссылается на поле ввода, в код создания которого вложен тег `f:ajax`) и подготавливается к отображению на клиенте компонент с идентификатором `nameError` при возврате из вызова Ajax.

Предусмотрена также возможность выполнять и подготавливать к отображению сразу несколько компонентов, как в следующем примере:

```
<h:inputText id="nameInput" value="#{user.name}">
   <f:ajax event="blur" execute="@this passwordInput"
                        render="nameError passwordError"/>
</h:inputText>
<h:outputText id="nameError"/>
...
<h:inputText id="passwordInput"/>
<h:outputText id="passwordError" value="#{user.passwordError}"/>
...
```

В приведенном выше коде выполняются и компонент `nameInput`, и компонент `passwordInput`. В нем подготавливаются к отображению компоненты `passwordError` и `nameError`.

Поддержка Ajax JSF осуществляется на низком уровне, но представляет собой развитую и всестороннюю реализацию Ajax. Чтобы успешно использовать Ajax в сочетании с JSF, необходимо всегда помнить, что в основе взаимодействия этих технологий лежит выполнение компонента на сервере во время вызова Ajax.

Например, при выполнении в реализации JSF компонента ввода происходит копирование значения поля ввода в свойство вспомогательного бина. Это изменение свойства, как правило, влияет на то, как происходит подготовка к отображению ответа Ajax. Возможно, это значение просто повторно отображается или вызывает более ярко выраженное изменение в пользовательском интерфейсе.

Тег f:ajax

Авторы страниц закрепляют правила поведения, такие как проверка правильности, за компонентами JSF с помощью тегов из основной библиотеки JSF. В качестве примера ниже показано, как проверить, что поле ввода текста содержит по меньшей мере пять символов.

```
<h:inputText value="#{user.name}">
   <f:validateLength minimum="5"/>
</h:inputText>
```

Поддержка Ajax JSF вполне позволяет справиться с этой задачей. Чтобы закрепить правило поведения Ajax за компонентом, можно добавить тег f:ajax в текст компонента, например:

```
<h:inputText id="name" value="#{user.name}">
    <f:ajax event="keyup" execute="@this" render="echo"/>
</h:inputText>
...
<h:outputText id="echo" value="#{user.name}"/>
```

В приведенной выше разметке правило поведения Ajax закрепляется за полем ввода. Это правило поведения предусматривает эхо-повтор всего, что вводит пользователь в поле имени (рис. 10.4).

Для каждого события keyup, активизированного полем ввода, реализация JSF отправляет вызов Ajax на сервер. На сервере вызов Ajax приводит к выполнению компонента name (обозначенного встроенным ключевым словом @this), а после возврата вызова Ajax реализация JSF подготавливает к отображению только компонент эхо-повтора на клиенте.

В тексте поля вывода, предназначенного для эхо-повтора, повторяется все, что находится в поле ввода имени, поскольку при обработке реализацией JSF поля ввода имени на сервере происходит копирование имени в свойство вспомогательного бина, ассоциированное с полем ввода имени, т.е. с полем name управляемого бина с именем user. После этого реализация JSF подготавливает к отображению поле эхо-повтора, в котором появляется обновленное значение user.name.

Чаще всего, как и в случае рассматриваемого простого примера эхо-повтора, возникает необходимость в выполнении компонента ввода, в который вложен тег f:ajax, поэтому реализация JSF выполняет @this по умолчанию. Из того, что выполнение происходит по умолчанию, следует, что можно исключить атрибут execute="@this" из приведенной выше разметки:

```
<h:inputText id="name" value="#{user.name}">
   <f:ajax event="keyup" render="echo"/>  <!-- Значение execute="@this" задано неявно -->
</h:inputText>
```

Кроме @this, можно также использовать @form, @all или @none в качестве значений атрибута execute тега f:ajax, как показано в табл. 10.1, в которой перечислены атрибуты тега f:ajax.

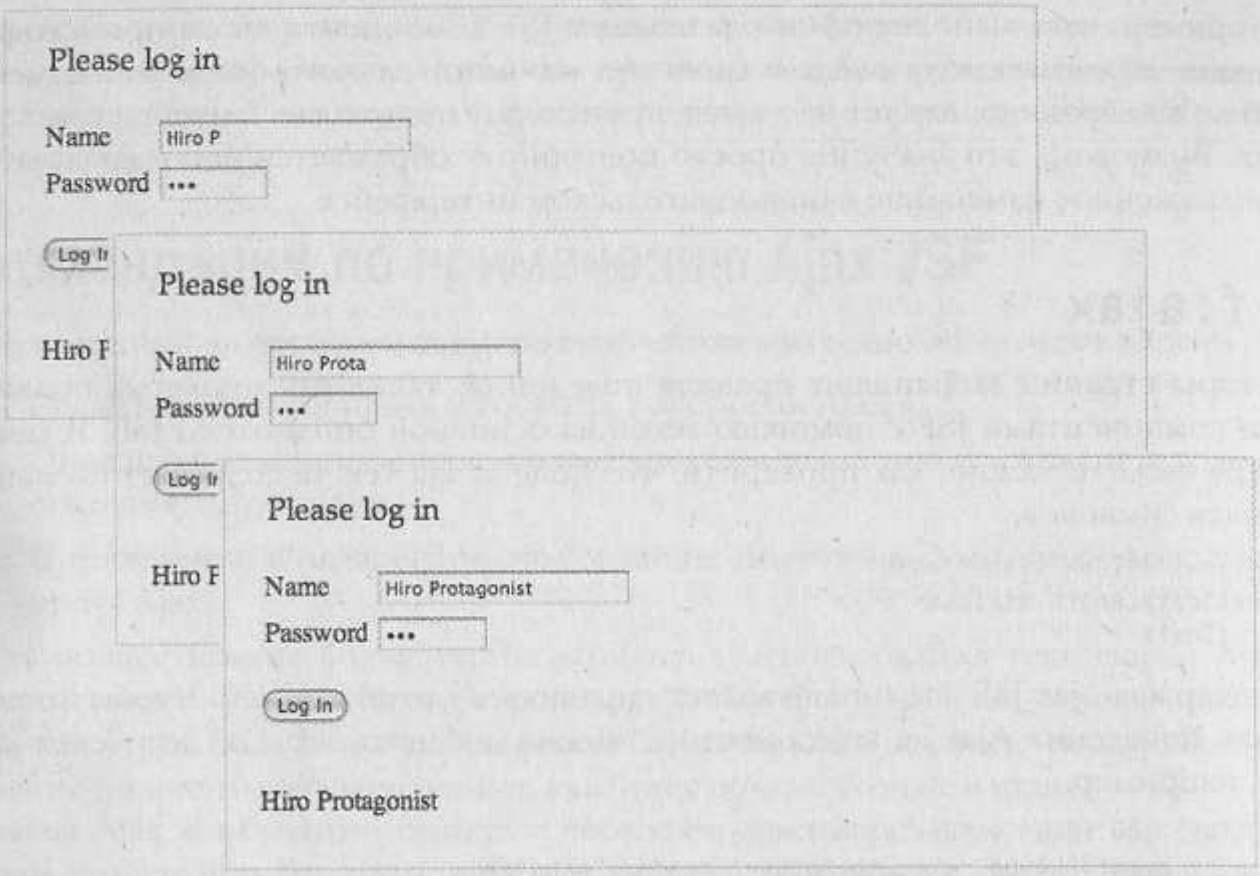


Рис. 10.4. Использование технологии Ajax для эхо-повтора значения в поле ввода по мере набора данных пользователем

Таблица 10.1. Атрибуты тега `f:ajax`

Атрибут	Описание
`disabled`	Отключает тег, задавая значение `true` атрибута `disabled`
`event`	Событие, которое активизирует запрос Ajax. Именами событий могут быть имена событий JavaScript без префикса `on`. Например, для события `onblur` можно использовать атрибут `event="blur"`. Именами событий могут быть также имена следующих событий компонента: `action` и `valueChange`. Эти имена могут быть заданы для командных компонентов (кнопки и ссылки) и полей ввода соответственно
`execute`	Разделенный пробелами список компонентов, выполняемых реализацией JSF на сервере во время вызова Ajax. Допустимыми ключевыми словами для атрибута `execute` являются `@this`, `@form`, `@all`, `@none` Если атрибут `execute` не задан, то реализация JSF использует `@this` в качестве значения по умолчанию
`immediate`	Если для этого атрибута задано значение `true`, реализация JSF обрабатывает поля ввода на раннем этапе своего жизненного цикла
`onerror`	Функция JavaScript, вызываемая реализацией JSF, если вызов Ajax приводит к ошибке

`onevent`	Функция JavaScript, вызываемая реализацией JSF применительно к событиям Ajax. Эта функция вызывается три раза в течение срока существования успешного вызова Ajax:
	`begin`
	`complete`
	`success`
	Применительно к успешным вызовам Ajax реализация JSF вызывает функцию `onevent` с начала вызова Ajax (`begin`), после его обработки на сервере (`complete`) и непосредственно перед тем, как JSF подготавливает к отображению ответ Ajax (`success`).

Окончание табл. 10.1

Атрибут	Описание
	Если во время обработки запроса Ajax возникает ошибка, реализация JSF вызывает функцию `onevent` после завершения запроса Ajax и впоследствии вызывает обработчик ошибок, на который ссылается атрибут `onevent`
`listener`	Реализация JSF вызывает метод `processAjaxBehavior` этого прослушивателя по одному разу во время каждого вызова Ajax, в фазе вызова приложения жизненного цикла (в конце части выполнения жизненного цикла). Этот метод должен иметь следующую сигнатуру: `public void processAjaxBehavior(javax.faces.event.AjaxBehaviorEvent event) throws javax.faces.event.AbortProcessingException`
`render`	Разделенный пробелами список компонентов, которые реализация JSF подготавливает к отображению на клиенте после возврата вызова Ajax с сервера. Можно использовать те же ключевые слова (`@all`, `@this`, `@form` и `@none`), которые допустимы для атрибута `execute`. Если атрибут `render` не определен, то приобретает значение по умолчанию `@none`; это означает, что реализация JSF не будет подготавливать к отображению никаких компонентов после завершения запроса Ajax

Следует отметить, что для событий используется следующее соглашение об именах JSF: берется имя события JavaScript и отсекается ведущая подстрока `on`. Таким образом, `onblur` преобразуется в `blur`, `onkeyup` — в `keyup` и т.д. Необходимо также учитывать, что в качестве событий могут применяться события компонента `action` и `valueChange` вместо событий JavaScript.

Атрибуты `onerror` и `onevent` — это функции JavaScript, вызываемые реализацией JSF, когда в жизненном цикле Ajax происходят некоторые предопределенные события. Применительно к успешному запросу Ajax реализация JSF вызывает функцию `onevent` три раза: с началом запроса Ajax, после его завершения и еще раз после завершения, если запрос оказался успешным. Реализация JSF после неудачного запроса Ajax вызывает функцию `onerror` языка JavaScript.

Значением для атрибута `listener` является выражение метода. Реализация JSF вызывает этот метод Java по одному разу в каждом вызове Ajax (в фазе вызова приложения жизненного цикла JSF).

Группы Ajax

Как было описано в разделе "Тег `f:ajax`" на стр. 341, теги `f:ajax` должны быть помещены в тег компонента в целях связывания запроса Ajax с компонентом. Технология JSF позволяет также связывать запрос Ajax с группой компонентов Ajax путем обращения этой структуры:

```
<f:ajax>
  <h:form>
    <h:panelGrid columns="2">
      ...
      <h:inputText id="name" value="#{user.name}"/>
      ...
      <h:inputText id="password" value="#{user.password}"/>
      ...
      </h:commandButton value="Submit" action="#{user.login}"/>
```

```
      </h:panelGrid>
    </h:form>
</f:ajax>
```

В приведенной выше разметке происходит подготовка к использованию в Ajax всех компонентов формы. Если значения полей ввода изменяются или пользователь активизирует кнопку, то реализация JSF передает запрос Ajax на сервер. (Следует учитывать, что в приведенном выше листинге кода реализация JSF не будет подготавливать к отображению никаких компонентов после возврата вызова Ajax, поскольку не задан атрибут `render` для тега `f:ajax`.)

Для некоторых встроенных компонентов JSF предусмотрено событие Ajax по умолчанию. В табл. 10.2 показаны заданные по умолчанию события для компонентов, имеющих тип события по умолчанию.

Таблица 10.2. События Ajax по умолчанию для компонентов JSF

Компоненты	Событие Ajax по умолчанию
Кнопки и ссылки	`action`
Поля ввода текста, текстовые области, секретные поля ввода и все компоненты выбора	`valueChange`

В реализации JSF применяются события Ajax по умолчанию, перечисленные в табл. 10.2, если событие не задано явно. Предусмотрена возможность приблизительно так определить событие:

```
<f:ajax event="click">
  <h:form>
    ...
    <h:inputText id="name" value="#{user.name}">
    ...
    <h:inputText id="password" value="#{user.password}"/>
    ...
    <h:commandButton value="Submit" action="#{user.login}"/>
  </h:form>
</f:ajax>
```

Таким образом, в настоящее время технология Ajax применима только для события щелчка: при щелчке на одном из полей ввода или на кнопке реализация JSF активизирует запрос Ajax. Можно даже вкладывать теги `f:ajax` друг в друга, причем результат становится совокупным, например:

```
<f:ajax event="click">
  <h:form>
    ...
    <h:inputText id="name" value="#{user.name}">
    ...
    <h:inputText id="password" value="#{user.password}"/>
    ...
    <h:commandButton value="Submit" action="#{user.login}">
      <f:ajax event="mouseover"/>
    </h:commandButton>
  </h:form>
</f:ajax>
```

В приведенной выше разметке реализация JSF активизирует запросы Ajax, когда пользователь проводит курсор мыши над кнопкой или активизирует кнопку.

Проверка правильности полей в Ajax

При прочих равных условиях, лучше всего немедленно предоставлять пользователям обратную связь в виде результатов проверки правильности по мере набора ими текста в поле ввода. Проверка правильности значения поля представляет собой один из распространенных вариантов использования Ajax. На рис. 10.5 показан пример проверки правильности значения поля в Ajax. Компонент `name` имеет средство проверки, которое позволяет узнать, содержит ли имя символ подчеркивания; в случае положительного ответа это средство проверки активизирует исключительную ситуацию средства проверки, предусматривающую вывод сообщения об ошибке.

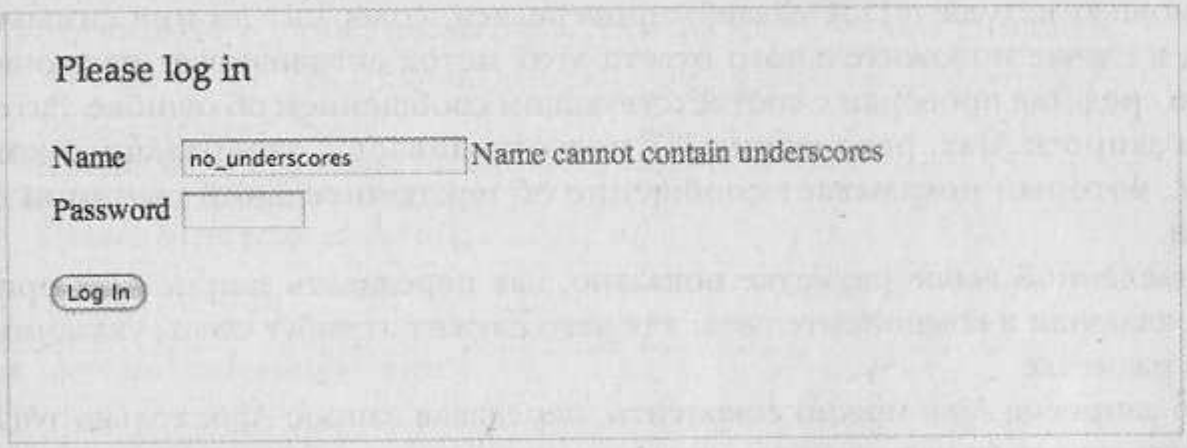


Рис. 10.5. Проверка правильности с помощью технологии Ajax

Ниже показаны разметка для поля ввода имени и связанный с ним тег `h:message`.

```
<h:inputText id="name" value="#{user.name}" validator="#{user.validateName}">
   <f:ajax event="keyup" render="nameError"/>
</h:inputText>
<h:message for="name" id="nameError" style="color: red"/>
```

Для каждого события `keyup` в компоненте `name` реализация JSF передает запрос Ajax и выполняет обработку поля ввода имени.

При выполнении обработки поля ввода имени на сервере реализация JSF вызывает средство проверки поля ввода — метод `validateName()` управляемого бина `user`. Класс `User` представлен в листинге 10.1.

Листинг 10.1. Файл `echo/src/java/com/corejsf/UserBean.java`

```
 1. package com.corejsf;
 2.
 3. import java.io.Serializable;
 4.
 5. import javax.enterprise.context.SessionScoped;
 6. import javax.faces.application.FacesMessage;
 7. import javax.faces.component.UIComponent;
 8. import javax.faces.context.FacesContext;
 9. import javax.faces.validator.ValidatorException;
10. import javax.inject.Named;
11.    // или import javax.faces.bean.SessionScoped;
12.
13. @Named("user") // или @ManagedBean(name="user")
14. @SessionScoped
15. public class UserBean implements Serializable {
16.    private String name = "";
```

```
18.
19.     public String getName() { return name; }
20.     public void setName(String newValue) { name = newValue; }
21.
22.     public String getPassword() { return password; }
23.     public void setPassword(String newValue) { password = newValue; }
24.
25.     public void validateName(FacesContext fc, UIComponent c, Object value) {
26.       if ( ((String)value).contains("_") )
27.          throw new ValidatorException(
28.             new FacesMessage("Name cannot contain underscores"));
29.     }
30. }
```

С помощью метода validateName() проверяется, содержит ли имя символ подчеркивания; в случае положительного ответа этот метод активизирует исключительную ситуацию средства проверки с соответствующим сообщением об ошибке. Затем, после возврата запроса Ajax, реализация JSF подготавливает к отображению компонент nameError, который показывает сообщение об исключительной ситуации средства проверки.

В приведенной выше разметке показано, как передавать запрос Ajax при каждом нажатии клавиши в компоненте name, для чего служит атрибут event, указанный в предыдущей разметке.

Число запросов Ajax можно сократить, передавая запрос Ajax только тогда, когда поле имени теряет фокус, как в следующем примере:

```
<h:inputText id="name" value="#{user.name}" validator="#{user.validateName}">
   <f:ajax event="blur" render="nameError"/>
</h:inputText>
```

Задача проверки полей ввода с помощью технологии Ajax является простой. Теперь рассмотрим немного более сложную задачу.

Мониторинг запросов Ajax

Предусмотрена возможность отслеживать запросы Ajax с помощью атрибута onevent тега f:ajax. Значением этого атрибута должна быть функция JavaScript. Реализация JSF вызывает эту функцию на каждой стадии запроса Ajax: begin, complete и success.

На рис. 10.6 показано приложение, в котором контролируется проверка правильности в Ajax поля ввода имени, описанная в разделе "Проверка правильности поля Ajax" на стр. 345.

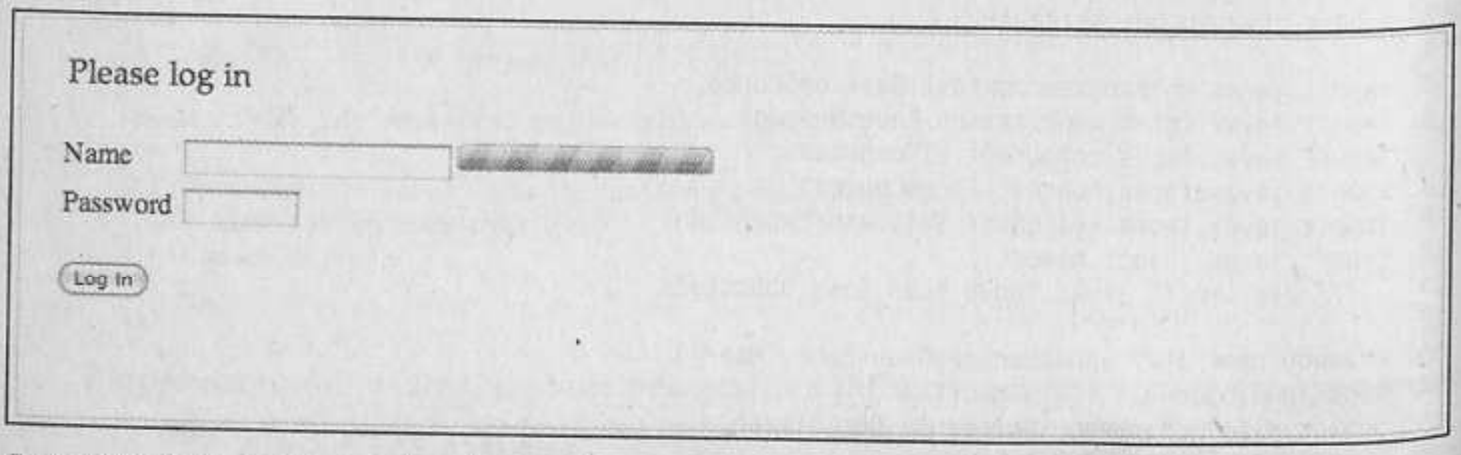


Рис. 10.6. Приложение, контролирующее проверку правильности в Ajax

В то время как проверка правильности происходит на сервере, приложение показывает анимационный индикатор выполнения на клиенте. Необходимо отметить, что этот пример является довольно надуманным; на практике проверка правильности обычно не занимает столько времени, чтобы имело смысл следить за ее ходом с помощью индикатора выполнения.

После возврата вызова Ajax приложение скрывает индикатор выполнения. Относящаяся к этому разметка приведена ниже.

```
<h:outputScript library="javascript" name="prototype-1.6.0.2.js"/>

<script type="text/javascript">
   function showProgress(data) {
      var inputId = data.source.id
      var progressbarId = inputId.substring(0, inputId.length - "name".length)
         + "pole";

        if (data.status == "begin")
           Element.show(progressbarId);
        else if (data.status == "success")
           Element.hide(progressbarId);
    }
</script>

   <h:form id="form" prependId="false">
      ...
      <h:panelGrid columns="2">
         #{msgs.namePrompt}
         <h:panelGroup>
            <h:inputText id="name" value="#{user.name}"
               validator="#{user.validateName}">

               <f:ajax event="blur" render="nameError" onevent="showProgress"/>

               </h:inputText>

               <h:graphicImage id="pole"
                  library="images" name="orange-barber-pole.gif"
                  style="display: none"/>
               <h:message for="name" id="nameError"
                  value="#{user.nameError}" style="color: red"/>
          </h:panelGroup>

   ...

   <p>
      <h:commandButton id="loginButton"
      value="#{msgs.loginButtonText}"
      action="#{user.loginAction}"/>
   </p>

   ...

</h:form>
```

В приведенной выше разметке используется широко известная библиотека Prototype языка JavaScript (см. http://www.prototypejs.org/, чтобы получить дополнительную информацию о библиотеке Prototype) для отображения и сокрытия индикатора выполнения с помощью объекта Element библиотеки Prototype. Для достижения той же цели можно было бы ввести в действие средства языка JavaScript, но для этого требуется применение кода, предназначенного строго для определенного браузера (а такой код авторы предпочитают не использовать в данной книге). Чтобы восполь-

зоваться библиотекой Prototype, можно скопировать файл JavaScript библиотеки Prototype в каталог resources/javascript, а затем обратиться к коду JavaScript с помощью тега h:outputScript.

Кроме того, к компоненту name был добавлен тег f:ajax, а функция showProgress языка JavaScript зарегистрирована как функция обратного вызова. Эта функция проверяет, имеет ли состояние status запроса Ajax значение begin; в случае положительного ответа эта функция отображает индикатор выполнения. С другой стороны, если состоянием запроса Ajax является success, из чего следует, что возврат вызова Ajax и подготовка к отображению были выполнены успешно, то функция скрывает индикатор выполнения. Более строгие определения значений begin, success и complete показаны в табл. 10.3.

Таблица 10.3. Атрибуты объекта данных

Атрибут	Описание
begin	Непосредственно перед тем, как реализация JSF отправляет вызов Ajax на сервер
success	Сразу после подготовки ответа Ajax к отображению
complete	Применительно к успешному вызову реализация JSF вызывает этот метод только после части выполнения жизненного цикла, что по определению означает — непосредственно перед частью подготовки к отображению. Что касается ошибок, то реализация JSF вызывает этот метод непосредственно перед вызовом обработчика ошибок. Обработчик ошибок, как правило, задается с помощью атрибута onevent тега f:ajax или с помощью API-интерфейсов JavaScript технологии JSF

Реализация JSF передает объект данных любой функции JavaScript, зарегистрированной с вызовом Ajax, с помощью атрибута onevent тега f:ajax. Эти атрибуты объекта данных приведены в табл. 10.4.

Таблица 10.4. Дополнительный перечень атрибутов объекта данных

Атрибут	Описание
status	Состояние текущего вызова Ajax. Должно быть одним из следующих: begin, complete или error
Type	Либо event, либо status
Source	Элемент DOM, который является источником события
ResponseXML	Ответ на запрос Ajax. Этот объект в фазе начала запроса Ajax является неопределенным
ResponseText	Ответ XML в виде текста. Этот объект в фазе начала запроса Ajax также является неопределенным
ResponseCode	Числовой код ответа на запрос Ajax. Как и responseXML, и responseText, этот объект является неопределенным в фазе begin запроса Ajax

Пространства имен JavaScript

В разделе "Тег f:ajax" на стр. 341 было показано, как реализовать функцию showProgress() языка JavaScript, в которой используется библиотека Prototype для отображения и сокрытия элемента DOM. Но этот способ ненадежен, поскольку указанный метод может быть перекрыт другим методом JavaScript с тем же именем.

Нельзя исключить вероятность того, что другой программист может реализовать еще одну функцию JavaScript с именем showProgress(), заменив таким образом нашу версию метода своей. Чтобы исключить возможность такой замены, можно сделать имя этой функции более уникальным, например, назвать ее com.corejsf.showProgress().

Руководствуясь этой стратегией защиты разрабатываемого метода JavaScript от переопределения, мы реализовали простую карту, которая служит пространством имен, и определили функции в этой карте[1]:

```
if (!com) var com = {}
if (!com.corejsf) {
   com.corejsf = {
      showProgress: function(data) {
         var inputId = data.source.id
         var progressbarId = inputId.substring(0, inputId.length - "name".length)
            + "pole";

          if (data.status == "begin")
             Element.show(progressbarId);
          else if (data.status == "success")
            Element.hide(progressbarId);
         }
      }
}
```

Поэтому теперь вызывающая программа обращается к указанной функции через пространство имен:

```
<f:ajax event="blur" render="nameError" onevent="com.corejsf.showProgress"/>
```

Определение пространства имен в языке JavaScript не только предотвращает перезапись собственных функций другими программистами, но и служит для читателей конкретного кода указанием на то, что перед ними — программа, составленная программистом на JavaScript высокого уровня.

На заметку! Кроме создания для конкретного случая пространства имен путем помещения функций JavaScript в карту, можно также помещать в отображение данные. К определению пространства имен для данных можно прибегнуть при реализации пользовательских компонентов, которые сопровождают данные на клиенте. Если в программе применяются многочисленные экземпляры одного и того же компонента, они могут перезаписывать данные друг друга. Одним из способов обеспечения того, чтобы многочисленные пользовательские компоненты одного того же типа на основе Ajax, запоминающие данные на клиенте, могли сосуществовать на единственной странице, является помещение данных в карту, ключами для которой служат клиентские идентификаторы компонентов, содержащих данные.

Обработка ошибок в Ajax

Для обработки ошибок можно использовать атрибут onerror тега f:ajax:

```
<f:ajax onerror="handleAjaxError"/>
```

Значением атрибута onerror является функция JavaScript. Реализация JSF вызывает эту функцию, если во время обработки запроса Ajax возникает ошибка. Как и в случае атрибута onevent тега f:ajax, реализация JSF передает функции onerror объект данных. Значения для этого объекта представляют собой то же, что и значения для объекта

[1] Карта фактически является объектным литералом, поскольку в языке JavaScript не предусмотрены структуры данных типа карты, но семантически объектный литерал JavaScript аналогичен карте.

данных, передаваемого реализацией JSF функции обработки событий, как показано в табл. 10.4, за исключением свойства status. Допустимые значения для свойства status приведены в табл. 10.5.

Таблица 10.5. Значения data.status для функций обработки ошибок

Атрибут	Описание
httpError	Значением status ответа является null, или undefined, или status < 200, или status >= 300
emptyResponse	Нет ответа от сервера
malformedXML	Ответ не является формально правильным кодом XML
serverError	Ответ Ajax содержит элемент error, полученный с сервера

Что касается ошибок, то объект данных содержит также три свойства, которыми не обладают объекты событий:

- description.
- errorName.
- errorMessage.

Выше было показано, как использовать тег f:ajax для реализации некоторых обычных вариантов применения Ajax, таких как проверка правильности и мониторинг запросов. Тег f:ajax представляет собой простое и вместе с тем достаточно гибкое инструментальное средство, особенно, когда он применяется для поддержки группирования; но для достижения большего успеха в разработке следует ознакомиться с принципами его работы. Для этого рассмотрим библиотеку JavaScript, которая используется в технологии JSF для реализации тега f:ajax, а также постараемся лучше узнать, как выглядят ответы Ajax JSF.

Ответы Ajax

Ответ на запрос Ajax JSF представляет собой документ XML, который указывает реализации JSF, как обновить страницу XHTML, с которой был запущен запрос. Этот ответ обрабатывается функцией jsf.ajax.response(). Для просмотра ответов Ajax JSF можно воспользоваться расширением Firebug браузера Firefox, как показано на рис. 10.7.

Ответ Ajax JSF представлен в коде XML. В качестве примера ниже приведен ответ, возвращенный после обработки запроса проверки правильности Ajax, который рассматривался в разделе "Проверка правильности поля Ajax" на стр. 345, после того как проверка правильности окончилась неудачей в связи с наличием в поле Name символа подчеркивания.

```
<?xml version="1.0" encoding="utf-8"?>
<partial-response>
   <changes>
      <update id="j_idt18:nameError">
         <![CDATA[<span id="j_idt18:nameError" style="color: red">
            Name cannot contain underscores
         </span>]]>
      </update>
      <update id="javax.faces.ViewState">
```

```
        <![CDATA[-4047143760309857992:5238789135448605596]]>
      </update>
    </changes>
</partial-response>
```

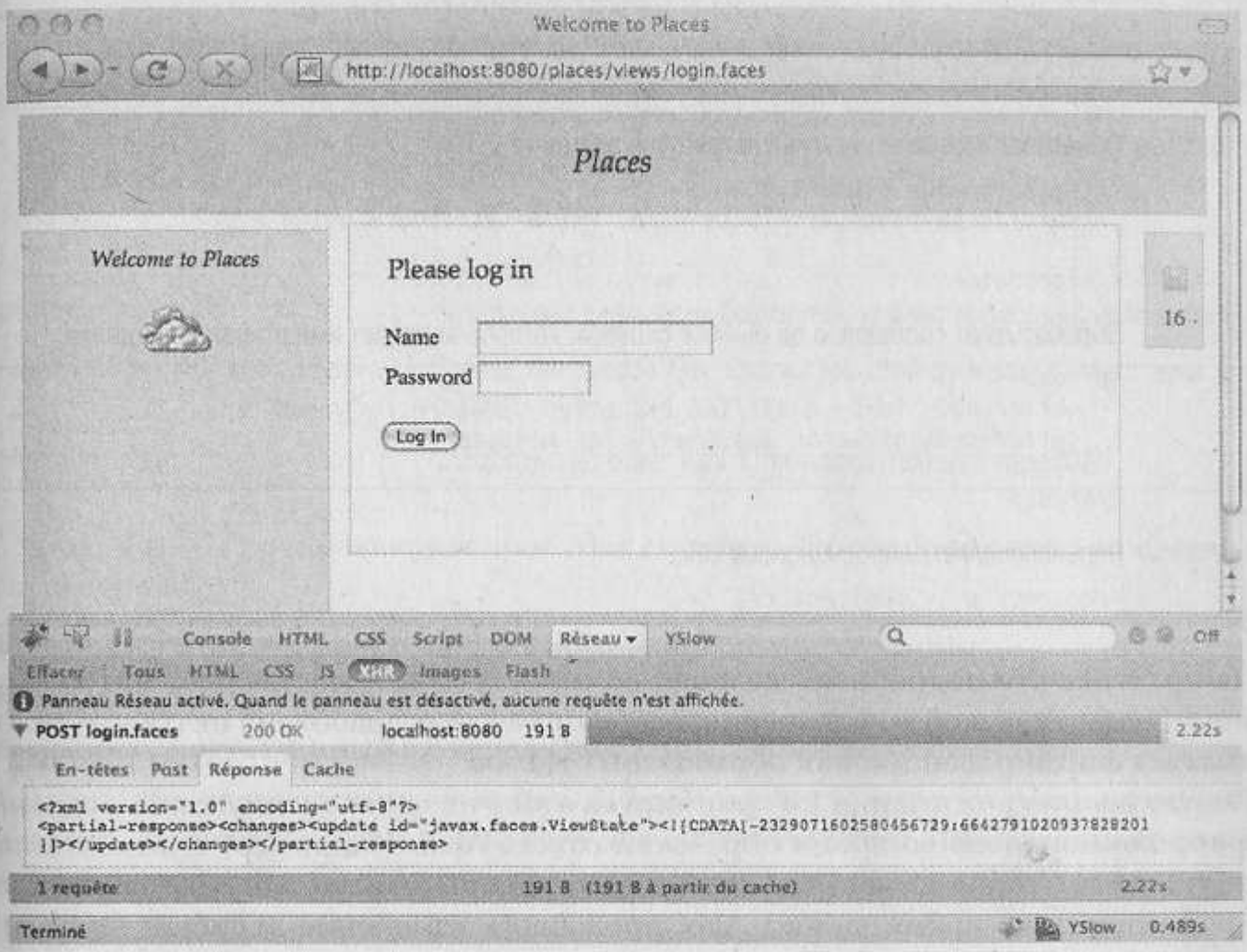


Рис. 10.7. Просмотр ответа Ajax с использованием расширения Firebug

В табл. 10.6 приведены допустимые элементы XML, применимые в ответах Ajax JSF.

Таблица 10.6. Элементы ответов Ajax JSF

Элемент	Описание

`insert`	Вставляет элемент DOM с указанным идентификатором перед существующим элементом:
	`<insert id="insert id" before="before id">`
	`  <![CDATA[...]]>`
	`</insert>`

`update`	Обновляет элемент DOM:
	`<update id="update id">`
	`  <![CDATA[...]]>`
	`</insert>`
	В дополнение к определению подлежащего обновлению клиентского идентификатора элемента DOM идентификатором обновления может быть один из следующих:
	• `javax.faces.ViewRoot`: обновляет весь элемент DOM;
	• `javax.faces.ViewState`: обновляет все состояние отправки формы;
	• `javax.faces.ViewBody`: обновляет текст страницы

Окончание табл. 10.6

Элемент	Описание

`delete`	Удаляет элемент DOM с указанным идентификатором:
	`<delete id="delete id">`
	`  <![CDATA[...]]>`
	`</attribute>`

`attributes`	Обновляет один или несколько атрибутов элемента DOM:
	`<attributes id="element id">`
	`  <attribute name="attribute name" value="attribute value"/>`
	`  ...`
	`</attribute>`

`error`	Вырабатывает сообщение об ошибке сервера, которое включает имя и текст сообщения:
	`<error>`
	`  <error-name>fully qualified exception class</error-name>`
	`  <error-message>error message</error-message>`
	`  ...`
	`</error>`

`redirect`	Перенаправляет запрос к новому URL: `<redirect url="redirect url"/>`

На практике эти подробные сведения об ответах Ajax представляют интерес в основном с теоретической точки зрения, особенно, если разработчик не собирается сам заниматься формированием или обработкой ответов.

По умолчанию реализация JSF формирует ответ на основе изменений, внесенных в дерево компонентов во время обработки ответа Ajax. Например, если с помощью атрибута `render` тега `h:ajax` задан компонент, который должен быть подготовлен к отображению, а во время вызова Ajax произошло изменение атрибута `style` этого компонента, то реализация JSF вырабатывает частичный ответ, применимый для обновления элемента DOM, связанного с данным компонентом.

Реализация JSF обрабатывает ответ с помощью функции `jsf.ajax.response()`, которая определена в коде JavaScript, предусмотренном в реализации JSF.

На заметку! Схему XML для ответов Ajax JSF можно найти в приложении A, раздел 1.3 спецификации JSF.

Библиотека JavaScript в версии JSF 2.0

Тег `f:ajax` представляет собой удобный способ реализации простых функциональных средств Ajax. Но функциональные возможности `f:ajax` являются ограниченными именно потому, что это — тег. Но этот тег основан на библиотеке JavaScript, встроенной в реализацию JSF, поэтому предусмотрена возможность непосредственно использовать эту библиотеку JavaScript для реализации более сложных сценариев Ajax.

Доступ к встроенной библиотеке JavaScript реализации JSF можно обеспечить в конкретных файлах XHTML:

```
<h:outputScript library="javax.faces" name="jsf.js"/>
```

В состав этой библиотеки входит набор функций Ajax, перечисленных в табл. 10.7.

Таблица 10.7. Функции jsf.ajax

Функция	Описание
`addOnError(callback)`	Функция JavaScript, вызываемая реализацией JSF, когда вызов Ajax приводит к ошибке
`addOnEvent(callback)`	Функция JavaScript, вызываемая реализацией JSF применительно к событиям Ajax. Эта функция вызывается три раза на протяжении срока существования успешного вызова Ajax: • `begin` • `complete` • `success`
`isAutoExec()`	Возвращает `true`, если браузер выполняет обработанный код JavaScript
`request(source, event, options)`	Отправляет запрос Ajax. Параметры этой функции соответствуют атрибутам `f:ajax`
`response(request, context)`	Обрабатывает ответ Ajax. Ответ представлен в коде XML

Метод `request()` отправляет запрос Ajax серверу. Этот запрос всегда представляет собой следующее:

- запрос `POST` к действию, вложенному в окружающую форму;
- асинхронный запрос;
- запрос, поставленный в очередь вместе с другими запросами Ajax.

В теге `f:ajax` используются функции API-интерфейса JavaScript Ajax, приведенные в табл. 10.7, поэтому можно обойти этот тег и использовать API непосредственно на конкретных страницах XHTML. Например, вместо следующего кода:

```
<h:inputText...>
   <f:ajax event="blur" render="nameError" onevent="com.corejsf.showProgress"/>
</h:inputText>
```

можно использовать такой код:

```
<h:outputScript library="javax.faces" name="jsf.js"/>

<h:inputText
   onblur="jsf.ajax.request(this, event,
            { render: 'nameError',
              onevent: com.corejsf.showProgress
            })"/>
```

Необходимо отметить одно тонкое различие между двумя приведенными выше идентичными вариантами использования тега `f:ajax` и API-интерфейса JavaScript: атрибуты `f:ajax` всегда представляют собой строки, тогда как атрибут `onevent` параметров, передаваемых методу `jsf.ajax.request()`, является методом, а не строкой.

В табл. 10.8 перечислены допустимые ключи и значения для карты параметров `jsf.ajax.request()`.

Подробное описание библиотеки JavaScript JSF можно найти в спецификации JSF и в документации JavaScript, которая входит в поставку JSF. Можно также загрузить справочный исходный код реализации JSF, который включает библиотеку JavaScript.

Для запроса Ajax можно не только задать параметры, приведенные в табл. 10.8, но и ввести дополнительные параметры для любого запроса Ajax.

Таблица 10.8. Ключи и значения для параметров метода
`jsf.ajax.request(source, event, options)`

Ключ	Значение
`execute`	Разделенный пробелами список компонентов, выполняемых реализацией JSF в фазе выполнения жизненного цикла. Ключевые слова: • `@this` • `@form` • `@all` • `@none`
`render`	Разделенный пробелами список компонентов, выполняемых реализацией JSF в фазе подготовки к отображению жизненного цикла
`onevent`	Функция JavaScript, вызываемая реализацией JSF применительно к событиям Ajax. Эта функция вызывается три раза на протяжении срока существования вызова Ajax: • `begin` • `complete` • `success`
`onerror`	Функция JavaScript, вызываемая реализацией JSF, если вызов Ajax приводит к ошибке

Передача дополнительных параметров запроса Ajax

Иногда возникает необходимость в выполнении дополнительных операций вместе с вызовом Ajax. Например, в коде компонента текстового поля с автоматическим завершением, который будет рассматриваться в разделе "Использование Ajax в составных компонентах" на стр. 357, вычисляется местоположение окна списка, содержащего завершенные элементы, которые отображаются под полем ввода текста. Затем эта информация передается наряду с запросом Ajax, как показано ниже.

```
if (!com) var com = {};

if (!com.corejsf) {
   var focusLostTimeout;

      com.corejsf = {
         updateCompletionItems: function(input, event) {
            var keystrokeTimeout;

            var ajaxRequest = function() {
               jsf.ajax.request(input, event,
               { render: com.corejsf.getListboxId(input),
                     x: Element.cumulativeOffset(input)[0],
                     y: Element.cumulativeOffset(input)[1] + Element.getHeight(input)
               });
            }

            window.clearTimeout(keystrokeTimeout);
            keystrokeTimeout = window.setTimeout(ajaxRequest, 350);
      }, ...
   ...
}
```

Реализация JSF передает все пары "ключ–значение" в опциях Ajax, которые не перечислены в табл. 10.8, в запрос Ajax в виде параметров. Именами параметров являются ключи, а значениями параметров — значения, связанные с ключами. В приведенном выше коде эти значения представляют собой координаты x и y окна списка. Мы используем функцию Element.cumulativeOffset() библиотеки Prototype для вычисления этих координат. Для предыдущего вызова Ajax эти координаты x и y используются на сервере следующим образом:

```
package com.corejsf;

...

@ManagedBean()
@SessionScoped()
public class AutocompleteListener {
   ...
   private void setListboxStyle(int rows, Map<String, Object> attrs) {
      if (rows > 0) {
         Map<String, String> reqParams = FacesContext.getCurrentInstance()
           .getExternalContext().getRequestParameterMap();

         attrs.put("style", "display: inline; position: absolute; left: "
   + reqParams.get("x") + "px;" + " top: " + reqParams.get("y") + "px");
            }
            else
               attrs.put("style", "display: none;");
      }

   ...
}
```

Организация очередей событий

Реализация JSF автоматически ставит запросы Ajax в очередь и выполняет их последовательно так, что предыдущий запрос Ajax всегда завершается до начала следующего.

Но реализация JSF ставит в очередь только вызовы Ajax; постановка в очередь обычных запросов HTTP не осуществляется. Поскольку реализация JSF не выполняет постановку в очередь обычных запросов HTTP, смешивание запросов Ajax и HTTP может привести к непредсказуемому поведению программы. В качестве примера рассмотрим следующий код:

```
<h:form>
   <h:inputText ...>
      <f:ajax onblur="..."/>
   </h:inputText>

   <h:commandButton value="submit" action="nextPage"/>
</h:form>
```

В приведенной выше разметке применяется поле ввода Ajax в форме с кнопкой, при создании которой не используется Ajax. Попробуем представить себе, что произойдет, если поле ввода получит фокус и пользователь щелкнет на кнопке. Поле ввода потеряет фокус и произойдет запуск вызова Ajax, а непосредственно после этого реализация JSF выполнит обычную отправку формы в результате активизации кнопки. Завершится ли вызов Ajax перед отправкой формы? Это возможно, особенно, если вызов Ajax не потребует много времени, но велика вероятность того, что отправка формы приведет к прерыванию вызова Ajax, а в связи с тем, что вся суть технологии

Ajax состоит в манипулировании серверными данными, можно легко перевести программу в непредсказуемое состояние.

Рекомендация, касающаяся смешивания запросов Ajax и обычных запросов HTTP, проста: так поступать не следует. Вместо этого необходимо выполнять обычные запросы HTTP по такому же принципу, как и запросы Ajax, благодаря чему реализация JSF будет ставить их в очередь как запросы Ajax, гарантируя завершение предыдущего запроса Ajax до начала следующего. В приведенной выше разметке достаточно просто подготовить кнопку к использованию в Ajax:

```
<h:form>
   <h:inputText ...>
      <f:ajax onblur="..."/>
   </h:inputText>
   <h:commandButton value="submit" action="nextPage">
      <f:ajax/>
   </h:commandButton>
</h:form>
```

Объединение событий

Вообще говоря, иногда в работе с технологией Ajax возникают определенные нюансы. Например, если речь идет о поле ввода текста с поддержкой Ajax, то теоретически не исключена ситуация, что пользователь будет набирать текст настолько быстро, что обнаружится заметная задержка между нажатием клавиш и отображением введенных символов.

В таких случаях может потребоваться объединять вызовы Ajax в целях периодического выполнения вызовов к серверу, а не создавать вызовы для каждого события. В версии JSF 2.0 не предусмотрена явная поддержка объединения вызовов Ajax, но это несложно сделать самому с помощью таймера JavaScript. Например, в компоненте с автоматическим завершением, который рассматривается в разделе "Использование Ajax в составных компонентах" на стр. 357, вызовы Ajax объединяются с помощью функции JavaScript:

```
<h:inputText id="input" value="#{cc.attrs.value}"
   onkeyup="com.corejsf.updateCompletionItems(this, event)" ... />
```

Для объединения событий служит функция `updateCompletionItems`:

```
updateCompletionItems: function(input, event) {
   var keystrokeTimeout;
      var ajaxRequest = function() {
         jsf.ajax.request(input, event,
            { render: corejsf.getListboxId(input),
                  x: Element.cumulativeOffset(input)[0],
                  y: Element.cumulativeOffset(input)[1] + Element.getHeight(input)
            });
      }

      window.clearTimeout(keystrokeTimeout);
      keystrokeTimeout = window.setTimeout(ajaxRequest, 350);
   }
   ...
}
```

При возникновении события `keyup` в поле ввода мы планируем вызов Ajax, который должен произойти по истечении 350 мс. Если следующее событие `keyup` происходит на протяжении этих 350 мс, то предыдущий вызов Ajax отменяется и планируется

следующий, через новый интервал в 350 мс. Поэтому отправка вызова Ajax на сервер осуществляется только после того, как пользователь выдерживает паузу в 350 мс или больше между нажатиями клавиш.

Перехват значений функции jsf.ajax.request()

Технология JSF позволяет связать функцию Java с запросом Ajax с помощью атрибута listener тега f:ajax:

```
<h:inputText value="...">
   <f:ajax event="keyup" listener="#{someBean.someMethod}"/>
</h:inputText>
```

Однако тег f:ajax позволяет добавить прослушиватель лишь для конкретного события, связанного с конкретным компонентом. Но иногда возникает необходимость в добавлении определенных функциональных возможностей к каждому запросу Ajax. Это может потребоваться по весьма многим причинам. Возможно, есть необходимость в повышении безопасности каждого вызова Ajax или требуется регистрировать каждый вызов Ajax на сервере. В технологии JSF отсутствует конкретное решение по реализации таких функциональных средств.

В разделе "Пространства имен JavaScript" на стр. 348 было показано, как использовать карту для реализации произвольного пространства имен JavaScript, которое помогало бы защищать пользовательские функции JavaScript от перезаписи другими функциями с такими же именами.

Запросы Ajax JSF реализуются с помощью функции JavaScript, а именно jsf.ajax.request(), и, как любая функция JavaScript, она легко допускает перехват своих результатов, как показано ниже.

```
var builtinAjaxRequestFunction = jsf.ajax.request;
jsf.ajax.request = function(c,e,o) {
    alert("hello")
    builtinAjaxRequestFunction(c,e,o)
    alert("bye")
}
```

После перехвата результатов выполнения функции jsf.ajax.request() с помощью кода JavaScript, приведенного выше, каждый вызов Ajax, будь то вызов, инициированный тегом f:ajax или активизированный функцией jsf.ajax.request(), приводит к вызову перехваченной функции, которая в данном случае лишь показывает предупреждения до и после отправки вызова Ajax на сервер. Читателю, возможно, потребуется внести изменения в эту разметку в целях осуществления более полезных функций по сравнению лишь с показом предупреждений.

Использование Ajax в составных компонентах

Несомненно, двумя наиболее важными особенностями версии JSF 2.0 являются встроенные средства Ajax и поддержка составных компонентов. Эти возможности достаточно несложно объединить, поэтому легко решается задача реализации составных компонентов на основе Ajax. В качестве примера на рис. 10.8 показано текстовое поле с автоматическим завершением, которое реализовано как составной компонент JSF со встроенными средствами Ajax.

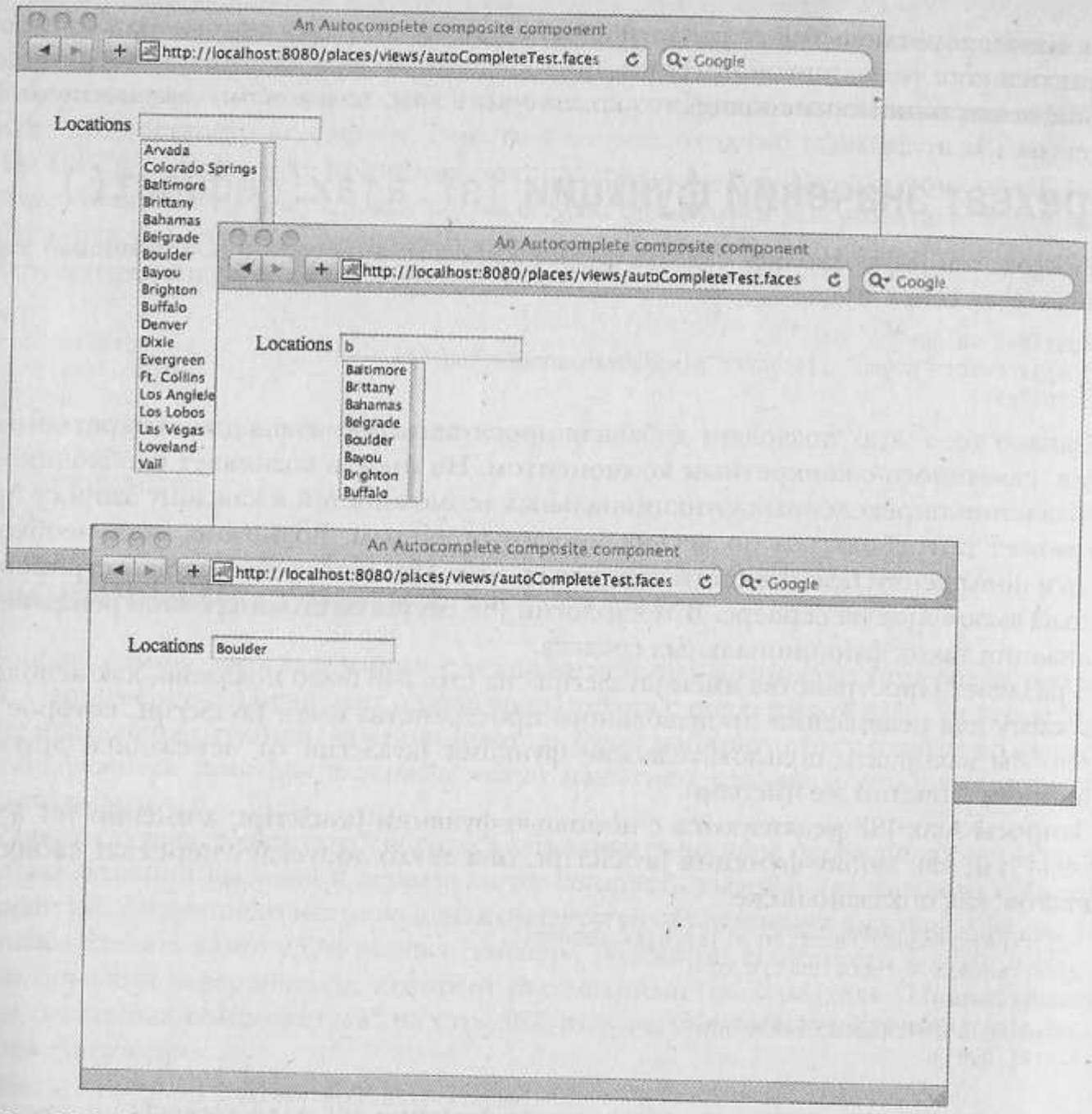


Рис. 10.8. Составной компонент с автоматическим завершением

Компонент с автоматическим завершением состоит из четырех файлов.

1. Файл XHTML, который определяет интерфейс компонента и реализацию.
2. Файл JavaScript с кодом, характерным для этого компонента.
3. Файл JavaScript для библиотеки Prototype на языке JavaScript.
4. Класс Java, который реализует методы прослушивателя для компонента поля ввода текста с автоматическим завершением и окно списка.

Компонент с автоматическим завершением также состоит из поля ввода текста и окна списка. Первоначально окно списка скрыто.

Впоследствии в ответ на события `keyup` в поле ввода текста реализация JSF выполняет вызовы Ajax к серверу. На сервере прослушиватель изменений значений, связанный с полем ввода текста, проверяет значение поля ввода по списку завершенных элементов, связанных с полем ввода, и заполняет окно списка соответствующими элементами. Если какие-либо соответствующие элементы отсутствуют, прослушива-

тель изменений значений задает стиль окна списка так, что это окно отображается под полем ввода, с которым оно связано.

Ниже показано, как используется компонент с автоматическим завершением, приведенный на рис. 10.8.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:util="http://java.sun.com/jsf/composite/util">

  <h:head><title>An Autocomplete composite component</title></h:head>

  <h:body>
    <div style="padding: 20px;">
      <h:form>
        <h:panelGrid columns="2">

            Locations

            <util:autoComplete value="#{user.city}"
            completionItems="#{autoComplete.locations}" />

        </h:panelGrid>
      </h:form>
    </div>
  </h:body>
</html>
```

В приведенной выше разметке просто объявляется пространство имен для компонента (`util`) и используется тег компонента (`util:autoComplete`). Атрибут `completionItems` указывает на свойство `String[]` управляемого бина `autoComplete`. Этот бин имеет простую реализацию:

```
@ApplicationScoped
public class AutoComplete {
   public String[] getLocations() {
      return new String[] {
          "Arvada", "Colorado Springs", "Baltimore", "Brittany", "Bahamas",
          "Belgrade", "Boulder", "Bayou", "Brighton", "Buffalo", "Denver", "Dixie",
          "Evergreen", "Ft. Collins", "Los Angeles", "Los Lobos", "Las Vegas",
          "Loveland", "Vail"
      };
   }
}
```

В листингах 10.2–10.4 показан код для компонента с автоматическим завершением, включая определение компонента, а также связанные с ним код JavaScript и вспомогательный бин.

В отношении этого компонента с автоматическим завершением необходимо сделать несколько замечаний.

- Листинг 10.2. В компоненте с автоматическим завершением используются и тег `f:ajax`, и API-интерфейс JavaScript JSF.
- Листинг 10.3. Компонент с автоматическим завершением добавляет два параметра запроса к запросу Ajax: координаты `x` и `y` верхнего левого угла окна списка.
- Листинг 10.3. В приложении события `keyup` объединяются, поэтому вызов Ajax активизируется, только если пользователь приостанавливает набор символов

на 350 мс. Это позволяет предотвратить передачу подряд многочисленных запросов на сервер в том случае, если над вводом текста работает умелый наборщик.

- Листинг 10.3. После того как поле ввода текста с автоматическим завершением теряет фокус, этот составной компонент скрывает окно списка.
- Листинг 10.4. Каждый компонент с автоматическим завершением хранит свой список завершенных элементов (отображаемых в окне списка) в одном из атрибутов окна списка компонента с автоматическим завершением. Благодаря этому обеспечивается возможность размещать несколько компонентов с автоматическим завершением на одной странице.

Но если поле ввода текста теряет фокус, то в окне списка ни в коем случае не отображается выбор элемента, поскольку окно списка становится скрытым после перемещения фокуса с поля ввода. Таким образом, благодаря использованию такого же приема, как и при объединении событий, в составном компоненте обеспечивается ожидание в течение 200 мс, когда поле ввода теряет фокус, и только после этого фактически происходит сокрытие окна списка. Поэтому окно списка остается видимым, когда пользователь выбирает элемент.

Листинг 10.2. Файл `autoComplete/web/resources/util/autoComplete.xhtml`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:composite="http://java.sun.com/jsf/composite">
<head><title>IGNORED</title></head>
<body>
   <ui:composition>
     <composite:interface>
       <composite:attribute name="value" required="true"/>
       <composite:attribute name="completionItems" required="true"/>
     </composite:interface>
     <composite:implementation>
       <h:outputScript library="javascript"
          name="prototype-1.6.0.2.js" target="head"/>
       <h:outputScript library="javascript"
          name="autoComplete.js" target="head"/>

       <h:inputText id="input" value="#{cc.attrs.value}"
           valueChangeListener="#{autocompleteListener.valueChanged}"
           onkeyup="com.corejsf.updateCompletionItems(this, event)"
           onblur="com.corejsf.inputLostFocus(this)"/>

       <h:selectOneListbox id="listbox" style="display: none"
           valueChangeListener="#{autocompleteListener.completionItemSelected}"
           onfocus="com.corejsf.listboxGainedFocus()">

          <f:selectItems value="#{cc.attrs.completionItems}"/>
          <f:ajax render="input"/>

       </h:selectOneListbox>
     </composite:implementation>
   </ui:composition>
  </body>
</html>
```

Листинг 10.3. Файл autoComplete/web/resources/javascript/autoComplete.js

```
if   (!com) var com = {}
if   (!com.corejsf) {
     var focusLostTimeout
     com.corejsf = {
        errorHandler: function(data) {
            alert("Error occurred during Ajax call: " + data.description)
        },

       updateCompletionItems: function(input, event) {
          var keystrokeTimeout

            jsf.ajax.addOnError(com.corejsf.errorHandler)

            var ajaxRequest = function() {
               jsf.ajax.request(input, event, {
                  render: com.corejsf.getListboxId(input),
                  x: Element.cumulativeOffset(input)[0],
                  y: Element.cumulativeOffset(input)[1] + Element.getHeight(input)
               })
            }

            window.clearTimeout(keystrokeTimeout)
            keystrokeTimeout = window.setTimeout(ajaxRequest, 350)
        },

        inputLostFocus: function(input) {
           var hideListbox = function() {
              Element.hide(com.corejsf.getListboxId(input))
           }

            focusLostTimeout = window.setTimeout(hideListbox, 200)
        },

        getListboxId: function(input) {
           var clientId = new String(input.name)
           var lastIndex = clientId.lastIndexOf(":")
           return clientId.substring(0, lastIndex) + ":listbox"
        }
   }
}
```

Листинг 10.4. Файл autoComplete/src/java/com/corejsf/AutoCompleteListener.java

```
package com.corejsf;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;
import javax.faces.component.UIInput;
import javax.faces.component.UISelectItems;
import javax.faces.component.UISelectOne;
import javax.faces.context.FacesContext;
import javax.faces.event.ValueChangeEvent;

```

```
17.
18. @Named // @ManagedBean
19. @SessionScoped
20. public class AutocompleteListener {
21.    private static String COMPLETION_ITEMS_ATTR = "corejsf.completionItems";
22.
23.    public void valueChanged(ValueChangeEvent e) {
24.       UIInput input = (UIInput)e.getSource();
25.       UISelectOne listbox = (UISelectOne)input.findComponent("listbox");
26.
27.       if (listbox != null) {
28.          UISelectItems items = (UISelectItems)listbox.getChildren().get(0);
29.            Map<String, Object> attrs = listbox.getAttributes();
30.            List<String> newItems = getNewItems((String)input.getValue(),
31.               getCompletionItems(listbox, items, attrs));
32.
33.            items.setValue(newItems.toArray());
34.            setListboxStyle(newItems.size(), attrs);
35.       }
36.    }
37.
38.    private List<String> getNewItems(String inputValue, String[] completionItems) {
39.       List<String> newItems = new ArrayList<String>();
40.
41.       for (String item : completionItems) {
42.          String s = item.substring(0, inputValue.length());
43.          if (s.equalsIgnoreCase(inputValue))
44.             newItems.add(item);
45.       }
46.
47.       return newItems;
48.    }
49.
50.    private void setListboxStyle(int rows, Map<String, Object> attrs) {
51.       if (rows > 0) {
52.          Map<String, String> reqParams = FacesContext.getCurrentInstance()
53.             .getExternalContext().getRequestParameterMap();
54.
55.          attrs.put("style", "display: inline; position: absolute; left: "
56.             + reqParams.get("x") + "px;" + " top: " + reqParams.get("y") + "px");
57.       }
58.       else
59.          attrs.put("style", "display: none;");
60.    }
61.
62.    private String[] getCompletionItems(UISelectOne listbox,
63.       UISelectItems items, Map<String, Object> attrs) {
64.          String[] completionItems = (String[])attrs.get(COMPLETION_ITEMS_ATTR);
65.
66.          if (completionItems == null) {
67.             completionItems = (String[])items.getValue();
68.             attrs.put(COMPLETION_ITEMS_ATTR, completionItems);
69.          }
70.       return completionItems;
71.    }
72.
73.    public void completionItemSelected(ValueChangeEvent e) {
74.       UISelectOne listbox = (UISelectOne)e.getSource();
75.       UIInput input = (UIInput)listbox.findComponent("input");
76.
77.       if(input != null) {
78.          input.setValue(listbox.getValue());
```

```
79.         }
80.         Map<String, Object> attrs = listbox.getAttributes();
81.         attrs.put("style", "display: none");
82.     }
83. }
```

Резюме

В версии JSF 2.0 предусмотрена возможность применения надежной инфраструктуры Ajax, с помощью которой можно реализовать весьма развитые пользовательские интерфейсы. На самом высоком уровне абстракции в технологии JSF предусмотрен тег `f:ajax`, который позволяет закреплять правила поведения Ajax за компонентами. Благодаря использованию тега `f:ajax` обеспечивается единообразие подхода к осуществлению также других правил поведения JSF, подобных тем, что реализуются с помощью средств проверки и преобразователей, которые также закрепляются за компонентами с помощью встроенных тегов из основной библиотеки JSF (`f:validator` и `f:converter` соответственно). Такое единообразие становится причиной того, что средства Ajax, доступные с помощью тега `f:ajax`, выглядят вполне естественными для разработчиков на JSF.

На самом нижнем уровне абстракции можно непосредственно использовать API-интерфейс JavaScript JSF (применяемый в технологии JSF для реализации тега `f:ajax`). Если API-интерфейс JavaScript используется непосредственно, то приходится создавать чуть больший объем кода, чем при использовании тега `f:ajax`, но разработчик получает возможность добиться большего разнообразия, поскольку может реализовать дополнительные функциональные средства применительно к вызовам Ajax.

Встроенные функциональные возможности поддержки Ajax в версии JSF 2.0 могут служить для воплощения в жизнь многих вариантов применения Ajax. Несомненно, последующие версии JSF также будут основаны на этих функциональных возможностях и дополнят средства Ajax, предоставляемые платформой JSF.

Пользовательские компоненты, преобразователи и средства проверки

В этой главе...

- Реализация классов компонентов
- Кодирование: формирование разметки
- Декодирование: обработка значений запроса
- Файл описания библиотеки тегов JSF 1.2
- Использование внешнего средства подготовки к отображению
- Обработка атрибутов тегов JSF 2.0
- Разработка кода JavaScript
- Использование дочерних компонентов и аспектов
- Сохранение и восстановление состояния
- Создание компонентов Ajax JSF 2.0

Глава 11

В технологии JSF предусмотрен основной набор компонентов для создания веб-приложений на основе HTML, таких как текстовые поля, флажки, кнопки и т.д. В главе 9 было показано, как создавать из этих компонентов более сложные составные компоненты. Однако составные компоненты ограничены относительно простыми конфигурациями. Например, с помощью составного компонента нельзя отобразить дерево или календарь в виде таблицы. К счастью, технология JSF позволяет также создавать пользовательские компоненты с более развитым поведением.

В API-интерфейсе JSF предусмотрены возможности, позволяющие реализовать пользовательские компоненты и связанные с ними теги с такими же функциями, которыми обладают стандартные теги JSF. Например, в теге `h:inputText` используется выражение значения для связывания значения текстового поля со свойством бина, что позволяет применять выражение значения типа `java.util.Date` в компоненте календаря. Стандартные компоненты ввода JSF активизируют события изменения значения при изменении их значений, что дает возможность активизировать события изменения значения, например, после выбора другой даты в календаре.

В начале этой главы будет рассматриваться компонент счетчика (рис. 11.1), иллюстрирующий важные проблемы, с которыми приходится сталкиваться во всех пользовательских компонентах. Затем проводится усовершенствование этого счетчика для ознакомления с более сложными проблемами, как указано в соответствующих разделах главы.

- Использование внешнего средства подготовки к отображению.
- Обработка атрибутов тегов.
- Разработка кода JavaScript.

После этого мы перейдем к описанию компонента области окна с вкладками (также показанного на рис. 11.1), который иллюстрирует следующие особенности разработки пользовательского компонента.

- Обработка дочерних тегов `SelectItem`.
- Обработка аспектов.
- Использование скрытых полей.
- Сохранение и восстановление состояния.

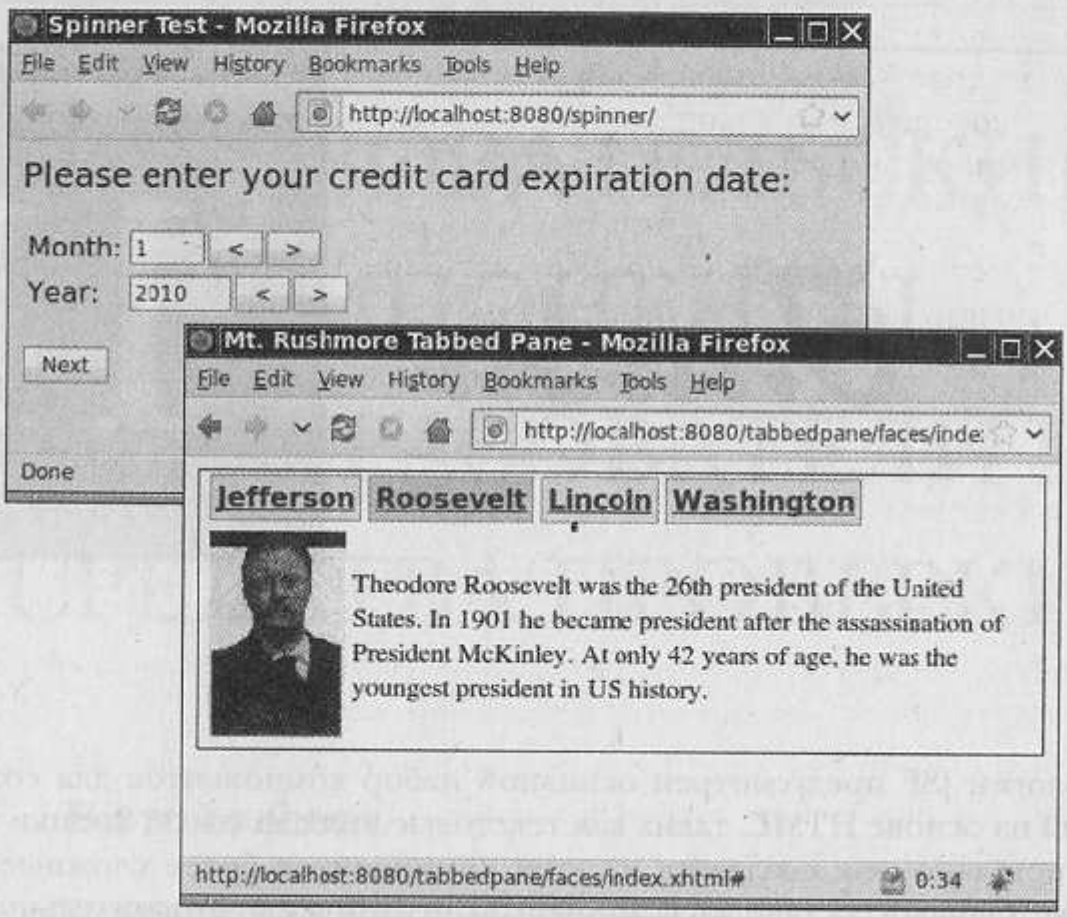


Рис. 11.1. Счетчик и область окна с вкладками

Наконец, мы возвратимся к счетчику, чтобы показать, как можно ввести в пользовательских компонентах средства поддержки Ajax (раздел "Создание компонентов Ajax" на стр. 409).

Реализация классов компонентов

При разработке пользовательского компонента необходимо реализовать класс компонента, который должен выполнять ряд функций.

- Поддержка состояния компонента (например, минимальное, максимальное и текущее значения счетчика).
- Выработка кода пользовательского интерфейса путем записи разметки (в случае счетчика – кода HTML для поля ввода и кнопок).
- Декодирование запросов HTTP (таких, которые формируются после щелчка на кнопках счетчика).

В соответствии с принятым соглашением имя класса компонента имеет префикс `UI`, например `UISpinner`.

Классы компонента могут передавать функции кодирования и декодирования отдельному средству подготовки к отображению. Используя разные средства подготовки к отображению, можно поддерживать различные клиенты, такие как веб-браузеры и сотовые телефоны. Первоначально рассматриваемый компонент счетчика сам выполняет задачу подготовки к отображению, но в разделе "Использование внешнего средства подготовки к отображению" на стр. 380 будет показано, как реализовать отдельное средство подготовки к отображению для счетчика.

Класс компонента должен быть подклассом класса UIComponent. Этот класс определяет более 40 абстрактных методов, поэтому может потребоваться расширить существующий класс, который их реализует. Предусмотрена возможность выбирать среди классов, показанных на рис. 11.2. Обычно создается подкласс одного из трех стандартных классов компонентов.

- UIOutput, если компонент отображает значение, но не позволяет пользователю его редактировать.
- UIInput, если компонент читает значение, введенное пользователем (такое как значение счетчика).
- UICommand, если компонент производит действия, аналогичные действиям командной кнопки или ссылки.

Как показано на рис. 11.2, эти три класса реализуют интерфейсы, которые определяют следующие разнообразные функции.

- Интерфейс ValueHolder определяет методы, позволяющие управлять значением компонента, локальным значением и преобразователем.
- Интерфейс EditableValueHolder расширяет интерфейс ValueHolder и добавляет методы, предназначенные для управления средствами проверки и прослушивателями изменений значений.
- Интерфейс ActionSource определяет методы, обеспечивающие управление прослушивателями действий.
- Интерфейс ActionSource2 определяет методы управления действиями.

На заметку! Интерфейс ActionSource2 был введен в версии JSF 1.2. В версии JSF 1.1 интерфейс ActionSource определял методы управления и действиями, и прослушивателями действий.

Класс UIComponent управляет несколькими важными категориями данных.

- Список дочерних компонентов. В качестве примера можно указать, что дочерние компоненты компонента h:panelGrid представляют собой компоненты, которые размещаются в местоположении, указанном с помощью сетки. Однако компонент не обязательно должен иметь какие-либо дочерние компоненты.
- Карта компонентов аспекта. Аспекты аналогичны дочерним компонентам, но для доступа к каждому аспекту применяется ключ, а не позиция в списке. Задача размещения собственных аспектов возлагается на компонент. Например, компонент h:dataTable имеет аспекты заголовка и нижнего колонтитула.
- Карта атрибутов. Это универсальная карта, которую можно использовать для хранения произвольных пар "ключ–значение".
- Карта выражений значения. Это еще одна универсальная карта, которая может служить для хранения произвольных выражений значения. Например, если тег счетчика имеет атрибут value="#{cardExpirationDate.month}", то компонент хранит объект ValueExpression для данного выражения значения под ключом "value".
- Коллекция прослушивателей событий. Прослушиватели получают уведомление, когда происходит рассылка события, источником которого является данный компонент.

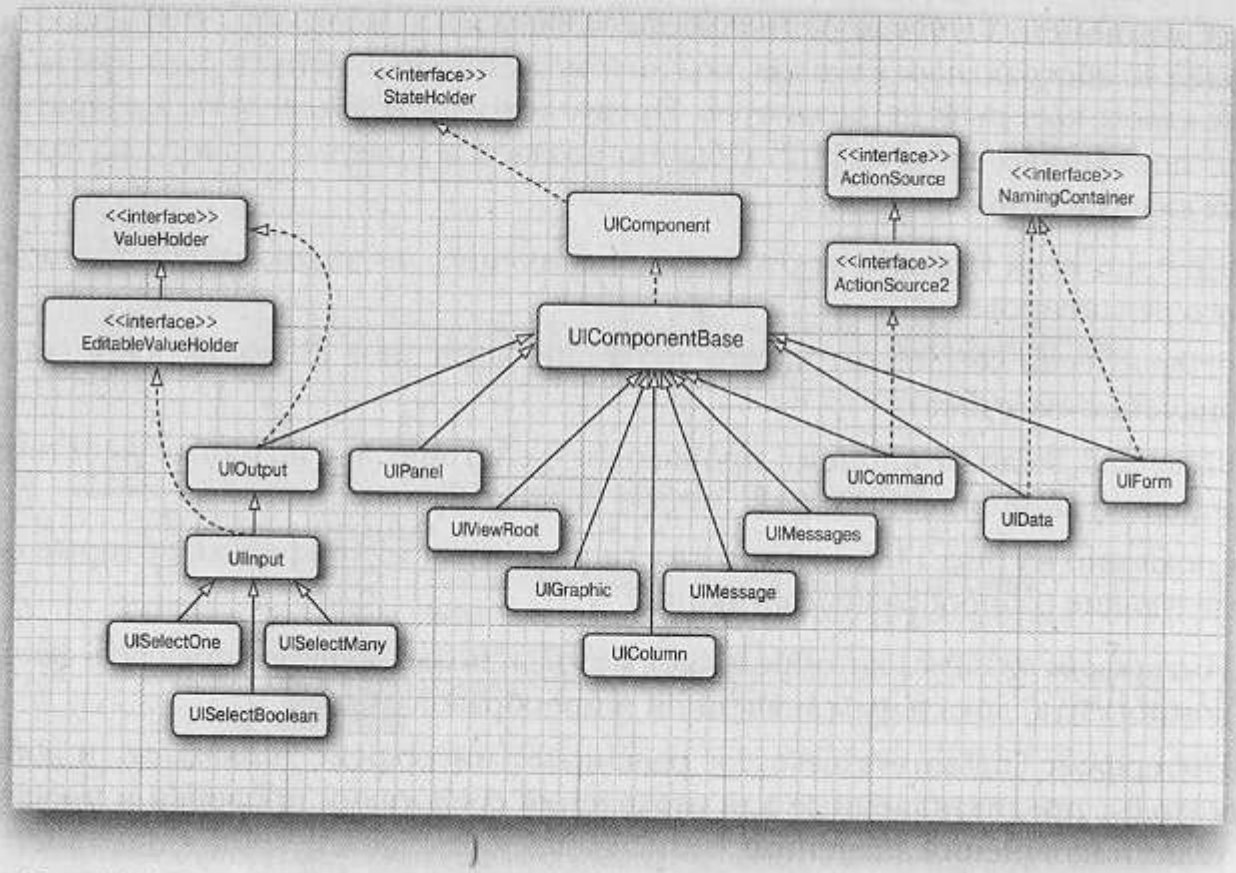


Рис. 11.2. Иерархия компонентов JSF (показаны не все классы)

Теперь рассмотрим компонент счетчика более подробно. Счетчик позволяет вводить число в текстовое поле, либо набирая его непосредственно в поле, либо активизируя кнопку декремента или инкремента. На рис. 11.3 показано приложение, в котором используются два счетчика для указания даты истечения срока действия кредитной карточки; один счетчик показывает месяц, другой — год.

На рис. 11.3 обработка всех полей сверху вниз происходит в соответствии с ожидаемым. Пользователь вводит допустимые значения, в результате чего средства навигации открывают определенную страницу JSF, на которой повторяются эти значения.

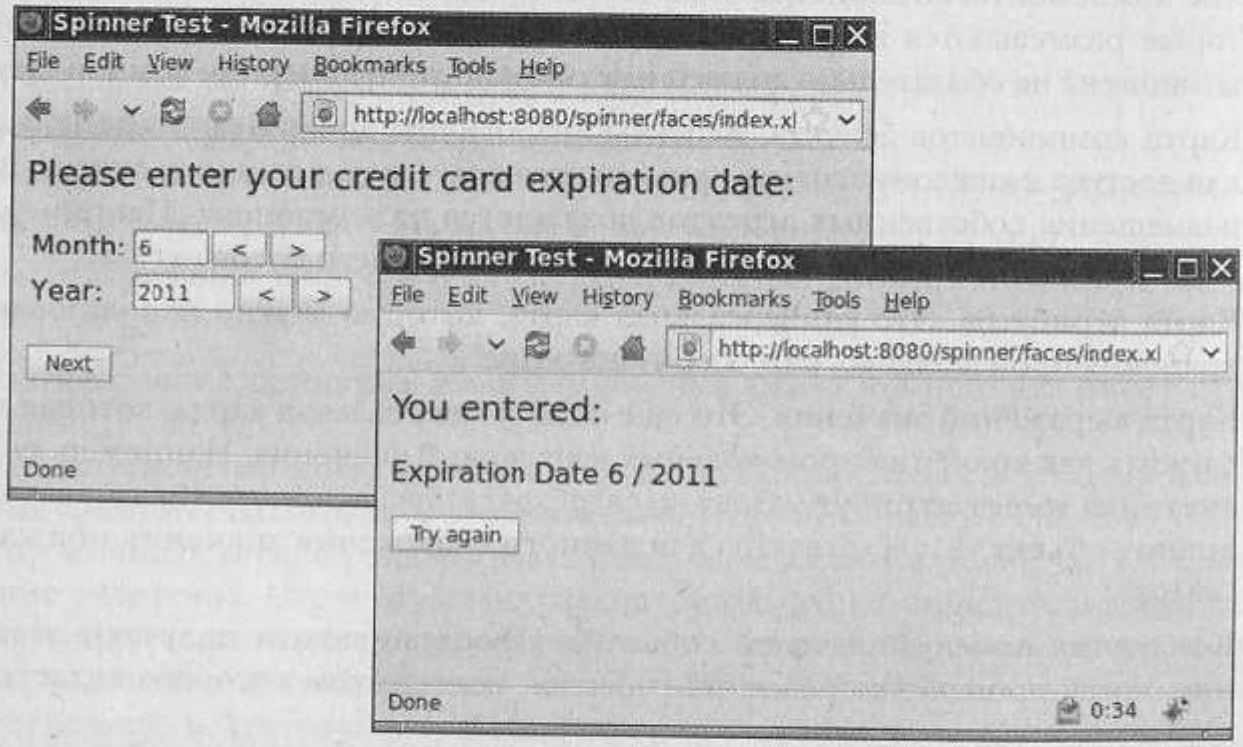


Рис. 11.3. Использование компонента счетчика

Ниже показано, как используется компонент corejsf:spinner.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:corejsf="http://corejsf.com">
...
<corejsf:spinner value="#{cardExpirationDate.month}"
    minimum="1" maximum="12" size="3"/>
...
<corejsf:spinner value="#{cardExpirationDate.year}"
    minimum="1900" maximum="2100" size="5"/>
```

Атрибуты minimum и maximum позволяют назначить диапазон допустимых значений; например, счетчик месяца имеет минимум 1 и максимум 12. Можно также ограничить размер текстового поля счетчика с помощью атрибута size. В следующих разделах будет показано, как реализовать компонент UISpinner, выполняющий функции кодирования и декодирования.

Кодирование: формирование разметки

Компоненты JSF вырабатывают разметку для своих пользовательских интерфейсов. По умолчанию стандартные компоненты JSF формируют код HTML. Компоненты могут сами подготавливать требуемый код или передавать функции кодирования отдельному средству подготовки к отображению. Последний подход является более перспективным, поскольку позволяет подключать по мере необходимости разные средства подготовки к отображению, например, для формирования кода разметки на языке, отличном от HTML. Однако в целях упрощения в первую очередь рассмотрим счетчик, который сам выполняет функции подготовки к отображению.

Для формирования кода разметки в компонентах применяются три метода.

- encodeBegin().
- encodeChildren().
- encodeEnd().

Эти методы вызываются реализацией JSF в конце жизненного цикла в том порядке, в котором они перечислены выше. Реализация JSF вызывает метод encodeChildren, только если компонент возвращает true из своего метода getRendersChildren. (Большинство стандартных компонентов возвращает false, предоставляя реализации JSF самой подготавливать дочерние компоненты к отображению.)

Что касается простых компонентов, таких как рассматриваемый здесь счетчик, не имеющих дочерних компонентов, то реализовывать метод encodeChildren нет необходимости. В данном случае мы можем не задумываться над тем, следует ли вырабатывать код компонента до или после кода его дочерних компонентов, поэтому все кодирование выполняется в методе encodeBegin.

Счетчик вырабатывает код HTML для текстового поля и двух кнопок; этот код HTML выглядит следующим образом:

```
<input type="text" name="..." size="..." value="current value"/>
<input type="submit" name="..." value="<"/>
<input type="submit" name="..." value=">"/>
```

Ниже показано, как код HTML вырабатывается в компоненте UISpinner.

```
public void encodeBegin(FacesContext context) throws IOException {
   ResponseWriter writer = context.getResponseWriter();
   String clientId = getClientId(context);
```

```
    // Кодирование для поля ввода
    writer.startElement("input", this);
    writer.writeAttribute("name", clientId, null);
    Object v = getValue();
    if (v != null) writer.writeAttribute("value", v, "value");
    Object size = getAttributes().get("size");
    if (size != null) writer.writeAttribute("size", size, "size");
    writer.endElement("input");
    // Кодирование для кнопки декремента
    writer.startElement("input", this);
    writer.writeAttribute("type", "submit", null);
    writer.writeAttribute("name", clientId + ".less", null);
    writer.writeAttribute("value", "<", "value");
    writer.endElement("input");
    // Кодирование для кнопки инкремента
    ...
}
```

Класс `ResponseWriter` используется для записи разметки. Он имеет вспомогательные методы для вывода начальных и конечных элементов HTML и для записи атрибутов элементов. Методы `startElement` и `endElement` служат для формирования разграничителей элементов. Эти методы отслеживают дочерние элементы, поэтому программисту не приходится задумываться над тем, что разные теги, допустим, `<input .../>` и `<input ...>...</input>`, по сути, являются равнозначными. Метод `writeAttribute` записывает пары "имя атрибута–значение" с соответствующими экранирующими символами.

Последний параметр методов `startElement` и `writeAttribute` предназначен для поддержки инструментальных средств. Предполагается, что должны быть переданы объект компонента, или имя атрибута, или `null`, если вывод непосредственно не соответствует компоненту или атрибуту. Этот параметр не используется справочной реализацией, но другие реализации могут заменять метод `ResponseWriter` и использовать его.

В ходе работы метода `UISpinner.encodeBegin` приходится решать две задачи. Во-первых, этот метод должен возвращать текущее состояние счетчика. Числовое значение можно легко получить с помощью метода `getValue`, который счетчик наследует от класса `UIInput`. Для получения данных о размере применяется карта атрибутов компонента, для чего используется метод `getAttributes`.

Во-вторых, метод кодирования должен предусматривать присваивание имен для элементов HTML, которые счетчик вырабатывает в коде. Для этого вызывается метод `getClientId` в целях получения клиентского идентификатора компонента, который состоит из идентификатора включающей формы и идентификатора такого компонента, как `_id1:monthSpinner`.

Этот идентификатор создается реализацией JSF. Имена кнопок инкремента и декремента начинаются с клиентского идентификатора и оканчиваются суффиксами `.more` и `.less` соответственно. Ниже приведен законченный пример кода HTML, сформированного счетчиком.

```
<input type="text" name="_id1:monthSpinner" value="1" size="3"/>
<input type="submit" name="_id1:monthSpinner.less" value="<"/>
<input type="submit" name="_id1:monthSpinner.more" value=">"/>
```

В следующем разделе будет описано, как эти имена используются методом декодирования счетчика.

`javax.faces.component.UIComponent`

- `void encodeBegin(FacesContext context) throws IOException`

Этот метод вызывается в фазе подготовки ответа к отображению жизненного цикла JSF, когда тип средства подготовки к отображению компонента обозначен как `null`.

- `String getClientId(FacesContext context)`

  Возвращает клиентский идентификатор для данного компонента. Платформа JSF создает клиентский идентификатор из идентификатора включающей формы (или, вообще говоря, включающего контейнера именования) и идентификатора этого компонента.

- `Map<String, Object> getAttributes()`

  Возвращает изменяемую карту атрибутов и свойств компонента. Этот метод используется для просмотра, добавления, обновления или удаления атрибутов из компонента. Можно также использовать эту карту для просмотра или обновления свойств. Методы `get` и `put` карты применяются для проверки того, соответствует ли ключ свойству компонента. В случае положительного ответа на этот вопрос вызывается метод получения свойства или метод задания свойства.

  Начиная с версии JSF 1.2 карта возвращает также атрибуты, которые определены выражениями значения. Если метод `get` вызывается с именем, не относящимся к свойству или атрибуту, а являющимся ключом в карте выражений значения компонента, то возвращается значение ассоциированного выражения.

На заметку! Счетчик — простой компонент без дочерних компонентов, поэтому задача выработки его кода является довольно несложной. В качестве более сложного примера рассмотрим, как вырабатывается код разметки средством подготовки к отображению области окна с вкладками. Это средство подготовки к отображению показано в листинге 11.11 на стр. 400.

На заметку! Реализация JSF вызывает метод `encodeChildren` компонента, если компонент возвращает `true` из метода `getRendersChildren`. Любопытно отметить, что не имеет значения, содержит ли компонент фактически дочерние компоненты; при условии, что метод `getRendersChildren` компонента возвращает `true`, реализация JSF вызывает метод `encodeChildren`, даже если компонент не имеет дочерних компонентов.

`javax.faces.context.FacesContext`

- `ResponseWriter getResponseWriter()`

  Возвращает ссылку на средство записи ответа. По желанию разработчик может включить в реализацию JSF собственное средство записи ответа. По умолчанию в реализации JSF используется средство записи ответа, которое может записывать теги HTML.

`javax.faces.context.ResponseWriter`

- `void startElement(String elementName, UIComponent component)`

  Записывает начальный тег для указанного элемента. Параметр `component` позволяет инструментальным средствам связывать компонент и его разметку. В настоящее время в справочной реализации JSF этот атрибут игнорируется.

- `void endElement(String elementName)`

  Записывает конечный тег для указанного элемента.

- `void writeAttribute(String attributeName, String attributeValue, String componentProperty)`

  Записывает атрибут и его значение. Этот метод может вызываться только между вызовами `startElement()` и `endElement()`. Здесь `componentProperty` — имя свойства компонента, которое соответствует атрибуту. Этот параметр предназначен для инструментальных средств и не используется справочной реализацией JSF.

Декодирование: обработка значений запроса

Чтобы лучше понять, как происходит процесс декодирования, необходимо рассмотреть весь ход работы веб-приложения. Сервер отправляет форму HTML в браузер. После того как пользователь отправляет эту форму, браузер передает обратно запрос POST, который состоит из пар "имя–значение". Этот запрос POST представляет собой единственные данные, которые сервер может использовать для интерпретации действий пользователя в браузере.

Если пользователь щелкает на кнопке инкремента или декремента, то за этим следует запрос POST, который включает имена и значения всех текстовых полей, но содержит имя и значение только нажатой кнопки. Например, если пользователь щелкнул на кнопке инкремента счетчика месяцев в приложении, показанном на рис. 11.1 на стр. 366, серверу от браузера передаются следующие параметры запроса:

Имя	Значение
_id1:monthSpinner	1
_id1:yearSpinner	2010
_id1:monthSpinner.more	>

В процессе декодирования запроса HTTP счетчик ищет имена параметров запроса, которые соответствуют его клиентскому идентификатору, и обрабатывает связанные с ними значения. Метод декодирования счетчика приведен ниже.

```
public void decode(FacesContext context) {
   Map requestMap = context.getExternalContext().getRequestParameterMap();
   String clientId = getClientId(context);
    int increment;
    if (requestMap.containsKey(clientId + MORE)) increment = 1;
    else if (requestMap.containsKey(clientId + LESS)) increment = -1;
    else increment = 0;
    try {
       int submittedValue
          = Integer.parseInt((String) requestMap.get(clientId));
       int newValue = getIncrementedValue(submittedValue, increment);
       setSubmittedValue("" + newValue);
    }
    catch (NumberFormatException ex) {
       // Даже если обработку некачественного ввода обеспечивает преобразователь,
       // мы все равно обязаны задать переданное значение, поскольку иначе в распоряжении
       // преобразователя не будет никаких входных данных для обработки
       setSubmittedValue((String) requestMap.get(clientId));
    }
}
```

В методе декодирования рассматриваются параметры запроса для определения того, какой из кнопок счетчика (если таковая вообще имеется) был активизирован запрос. Если существует параметр запроса с именем clientId.less, где clientId — клиентский идентификатор счетчика, подлежащего декодированию, то известно, что была активизирована кнопка декремента. Если метод декодирования находит параметр запроса с именем clientId.more, это говорит о том, что была активизирована кнопка инкремента.

Если не существует ни тот ни иной параметр, то можно сделать вывод, что запрос не был инициирован счетчиком, поэтому инкременту присваивается нулевое значение. Но само значение счетчика все еще должно быть обновлено, поскольку пользо-

ватель мог ввести некоторое значение в текстовом поле и щелкнуть на кнопке Next (Далее).

Указанное соглашение об именах остается применимым, даже если на странице имеется несколько счетчиков, поскольку каждый счетчик кодируется с клиентским идентификатором компонента счетчика, в отношении которого можно гарантировать его уникальность. Если на одной и той же странице применяется несколько счетчиков, то каждый компонент счетчика декодирует свой собственный запрос.

После того как метод декодирования определяет, что была нажата одна из кнопок счетчика, этот метод увеличивает значение счетчика на 1 или –1, в зависимости от того, какую кнопку активизировал пользователь. Это увеличенное значение вычисляется с помощью закрытого метода getIncrementedValue:

```
private int getIncrementedValue(int submittedValue, int increment) {
   Integer minimum = toInteger(getAttributes().get("minimum"));
   Integer maximum = toInteger(getAttributes().get("maximum"));
   int newValue = submittedValue + increment;
   if ((minimum == null || newValue >= minimum.intValue()) &&
      (maximum == null || newValue <= maximum.intValue()))
      return newValue;
   else
      return submittedValue;
}
```

Метод getIncrementedValue проверяет значение, введенное пользователем в счетчике, по атрибутам minimum и maximum счетчика.

В данном случае используется вспомогательный метод toInteger, который преобразовывает значение атрибута в целое число. Следует учитывать, что значением атрибута может быть произвольное значение типа Object. Атрибут может быть также задан как строка:

```
minimum="1"
```

В этом случае он является объектом типа String. Еще один вариант состоит в том, что атрибут может быть результатом выражения значения:

```
minimum="#{someBean.someProperty}"
```

Если значение свойства имеет тип int или Integer, то атрибут имеет тип Integer. Метод toInteger применим в следующих случаях:

```
private static Integer toInteger(Object value) {
   if (value == null) return null;
   if (value instanceof Number) return ((Number) value).intValue();
   if (value instanceof String) return Integer.parseInt((String) value);
   throw new IllegalArgumentException("Cannot convert " + value);
}
```

После возврата им увеличенного значения метод декодирования вызывает метод setSubmittedValue компонента счетчика. Этот метод сохраняет отправленное значение в компоненте.

Следует учитывать, что отправленное значение должно быть задано как строка. В компоненте счетчика используется стандартный преобразователь целых чисел платформы JSF для преобразования строк в объекты Integer, и наоборот. Конструктор UISpinner просто вызывает метод setConverter, как в следующем примере:

```
public class UISpinner extends UIInput {
   public UISpinner() {
      setConverter(new IntegerConverter()); // Преобразование переданного значения.
      setRendererType(null);       // Этот компонент подготавливает к отображению сам себя
```

```
   }
   ...
}
```

Метод декодирования счетчика перехватывает недопустимые входные данные в выражении `catch` исключения `NumberFormatException`. Вместо формирования сообщения об ошибке он задает отправленное значение компонента в пользовательском поле ввода. На последующих этапах жизненного цикла JSF стандартный преобразователь целых чисел будет предпринимать попытки преобразовать это значение и сформирует соответствующее сообщение об ошибке, если поле ввода содержит неправильные данные.

В листинге 11.1 приведен полный код для класса `UISpinner`.

Листинг 11.1. Файл spinner/src/java/com/corejsf/UISpinner.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Map;
5.
6. import javax.faces.component.FacesComponent;
7. import javax.faces.component.UIInput;
8. import javax.faces.context.FacesContext;
9. import javax.faces.context.ResponseWriter;
10. import javax.faces.convert.IntegerConverter;
11.
12. @FacesComponent("com.corejsf.Spinner")
13. public class UISpinner extends UIInput {
14.    private static final String MORE = ".more";
15.    private static final String LESS = ".less";
16.
17.    public UISpinner() {
18.       setConverter(new IntegerConverter()); // Преобразование переданного значения.
19.       setRendererType(null); // Этот компонент подготавливает к отображению сам себя
20.    }
21.
22.    public void encodeBegin(FacesContext context) throws IOException {
23.       ResponseWriter writer = context.getResponseWriter();
24.       String clientId = getClientId(context);
25.
26.       encodeInputField(writer, clientId);
27.       encodeDecrementButton(writer, clientId);
28.       encodeIncrementButton(writer, clientId);
29.    }
30.
31.    public void decode(FacesContext context) {
32.       Map<String, String> requestMap
33.          = context.getExternalContext().getRequestParameterMap();
34.       String clientId = getClientId(context);
35.
36.       int increment;
37.       if (requestMap.containsKey(clientId + MORE)) increment = 1;
38.       else if(requestMap.containsKey(clientId + LESS)) increment = -1;
39.       else increment = 0;
40.
41.       try {
42.          int submittedValue
43.             = Integer.parseInt((String) requestMap.get(clientId));
44.
45.          int newValue = getIncrementedValue(submittedValue, increment);
```

```
            setSubmittedValue("" + newValue);
        }
        catch(NumberFormatException ex) {
         // Даже если обработку некачественного ввода обеспечивает преобразователь,
         // мы все равно обязаны задать переданное значение, поскольку иначе в
         // распоряжении преобразователя не будет никаких входных данных для обработки
         setSubmittedValue((String) requestMap.get(clientId));
        }
    }

    private void encodeInputField(ResponseWriter writer, String clientId)
          throws IOException {
       writer.startElement("input", this);
       writer.writeAttribute("name", clientId, null);

        Object v = getValue();
        if (v != null) writer.writeAttribute("value", v, "value");

        Object size = getAttributes().get("size");
        if (size != null) writer.writeAttribute("size", size, "size");

        writer.endElement("input");
    }

    private void encodeDecrementButton(ResponseWriter writer, String clientId)
          throws IOException {
       writer.startElement("input", this);
       writer.writeAttribute("type", "submit", null);
       writer.writeAttribute("name", clientId + LESS, null);
       writer.writeAttribute("value", "<", "value");
       writer.endElement("input");
    }

    private void encodeIncrementButton(ResponseWriter writer, String clientId)
          throws IOException {
       writer.startElement("input", this);
       writer.writeAttribute("type", "submit", null);
       writer.writeAttribute("name", clientId + MORE, null);
       writer.writeAttribute("value", ">", "value");
       writer.endElement("input");
    }

    private int getIncrementedValue(int submittedValue, int increment) {
       Integer minimum = toInteger(getAttributes().get("minimum"));
       Integer maximum = toInteger(getAttributes().get("maximum"));
       int newValue = submittedValue + increment;

         if ((minimum == null || newValue >= minimum.intValue()) &&
            (maximum == null || newValue <= maximum.intValue()))
            return newValue;
         else
            return submittedValue;
    }

    private static Integer toInteger(Object value) {
       if (value == null) return null;
       if (value instanceof Number) return ((Number) value).intValue();
       if (value instanceof String) return Integer.parseInt((String) value);
       throw new IllegalArgumentException("Cannot convert " + value);
    }
}
```

javax.faces.component.UIComponent

- `void decode(FacesContext context)`

Этот метод вызывается реализацией JSF в начале жизненного цикла JSF, только если тип средства подготовки к отображению компонента обозначен как `null`, а это означает, что компонент сам подготавливает себя к отображению.

Метод декодирования декодирует параметры запроса. Как правило, компоненты передают значения параметров запроса свойствам или атрибутам компонентов. Компоненты, которые активизируют события действий, выполняют постановку их в очередь в этом методе.

javax.faces.context.FacesContext

- `ExternalContext getExternalContext()`

Возвращает ссылку на прокси-службу обработки контекста. Как правило, действительным контекстом является контекст сервлета или портлета. Если вместо непосредственного использования этого действительного контекста используется внешний контекст, то разрабатываемые приложения могут работать с сервлетами и портлетами.

javax.faces.context.ExternalContext

- `Map getRequestParameterMap()`

Возвращает карту параметров запроса. Пользовательские компоненты, как правило, вызывают этот метод в методе `decode()` для определения того, не являются ли они теми компонентами, которые активизировали запрос.

javax.faces.component.EditableValueHolder

- `void setSubmittedValue(Object submittedValue)`

Задает отправленное значение компонента; компоненты ввода имеют редактируемые значения, поэтому класс `UIInput` реализует интерфейс `EditableValueHolder`. Отправленным значением является значение, введенное пользователем, по всей видимости, на веб-странице. Применительно к приложениям на основе HTML это значение всегда представляет собой строку, но данный метод принимает также ссылку на значение типа `Object`, что позволяет использовать его с другими технологиями отображения.

javax.faces.component.ValueHolder

- `void setConverter(Converter converter)`

И компоненты ввода, и компоненты вывода имеют значения, поэтому те и другие реализуют интерфейс `ValueHolder`. Значения должны быть преобразованы, поэтому интерфейс `ValueHolder` определяет некоторый метод для задания преобразователя. Пользовательские компоненты используют этот метод, чтобы связать себя со стандартными или пользовательскими преобразователями.

Файл описания библиотеки тегов JSF 1.2

Кроме реализации пользовательского класса компонента, необходимо также предоставить файл описания, который определяет, как теги пользовательского компонента могут использоваться на странице JSF.

Разработчик, предоставляя пользовательский компонент, должен подготовить файл описания библиотеки тегов, который определяет следующее.

На заметку! Версия JSF 1.0 была основана на технологии JSP, поэтому разработчикам пользовательских компонентов приходилось выполнять огромный объем трудоемкой и напряженной работы по обработке тегов. Это направление разработки пользовательских компонентов в версии JSF 2.0 было существенно упрощено.

- Пространство имен (такое как `http://coresf.com`).
- Для каждого тега — имя (такое как `spinner`) и тип компонента.

Например:

```
<facelet-taglib ...>
   <namespace>http://corejsf.com</namespace>
   <tag>
      <tag-name>spinner</tag-name>
      <component>
         <component-type>com.corejsf.Spinner</component-type>
      </component>
   </tag>
</facelet-taglib>
```

Файлы описания библиотеки тегов уже рассматривались в главе 5. Напомним, что имя файла должно оканчиваться суффиксом `taglib.xml`, например `corejsf.taglib.xml`.

Тип компонента представляет собой идентификатор для класса компонента, который должен быть отображен на действительный класс. Можно задать этот идентификатор с помощью аннотации класса компонента:

```
@FacesComponent("com.corejsf.Spinner")
public class UISpinner extends UIInput
```

Тип компонента может рассматриваться как аналог идентификатора преобразователя или средства проверки, который был описан в главе 7.

Файл описания для компонента счетчика показан в листинге 11.2.

На заметку! Имена и типы атрибутов тегов можно задавать в файле описания `taglib`, но эти сведения применяются исключительно в целях документирования. Они не используются реализацией JSF.

Можно задать местоположение этого файла в файле `web.xml`:

```
<context-param>
  <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
  <param-value>/WEB-INF/corejsf.taglib.xml</param-value>
</context-param>
```

В рассматриваемом простом примере используется именно этот подход.

Но если необходимо упаковать компонент счетчика, чтобы можно было его использовать повторно во многих проектах, то следует предусмотреть файл JAR, который может быть добавлен к каталогу `WEB-INF/lib` любого веб-приложения.

Желательно сделать этот файл JAR содержащим все необходимое в себе, чтобы пользователям не приходилось заботиться о редактировании файлов описания библиотеки тегов или файлов конфигурации. Выполните следующие действия.

1. Разместите файл описания библиотеки тегов в каталоге `META-INF`.
2. Если требуется файл `faces-config.xml`, также разместите его в каталоге `META-INF`.
3. Разместите все необходимые ресурсы (такие как изображения, сценарии или файлы таблиц CSS) в каталоге `META-INF/resources`.

4. Предотвратите конфликты имен, применив соответствующий префикс для таких глобальных имен, как имена компонентов, ключи сообщений или имена регистраторов, используемых конкретной реализацией.

Выше были описаны все части, необходимые для приложения проверки счетчика, которое показано на рис. 11.1. Структура каталогов приведена на рис. 11.4. В листингах 11.3 и 11.4 показаны страницы JSF, а в листинге 11.5 приведен класс управляемого бина.

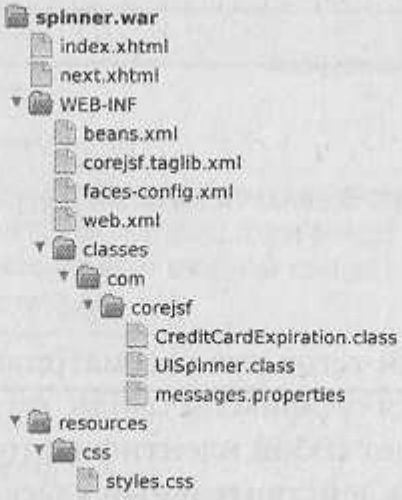


Рис. 11.4. Структура каталогов для примера счетчика

Листинг 11.2. Файл spinner/web/WEB-INF/corejsf.taglib.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<facelet-taglib version="2.0"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-facelettaglibary_2_0.xsd">
   <namespace>http://corejsf.com</namespace>
   <tag>
      <tag-name>spinner</tag-name>
      <component>
         <component-type>com.corejsf.Spinner</component-type>
      </component>
   </tag>
</facelet-taglib>
```

Листинг 11.3. Файл spinner/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:corejsf="http://corejsf.com">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.windowTitle}</title>
   </h:head>

   <h:body>
```

```
13.       <h:form id="spinnerForm">
14.          <h:outputText value="#{msgs.creditCardExpirationPrompt}"
15.                        styleClass="pageTitle"/>
16.          <p/>
17.          <h:panelGrid columns="3">
18.             #{msgs.monthPrompt}
19.             <corejsf:spinner value="#{cardExpirationDate.month}"
20.                id="monthSpinner" minimum="1" maximum="12" size="3"/>
21.             <h:message for="monthSpinner"/>
22.             #{msgs.yearPrompt}
23.             <corejsf:spinner value="#{cardExpirationDate.year}"
24.                id="yearSpinner" minimum="1900" maximum="2100" size="5"/>
25.             <h:message for="yearSpinner"/>
26.          </h:panelGrid>
27.          <p/>
28.          <h:commandButton value="#{msgs.nextButtonPrompt}" action="next"/>
29.       </h:form>
30.    </h:body>
31. </html>
```

Листинг 11.4. Файл spinner/web/next.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:h="http://java.sun.com/jsf/html">
6.    <h:head>
7.       <h:outputStylesheet library="css" name="styles.css"/>
8.       <title>#{msgs.windowTitle}</title>
9.    </h:head>
10.   <h:body>
11.      <h:form>
12.         <h:outputText value="#{msgs.youEnteredPrompt}" styleClass="pageTitle"/>
13.         <p>#{msgs.expirationDatePrompt} #{cardExpirationDate.month} /
14.         #{cardExpirationDate.year}</p>
15.         <p><h:commandButton value="Try again" action="index"/></p>
16.      </h:form>
17.   </h:body>
18. </html>
```

Листинг 11.5. Файл spinner/src/java/com/corejsf/CreditCardExpiration.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. import javax.inject.Named;
6.    // или import javax.faces.bean.ManagedBean;
7. import javax.enterprise.context.SessionScoped;
8.    // или import javax.faces.bean.SessionScoped;
9.
10. @SessionScoped
11. @Named("cardExpirationDate") // или @ManagedBean(name="cardExpirationDate")
12. public class CreditCardExpiration implements Serializable {
13.    private int month = 1;
14.    private int year = 2010;
15.
16.    public int getMonth() { return month; }
17.    public void setMonth(int newValue) { month = newValue; }
```

```
18.
19.    public int getYear() { return year; }
20.    public void setYear(int newValue) { year = newValue; }
21. }
```

Использование внешнего средства подготовки к отображению

В приведенном выше примере за собственную подготовку к отображению отвечал сам класс `UISpinner`. Но классы `UI` могут передавать функции подготовки к отображению отдельному классу. При первоначальном создании платформы JSF предполагалось, что она будет иметь средства подготовки к отображению, которые вырабатывают код разметки, отличный от HTML, или декодируют значения полей ввода, отличные от HTTP. Но этот общий подход так и не был задействован по-настоящему, поэтому мы не будем на нем останавливаться. Однако все еще может оказаться удобным разделение классов компонентов и средств подготовки к отображению, чтобы можно было использовать повторно те и другие по отдельности.

В полной аналогии с тем, что каждый компонент имеет идентификатор, вызываемый типом компонента, каждое средство подготовки к отображению имеет идентификатор, вызываемый типом средства подготовки к отображению.

Имена стандартных тегов HTML выбраны так, чтобы с их помощью можно было указать тип компонента и тип средства подготовки к отображению. Например, тег `h:selectOneMenu` имеет тип компонента `javax.faces.SelectOne`, а средство подготовки к отображению — тип `javax.faces.Menu`. Аналогичным образом, тег `h:selectManyMenu` имеет тип компонента `javax.faces.SelectMany`, а тип его средства подготовки к отображению является таким же, `javax.faces.Menu`.

К сожалению, эту схему именования нельзя назвать слишком удачной. Типом средства подготовки к отображению для тегов `h:inputText` и `h:outputText` является `javax.faces.Text`. Но возможность подготовки к отображению компонентов ввода и вывода таким же образом отсутствует. Чтобы подготовить к отображению компонент `h:inputText`, необходимо записать текстовое поле ввода в коде HTML. А для подготовки к отображению тега `h:outputText` достаточно записать текст и, возможно, тег `span`. Эти средства подготовки к отображению не имеют ничего общего!

Поэтому вместо идентификации средств подготовки к отображению по отдельным компонентам для определения средств подготовки к отображению используются семейство компонентов и тип средства подготовки к отображению. Для всех стандартных компонентов JSF семейство компонентов идентично типу компонента. (Само проведение различия между *типом* и *семейством* кажется не совсем оправданным; это — лишь еще один пример ввода дополнительного термина "на всякий случай", что так часто встречается в спецификации JSF.)

Тип средства подготовки к отображению задается в файле описания библиотеки тегов:

```
<tag>
   <tag-name>spinner</tag-name>
   <component>
      <component-type>com.corejsf.Spinner</component-type>
      <renderer-type>com.corejsf.Spinner</renderer-type>
   </component>
</tag>
```

Для определения семейства компонентов и типа средств подготовки к отображению для класса средств подготовки к отображению используется аннотация `@FacesRenderer`:

```
@FacesRenderer(componentFamily="javax.faces.Input",
   rendererType="com.corejsf.Spinner")
public class SpinnerRenderer extends Renderer {
```

На заметку! Идентификаторы компонентов и средств подготовки к отображению имеют отдельные пространства имен. Вполне допустимо использовать одну и ту же строку в качестве идентификатора компонента и идентификатора средства подготовки к отображению.

Рекомендуется также задавать тип средства подготовки к отображению в конструкторе компонента:

```
public UISpinner() {
   ...
   setRendererType("com.corejsf.Spinner"); // Этот компонент имеет средство
                                           // подготовки к отображению
}
```

В таком случае тип средства подготовки к отображению устанавливается правильно, если компонент вводится в действие программным путем без использования тегов.

Конечный шаг состоит в реализации самого средства подготовки к отображению. Класс `Renderers` расширяет класс `javax.faces.render.Renderer`. Этот класс имеет семь методов, с четырьмя из которых мы уже сталкивались.

- `void encodeBegin(FacesContext context, UIComponent component)`
- `void encodeChildren(FacesContext context, UIComponent component)`
- `void encodeEnd(FacesContext context, UIComponent component)`
- `void decode(FacesContext context, UIComponent component)`

Перечисленные выше методы средства подготовки к отображению почти идентичны своим аналогам для компонентов, не считая того, что методы средства подготовки к отображению принимают дополнительный параметр: ссылку на компонент, подготавливаемый к отображению.

Таким образом, методы средства подготовки к отображению получают ссылку на компонент в качестве универсального параметра `UIComponent`, поэтому необходимо применять приведение типов, чтобы использовать методы, характерные для конкретного компонента. Вместо приведения к определенному классу, такому как `UISpinner`, следует приводить тип к одному из типов интерфейсов `ValueHolder`, `EditableValueHolder`, `ActionSource` или `ActionSource2`. Благодаря этому упрощается повторное использование конкретного кода в других средствах подготовки к отображению. Например, в средстве подготовки к отображению `UISpinner` применяется интерфейс `EditableValueHolder`.

Ниже перечислены остальные методы средства подготовки к отображению.

- `boolean getRendersChildren()`
- `String convertClientId(FacesContext context, String clientId)`
- `Object getConvertedValue(FacesContext context, UIComponent component, Object submittedValue)`

Метод `getRendersChildren` определяет, является ли средство подготовки к отображению ответственным за подготовку к отображению дочерних компонентов его ком-

понента. Если этот метод возвращает true, то вызывается метод encodeChildren средства подготовки к отображению; если же он возвращает false (такое правило поведения предусмотрено по умолчанию), то реализация JSF не вызывает этот метод и подготовка кода для дочерних компонентов осуществляется отдельно.

Метод convertClientId преобразовывает строку идентификатора (такую как _id1:monthSpinner), чтобы ее можно было использовать на клиенте. Дело в том, что некоторые клиенты могут налагать ограничения на идентификаторы, например, не допускать применения специальных символов. В реализации по умолчанию происходит возврат исходной строки идентификатора, и это вполне допустимо для средств подготовки к отображению HTML.

Метод getConvertedValue преобразовывает отправленное значение компонента из строки в объект. В реализации по умолчанию в классе Renderer просто происходит возврат отправленного значения. К сожалению, это становится проблемой для компонентов, в которых используются преобразователи. После того как счетчик делегирует свои функции средству подготовки к отображению, он теряет возможность полагаться на механизм UIInput в целях проведения преобразований. В API-интерфейсе JSF доступ к коду преобразования не предоставляется, поэтому данный код необходимо каждый раз копировать в любое средство подготовки к отображению, которое использует преобразователь. Мы разместили этот код в статическом методе getConvertedValue класса com.corejsf.util.Renderers (листинг 11.8 на стр. 390) и вызываем его в методе getConvertedValue средства подготовки к отображению.

На заметку! В классе UIInput код преобразования находится в защищенном методе UIInput.getConvertedValue, который выглядит так же, как и в справочной реализация JSF 2.0:

```
// Это - код из объявления класса javax.faces.component.UIInput:
    protected void getConvertedValue(FacesContext context, Object
        newSubmittedValue) throws ConverterException {
      Object newValue = newSubmittedValue;
      if (renderer != null) {
        newValue = renderer.getConvertedValue(context, this, newSubmittedValue);
      } else if (newSubmittedValue instanceof String) {
        Converter converter = getConverterWithType(context); // приватный метод
        if (converter != null) {
          newValue = converter.getAsObject(context, this,
            (String) newSubmittedValue);
        }
      }
      return newValue;
    }
```

Закрытый метод getConverterWithType ищет соответствующий преобразователь для значения компонента.

Таким образом, код преобразования UIInput спрятан в защищенных и закрытых методах, поэтому он не может служить для повторного использования в средстве подготовки к отображению. В пользовательских компонентах, в которых применяются преобразователи, этот код приходится дублировать; см., например, реализацию com.sun.faces.renderkit.html_basic.HtmlBasicInputRenderer в справочной реализации. Применяемый нами класс com.corejsf.util.Renderers предоставляет код, который читатель может использовать в своих собственных классах.

Сравнение листингов 11.6 и 11.7 с листингом 11.1 показывает, что мы переместили большую часть кода из исходного класса компонента в новый класс средства подготовки к отображению.

Обработка атрибутов тегов JSF 2.0

При использовании тега компонента на странице JSF необходимо задать атрибуты тега в целях определения свойств тега. Например, тег spinner имеет атрибуты для задания минимального, максимального и текущего значений счетчика. В нашем первом примере мы просто получали эти значения тега из карты атрибутов компонента.

В данном разделе обработка атрибутов тега рассматривается более подробно. Обработчик тегов обрабатывает атрибуты тега и их значения. В обработчике тегов по умолчанию предусмотрены правила для самых распространенных атрибутов и предпринимаются допустимые действия по отношению к атрибутам, которые неизвестны обработчику. Если такое поведение по умолчанию не подходит для конкретного компонента, можно дополнительно предусмотреть пользовательский обработчик.

Задачу обработки атрибутов не следует считать тривиальной; чтобы убедиться в этом, рассмотрим тег счетчика из приведенного выше первого примера:

```
<corejsf:spinner value="#{cardExpirationDate.month}" minimum="1" maximum="12" />
```

Заслуживает внимания то, что компонент должен сохранять выражение значения #{cardExpirationDate.month} без его вычисления. В конце концов, каждый раз, когда реализация JSF декодирует и преобразовывает значение, в ней должен выполняться метод задания свойства setMonth бина cardExpirationDate. Обработчик тегов преобразовывает строку "#{cardExpirationDate.month}" в объект ValueExpression и сохраняет его в карте выражений значения компонентов, используя ключ "value". Класс UIInput применяет это выражение значения при обновлении значения модели после успешной проверки правильности.

Это — частный случай, но действия, связанные с его возникновением, заранее предусмотрены в коде обработчика тегов по умолчанию. Если в конкретном компоненте реализуется интерфейс ValueHolder, то обработка его атрибута value происходит правильно. Могут быть также определены частные случаи для следующих атрибутов.

- Атрибут value экземпляров ValueHolder.
- Атрибуты validator и valueChangeListener экземпляров EditableValueHolder.
- Атрибут actionListener экземпляров ActionSource.
- Атрибут action экземпляров ActionSource2.

Однако в этом применяемом по умолчанию обработчике отсутствуют какие-либо средства обработки наших атрибутов minimum и maximum. Прежде всего обработчик определяет, имеет ли компонент методы задания свойств setMinimum или setMaximum. В случае положительного ответа обработчик вызывает эти методы, вычисляет строки выражений значения и, если целевой тип является числовым или булевым, преобразует строковые литералы в целевой тип.

Если же метод задания свойства отсутствует, то обработчик тегов помещает имя и значение атрибута в карту атрибутов компонента. Он вычисляет выражения значения, но не преобразовывает строки.

Ниже описано, что происходит с атрибутами maximum и minimum. Они сохраняются в карте атрибутов компонента со значениями "1" и "12". Следует учитывать, что эти значения являются строковыми, а не числовыми, поскольку обработчику тегов неизвестно, каковым является намеченный тип. Подводя итоги, отметим, что выполняет обработчик тегов по умолчанию.

- Если атрибутом является value, validator, valueChangeListener, action или actionListener, используется специальное правило.
- В противном случае, если имеется метод задания свойства для атрибута, он вызывается, вычисляются выражения значения и строковые значения преобразуются по мере необходимости.
- В противном случае вводится запись в карту атрибутов компонента. Выражения значения вычисляются, но строки не преобразуются.

Теперь рассмотрим средство подготовки к отображению, в котором используются параметры атрибутов тега. В рассматриваемом классе счетчика используется вызов, такой, как

```
component.getAttributes().get("minimum")
```

для выборки значений атрибутов. Но в этом вызове не все очевидно при поверхностном рассмотрении. Карта, возвращенная методом UIComponent.getAttributes, является интеллектуальной: она предоставляет доступ к свойствам компонента, карте атрибутов и карте ссылок на значения. Например, если происходит вызов метода get карты с атрибутом, имеющим имя "value", вызывается метод getValue. Если атрибут имеет имя "minimum", а метод getMinimum отсутствует, то карта атрибутов компонента запрашивается на наличие записи с ключом "minimum".

В завершение этого продолжительного обсуждения отметим, что автор компонента должен правильно выбирать способ обработки атрибутов тегов. Он должен либо предусматривать метод задания свойства, либо использовать карту атрибутов. Подход на основе метода задания свойства имеет небольшое преимущество: он автоматически преобразовывает строки в числа или булевы значения. Недостатком его является то, что в этом случае разработчику приходится заботиться о сохранении состояния. (Подробнее об этом – в разделе "Сохранение и восстановление состояния" на стр. 405.)

Поддержка прослушивателей изменений значений

Если разрабатываемым пользовательским компонентом является компонент ввода, то можно активизировать события изменения значения, представляющие интерес для конкретных прослушивателей. Например, в приложении с календарем может потребоваться обновлять другой компонент каждый раз после изменения значения счетчика месяца. К счастью, задача поддержки прослушивателей изменений значений является несложной. Класс UIInput автоматически формирует события изменения значения после каждого изменения входного значения. Напомним, что предусмотрены два способа закрепления прослушивателей изменений значений. Можно добавить один или несколько прослушивателей с помощью тега f:valueChangeListener:

```
<corejsf:spinner ...>
   <f:valueChangeListener type="com.corejsf.SpinnerListener"/>
   ...
</corejsf:spinner>
```

Еще один вариант состоит в том, что можно воспользоваться атрибутом valueChangeListener:

```
<corejsf:spinner value="#{cardExpirationDate.month}"
   id="monthSpinner" minimum="1" maximum="12" size="3"
   valueChangeListener="#{cardExpirationDate.changeListener}"/>
```

В рассматриваемом примере программы мы демонстрируем применение прослушивателя изменений значений, сохраняя данные о количестве всех изменений значения, которые отображаются в форме (рис. 11.5):

```
public class CreditCardExpiration {
   private int changes = 0;
   // Применение прослушивателя изменений значений
   public void changeListener(ValueChangeEvent e) {
      changes++;
   }
}
```

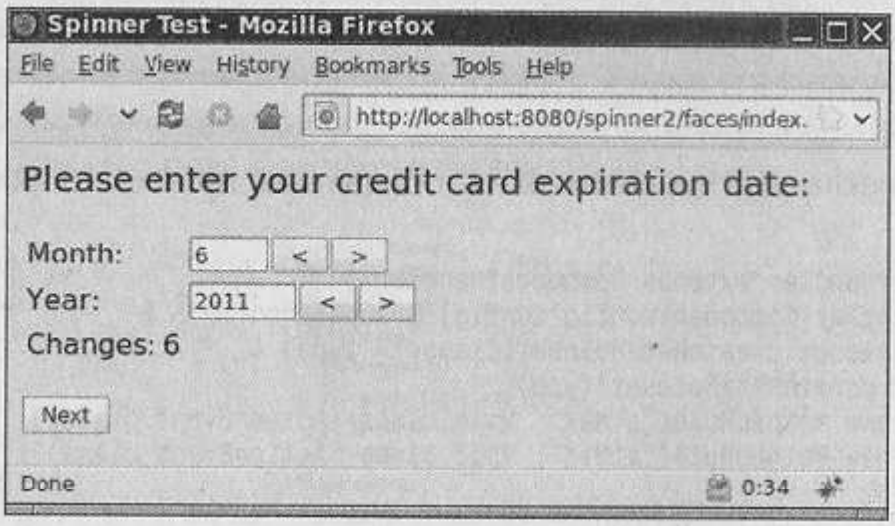


Рис. 11.5. Подсчет количества изменений значения

Поддержка выражений метода

Специальные атрибуты `valueChangeListener`, `validator`, `action` и `actionListener` автоматически поддерживают выражения метода. Если же возникает необходимость поддерживать выражения метода для собственных атрибутов, то следует проделать некоторый объем работы.

Ниже приведен немного надуманный пример. Во второй версии рассматриваемого счетчика поддерживаются атрибуты `atMax` и `atMin`, в качестве которых могут быть заданы выражения метода. Эти методы вызываются при попытке пользователя задать значение счетчика больше максимума или меньше минимума. В этом примере программы в качестве атрибута `atMax` задается метод, который увеличивает значение года и задает месяц равным 1. Таким образом, если пользователь увеличивает значение текущего месяца свыше 12, то дата автоматически переводится на январь следующего года.

Страница JSF содержит ссылку на метод:

```
<corejsf:spinner value="#{cardExpirationDate.month}"
   minimum="1" maximum="12" size="3"
   atMax="#{cardExpirationDate.incrementYear}" .../>
```

где метод `incrementYear` определен примерно так:

```
@Named("cardExpirationDate")
public class CreditCardExpiration {
   ...
   public void incrementYear(ActionEvent event) { year++; month = 1; }
}
```

К сожалению, обработчик тегов по умолчанию не может определить, что `"#{cardExpirationDate.incrementYear}"` представляет собой ссылку на метод. В этом со-

стоит недостаток API-интерфейса языка выражений. Поэтому необходимо внести изменения в обработчик тегов. Ниже показано, как это сделать.

Следует предусмотреть класс обработчика SpinnerHandler и объявить его в применяемом файле описания библиотеки тегов:

```
<facelet-taglib ...>
   <namespace>http://corejsf.com</namespace>
   <tag>
      <tag-name>spinner</tag-name>
      <component>
         <component-type>com.corejsf.Spinner</component-type>
         <renderer-type>com.corejsf.Spinner</renderer-type>
         <handler-class>com.corejsf.SpinnerHandler</handler-class>
      </component>
   </tag>
</facelet-taglib>
```

В классе SpinnerHandler должны быть объявлены правила обработки атрибутов atMin и atMax:

```
public class SpinnerHandler extends ComponentHandler {
   public SpinnerHandler(ComponentConfig config) { super(config); }
    protected MetaRuleset createMetaRuleset(Class<?> type) {
       return super.createMetaRuleset(type)
          .addRule(new MethodRule("atMax", Void.class, ActionEvent.class))
          .addRule(new MethodRule("atMin", Void.class, ActionEvent.class));
    }
}
```

Изложение подробных сведений о методе createMetaRuleset становится довольно сложным, поэтому мы не будем на этом останавливаться. Если необходимо обеспечить поддержку выражений метода в конкретном теге, то можно просто следовать этой модели.

Класс обработчика задает атрибуты atMin и atMax для объектов MethodExpression. В следующем разделе будет показано, как их использовать.

Постановка событий в очередь

Предположим, пользователь установил значение счетчика равным максимальному. В этом случае должно быть выполнено выражение метода atMax, но только после того, как будут отправлены и проверены значения запроса. Выполнять выражение метода непосредственно в методе декодирования бессмысленно, поскольку значения модели будут перезаписаны отправленными значениями.

Чтобы задержать выполнение выражения метода, необходимо поставить в очередь событие, которое реализация JSF выполнит в фазе вызова приложения:

```
FacesEvent event = new ActionEvent(spinner);
event.setPhaseId(PhaseId.INVOKE_APPLICATION);
spinner.queueEvent(event);
```

Если происходит выход за пределы минимального и максимального значений, средство подготовки к отображению счетчика добавляет метод MethodExpressionActionListener, созданный с помощью выражений метода atMax или atMin. После рассылки сообщения о событии для прослушивателя выполняется метод. Прослушиватель удаляется при подготовке компонента к отображению.

Изложение подробных сведений об этом становится довольно сложным, поскольку методы добавления и удаления прослушивателей представляют собой защищенные методы в классе UIComponent, как показано в листингах 11.6 и 11.7.

На заметку! Решение о динамической установке прослушивателей в средстве подготовки к отображению является довольно необычным. При наличии стандартного атрибута `actionListener` прослушиватель устанавливается обработчиком тегов.

Пример приложения

На рис. 11.6 показана структура каталогов второго приложения счетчика, в котором используется отдельное средство подготовки к отображению.

В листингах 11.6 и 11.7 приведен код компонента счетчика и средства подготовки к отображению соответственно.

Мы полагаемся на вспомогательный класс `Renderers` в листинге 11.8, который содержит код вызова преобразователя. (Класс `Renderers` содержит также метод `getSelectedItems`, необходимость в котором появится в ходе дальнейшего изложения материала данной главы; пока можно его не учитывать.)

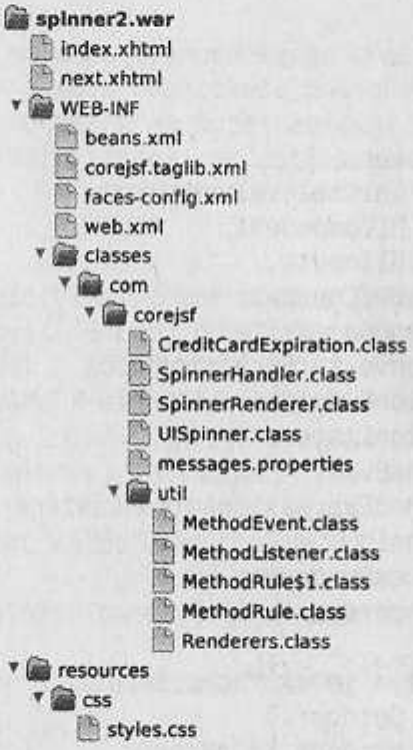


Рис. 11.6. Структура каталогов пересмотренного примера счетчика

Листинг 11.6. Файл `spinner2/src/java/com/corejsf/UISpinner.java`

```
1. package com.corejsf;
2.
3. import javax.faces.component.FacesComponent;
4. import javax.faces.component.UIInput;
5. import javax.faces.convert.IntegerConverter;
6. import javax.faces.event.FacesListener;
7.
8. @FacesComponent("com.corejsf.Spinner")
9. public class UISpinner extends UIInput {
10.    private FacesListener maxMinListener;
11.    public UISpinner() {
12.       setConverter(new IntegerConverter()); // Преобразование переданного значения
13.       setRendererType("com.corejsf.Spinner");
```

```
14.    }
15.
16.    public void addMaxMinListener(FacesListener listener) {
17.       if (listener != null) addFacesListener(listener);
18.       maxMinListener = listener;
19.    }
20.
21.    public void removeMaxMinListener() {
22.       if (maxMinListener != null) {
23.           removeFacesListener(maxMinListener);
24.           maxMinListener = null;
25.       }
26.    }
27. }
```

Листинг 11.7. Файл spinner2/src/java/com/corejsf/SpinnerRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Map;
5.
6. import javax.el.MethodExpression;
7. import javax.faces.component.EditableValueHolder;
8. import javax.faces.component.UIComponent;
9. import javax.faces.component.UIInput;
10. import javax.faces.context.FacesContext;
11. import javax.faces.context.ResponseWriter;
12. import javax.faces.convert.ConverterException;
13. import javax.faces.event.ActionEvent;
14. import javax.faces.event.ActionListener;
15. import javax.faces.event.FacesEvent;
16. import javax.faces.event.MethodExpressionActionListener;
17. import javax.faces.event.PhaseId;
18. import javax.faces.render.FacesRenderer;
19. import javax.faces.render.Renderer;
20.
21. @FacesRenderer(componentFamily="javax.faces.Input",
22.    rendererType="com.corejsf.Spinner")
23. public class SpinnerRenderer extends Renderer {
24.    private static final String MORE = ".more";
25.    private static final String LESS = ".less";
26.
27.    public Object getConvertedValue(FacesContext context, UIComponent component,
28.          Object submittedValue) throws ConverterException {
29.       return com.corejsf.util.Renderers.getConvertedValue(context, component,
30.          submittedValue);
31.    }
32.
33.    public void encodeBegin(FacesContext context, UIComponent spinner)
34.          throws IOException {
35.       ResponseWriter writer = context.getResponseWriter();
36.       String clientId = spinner.getClientId(context);
37.
38.       encodeInputField(spinner, writer, clientId);
39.       encodeDecrementButton(spinner, writer, clientId);
40.       encodeIncrementButton(spinner, writer, clientId);
41.
42.       ((UISpinner) spinner).removeMaxMinListener();
43.    }
44.
```

```
45.    public void decode(FacesContext context, UIComponent component) {
46.       EditableValueHolder spinner = (EditableValueHolder) component;
47.       Map<String, String> requestMap
48.          = context.getExternalContext().getRequestParameterMap();
49.       String clientId = component.getClientId(context);
50.
51.       int increment;
52.       if (requestMap.containsKey(clientId + MORE)) increment = 1;
53.       else if (requestMap.containsKey(clientId + LESS)) increment = -1;
54.       else increment = 0;
55.
56.       try {
57.          int submittedValue
58.             = Integer.parseInt((String) requestMap.get(clientId));
59.
60.          int newValue = getIncrementedValue(component, submittedValue,
61.             increment);
62.          spinner.setSubmittedValue("" + newValue);
63.       }
64.       catch(NumberFormatException ex) {
65.       // Даже если обработку некачественного ввода обеспечивает преобразователь,
66.       // мы все равно обязаны задать переданное значение, поскольку иначе в
67.       // распоряжении преобразователя не будет никаких входных данных для обработки
68.       spinner.setSubmittedValue((String) requestMap.get(clientId));
69.       }
70.    }
71.
72.    private void encodeInputField(UIComponent spinner, ResponseWriter writer,
73.          String clientId) throws IOException {
74.       writer.startElement("input", spinner);
75.       writer.writeAttribute("name", clientId, null);
76.
77.       Object v = ((UIInput) spinner).getValue();
78.       if (v != null)
79.          writer.writeAttribute("value", v, "value");
80.
81.       Object size = spinner.getAttributes().get("size");
82.       if (size != null)
83.          writer.writeAttribute("size", size, "size");
84.
85.       writer.endElement("input");
86.    }
87.
88.    private void encodeDecrementButton(UIComponent spinner,
89.          ResponseWriter writer, String clientId) throws IOException {
90.       writer.startElement("input", spinner);
91.       writer.writeAttribute("type", "submit", null);
92.       writer.writeAttribute("name", clientId + LESS, null);
93.       writer.writeAttribute("value", "<", "value");
94.       writer.endElement("input");
95.    }
96.
97.    private void encodeIncrementButton(UIComponent spinner,
98.          ResponseWriter writer, String clientId) throws IOException {
99.       writer.startElement("input", spinner);
100.      writer.writeAttribute("type", "submit", null);
101.      writer.writeAttribute("name", clientId + MORE, null);
102.      writer.writeAttribute("value", ">", "value");
103.      writer.endElement("input");
104.   }
105.
106.   private int getIncrementedValue(UIComponent spinner, int submittedValue,
```

```
107.          int increment) {
108.       Integer minimum = toInteger(spinner.getAttributes().get("minimum"));
109.       Integer maximum = toInteger(spinner.getAttributes().get("maximum"));
110.       int newValue = submittedValue + increment;
111.
112.        ActionListener listener = null;
113.
114.        MethodExpression minMethod
115.           = (MethodExpression) spinner.getAttributes().get("atMin");
116.        if (minimum != null && newValue < minimum && minMethod != null) {
117.           listener = new MethodExpressionActionListener(minMethod);
118.           FacesEvent event = new ActionEvent(spinner);
119.           event.setPhaseId(PhaseId.INVOKE_APPLICATION);
120.           spinner.queueEvent(event);
121.        }
122.
123.        MethodExpression maxMethod
124.           = (MethodExpression) spinner.getAttributes().get("atMax");
125.        if (maximum != null && newValue > maximum && maxMethod != null) {
126.           listener = new MethodExpressionActionListener(maxMethod);
127.       FacesEvent event = new ActionEvent(spinner);
128.       event.setPhaseId(PhaseId.INVOKE_APPLICATION);
129.       spinner.queueEvent(event);
130.    }
131.
132.    ((UISpinner) spinner).addMaxMinListener(listener);
133.
134.    if ((minimum == null || newValue >= minimum.intValue()) &&
135.       (maximum == null || newValue <= maximum.intValue()))
136.       return newValue;
137.    else
138.       return submittedValue;
139. }
140.
141. private static Integer toInteger(Object value) {
142.    if (value == null) return null;
143.    if (value instanceof Number) return ((Number) value).intValue();
144.    if (value instanceof String) return Integer.parseInt((String) value);
145.    throw new IllegalArgumentException("Cannot convert " + value);
146. }
147.}
```

Листинг 11.8. Файл spinner2/src/java/com/corejsf/util/Renderers.java

```
1. package com.corejsf.util;
2.
3. import java.util.ArrayList;
4. import java.util.Arrays;
5. import java.util.Collection;
6. import java.util.List;
7. import java.util.Map;
8.
9. import javax.el.ValueExpression;
10. import javax.faces.application.Application;
11. import javax.faces.component.UIComponent;
12. import javax.faces.component.UIForm;
13. import javax.faces.component.UISelectItem;
14. import javax.faces.component.UISelectItems;
15. import javax.faces.component.ValueHolder;
16. import javax.faces.context.FacesContext;
17. import javax.faces.convert.Converter;
```

```
import javax.faces.convert.ConverterException;
import javax.faces.model.SelectItem;

public class Renderers {
   public static Object getConvertedValue(FacesContext context,
         UIComponent component, Object submittedValue)
         throws ConverterException {
      if (submittedValue instanceof String) {
         Converter converter = getConverter(context, component);
         if (converter != null) {
            return converter.getAsObject(context, component,
                  (String) submittedValue);
         }
      }
      return submittedValue;
   }

   public static Converter getConverter(FacesContext context,
         UIComponent component) {
      if (!(component instanceof ValueHolder)) return null;
      ValueHolder holder = (ValueHolder) component;

      Converter converter = holder.getConverter();
      if (converter != null)
         return converter;

      ValueExpression expr = component.getValueExpression("value");
      if (expr == null) return null;

      Class<?> targetType = expr.getType(context.getELContext());
      if (targetType == null) return null;
      // В версии 1.0 этой справочной реализации преобразователь не применяется,
      // если целевым типом является String или Object, но это - программная ошибка

      Application app = context.getApplication();
      return app.createConverter(targetType);
   }

   public static String getFormId(FacesContext context, UIComponent component) {
      UIComponent parent = component;
      while (!(parent instanceof UIForm))
         parent = parent.getParent();
      return parent.getClientId(context);
   }

   public static List<SelectItem> getSelectItems(UIComponent component) {
      ArrayList<SelectItem> list = new ArrayList<SelectItem>();
      for (UIComponent child : component.getChildren()) {
         if (child instanceof UISelectItem) {
            Object value = ((UISelectItem) child).getValue();
            if (value == null) {
               UISelectItem item = (UISelectItem) child;
               list.add(new SelectItem(item.getItemValue(),
                     item.getItemLabel(),
                     item.getItemDescription(),
                     item.isItemDisabled()));
            } else if (value instanceof SelectItem) {
               list.add((SelectItem) value);
            }
         } else if (child instanceof UISelectItems) {
            Object value = ((UISelectItems) child).getValue();
            if (value instanceof SelectItem)
```

```
80.             list.add((SelectItem) value);
81.          else if (value instanceof SelectItem[])
82.             list.addAll(Arrays.asList((SelectItem[]) value));
83.          else if (value instanceof Collection<?>) {
84.             @SuppressWarnings("unchecked")
85.             Collection<SelectItem> values = (Collection<SelectItem>) value;
86.             list.addAll(values);
87.          }
88.          // Предупреждение
89.          else if (value instanceof Map<?, ?>) {
90.             for (Map.Entry<?, ?> entry : ((Map<?, ?>) value).entrySet())
91.                list.add(new SelectItem(entry.getKey(),
92.                      "" + entry.getValue()));
93.          }
94.       }
95.    }
96.    return list;
97. }
98. }
```

javax.faces.component.UIComponent

- ValueExpression getValueExpression(String name)

  Возвращает выражение значения, связанное с указанным именем.

javax.faces.component.ValueHolder

- Converter getConverter()

  Возвращает преобразователь, связанный с компонентом.

javax.faces.context.FacesContext

- ELContext getELContext()

  Возвращает "объект контекста языка выражений", который необходим для вычисления выражений.

javax.el.ValueExpression

- Class getType(ELContext context)

  Возвращает тип данного выражения значения.

javax.faces.application.Application

- Converter createConverter(Class targetClass)

  Создает преобразователь, учитывая его целевой класс. Реализации JSF поддерживают карту допустимых типов преобразователей, которые обычно определяются в файле конфигурации faces. Если метод targetClass является ключом в данной карте, то с помощью этого метода создается экземпляр связанного преобразователя (указанный как значение для ключа targetClass) и происходит его возврат.

  Если метод targetClass не находится в карте, то он ищет в карте ключ, который соответствует интерфейсам и суперклассам targetClass, в указанном порядке до тех пор, пока не обнаруживается соответствующий класс. Как только соответствующий класс будет найден, этот метод создает связанный с ним преобразователь и выполняет его возврат. Если не удается найти преобразователь для метода targetClass, его интерфейсов или суперклассов, этот метод возвращает null.

Разработка кода JavaScript

Компонент счетчика выполняет обмен данными с сервером каждый раз, когда выполняется щелчок на одной из его кнопок. Этот обмен данными приводит к обновлению значения счетчика на сервере.

Подобные обмены данными могут стать существенной причиной снижения производительности работы счетчика, поэтому почти во всех обстоятельствах лучше сохранять значение счетчика на клиенте и обновлять значение этого компонента только при отправке формы, в которой находится счетчик. Это можно сделать с помощью кода JavaScript, который выглядит следующим образом:

```
<input type="text" name="_id1:monthSpinner" value="1"/>
<script language="JavaScript">
   document.forms['_id1']['_id1:monthSpinner'].min = 1;
   document.forms['_id1']['_id1:monthSpinner'].max = 12;
</script>
<input type="button" value="<" onclick=
   "com.corejsf.spinner.spin(document.forms['_id1']['_id1:monthSpinner'], -1);"/>
<input type="button" value=">" onclick=
   "com.corejsf.spinner.spin(document.forms['_id1']['_id1:monthSpinner'], 1);"/>
```

Функция `spin` определена в файле `spinner-js/web/resources/javascript/spinner.js` на языке JavaScript, который показан в листинге 11.9.

Чтобы гарантировать то, что этот ресурс JavaScript будет включен на каждую страницу JSF, на которой используется счетчик, можно просто аннотировать средство подготовки к отображению с помощью аннотации `@ResourceDependency`:

```
@FacesRenderer(...)
@ResourceDependency(library="javascript", name="spinner.js")
public class JSSpinnerRenderer extends Renderer
```

При разработке кода JavaScript, который обращается к полям в форме, необходимо задавать поле с помощью примерно такого выражения:

```
document.forms['_id1']['_id1:monthSpinner']
```

Первый индекс массива представляет собой клиентский идентификатор формы, а второй индекс – клиентский идентификатор компонента.

Задача получения идентификатора формы является распространенной, поэтому мы добавили к классу `com.corejsf.util.Renderers` вспомогательный метод, который выполняет эту функцию:

```
public static String getFormId(FacesContext context, UIComponent component) {
   UIComponent parent = component;
   while (!(parent instanceof UIForm)) parent = parent.getParent();
   return parent.getClientId(context);
}
```

В данной книге мы не приводим подробные сведения о программировании на языке JavaScript, но отметим, что имеем весьма значительные опасения по поводу внедрения глобальных функций и переменных JavaScript в заранее не известную страницу.

Вместо написания глобальной функции `spin` мы определяем `spin` как метод объекта `com.corejsf.spinner`. Аналогичный подход используется по отношению к минимальному и максимальному значениям каждого счетчика, для чего к каждому полю ввода добавляются переменные `min` и `max`.

Средство подготовки к отображению счетчика, в котором вырабатывается приведенный выше код JavaScript, показан в листинге 11.10.

Следует отметить, что компонент UISpinner совершенно не затрагивается этим изменением. Было обновлено только средство подготовки к отображению, что может служить демонстрацией широких возможностей использования подключаемых средств подготовки к отображению.

Листинг 11.9. Файл spinner-js/web/resources/javascript/spinner.js

```
if (!com) var com = {};
if (!com.corejsf) com.corejsf = {};
com.corejsf.spinner = {
   spin: function(field, increment) {
      var v = parseInt(field.value) + increment;
      if (isNaN(v)) return;
      if ('min' in field && v < field.min) return;
      if ('max' in field && v > field.max) return;
      field.value = v;
   }
};
```

Листинг 11.10. Файл spinner-js/src/java/com/corejsf/JSSpinnerRenderer.java

```
package com.corejsf;

import java.io.IOException;
import java.text.MessageFormat;
import java.util.Map;

import javax.faces.application.ResourceDependency;
import javax.faces.component.EditableValueHolder;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.convert.ConverterException;
import javax.faces.render.FacesRenderer;
import javax.faces.render.Renderer;

@FacesRenderer(componentFamily="javax.faces.Input",
   rendererType="com.corejsf.Spinner")
@ResourceDependency(library="javascript", name="spinner.js")
public class JSSpinnerRenderer extends Renderer {
   public Object getConvertedValue(FacesContext context, UIComponent component,
         Object submittedValue) throws ConverterException {
      return com.corejsf.util.Renderers.getConvertedValue(context, component,
         submittedValue);
   }

   public void encodeBegin(FacesContext context, UIComponent component)
         throws IOException {
      ResponseWriter writer = context.getResponseWriter();
      String clientId = component.getClientId(context);
      String formId = com.corejsf.util.Renderers.getFormId(context, component);

      UIInput spinner = (UIInput) component;
      String min = component.getAttributes().get("minimum").toString();
      String max = component.getAttributes().get("maximum").toString();
      String size = component.getAttributes().get("size").toString();
```

```
37.
38.       writer.startElement("input", spinner);
39.       writer.writeAttribute("type", "text", null);
40.       writer.writeAttribute("name", clientId , null);
41.       writer.writeAttribute("value", spinner.getValue().toString(), "value");
42.       if (size != null) writer.writeAttribute("size", size , null);
43.       writer.endElement("input");
44.
45.       writer.startElement("script", spinner);
46.       writer.writeAttribute("language", "JavaScript", null);
47.       if (min != null) {
48.          writer.write(MessageFormat.format(
49.             "document.forms[''{0}''][''{1}''].min = {2};",
50.             formId, clientId, min));
51.       }
52.       if (max != null) {
53.          writer.write(MessageFormat.format(
54.             "document.forms[''{0}''][''{1}''].max = {2};",
55.             formId, clientId, max));
56.       }
57.       writer.endElement("script");
58.
59.       writer.startElement("input", spinner);
60.       writer.writeAttribute("type", "button", null);
61.       writer.writeAttribute("value", "<", null);
62.       writer.writeAttribute("onclick",
63.          MessageFormat.format(
64.             "com.corejsf.spinner.spin(document.forms[''{0}''][''{1}''], -1);",
65.             formId, clientId),
66.          null);
67.       writer.endElement("input");
68.
69.       writer.startElement("input", spinner);
70.       writer.writeAttribute("type", "button", null);
71.       writer.writeAttribute("value", ">", null);
72.       writer.writeAttribute("onclick",
73.          MessageFormat.format(
74.             "com.corejsf.spinner.spin(document.forms[''{0}''][''{1}''], 1);",
75.             formId, clientId),
76.          null);
77.       writer.endElement("input");
78.    }
79.
80.    public void decode(FacesContext context, UIComponent component) {
81.       EditableValueHolder spinner = (EditableValueHolder) component;
82.       Map<String, String> requestMap
83.          = context.getExternalContext().getRequestParameterMap();
84.       String clientId = component.getClientId(context);
85.       spinner.setSubmittedValue((String) requestMap.get(clientId));
86.       spinner.setValid(true);
87.    }
88. }
```

Использование дочерних компонентов и аспектов

Счетчик, который обсуждался в первой половине этой главы, представляет собой простой компонент, не имеющий дочерних компонентов. Чтобы проиллюстрировать, как обеспечить управление с помощью компонента другими компонентами, реализуем область окна с вкладками, которая показана на рис. 11.7.

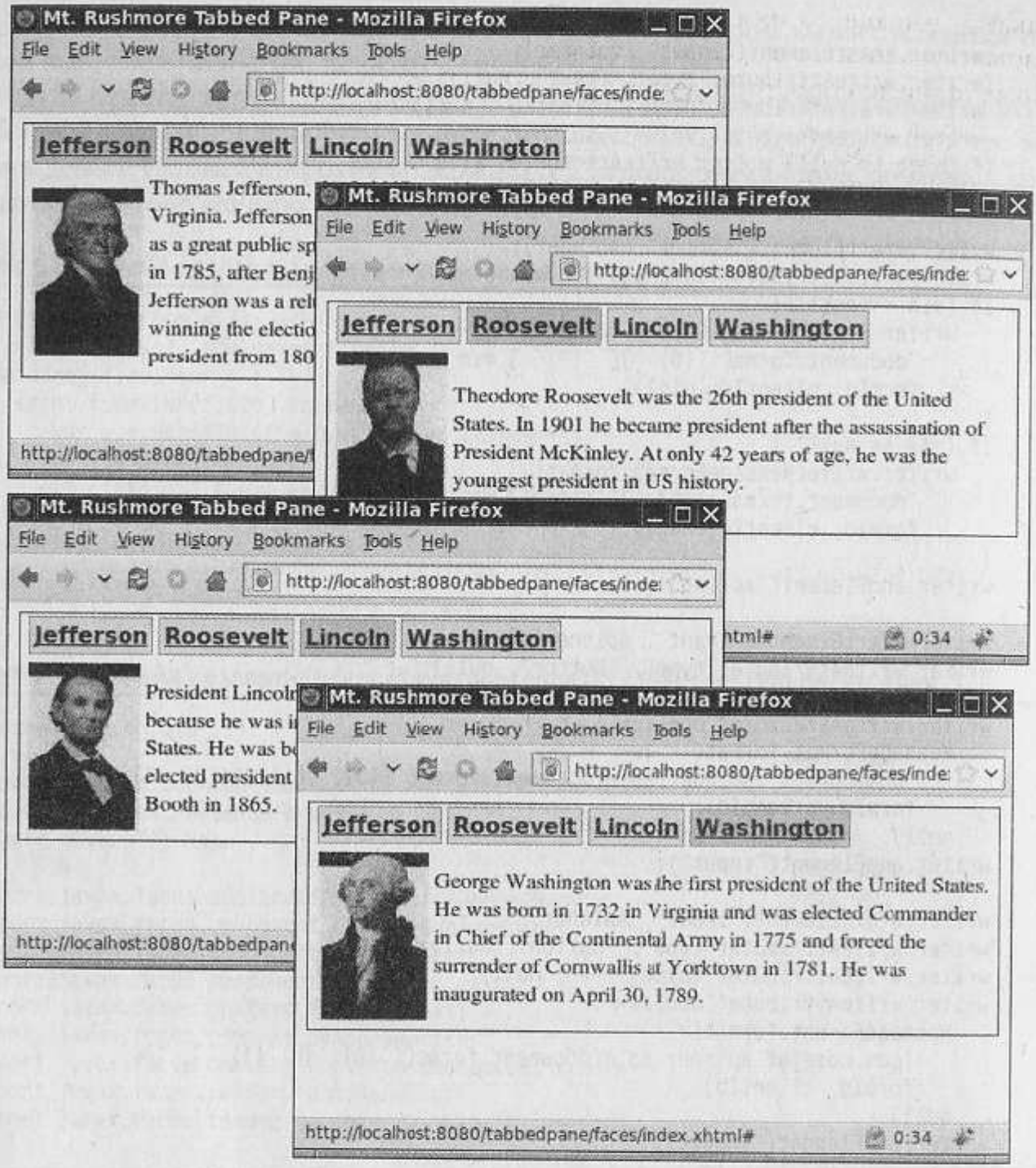


Рис. 11.7. Компонент области окна с вкладками

Компонент области окна с вкладками весьма существенно отличается от реализации области окна с вкладками, которая была показана в главе 8. Реализация в главе 8 характеризовалась тем, что была выполнена исключительно под конкретную задачу, для чего использовались стандартные теги JSF, такие как `h:graphicImage` и `h:commandLink`. А в этом разделе будет разработан повторно используемый компонент, который авторы страниц могут просто перетаскивать на свои страницы.

Для определения вкладок используются теги `f:selectItem` (или тег `f:selectItems`), причем именно такой способ используется в стандартных тегах меню и окон со списками JSF для определения элементов меню или окон со списками. Эти компоненты являются дочерними компонентами области окна с вкладками.

Содержимое области окна с вкладками определяется с помощью аспекта, подготавливаемого к выводу средством подготовки к отображению. Например, можно определить содержимое для вкладки "`Washington`" на рис. 11.7 как `washington`. После этого средство подготовки к отображению выполняет поиск аспекта области окна с

вкладками, имеющего имя washington. Такое использование аспектов аналогично использованию аспектов заголовка и нижнего колонтитула в теге h:dataTable.

Ниже приведен простой пример применения компонента области окна с вкладками.

```
<corejsf:tabbedPane >
   <f:selectItem itemLabel="Jefferson" itemValue="jefferson"/>
   <f:selectItem itemLabel="Roosevelt" itemValue="roosevelt"/>
   <f:selectItem itemLabel="Lincoln"    itemValue="lincoln"/>
   <f:selectItem itemLabel="Washington" itemValue="washington"/>
   <f:facet name="jefferson">
      <h:panelGrid columns="2">
         <h:graphicImage value="/images/jefferson.jpg"/>
         <h:outputText value="#{msgs.jeffersonDiscussion}"/>
      </h:panelGrid>
   </f:facet>
   <!-- Еще три аспекта -->
   ...
</corejsf:tabbedPane>
```

Выполнение приведенного выше кода приводит к созданию области окна с вкладками, показанной на рис. 11.8, которая выглядит как довольно упрощенная.

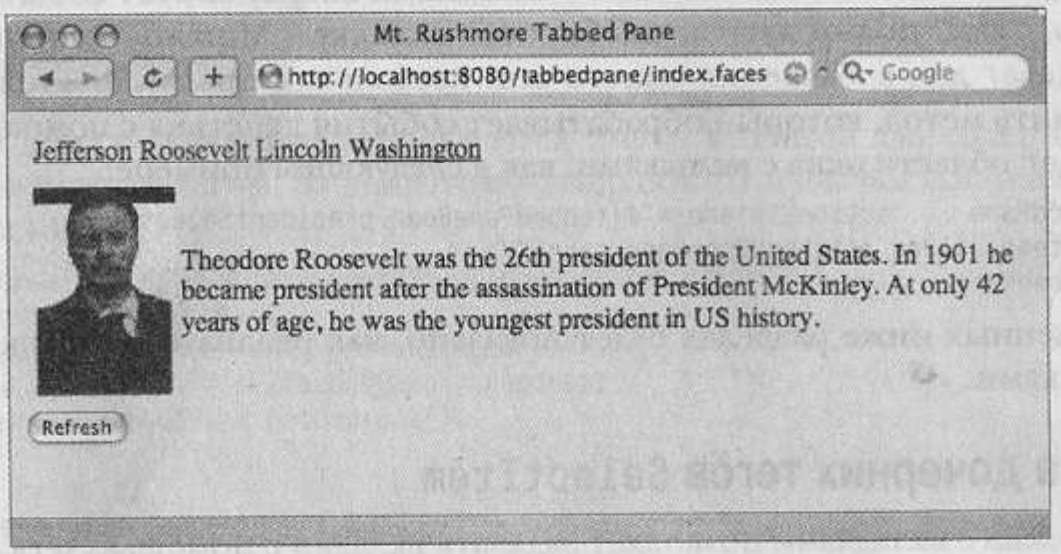


Рис. 11.8. Простая область окна с вкладками

Чтобы продемонстрировать эффекты, показанные на рис. 11.7, можно использовать стили CSS примерно так:

```
<corejsf:tabbedPane styleClass="tabbedPane"
      tabClass="tab" selectedTabClass="selectedTab">
```

Можно также применить единственный тег f:selectItems вместо нескольких тегов f:selectitem:

```
<corejsf:tabbedPane styleClass="tabbedPane"
      tabClass="tab" selectedTabClass="selectedTab">
   <f:selectItems value="#{myBean.tabs}"/>
   ...
</corejsf:tabbedPane>
```

В данном случае вкладки определены в бине.

В приведенном выше примере мы непосредственно определили текст, отображаемый в каждой вкладке в качестве меток элементов выбора: "Jefferson", "Roosevelt" и т.д. Прежде чем средство подготовки к отображению области окна с вкладками выработает код вкладки, оно проверяет, являются ли данные метки ключами в связке ресурсов; в случае положительного ответа средство подготовки к отображению кодирует значение ключа. Если метки не являются ключами в связке ресурсов, то средство

подготовки к отображению выводит в код метки в том виде, в котором они представлены. Для определения связки ресурсов применяется атрибут resourceBundle, как в следующем примере:

```
<corejsf:tabbedPane resourceBundle="com.corejsf.messages">
   <f:selectItem itemLabel="jeffersonTabText" itemValue="jefferson"/>
   <f:selectItem itemLabel="rooseveltTabText" itemValue="roosevelt"/>
   <f:selectItem itemLabel="lincolnTabText"    itemValue="lincoln"/>
   <f:selectItem itemLabel="washingtonTabText" itemValue="washington"/>
   ...
</corejsf:tabbedPane>
```

Заслуживают внимания метки элементов, поскольку все они являются ключами в связке ресурсов сообщений:

```
...
jeffersonTabText=Jefferson
rooseveltTabText=Roosevelt
lincolnTabText=Lincoln
washingtonTabText=Washington
...
```

Наконец, компонент области окна с вкладками активизирует событие действия после того, как пользователь выбирает вкладку. Можно использовать тег f:actionListener для добавления одного или нескольких прослушивателей действий или определить метод, который обрабатывает события действия с помощью атрибута actionListener области окна с вкладками, как в следующем примере:

```
<corejsf:tabbedPane ... actionListener="#{tabbedPaneBean.presidentSelected}">
   <f:selectItems value="#{tabbedPaneBean.tabs}"/>
</corejsf:tabbedPane>
```

В приведенных ниже разделах будет показано, как реализуются функции области окна с вкладками.

Обработка дочерних тегов SelectItem

Область окна с вкладками позволяет задавать вкладки с помощью тега f:selectItem или f:selectItems. Эти теги создают компоненты UISelectItem и добавляют их к области окна с вкладками в качестве дочерних компонентов. Таким образом, это средство подготовки к отображению области окна с вкладками имеет дочерние компоненты и подготавливает к выводу эти дочерние компоненты, поэтому переопределяет методы getRendersChildren() и encodeChildren():

```
public boolean getRendersChildren() {
   return true;
}
public void encodeChildren(FacesContext context, UIComponent component)
      throws java.io.IOException {
   // Если компонент окна с вкладками не имеет дочерних компонентов,
   // этот метод все равно вызывается
   if (component.getChildCount() == 0) {
      return;
   }
   ...
   for (SelectItem item : com.corejsf.util.Renderers.getSelectItems(component))
      encodeTab(context, writer, item, component);
      ...
   }
   ...
}
```

Вообще говоря, в компоненте, который обрабатывает свои дочерние компоненты, должен содержаться код, подобный следующему:

```
for (UIComponent child : component.getChildren())
   processChild(context, writer, child, component);
```

Но в рассматриваемом примере ситуация более сложная. Напомним, что в качестве значения для тега f:selectItems можно определить единственный элемент выбора, коллекцию элементов выбора, массив элементов выбора или карту объектов Java (см. главу 4). В каждом случае, в котором в классе обрабатываются дочерние компоненты, относящиеся к типу SelectItem или SelectItems, приходится учитывать это сочетание возможностей.

В методе com.corejsf.util.Renderers.getSelectItems учитываются все эти типы данных и происходит их объединение в виде списка объектов SelectItem. Код этого вспомогательного метода см. в листинге 11.8.

Метод encodeChildren класса TabbedPaneRenderer вызывает метод getSelectItems и вырабатывает для каждого дочернего компонента код его отображения в виде вкладки. Дополнительные сведения об этом будут приведены в разделе "Использование скрытых полей" на стр. 400.

Обработка аспектов

В области окна с вкладками используются имена аспектов для содержимого, связанного с конкретным тегом. За подготовку выбранного аспекта к отображению отвечает метод encodeEnd:

```
public void encodeEnd(FacesContext context, UIComponent component)
      throws java.io.IOException {
   ResponseWriter writer = context.getResponseWriter();
   UITabbedPane tabbedPane = (UITabbedPane) component;
   String content = tabbedPane.getContent();
   ...
   if (content != null) {
      UIComponent facet = component.getFacet(content);
      if (facet != null) {
         if (facet.isRendered()) {
            facet.encodeBegin(context);
            if (facet.getRendersChildren())
               facet.encodeChildren(context);
            facet.encodeEnd(context);
         }
      }
   }
}
```

Класс UITabbedPane имеет содержимое переменной экземпляра, в котором хранится имя аспекта или URL вкладки, отображаемой в настоящее время.

Метод encodeEnd проверяет, является ли содержимое вкладки, выбранной в настоящее время, именем аспекта этого компонента. В случае положительного ответа он вырабатывает код аспекта, вызывая его методы encodeBegin, encodeChildren и encodeEnd. В любом случае, если средство подготовки к отображению подготавливает для вывода свои собственные дочерние компоненты, оно должно брать на себя эту ответственность.

javax.faces.component.UIComponent

- UIComponent getFacet(String facetName)

Возвращает ссылку на аспект, если он существует. Если аспект не существует, метод возвращает `null`.

- `boolean getRendersChildren()`

  Возвращает `true`, если компонент подготавливает к отображению свои дочерние компоненты; в противном случае возвращает `false`. Метод `encodeChildren` компонента не вызывается, если этот метод не возвращает `true`. По умолчанию метод `getRendersChildren` возвращает `false`.

- `boolean isRendered()`

  Возвращает свойство `rendered`. Компонент подготавливается к отображению, только если свойство `rendered` имеет значение `true`.

Использование скрытых полей

Код каждой вкладки в области окна с вкладками представляется в виде гиперссылки:

```
<a href="#" onclick="document.forms[formId][clientId].value=content;
   document.forms[formId].submit();"/>
```

После того как пользователь щелкнет на конкретной гиперссылке, происходит отправка формы (значение `href` соответствует текущей странице). Безусловно, сервер должен иметь информацию о том, какая вкладка была выбрана. Эти сведения сохраняются в скрытом поле, которое размещается после всех вкладок:

```
<input type="hidden" name="clientId"/>
```

Вслед за отправкой формы имя и значение скрытого поля передаются назад на сервер, благодаря чему метод декодирования получает возможность активизировать выбранную вкладку. Метод `encodeTab` средства подготовки к отображению формирует теги гиперссылок. Метод `encodeEnd` вызывает метод `encodeHiddenFields()`, который вырабатывает код для скрытого поля. С подробными сведениями об этом можно ознакомиться в листинге 11.11.

При декодировании средством подготовки к отображению области окна с вкладками входящего запроса используется параметр запроса, связанный со скрытым полем, для задания содержимого компонента области окна с вкладками.

Мы также осуществляем постановку в очередь событий действий, чтобы обеспечить вызов всех закрепленных за ними прослушивателей действий:

```
public void decode(FacesContext context, UIComponent component) {
   Map<String, String> requestParams =
      context.getExternalContext().getRequestParameterMap();
   String clientId = component.getClientId(context);
   String content = (String) (requestParams.get(clientId));
   if (content != null && !content.equals("")) {
      UITabbedPane tabbedPane = (UITabbedPane) component;
      tabbedPane.setContent(content);
   }
   component.queueEvent(new ActionEvent(component));
}
```

На этом обсуждение класса `TabbedPaneRenderer` завершается. Полный код этого класса можно найти в листинге 11.11.

Листинг 11.11. Файл `tabbedpane/src/java/com/corejsf/TabbedPaneRenderer.java`

```
1. package com.corejsf;
2.
3. import java.io.IOException;
```

```
4. import java.util.Map;
5. import java.util.logging.Level;
6. import java.util.logging.Logger;
7. import javax.faces.component.UIComponent;
8. import javax.faces.context.ExternalContext;
9. import javax.faces.context.FacesContext;
10. import javax.faces.context.ResponseWriter;
11. import javax.faces.event.ActionEvent;
12. import javax.faces.model.SelectItem;
13. import javax.faces.render.FacesRenderer;
14. import javax.faces.render.Renderer;
15. import javax.servlet.ServletContext;
16. import javax.servlet.ServletException;
17. import javax.servlet.ServletRequest;
18. import javax.servlet.ServletResponse;
19.
20. // Средство подготовки к отображению для компонента UITabbedPane
21.
22. @FacesRenderer(componentFamily="javax.faces.Command",
23.    rendererType="com.corejsf.TabbedPane")
24. public class TabbedPaneRenderer extends Renderer {
25.    private static Logger logger = Logger.getLogger("com.corejsf.util");
26.
27.    // По умолчанию метод getRendersChildren() возвращает false, поэтому метод
28.    // encodeChildren() не будет вызываться, если мы не преопределим
29.    // getRendersChildren(), чтобы он возвращал true
30.    public boolean getRendersChildren() {
31.       return true;
32.    }
33.
34.    // Метод decode получает значение параметра запроса, именем которого является
35.    // клиентский идентификатор компонента окна с вкладками. Параметр запроса
36.    // кодируется как скрытое поле методом encodeHiddenField, который вызывается
37.    // методом encodeEnd. Значение параметра задается с помощью кода JavaScript,
38.    // вырабатываемого с помощью метода encodeTab. Оно представляет собой имя
39.    // аспекта или страницы JSP.
40.    // В методе decode используется значение параметра запроса для задания атрибута
41.    // content компонента окна с вкладками.
42.    // Наконец, метод decode() ставит в очередь событие действия, которое
43.    // активизировано для зарегистрированных прослушивателей в фазе вызова приложения
44.    // жизненного цикла JSF. Прослушиватели действий могут быть заданы с помощью
45.    // атрибута actionListener компонента <corejsf:tabbedpane>или тегов
46.    //  <f:actionListener>, вложенных в компонент <corejsf:tabbedpane>.
47.
48.    public void decode(FacesContext context, UIComponent component) {
49.       Map<String, String> requestParams
50.          = context.getExternalContext().getRequestParameterMap();
51.       String clientId = component.getClientId(context);
52.
53.          String content = (String) (requestParams.get(clientId));
54.          if (content != null && !content.equals("")) {
55.             UITabbedPane tabbedPane = (UITabbedPane) component;
56.             tabbedPane.setContent(content);
57.          }
58.
59.          component.queueEvent(new ActionEvent(component));
60.    }
61.
62.    // Метод encodeBegin записывает начальный элемент <table> языка HTML
63.    // с применением класса CSS, указанного атрибутом styleClass компонента
64.    //  <corejsf:tabbedpane> (если он задан)
65.
```

```
66.    public void encodeBegin(FacesContext context, UIComponent component)
67.          throws java.io.IOException {
68.       ResponseWriter writer = context.getResponseWriter();
69.       writer.startElement("table", component);
70.
71.         String styleClass = (String) component.getAttributes().get("styleClass");
72.         if (styleClass != null)
73.            writer.writeAttribute("class", styleClass, null);
74.
75.         writer.write("\n"); // Повышение удобства чтения сформированного кода HTML
76.    }
77.
78.    // Метод encodeChildren() вызывается реализацией JSF после encodeBegin().
79.    // Дочерними по отношению к компоненту <corejsf:tabbedpane> являются компоненты
80.    // UISelectItem, заданные с помощью одного или нескольких тегов <f:selectItem> или
81.    // одного тега <f:selectItems>, вложенных в <corejsf:tabbedpane>
82.
83.    public void encodeChildren(FacesContext context, UIComponent component)
84.          throws java.io.IOException {
85.       // Даже если компонент окна с вкладками не имеет дочерних компонентов, этот
86.       // метод все равно вызывается
87.       if (component.getChildCount() == 0) {
88.          return;
89.       }
90.
91.         ResponseWriter writer = context.getResponseWriter();
92.         writer.startElement("thead", component);
93.         writer.startElement("tr", component);
94.         writer.startElement("th", component);
95.
96.         writer.startElement("table", component);
97.         writer.startElement("tbody", component);
98.         writer.startElement("tr", component);
99.
100.      for (SelectItem item : com.corejsf.util.Renderers.getSelectItems(component))
101.         encodeTab(context, writer, item, component);
102.
103.      writer.endElement("tr");
104.      writer.endElement("tbody");
105.      writer.endElement("table");
106.
107.      writer.endElement("th");
108.      writer.endElement("tr");
109.      writer.endElement("thead");
110.      writer.write("\n"); // Повышение удобства чтения сформированного кода HTML
111.   }
112.
113.   // Метод encodeEnd() вызывается реализацией JSF после encodeChildren().
114.   // Метод encodeEnd() записывает текст таблицы и кодирует содержимое
115.   // окна с вкладками в одной строке таблицы
116.
117.   // Содержимое окна с вкладками может быть задано либо с помощью URL для страницы
118.   // JSP, либо с указанием имена аспекта, поэтому метод encodeEnd() определяет,
119.   //не задан ли аспект;в случае положительного ответа производится кодирование
120.   // аспекта; в противном случае происходит включение страницы JSP
121.   public void encodeEnd(FacesContext context, UIComponent component)
122.         throws java.io.IOException {
123.      ResponseWriter writer = context.getResponseWriter();
124.      UITabbedPane tabbedPane = (UITabbedPane) component;
125.      String content = tabbedPane.getContent();
126.
127.       writer.startElement("tbody", component);
```

```
128.        writer.startElement("tr", component);
129.        writer.startElement("td", component);
130.
131.        if (content != null) {
132.           UIComponent facet = component.getFacet(content);
133.           if (facet != null) {
134.              if (facet.isRendered()) {
135.                 facet.encodeBegin(context);
136.                 if (facet.getRendersChildren())
137.                    facet.encodeChildren(context);
138.                 facet.encodeEnd(context);
139.              }
140.           } else
141.              includePage(context, component);
142.        }
143.
144.        writer.endElement("td");
145.        writer.endElement("tr");
146.        writer.endElement("tbody");
147.
148.         // Закрытие столбца, строки и элементов таблицы
149.         writer.endElement("table");
150.
151.         encodeHiddenField(context, writer, component);
152.     }
153.
154.     // Метод encodeHiddenField вызывается в конце работы метода encodeEnd().
155.     // См. описание метода decode для получения сведений об этом поле и его значении
156.
157.     private void encodeHiddenField(FacesContext context, ResponseWriter writer,
158.           UIComponent component) throws java.io.IOException {// Запись скрытого поля,
159.        // именем которого является клиентский идентификатор окна с вкладками
160.        writer.startElement("input", component);
161.        writer.writeAttribute("type", "hidden", null);
162.        writer.writeAttribute("name", component.getClientId(context), null);
163.        writer.endElement("input");
164.     }
165.
166.     // Метод encodeTab, который вызывается методом encodeChildren, кодирует элемент
167.     // анкера HTML с помощью атрибута onclick, который задает значение скрытого поля,
168.     // закодированного методом encodeHiddenField, и передает форму, включающую окно
169.     // Дополнительные сведения о скрытом поле приведены в описании метода decode.
170.     // с вкладками. Кроме того, метод encodeTab записывает по одному атрибуту класса
171.     // для каждой вкладки, соответствующей либо атрибуту tabClass (для невыбранных
172.     // вкладок), либо атрибуту selectedTabClass (для выбранной вкладки).
173.
174.     private void encodeTab(FacesContext context, ResponseWriter writer,
175.           SelectItem item, UIComponent component) throws java.io.IOException {
176.        String tabText = getLocalizedTabText(component, item.getLabel());
177.        String content = (String) item.getValue();
178.
179.         writer.startElement("td", component);
180.         writer.startElement("a", component);
181.         writer.writeAttribute("href", "#", "href");
182.
183.         String clientId = component.getClientId(context);
184.         String formId = com.corejsf.util.Renderers.getFormId(context, component);
185.
186.         writer.writeAttribute("onclick",
187.         // Запись значения для выбранного поля, именем которого является клиентский
188.         // идентификатор окна с вкладками
189.               "document.forms['" + formId + "']['" + clientId + "'].value='"
```

```
                    + content + "'; " +

                     // Передача формы, в которой находится окно с вкладками
                     "document.forms['" + formId + "'].submit(); ", null);

        UITabbedPane tabbedPane = (UITabbedPane) component;
        String selectedContent = tabbedPane.getContent();

        String tabClass = null;
        if (content.equals(selectedContent))
           tabClass = (String) component.getAttributes().get("selectedTabClass");
        else
           tabClass = (String) component.getAttributes().get("tabClass");

        if (tabClass != null)
           writer.writeAttribute("class", tabClass, null);

        writer.write(tabText);

        writer.endElement("a");
        writer.endElement("td");
        writer.write("\n"); // Повышение удобства чтения сформированного кода HTML
    }

    // Текст для вкладок в компоненте окна с вкладками может быть задан как ключ в
    // связке ресурсов или как действительный текст, который отображается на вкладке.
    // После получения этого текста метод getLocalizedTabText предпринимает попытку
    // получить значение из связки ресурсов, заданной атрибутом resourceBundle
    // компонента <corejsf:tabbedpane>. Если такое значение не обнаруживается,
    // метод getLocalizedTabText просто возвращает переданную ему строку

    private String getLocalizedTabText(UIComponent tabbedPane, String key) {
       String bundle = (String) tabbedPane.getAttributes().get("resourceBundle");
       String localizedText = null;

        if (bundle != null) {
           localizedText = com.corejsf.util.Messages.getString(bundle, key, null);
        }
        if (localizedText == null)
           localizedText = key;
        // Параметр key фактически не является ключом в связке ресурсов,
        // поэтому просто возвращается как строка в неизменном виде
        return localizedText;
    }

    // В методе includePage используется диспетчер запросов к сервлету для включения
    // страницы, соответствующей выбранной вкладке

    private void includePage(FacesContext fc, UIComponent component) {
       ExternalContext ec = fc.getExternalContext();
       ServletContext sc = (ServletContext) ec.getContext();
         UITabbedPane tabbedPane = (UITabbedPane) component;
         String content = tabbedPane.getContent();

         ServletRequest request = (ServletRequest) ec.getRequest();
         ServletResponse response = (ServletResponse) ec.getResponse();
         try {
            sc.getRequestDispatcher(content).include(request, response);
         } catch (ServletException ex) {
            logger.log(Level.WARNING, "Couldn't load page: " + content, ex);
         } catch (IOException ex) {
            logger.log(Level.WARNING, "Couldn't load page: " + content, ex);
```

```
252.         }
253.  }
254.}
```

Сохранение и восстановление состояния

Реализация JSF сохраняет и восстанавливает состояние представления между запросами. Это состояние включает компоненты, преобразователи, средства проверки и прослушиватели событий. Разработчик должен убедиться в том, что создаваемые им пользовательские компоненты могут участвовать в процессе сохранения состояния.

После того как конкретное приложение сохраняет состояние на сервере, объекты представления сохраняются в памяти. Но если состояние сохраняется на клиенте, то объекты представления кодируются и сохраняются в скрытом поле, в очень длинной строке, которая выглядит примерно таким образом:

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
   value="rO0ABXNyACBjb20uc3VuLmZhY2VzLnVOaWwuVHJlZVNOcnVjdHVyZRRmGOQclWAgAgAETAAI...
   ...4ANXBwcHBwcHBwcHBwcHBwcHBxAH4ANXEAfgA1cHBwcHQABnN1Ym1pdHVxAH4ALAAAAAA=" />
```

Если состояние сохраняется на клиенте, то приходится обеспечивать нормальную работу приложения, даже если пользователи запрещают применение cookie-файлов, но при этом количество данных, которые сервер должен держать у себя для каждого пользователя веб-приложения, существенно уменьшается.

Один из подходов к решению этой задачи состоит в реализации методов `saveState` и `restoreState` интерфейса `StateHolder`.

Эти методы имеют следующую форму:

```
public Object saveState(FacesContext context) {
   Object[] values = new Object[n];
   values[0] = super.saveState(context);
   values[1] = instance variable #1;
   values[2] = instance variable #2;
   ...
   return values;
}
public void restoreState(FacesContext context, Object state) {
    Object values[] = (Object[]) state;
    super.restoreState(context, values[0]);
    Переменная экземпляра №1 = (Type) values[1];
    Переменная экземпляра №2 = (Type) values[2];
    ...
}
```

В данном случае предполагается, что значения в переменных экземпляра являются сериализуемыми. Если бы это условие не соблюдалось, то пришлось бы предусматривать сериализуемое представление состояния компонента.

На заметку! У читателя может возникнуть вопрос, почему в этих средствах реализации нельзя было просто использовать стандартный алгоритм сериализации Java. Дело в том, что средства сериализации Java, несмотря на свою весьма значительную универсальность, не обязательно предоставляют наиболее эффективный формат для кодирования состояния компонента. Архитектура JSF позволяет использовать реализации JSF для создания более эффективных механизмов.

Совет. Если вся информация о состоянии компонента хранится в виде атрибутов, то нет необходимости реализовывать методы `saveState` и `restoreState`, поскольку атрибуты компонента сохраняются авто-

матически реализацией JSF. В качестве примера отметим, что для хранения содержимого области окна с вкладками может использоваться атрибут `content` вместо переменной экземпляра `content`.

В этом случае класс `UITabbedPane` вообще не требуется. Достаточно воспользоваться суперклассом `UICommand` и объявить класс компонента:

```
<component>
        <component-type>com.corejsf.TabbedPane</component-type>
        <component-class>javax.faces.component.UICommand</component-class>
    </component>
```

Частичное сохранение состояния JSF 2.0

В версии JSF 2.0 предусмотрен улучшенный алгоритм сохранения состояния, в основе которого лежит простое наблюдение. В том, чтобы компоненты страницы хранили начальное состояние, устанавливаемое при создании представления, нет необходимости. Дело в том, что это состояние всегда может быть получено путем повторного создания представления. Требуют сохранения только различия между начальным и текущим состояниями. Для отслеживания того, произошло ли добавление или изменение значений после формирования представления, может применяться класс средства поддержки состояния, `StateHelper`.

Чтобы воспользоваться таким частичным сохранением состояния, необходимо сохранять применяемые свойства компонента в средстве поддержки состояния компонента. Класс `UITabbedPane` имеет одно свойство: имя аспекта вкладки, отображаемой в настоящее время. Ниже показано, как можно сохранить это свойство в средстве поддержки состояния.

```
private enum PropertyKeys { content };
public String getContent() {
   return (String) getStateHelper().get(PropertyKeys.content);
}
public void setContent(String newValue) {
   getStateHelper().put(PropertyKeys.content, newValue);
}
```

Если используется средство поддержки состояния, то нет необходимости реализовывать методы `saveState` и `restoreState`.

Чтобы проверить, действительно ли необходимо сохранение состояния, выполните такой эксперимент.

1. Замените методы `getContent` и `setContent` следующим:

   ```
   private String content;
   public String getContent() { return content; }
   public void setContent(String newValue) { content = newValue; }
   ```

2. Активизируйте клиентские средства сохранения состояния, добавив следующие строки в файл `web.xml`:

   ```
   <context-param>
      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
      <param-value>client</param-value>
   </context-param>
   ```

3. Добавьте кнопку `<h:commandButton value="Test State Saving"/>` в файл `index.faces`.

4. Запустите приложение и щелкните на одной из вкладок.

5. Щелкните на кнопке Test State Saving (Проверка сохранения состояния). Текущая страница отобразится снова, но ни одна из вкладок не будет выбрана!

В листинге 11.12 показано, как класс UITabbedPane сохраняет и восстанавливает свое состояние. В листинге 11.13 приведена страница JSF для приложения области окна с вкладками.

Листинг 11.12. Файл tabbedpane/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:corejsf="http://corejsf.com">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.windowTitle}</title>
   </h:head>
   <h:body>
      <h:form>
         <corejsf:tabbedPane styleClass="tabbedPane" tabClass="tab"
                             selectedTabClass="selectedTab">
            <f:facet name="jefferson">
               <h:panelGrid columns="2">
                  <h:graphicImage library="images" name="jefferson.jpg"/>
                  <h:outputText value="#{msgs.jeffersonDiscussion}"
                                styleClass="tabbedPaneContent"/>
               </h:panelGrid>
            </f:facet>
            <f:facet name="roosevelt">
               <h:panelGrid columns="2">
                  <h:graphicImage library="images" name="roosevelt.jpg"/>
                  <h:outputText value="#{msgs.rooseveltDiscussion}"
                                styleClass="tabbedPaneContent"/>
               </h:panelGrid>
            </f:facet>
            <f:facet name="lincoln">
               <h:panelGrid columns="2">
                  <h:graphicImage library="images" name="lincoln.jpg"/>
                  <h:outputText value="#{msgs.lincolnDiscussion}"
                                styleClass="tabbedPaneContent"/>
               </h:panelGrid>
            </f:facet>
            <f:facet name="washington">
               <h:panelGrid columns="2">
                  <h:graphicImage library="images" name="washington.jpg"/>
                  <h:outputText value="#{msgs.washingtonDiscussion}"
                                styleClass="tabbedPaneContent"/>
               </h:panelGrid>
            </f:facet>

            <f:selectItem itemLabel="#{msgs.jeffersonTabText}"
                          itemValue="jefferson"/>
            <f:selectItem itemLabel="#{msgs.rooseveltTabText}"
                          itemValue="roosevelt"/>
            <f:selectItem itemLabel="#{msgs.lincolnTabText}"
                          itemValue="lincoln"/>
            <f:selectItem itemLabel="#{msgs.washingtonTabText}"
```

```
53.             </corejsf:tabbedPane>
54.             <!-- <h:commandButton value="Test State Saving"/> -->
55.         </h:form>
56.     </h:body>
57. </html>
```

Листинг 11.13. Файл tabbedpane/src/java/com/corejsf/UITabbedPane.java

```
1. package com.corejsf;
2.
3. import javax.faces.component.FacesComponent;
4. import javax.faces.component.UICommand;
5.
6. @FacesComponent("com.corejsf.TabbedPane")
7. public class UITabbedPane extends UICommand {
8.    private enum PropertyKeys { content };
9.
10.   public String getContent() {
11.      return (String) getStateHelper().get(PropertyKeys.content);
12.   }
13.
14.   public void setContent(String newValue) {
15.      getStateHelper().put(PropertyKeys.content, newValue);
16.   }
17.
18. /*
19.    Эта версия используется для проверки того, что происходит
20.    при отсутствии сохранения состояния
21.    private String content;
22.
23.    public String getContent() { return content; }
24.    public void setContent(String newValue) { content = newValue; }
25. */
26. }
```

`javax.faces.component.StateHolder`

- `Object saveState(FacesContext context)`

  Возвращает объект `Serializable`, который сохраняет состояние этого объекта.

- `void restoreState(FacesContext context, Object state)`

  Восстанавливает состояние данного объекта из указанного объекта состояния, который является копией объекта, ранее полученного путем вызова метода `saveState`.

- `void setTransient(boolean newValue)`
- `boolean isTransient()`

  Задает и возвращает промежуточное свойство `transient`. Если это свойство задано, состояние не сохраняется.

`javax.faces.component.StateHelper` JSF 2.0

- `Object put(Serializable key, Object value)`

  Помещает пару "ключ-значение" в эту карту средства поддержки состояния и возвращает предыдущее значение или `null`, если такая пара не была задана.

- `Object get(Serializable key)`

  Возвращает значение, связанное с ключом в этой карте средства поддержки состояния, или `null`, если таковое отсутствует.

Создание компонентов Ajax JSF 2.0

Предусмотрены два способа добавления функциональных средств Ajax к конкретным пользовательским компонентам.

1. Внедрение кода Ajax в пользовательский компонент. Как правило, для этого необходимо реализовать пользовательское средство подготовки к отображению, вырабатывающее код JavaScript, в котором, в свою очередь, используется библиотека Ajax JSF 2.0 для выполнения вызовов Ajax.
2. Поддержка пользовательским компонентом средств Ajax путем поддержки тега `f:ajax`. Затем авторы страницы могут использовать тег `f:ajax` в сочетании с конкретным компонентом для реализации собственных функциональных средств Ajax.

Оба варианта являются в равной степени применимыми. Например, может потребоваться реализовать средство выбора даты, в котором предусмотрены разворачивающиеся меню для выбора месяца и числа. После того как пользователь выберет месяц, в данном средстве выбора даты вводятся в действие функции Ajax для обновления списка элементов выбора в разворачивающемся меню чисел. В этом случае необходимо внедрить указанные функциональные средства Ajax в данное средство выбора даты.

Для авторов страницы удобно также, если применяемые ими пользовательские компоненты работают с тегом `f:ajax`, поскольку при этом авторы страницы получают возможность дополнительно закреплять свои собственные функциональные возможности Ajax, необходимые на данный момент, за этими пользовательскими компонентами.

В следующих двух разделах будет показано, как внедрять функциональные средства Ajax в пользовательские компоненты и как обеспечить поддержку тега `f:ajax`. Эти решения будут проиллюстрированы на примере счетчика размера шрифта, который показан на рис. 11.9.

Счетчик размера шрифта является несложным. После того как пользователь изменяет значение счетчика, выполняется вызов Ajax к серверу, а затем обновляется размер шрифта символов, отображаемых под счетчиком, чтобы было видно, каким стал новый размер шрифта.

В примере, показанном на рис. 11.9, применяется пользовательский компонент счетчика размера шрифта, который реализует средство Ajax, содержащее все необходимое в самом себе. Для автора страницы достаточно добавить тег `corejsf:fontSpinner` к странице, после чего компонент счетчика размера шрифта обеспечит реализацию всех нюансов поддержки технологии Ajax.

Был также реализован второй пример, который выглядит и действует по аналогии с приложением, показанным на рис. 11.9, но во втором примере добавлена поддержка тега `f:ajax` к счетчику, о чем речь шла в разделе "Формирование кода JavaScript" на стр. 393, а функциональные средства Ajax реализованы автором страницы в файле XHTML.

Обсуждение обеих реализаций счетчика размера шрифта приведено в следующих двух разделах.

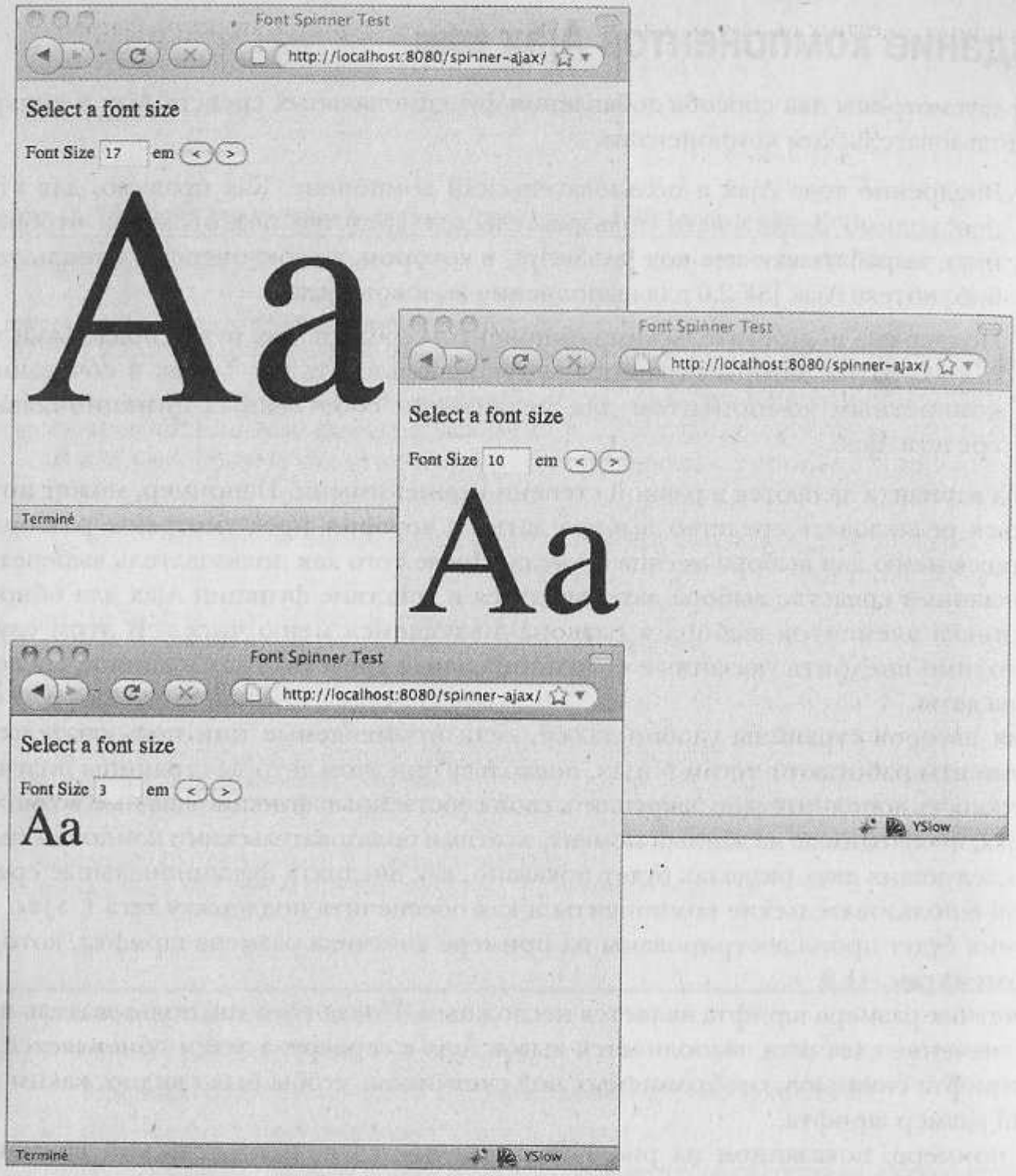


Рис. 11.9. Счетчик размера шрифта

Реализация самодостаточного кода Ajax в пользовательских компонентах

В этой самодостаточной версии счетчика размера шрифта предусмотрено, что счетчик размера шрифта инкапсулирует все необходимые для него функциональные средства Ajax, поэтому автору страницы достаточно лишь добавить тег счетчика размера шрифта к представлению:

```
<h:form id="spinnerForm">
   #{msgs.fontSizePrompt}
   <corejsf:fontSpinner value="#{fontSpecifics.size}"
      id="fontSpinner" minimum="1" maximum="100" size="3"/>
</h:form>
```

Значение счетчика размера шрифта указывает на свойство простого управляемого бина:

```
public class FontSpecifics implements Serializable {
   private int size = 1;
   public int getSize() { return size; }
   public void setSize(int size) { this.size = size; }
}
```

Этот счетчик размера шрифта аналогичен компоненту счетчика, который рассматривался ранее в этой главе, за исключением того, что под счетчиком добавлен двухсимвольный дисплей. Код, представляющий наибольший интерес, относится к средству подготовки к отображению счетчика размера шрифта и приведен в листинге 11.14.

В средстве подготовки к отображению счетчика размера шрифта формируется код JavaScript, который выполняет вызов Ajax к серверу. В этом коде JavaScript используется встроенная библиотека JavaScript, которая входит в состав версии JSF 2.0. В средстве подготовки к отображению проверяется, внедрена ли ссылка на встроенную библиотеку JavaScript в код страницы с помощью аннотации `ResourceDependency`:

```
@ResourceDependencies( {
   @ResourceDependency(library="javascript", name="spinner.js"),
   @ResourceDependency(library="javax.faces", name="jsf.js")
})
```

Заслуживает внимания то, что для рассматриваемого счетчика размера шрифта фактически нужны две библиотеки JavaScript, поэтому для соблюдения этого требования используется аннотация `ResourceDependencies`.

После подключения требуемых библиотек JavaScript создается определенный объем кода JavaScript:

```
String ajaxScript = MessageFormat.format(
   "if(document.forms[''{0}''][''{1}''].value != '''') {2};",
   formId, clientId, getAjaxScript(context, spinner));
```

Этот сценарий формируется с помощью метода `getAjaxScript()`, но выполняется, только если значением счетчика не является пустая строка. Это позволяет избежать возникновения такого крайнего случая, в котором предпринимается попытка показать на счетчике размер шрифта, равный 0 пунктам.

Метод `getAjaxScript()` возвращает сценарий, который выполняет вызов Ajax:

```
"jsf.ajax.request('" + component.getClientId() +
   "', null, {'render': '" + component.getParent().getClientId() + "' })"
```

В сценарии вызывается функция `jsf.ajax.request()` с указанием счетчика, который рассматривается как компонент, активизировавший вызов Ajax, и неопределенного события (параметр `null`). В сценарии также определяется, что родительский компонент компонента счетчика (в данном случае форма) подготовливается к отображению после вызова Ajax.

В средстве подготовки к отображению счетчика размера шрифта этот сформированный код JavaScript используется для вызова Ajax, после того, как происходит событие `keyup` в поле ввода текста счетчика и возникает событие `onclick` в кнопках.

Вызов Ajax, выполняемый счетчиком размера шрифта, приводит к выполнению кода счетчика на сервере, который сохраняет значение счетчика в компоненте счетчика. После возврата из этого вызова Ajax реализация JSF подготавливает счетчик к отображению, включая текст, который появляется под счетчиком:

```
writer.startElement("span", spinner);
String s = ((Integer) spinner.getValue()).toString();
writer.writeAttribute("style", "font-size: " + s + "em;", null);
writer.write("Aa");
writer.endElement("span");
```

Средство подготовки к отображению корректирует размер шрифта символов под счетчиком в соответствии со значением счетчика.

Листинг 11.14. Файл spinner-ajax/src/java/com/corejsf/FontSpinnerRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.text.MessageFormat;
5. import java.util.Map;
6.
7. import javax.faces.application.ResourceDependencies;
8. import javax.faces.application.ResourceDependency;
9. import javax.faces.component.EditableValueHolder;
10. import javax.faces.component.UIComponent;
11. import javax.faces.component.UIInput;
12. import javax.faces.context.FacesContext;
13. import javax.faces.context.ResponseWriter;
14. import javax.faces.convert.ConverterException;
15. import javax.faces.render.FacesRenderer;
16. import javax.faces.render.Renderer;
17.
18. @FacesRenderer(componentFamily="javax.faces.Input",
19.    rendererType="com.corejsf.FontSpinner")
20.
21. @ResourceDependencies( {
22.    @ResourceDependency(library="javascript", name="spinner.js"),
23.    @ResourceDependency(library="javax.faces", name="jsf.js")
24. })
25. public class FontSpinnerRenderer extends Renderer {
26.    public Object getConvertedValue(FacesContext context, UIComponent component,
27.          Object submittedValue) throws ConverterException {
28.       return com.corejsf.util.Renderers.getConvertedValue(context, component,
29.          submittedValue);
30.    }
31.
32.    public void encodeBegin(FacesContext context, UIComponent component)
33.          throws IOException {
34.       ResponseWriter writer = context.getResponseWriter();
35.       String clientId = component.getClientId(context);
36.       String formId = com.corejsf.util.Renderers.getFormId(context, component);
37.       UIInput spinner = (UIInput) component;
38.       String ajaxScript = MessageFormat.format(
39.          "if(document.forms[''{0}''][''{1}''].value != '''') {2};",
40.          formId, clientId, getChangeScript(context, spinner));
41.
42.       String min = component.getAttributes().get("minimum").toString();
43.       String max = component.getAttributes().get("maximum").toString();
44.       String size = component.getAttributes().get("size").toString();
45.
46.       writer.startElement("input", spinner);
47.       writer.writeAttribute("type", "text", null);
48.       writer.writeAttribute("name", clientId , null);
49.       writer.writeAttribute("id", clientId, null);
50.       writer.writeAttribute("value", spinner.getValue().toString(), "value");
```

```
51.       if (size != null) writer.writeAttribute("size", size , null);
52.
53.       writer.writeAttribute("onkeyup", ajaxScript, null);
54.       writer.endElement("input");
55.
56.       writer.startElement("script", spinner);
57.       writer.writeAttribute("language", "JavaScript", null);
58.       if (min != null) {
59.          writer.write(MessageFormat.format(
60.             "document.forms[''{0}''][''{1}''].min = {2};",
61.             formId, clientId, min));
62.       }
63.       if (max != null) {
64.          writer.write(MessageFormat.format(
65.             "document.forms[''{0}''][''{1}''].max = {2};",
66.             formId, clientId, max));
67.       }
68.       writer.endElement("script");
69.       writer.write(" em ");
70.
71.       String spinScript = MessageFormat.format(
72.          "com.corejsf.spinner.spin(document.forms[''{0}''][''{1}''], -1);",
73.          formId, clientId);
74.
75.       writer.startElement("input", spinner);
76.       writer.writeAttribute("type", "button", null);
77.       writer.writeAttribute("value", "<", null);
78.       writer.writeAttribute("onclick", spinScript + ajaxScript, null);
79.       writer.endElement("input");
80.
81.       spinScript = MessageFormat.format(
82.          "com.corejsf.spinner.spin(document.forms[''{0}''][''{1}''], 1);",
83.          formId, clientId);
84.
85.       writer.startElement("input", spinner);
86.       writer.writeAttribute("type", "button", null);
87.       writer.writeAttribute("value", ">", null);
88.       writer.writeAttribute("onclick", spinScript + ajaxScript, null);
89.       writer.endElement("input");
90.
91.       writer.startElement("br", spinner);
92.       writer.endElement("br");
93.
94.       writer.startElement("span", spinner);
95.       String s = ((Integer)spinner.getValue()).toString();
96.       writer.writeAttribute("style", "font-size: " + s + "em;", null);
97.       writer.write("Aa");
98.       writer.endElement("span");
99.    }
100.
101.   public void decode(FacesContext context, UIComponent component) {
102.      EditableValueHolder spinner = (EditableValueHolder) component;
103.      Map<String, String> requestMap
104.         = context.getExternalContext().getRequestParameterMap();
105.      String clientId = component.getClientId(context);
106.      spinner.setSubmittedValue((String) requestMap.get(clientId));
107.      spinner.setValid(true);
108.   }
109.
110.   protected final String getChangeScript(FacesContext context, UIInput component)
111.      throws IOException {
112. return
```

```
113.     "jsf.ajax.request('" + component.getClientId() +
114.     "', null, { 'render': '" +
115.     component.getParent().getClientId() + "' })";
116. }
117.}
```

Поддержка тега f:ajax в пользовательских компонентах

Во многих случаях имеет смысл реализовывать самодостаточные функциональные средства Ajax в пользовательском компоненте, как было показано в предыдущем разделе. Но целесообразно также обеспечивать поддержку тега f:ajax для конкретных пользовательских компонентов, чтобы авторы страниц могли закреплять дополнительные функциональные средства Ajax за предлагаемыми им пользовательскими компонентами.

В этом разделе рассматривается реализация счетчика размера шрифта, альтернативная по отношению к той, которая обсуждалась в предыдущем разделе. Вместо непосредственного внедрения функциональных средств Ajax в целях обновления текстового дисплея при изменении пользователем значения счетчика в данном случае возможность определить это правило поведения предоставляется автору страницы:

```
<h:form id="spinnerForm">
   #{msgs.fontSizePrompt}
   <corejsf:spinner value="#{fontSpecifics.size}"
         id="fontSpinner" minimum="1" maximum="100" size="3">
      <f:ajax render="fontPreview"/>
   </corejsf:spinner>
   <br/>
   <h:outputText id="fontPreview" value="Aa"
         style="font-size: #{fontSpecifics.size}em"/>
</h:form>
```

В приведенной выше разметке за создание двухсимвольного дисплея под счетчиком отвечает автор страницы, а не разработчик компонента счетчика. Кроме того, функциональные средства Ajax реализует автор страницы, а не разработчик компонента, для чего просто тег f:ajax добавляется к счетчику.

После того как в счетчике происходит событие Ajax по умолчанию (которое показывает, что пользователь изменил значение счетчика), клиент выполняет вызов Ajax к серверу, приводящий к сохранению значения размера шрифта в свойстве управляемого бина. После возврата из этого вызова Ajax клиент повторно подготавливает к отображению двухсимвольный дисплей и определяет размер шрифта выводимого текста в соответствии со значением, введенным пользователем в счетчике.

Чтобы обеспечить поддержку тега f:ajax, в пользовательских компонентах необходимо реализовать интерфейс ClientBehaviorHolder. Тег f:ajax закрепляет клиентские правила поведения за компонентом, включающим этот тег, поэтому разрабатываемый тег счетчика должен придерживать это клиентское правило поведения: интерфейс носит такое имя как раз в связи с этим.

Интерфейс ClientBehaviorHolder определяет четыре метода:

- void addClientBehavior(String event, ClientBehavior behavior)
- Map<String, List<ClientBehavior>> getClientBehaviors()
- String getDefaultEventName()
- Collection<String> getEventNames()

Для удобства в классе UIComponentBase (от которого в конечном счете наследует компонент счетчика) предусмотрены реализации по умолчанию этих четырех методов, даже притом, что сам класс UIComponentBase не реализует интерфейс ClientBehaviorHolder.

Реализации методов addClientBehavior() и getClientBehaviors() в классе UIComponentBase являются вполне приемлемыми, но необходимо сообщить реализации JSF, какие события должны поддерживаться для вызовов Ajax и какое событие рассматривается как событие по умолчанию, которое активизирует вызов Ajax. В данном случае событием по умолчанию является щелчок, поэтому после добавления автором страницы тега f:ajax без указания события для счетчика реализация JSF связывает это правило поведения Ajax с событием щелчка.

Класс UISpinner показан ниже.

```
@FacesComponent("com.corejsf.Spinner")
public class UISpinner extends UIInput implements ClientBehaviorHolder {
   private static List<String> eventNames = Arrays.asList("click");
   public UISpinner() {
      setConverter(new IntegerConverter()); // Преобразование переданного значения.
      setRendererType("com.corejsf.JSSpinner"); // Этот компонент имеет средство
                                                // подготовки к отображению
   }
   @Override public String getDefaultEventName() { return "click"; }
   @Override public Collection<String> getEventNames() { return eventNames; }
}
```

Но основная часть средств поддержки счетчика для тега f:ajax реализована в средстве подготовки к отображению счетчика, которое показано в листинге 11.15.

Эта версия средства подготовки к отображению счетчика аналогична средству подготовки к отображению, представленному в листинге 11.14. Оба средства подготовки к отображению формируют код JavaScript, который выполняет вызов Ajax; кроме того, оба средства подготовки к отображению закрепляют этот код JavaScript за полем ввода и кнопками счетчика. Таким образом, после выполнения пользователем действия keyup (отжатия клавиши) в поле ввода счетчика или щелчка на одной из кнопок реализация JSF выполняет вызов Ajax к серверу.

Различие между этими двумя средствами подготовки к отображению состоит в том, в каком месте формируется код JavaScript, который выполняет вызов Ajax. В средстве подготовки к отображению, показанном в листинге 11.14, сценарий формируется непосредственно с использованием встроенных средств JavaScript версии JSF 2.0. А в средстве подготовки к отображению, показанном в листинге 11.15, сценарий формируется на основе клиентского правила поведения, закрепленного тегом f:ajax за компонентом счетчика:

```
public final String getAjaxScript(FacesContext context, UIInput component)
   throws IOException {
    String script = null;
    ClientBehaviorContext behaviorContext
       = ClientBehaviorContext.createClientBehaviorContext(context,
          component, "click", component.getClientId(context), null);
    Map<String,List<ClientBehavior>> behaviors =
       ((UIInput) component).getClientBehaviors();
    if (behaviors.containsKey("click"))
       script = behaviors.get("click").get(0).getScript(behaviorContext);
    return script;
}
```

В приведенном выше коде происходит поиск правила поведения "click", закрепленного за компонентом счетчика. Еще раз отметим — компонент счетчика указывает реализации JSF, что событием Ajax по умолчанию является щелчок, поэтому автор страницы не обязан явно определять событие для тега f:ajax, поскольку реализация JSF связывает правило поведения тега f:ajax с событием щелчка. Безусловно, в данном случае указания, полученные реализацией JSF, не полностью соответствуют действительности, поскольку сценарий правила поведения "click" выполняется не только после щелчков, которые осуществляет пользователь на кнопках счетчика, но и после отжатия пользователем клавиши в поле ввода счетчика. Но об этом должны задумываться мы сами, так как такие нюансы не имеют значения для реализации JSF.

Наконец, средство подготовки к отображению, показанное в листинге 11.15, декодирует все клиентские правила поведения, закрепленные за компонентом счетчика. Клиентские правила поведения, так же как компоненты и средства подготовки к отображению, обладают способностью декодировать параметры запроса. В рассматриваемом примере клиентское правило поведения, закрепленное за счетчиком с помощью тега f:ajax, не выполняет никаких действий при получении указания о проведении декодирования, поэтому можно удалить вызов метода decodeBehaviors() в этом средстве подготовки к отображению, а пример по-прежнему останется работоспособным. Но рекомендуется всегда предусматривать декодирование во всех правилах поведения, закрепленных за конкретным пользовательским компонентом, на тот случай, что в одном или нескольких из этих правил поведения потребуется получение той или иной информации из запроса при выполнении вызова к серверу.

Листинг 11.15. Файл spinner-ajax2/src/java/com/corejsf/JSSpinnerRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.text.MessageFormat;
5. import java.util.List;
6. import java.util.Map;
7.
8. import javax.faces.application.ResourceDependencies;
9. import javax.faces.application.ResourceDependency;
10. import javax.faces.component.EditableValueHolder;
11. import javax.faces.component.UIComponent;
12. import javax.faces.component.UIInput;
13. import javax.faces.component.behavior.ClientBehavior;
14. import javax.faces.component.behavior.ClientBehaviorContext;
15. import javax.faces.component.behavior.ClientBehaviorHolder;
16. import javax.faces.context.ExternalContext;
17. import javax.faces.context.FacesContext;
18. import javax.faces.context.ResponseWriter;
19. import javax.faces.convert.ConverterException;
20. import javax.faces.render.FacesRenderer;
21. import javax.faces.render.Renderer;
22.
23. @FacesRenderer(componentFamily="javax.faces.Input",
24.    rendererType="com.corejsf.Spinner")
25. @ResourceDependencies( {
26.    @ResourceDependency(library="javascript", name="spinner.js"),
27.    @ResourceDependency(library="javax.faces", name="jsf.js")
28. })
29. public class JSSpinnerRenderer extends Renderer {
30.    public Object getConvertedValue(FacesContext context, UIComponent component,
```

```
31.          Object submittedValue) throws ConverterException {
32.       return com.corejsf.util.Renderers.getConvertedValue(context, component,
33.          submittedValue);
34.    }
35.
36.    public void encodeBegin(FacesContext context, UIComponent component)
37.          throws IOException {
38.       ResponseWriter writer = context.getResponseWriter();
39.       String clientId = component.getClientId(context);
40.       String formId = com.corejsf.util.Renderers.getFormId(context, component);
41.       UIInput spinner = (UIInput) component;
42.       String ajaxScript = MessageFormat.format(
43.          "if(document.forms[''{0}''][''{1}''].value != '''') {2};",
44.          formId, clientId, getChangeScript(context, spinner));
45.
46.       String min = component.getAttributes().get("minimum").toString();
47.       String max = component.getAttributes().get("maximum").toString();
48.       String size = component.getAttributes().get("size").toString();
49.
50.       writer.startElement("input", spinner);
51.       writer.writeAttribute("type", "text", null);
52.       writer.writeAttribute("name", clientId , null);
53.       writer.writeAttribute("id", clientId, null);
54.       writer.writeAttribute("value", spinner.getValue().toString(), "value");
55.       if (size != null) writer.writeAttribute("size", size , null);
56.
57.       writer.writeAttribute("onkeyup", ajaxScript, null);
58.       writer.endElement("input");
59.
60.       writer.startElement("script", spinner);
61.       writer.writeAttribute("language", "JavaScript", null);
62.       if (min != null) {
63.          writer.write(MessageFormat.format(
64.             "document.forms[''{0}''][''{1}''].min = {2};",
65.             formId, clientId, min));
66.       }
67.       if (max != null) {
68.          writer.write(MessageFormat.format(
69.             "document.forms[''{0}''][''{1}''].max = {2};",
70.             formId, clientId, max));
71.       }
72.       writer.endElement("script");
73.
74.       String spinScript = MessageFormat.format(
75.          "com.corejsf.spinner.spin(document.forms[''{0}''][''{1}''], -1);",
76.          formId, clientId);
77.
78.       writer.startElement("input", spinner);
79.       writer.writeAttribute("type", "button", null);
80.       writer.writeAttribute("value", "<", null);
81.       writer.writeAttribute("onclick", spinScript + ajaxScript, null);
82.       writer.endElement("input");
83.
84.       spinScript = MessageFormat.format(
85.          "com.corejsf.spinner.spin(document.forms[''{0}''][''{1}''], 1);",
86.          formId, clientId);
87.
88.       writer.startElement("input", spinner);
89.       writer.writeAttribute("type", "button", null);
90.       writer.writeAttribute("value", ">", null);
91.       writer.writeAttribute("onclick", spinScript + ajaxScript, null);
92.       writer.endElement("input");
```

```
93.    }
94.
95.    public void decode(FacesContext context, UIComponent component) {
96.       EditableValueHolder spinner = (EditableValueHolder) component;
97.       Map<String, String> requestMap
98.          = context.getExternalContext().getRequestParameterMap();
99.       String clientId = component.getClientId(context);
100.      spinner.setSubmittedValue((String) requestMap.get(clientId));
101.      spinner.setValid(true);
102.
103.       decodeBehaviors(context, component);
104.   }
105.
106.   public final String getChangeScript(FacesContext context, UIInput component)
107.         throws IOException {
108.   String script = null;
109.      ClientBehaviorContext behaviorContext =
110.         ClientBehaviorContext.createClientBehaviorContext(context,
111.            component, "click", component.getClientId(context), null);
112.
113.         Map<String,List<ClientBehavior>> behaviors
114.            = ((UIInput)component).getClientBehaviors();
115.         if (behaviors.containsKey("click") )
116.            script = behaviors.get("click").get(0).getScript(behaviorContext);
117.         return script;
118.   }
119.
120.   public final void decodeBehaviors(FacesContext context, UIComponent component) {
121.      if (!(component instanceof ClientBehaviorHolder)) return;
122.      ClientBehaviorHolder holder = (ClientBehaviorHolder)component;
123.      Map<String, List<ClientBehavior>> behaviors = holder.getClientBehaviors();
124.      if (behaviors.isEmpty()) return;
125.
126.       ExternalContext external = context.getExternalContext();
127.       Map<String, String> params = external.getRequestParameterMap();
128.       String behaviorEvent = params.get("javax.faces.behavior.event");
129.
130.       if (behaviorEvent != null) {
131.          List<ClientBehavior> behaviorsForEvent = behaviors.get(behaviorEvent);
132.
133.             if (behaviors.size() > 0) {
134.                String behaviorSource = params.get("javax.faces.source");
135.                String clientId = component.getClientId();
136.                if (null != behaviorSource && behaviorSource.equals(clientId))
137.                   for (ClientBehavior behavior: behaviorsForEvent)
138.                      behavior.decode(context, component);
139.             }
140.       }
141.   }
142.}
```

Резюме

В данной главе было показано, как формировать код пользовательских компонентов JSF. Задача реализации пользовательских компонентов является сложной, поэтому целесообразно в первую очередь рассматривать возможность создания составных компонентов. Но иногда единственная возможность состоит лишь в применении пользовательских компонентов, особенно если речь идет о компонентах с широким

набором правил поведения. Дополнительные примеры по этой теме будут приведены в главе 13. В следующей главе будет показано, как обращаться к внешним службам, таким как базы данных и электронная почта, из приложения JSF.

Внешние службы

В этой главе...

- Доступ к базе данных с помощью JDBC
- Настройка источника данных
- Использование спецификации JPA
- Аутентификация и авторизация, управляемые контейнером
- Отправка почты
- Использование веб-служб

Глава 12

В этой главе представлены сведения о том, как обеспечить доступ к внешним службам из приложения JSF, как обращаться к базам данных, отправлять электронную почту и подключаться к веб-службам.

Доступ к базе данных с помощью JDBC

В следующих разделах приведены краткие сведения об API-интерфейсе JDBC (Java Database Connectivity), позволяющие освежить в памяти то, что известно об этой спецификации. Мы предполагаем, что читатель знаком с основными командами SQL (Structured Query Language). Более подробное введение в эту тему можно найти в книге Кея Хорстмена (Cay Horstmann) и Гэри Корнелла (Gary Cornell), *Core Java*™, 8th ed., Santa Clara, CA: Sun Microsystems Press/Prentice Hall, 2008, том 2, глава 4.

Передача инструкций SQL на выполнение

Чтобы выполнять инструкции SQL к базе данных, требуется объект соединения. Предусмотрены различные методы получения соединения. Наиболее удобный из них состоит в использовании источника данных.

```
DataSource source = . . .;
Connection conn = source.getConnection();
```

В разделе "Доступ к управляемому контейнером ресурсу" на стр. 428 описано, как получить источник данных в контейнерах Tomcat и GlassFish. На данный момент предполагается, что источник данных уже настроен должным образом для подключения к предпочтительной базе данных.

Сразу после получения объекта `Connection` можно создать объект `Statement`, применимый для отправки инструкций SQL в базу данных. Для выполнения инструкций SQL, которые обновляют базу данных, служит метод `executeUpdate`, а для выполнения запросов, возвращающих результирующий набор, — метод `executeQuery`:

```
Statement stat = conn.createStatement();
stat.executeUpdate("INSERT INTO Credentials VALUES ('troosevelt', 'jabberwock')");
ResultSet result = stat.executeQuery("SELECT * FROM Credentials");
```

Класс ResultSet имеет необычный итеративный протокол. Прежде всего вызывается метод next для перемещения курсора к первой строке. (Метод next возвращает false, если доступных строк больше нет.) Для возврата значения поля в виде строки вызывается метод getString. Например:

```
while (result.next()) {
   username = result.getString("username");
   password = result.getString("passwd");
   ...
}
```

На заметку! На протяжении всей данной главы в качестве имени столбца используется passwd, поскольку password — зарезервированное слово SQL, а некоторые базы данных (такие как PostgreSQL) не допускают его применения для обозначения имени столбца.

После завершения работы с базой данных следует обязательно закрыть соединение. Для того чтобы соединение закрывалось при всех обстоятельствах, даже при возникновении исключительной ситуации, заключите код запроса в блок try/finally, как в следующем примере:

```
Connection conn = source.getConnection();
try {
   ...
}
finally {
   conn.close();
}
```

Безусловно, описанию API-интерфейса JDBC можно отвести много больше места, но даже приведенных выше основных сведений достаточно для начала работы с ним.

На заметку! В настоящем разделе будет показано, как выполнять инструкции SQL из конкретного веб-приложения. Эти рекомендации превосходно подходят для простых приложений, имеющих умеренные требования к памяти. Что же касается сложных приложений, то может потребоваться использовать технологию объектно-реляционного сопоставления, такую как JPA (Java Persistence API), которая рассматривается ниже в главе.

Управление соединениями

Одной из наиболее беспокоящих проблем из тех, которые приходится решать разработчику веб-приложений, является управление соединениями с базой данных. При этом приходится сталкиваться с двумя конфликтующими задачами. Прежде всего, для открытия соединения с базой данных может требоваться много времени. До завершения процессов соединения, аутентификации и приобретения ресурсов может пройти даже несколько секунд. Таким образом, невозможно просто открывать новое соединение для каждого запроса страницы.

С другой стороны, отсутствует возможность поддерживать открытыми значительное число соединений с базой данных. Соединения расходуют ресурсы и в клиентской программе, и на сервере базы данных. Как правило, база данных налагает ограничение на максимальное число поддерживаемых ею параллельных соединений. Таким образом, в приложении нельзя просто открывать соединение после входа в систему каждого пользователя и оставлять его открытым до тех пор, пока пользователь не выйдет из системы. В конце концов, любой пользователь может покинуть рабочее место и так и не вернуться к нему вовремя.

Одним из механизмов, широко применяемых для решения этих задач, является объединение соединений с базой данных в пул. В пуле соединений хранятся соединения с базой данных, которые уже открыты. Прикладные программы получают соединения из пула. После того как необходимость в использовании того или иного соединения отпадает, оно возвращается в пул, но не закрывается. Таким образом, благодаря применению пула запаздывание при установлении соединений с базой данных сводится к минимуму.

Реализация пула соединений базы данных непроста, и ответственность за решение этой задачи, безусловно, не должна возлагаться на прикладного программиста. Начиная с версии 2.0 API-интерфейс JDBC поддерживает создание пула, для чего предусмотрен удобный и прозрачный способ. Объект `Connection`, полученный приложением из пула, фактически инструментирован так, что метод `close` этого объекта просто возвращает его в пул. Функции по установке пула и предоставлению для приложения источника данных, метод `getConnection` которого выдает соединения из пула, возлагаются на сервер приложений.

В серверах приложений разных версий применяются различные способы настройки пула соединений с базой данных. Регламентация всех нюансов реализации этих функций не предусмотрена ни в одном стандарте Java, поскольку в спецификации JDBC нет ни одного слова, касающегося этой темы. В следующем разделе описано, как настраивать платформы GlassFish и Tomcat для создания пула соединений. Основные принципы являются такими же, как и применительно к другим серверам приложений, но, безусловно, подробности реализации могут существенно различаться.

Тем не менее для успешной поддержки пула остается необходимым, чтобы каждый объект соединения был закрыт по окончании его использования. В противном случае ресурсы пула будут исчерпаны и потребуется открыть новые физические соединения с базой данных. Закрытие соединений должным образом является темой следующего раздела.

Устранение утечек ресурсов соединения

Рассмотрим приведенную ниже простую последовательность инструкций.

```
DataSource source = . . .;
Connection conn = source.getConnection();
Statement stat = conn.createStatement();
String command = "INSERT INTO Credentials VALUES ('troosevelt', 'jabberwock')";
stat.executeUpdate(command);
conn.close();
```

Этот код внешне кажется вполне безошибочным: мы открываем соединение, выполняем команду и сразу же закрываем соединение. Но в нем есть существенный недостаток. Если в одном из вызовов методов активизируется исключительная ситуация, то вызов метода `close` так и не произойдет!

В таком случае раздраженный пользователь, не понимая причин задержки, может повторно отправить запрос много раз, допуская все бо́льшую утечку ресурсов объекта соединения с каждым щелчком.

Чтобы такая проблема не возникала, следует всегда помещать вызов функции `close` в блок `finally`:

```
DataSource source = . . .;
Connection conn = source.getConnection();
try {
   Statement stat = conn.createStatement();
```

```
   String command = "INSERT INTO Credentials VALUES ('troosevelt', 'jabberwock')";
   stat.executeUpdate(command);
}
finally {
   conn.close();
}
```

Это простое правило полностью решает задачу утечки ресурсов соединений. Данное правило становится наиболее эффективным, если содержимое конструкции try/finally не комбинируется с каким-либо другим кодом обработки исключений. В частности, не следует пытаться перехватывать исключение SQLException в том же блоке try:

```
// Предостерегаем против применения подобного кода
Connection conn = null;
try {
   conn = source.getConnection();
   Statement stat = conn.createStatement();
   String command = "INSERT INTO Credentials VALUES ('troosevelt', 'jabberwock')";
   stat.executeUpdate(command);
}
catch (SQLException) {
   // Регистрация ошибки
}
finally {
   conn.close(); // ОШИБКА
}
```

В приведенном выше коде имеются две малозаметные ошибки. Прежде всего, если вызов метода getConnection приведет к активизации исключительной ситуации, то значение conn все еще останется равным null, поэтому нельзя будет вызвать метод close. Кроме того, вызов метода close может также привести к активизации исключения SQLException. Пытаясь исправить эту ситуацию, программист может применять в предложении finally все более и более сложный код, но результатом станет лишь дополнительная путаница. Вместо этого следует использовать два отдельных блока try:

```
// Рекомендуется использовать отдельные блоки try
try {
   Connection conn = source.getConnection();
   try {
      Statement stat = conn.createStatement();
      String command = "INSERT INTO Credentials VALUES ('troosevelt', 'jabberwock')";
      stat.executeUpdate(command);
   }
   finally {
      conn.close();
   }
}
catch (SQLException) {
   // Регистрация ошибки
}
```

Внутренний блок try гарантирует, что соединение будет закрыто, а внешний блок try гарантирует, что сведения об исключительной ситуации будут зарегистрированы в журнале.

На заметку! Безусловно, можно также обозначить применяемый метод параметром throws SQLException и возложить задачу применения внешнего блока try на вызывающую программу. Это решение часто становится самым лучшим.

Использование подготовленных инструкций

Один из широко применяемых способов оптимизации программ JDBC состоит в использовании класса `PreparedStatement` для создания подготовленных инструкций. Применение подготовленных инструкций способствует ускорению операций базы данных, если в коде много раз выполняются запросы одного и того же типа. Рассмотрим задачу поиска пользовательских паролей. При этом приходится неоднократно выполнять запросы, имеющие следующую форму:

```
SELECT passwd FROM Credentials WHERE username=...
```

Впервые проводимая передача подготовленной инструкции SQL на выполнение фактически равносильна обращению к базе данных, чтобы в ней была осуществлена предварительная компиляция запроса, т.е. анализ инструкции SQL и вычисление стратегии запроса. Полученные при этом сведения сохраняются вместе с подготовленной инструкцией и повторно используются при каждой следующей передаче на выполнение того же запроса.

Подготовленная инструкция создается с помощью метода `prepareStatement` класса `Connection`. Символ ? используется для обозначения каждого параметра:

```
PreparedStatement stat = conn.prepareStatement(
   "SELECT passwd FROM Credentials WHERE username=?");
```

После того как все будет готово для выполнения подготовленной инструкции, нужно прежде всего задать значения параметров:

```
stat.setString(1, name);
```

(Обратите внимание на то, что значение индекса 1 указывает на первый параметр.) Затем инструкция выполняется обычным образом:

```
ResultSet result = stat.executeQuery();
```

На первый взгляд кажется, будто подготовленные инструкции не предоставляют существенных преимуществ в веб-приложении. В конце концов, ведь соединение закрывается после завершения каждого пользовательского запроса. Подготовленная инструкция связана с соединением с базой данных, и вся выполненная работа удаляется после закрытия физического соединения с базой данных.

Но если физические соединения с базой данных сохраняются в пуле, то весьма велика вероятность того, что подготовленная инструкция все еще останется пригодной для использования после повторного получения соединения. Во многих реализациях пула соединений предусмотрено кэширование подготовленных инструкций.

После вызова метода `prepareStatement` в первую очередь происходит поиск рассматриваемой инструкции в кэше пула с использованием строки запроса в качестве ключа. Если подготовленная инструкция будет найдена, то она будет применена повторно. В противном случае создается новая подготовленная инструкция и добавляется к кэшу.

Внимание! Возможность сохранить объект `PreparedStatement` и повторно использовать его за пределами единственной области действия запроса отсутствует. После закрытия соединения, полученного из пула, все связанные с ним объекты `PreparedStatement` также возвращаются в пул. Таким образом, программист не обязан задумываться над тем, существуют ли объекты `PreparedStatement`, выходящие за рамки текущего запроса. Вместо этого он должен продолжать вызывать метод `prepareStatement` применительно к одной и той же повторно выполняемой строке запроса, вполне обоснованно рассчитывая на получение кэшированного объекта инструкции.

Все эти действия остаются прозрачными для прикладного программиста. Программист лишь запрашивает объекты PreparedStatement в надежде на то, что хотя бы время от времени в пуле будет обнаруживаться существующий объект для данного запроса.

На заметку! Даже если разработчика не волнуют проблемы производительности, есть еще одно весомое основание для использования подготовленных инструкций: защита против атак по принципу внедрения кода SQL. Когда запрос формируется путем конкатенации кода SQL и введенных пользователем данных, злонамеренный пользователь может ввести в составе данных код SQL, который изменит смысл запроса. Если же используются подготовленные инструкции, то пользовательский ввод никогда не интерпретируется как код SQL.

Транзакции

Предусмотрена возможность сгруппировать набор инструкций для формирования транзакции. После успешного выполнения транзакции производится ее фиксация. С другой стороны, если при выполнении одной из инструкций произошла ошибка, может быть произведен откат к тому состоянию, как если бы ни одна из инструкций не была выполнена.

Группирование инструкций в транзакции осуществляется по двум причинам: обеспечение целостности базы данных и параллельного доступа. В качестве примера предположим, что необходимо передать деньги с одного банковского счета на другой. При этом необходимо одновременно списать денежные средства с одного счета и зачислить на другой. Если произойдет аварийное завершение работы системы после списания денег с первого счета, но перед зачислением на второй, то операция списания должна быть отменена. Наряду с этим, если одновременно предпринимаются две попытки доступа к счету, то операции доступа должны быть организованы последовательно.

По умолчанию соединение с базой данных находится в режиме автоматической фиксации, поэтому фиксация каждой инструкции SQL в базе данных происходит сразу после ее выполнения. После фиксации инструкции ее откат становится невозможным. При использовании транзакций это значение по умолчанию должно быть отменено:

```
conn.setAutoCommit(false);
```

После этого все запросы и обновления выполняются обычным образом. Если все инструкции выполнены без ошибок, следует вызвать метод фиксации commit:

```
conn.commit();
```

Если же произошла ошибка, необходимо применить следующий вызов:

```
conn.rollback();
```

При этом автоматически отменяются все инструкции, выполненные со времени последней фиксации. Добиться того, чтобы вызов метода отката rollback происходил после каждой активизации исключительной ситуации, — довольно трудоемкая задача. Но при использовании следующей схемы кода решение этой задачи гарантируется:

```
conn.setAutoCommit(false);
boolean committed = false;
try {
   database operations
   conn.commit();
   committed = true;
} finally {
   if (!committed) conn.rollback();
}
```

Использование базы данных Derby

Чтобы читатель мог приступить к изучению программ в этой главе, рекомендуем воспользоваться базой данных Apache Derby, которая входит в поставку GlassFish и некоторых версий JDK. (Если у читателя еще нет базы данных Derby, то ему следует загрузить Apache Derby с адреса `http://db.apache.org/derby`.)

На заметку! В корпорации Oracle принято именовать версию Apache Derby, которая входит в состав JDK, как JavaDB. Во избежание путаницы эта версия в настоящей главе именуется Derby.

Прежде чем появится возможность подключиться к серверу базы данных, он должен быть запущен. Подробные сведения о том, как выполнить эту операцию, зависят от конкретной базы данных.

Применительно к базе данных Derby выполните следующие действия.

1. Если используется платформа GlassFish, выполните следующее:

   ```
   glassfish/bin/asadmin start-database
   ```

   В противном случае определите местонахождение файла `derbyrun.jar`. В некоторых версиях JDK он находится в каталоге `jdk/db/lib`, а в других — в отдельном инсталляционном каталоге JavaDB. Мы обозначаем каталог, содержащий файл `lib/derbyrun.jar`, как `derby`. Выполните следующую команду:

   ```
   java -jar derby/lib/derbyrun.jar server start
   ```

2. Еще раз тщательно проверьте, правильно ли работает база данных. Создайте файл `ij.properties`, который содержит такие строки:

   ```
   ij.driver=org.apache.derby.jdbc.ClientDriver
   ij.protocol=jdbc:derby://localhost:1527/
   ij.database=COREJSF;create=true
   ```

   Вызовите на выполнение интерактивный инструментарий поддержки сценариев Derby (называемый `ij`), применив команду

   ```
   java -jar derby/lib/derbyrun.jar ij -p ij.properties
   ```

   Теперь можно приступить к выполнению команд SQL:

   ```
   CREATE TABLE Greetings (Message VARCHAR(20));
   INSERT INTO Greetings VALUES ('Hello, World!');
   SELECT * FROM Greetings;
   DROP TABLE Greetings;
   ```

   Обратите внимание на то, что каждая команда должна оканчиваться точкой с запятой. Для выхода из программы введите

   ```
   EXIT;
   ```

3. Завершив работу с базой данных, остановите сервер с помощью команды

   ```
   glassfish/bin/asadmin stop-database
   ```

   или, если не используется платформа GlassFish, команды

   ```
   java -jar derby/lib/derbyrun.jar server shutdown
   ```

На заметку! Если применяется инструментальное средство `asadmin` платформы GlassFish, то базы данных располагаются в каталоге `glassfish/databases`. Но если оказалось, что в каталоге, из которого вызвана на выполнение команда `asadmin`, имеется файл `derby.log`, то базы данных созданы в этом каталоге. Если платформа GlassFish не используется, то базы данных располагаются в каталоге, из которого был запущен сервер Derby.

Если какая-либо база данных больше не требуется, остановите сервер базы данных и удалите каталог с базой данных.

Если читатель использует другую базу данных, то должен изучить ее документацию, чтобы узнать, как запускать и останавливать сервер этой базы данных, как подключиться к ней и выполнить команды SQL.

Настройка источника данных

В следующих разделах показано, как настроить источник данных в сервере приложений, таком как GlassFish и Tomcat, и как обратиться к источнику данных из веб-приложения.

Доступ к ресурсу, управляемому контейнером

В приложении доступ к ресурсу, такому как источник данных, осуществляется с помощью символического имени (например, `jdbc/mydb`). Предусмотрены два способа получения ресурса по имени. Наиболее удобный из них состоит во внедрении ресурса. Для этого необходимо объявить поле в управляемом бине, предварив объявление аннотацией:

```
@Resource(name="jdbc/mydb") private DataSource source;
```

После того как сервер приложений загрузит управляемый бин, произойдет автоматическая инициализация этого поля.

В табл. 12.1 перечислены аннотации, которые можно использовать для внедрения ресурсов в управляемый бин JSF.

Внимание! Ресурсы, приведенные в табл. 12.1, не являются сериализуемыми. Но при использовании спецификации CDI это не приводит к возникновению каких-либо проблем. Контейнер будет удалять внедренные ресурсы перед сериализацией бина и повторно внедрять их после десериализации. Но если используются управляемые бины JSF и разрешена кластеризация, то не следует внедрять эти ресурсы в бины с областью действия сеанса или приложения.

Таблица 12.1. Аннотации для внедрения ресурсов

Аннотация	Тип ресурса
`@Resource`, `@Resources`	Произвольный ресурс JNDI
`@WebServiceRef`, `@WebServiceRefs`	Порт веб-службы
`@EJB`, `@EJBs`	Бин сеанса EJB
`@PersistenceContext`, `@PersistenceContexts`	Диспетчер сохраняемых сущностей
`@PersistenceUnit`, `@PersistenceUnits`	Фабрика диспетчера сохраняемых сущностей

Настройка ресурса базы данных в технологии GlassFish

Платформа GlassFish имеет удобный веб-интерфейс администрирования, который можно использовать для настройки источника данных. Задайте в применяемом браузере адрес `http://localhost:4848` и войдите в систему. (По умолчанию именем пользователя является `admin`, а пароль по умолчанию имеет значение `adminadmin`.)

1. Настройте пул базы данных. Выберите Пулы соединений и установите новый пул. Присвойте имя пулу, выберите тип ресурса (`javax.sql.DataSource`) и выберите применяемый поставщик базы данных (рис. 12.1).

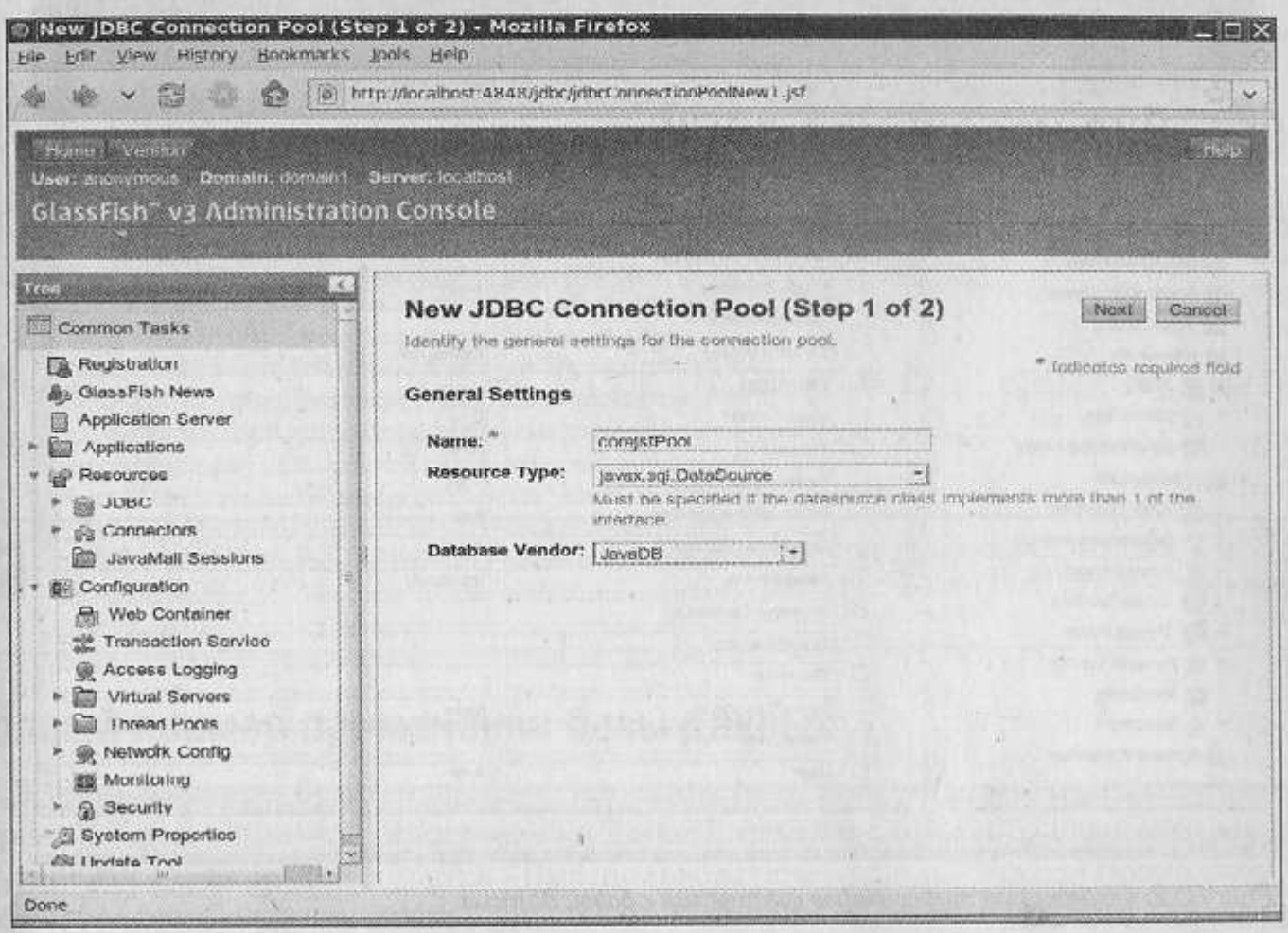


Рис. 12.1. Настройка пула соединений на платформе GlassFish

2. На открывшемся экране определите класс источника данных (`org.apache.derby.jdbc.ClientDataSource`) и параметры соединения с базой данных, такие как имя пользователя и пароль (рис. 12.2).
3. Затем настройте новый источник данных. Присвойте ему имя `jdbc/mydb` и выберите только что установленный пул (рис. 12.3).

На заметку! Если встроенная база данных Derby не используется, то необходимо поместить файл JAR, относящийся к файлу драйвера базы данных, в подкаталог `domains/domain1/lib/ext` конкретной инсталляции GlassFish.

Настройка ресурса базы данных на платформе Tomcat

В настоящем разделе показаны шаги настройки пула ресурсов базы данных для работы в контейнере Tomcat 6.

1. Скопируйте файл JAR с драйвером (`derbyclient.jar`, если применяется база данных Derby) в каталог `tomcat/lib`.
2. Добавьте файл `META-INF/context.xml`, который определяет параметры соединения. Ниже приведен пример для Derby.

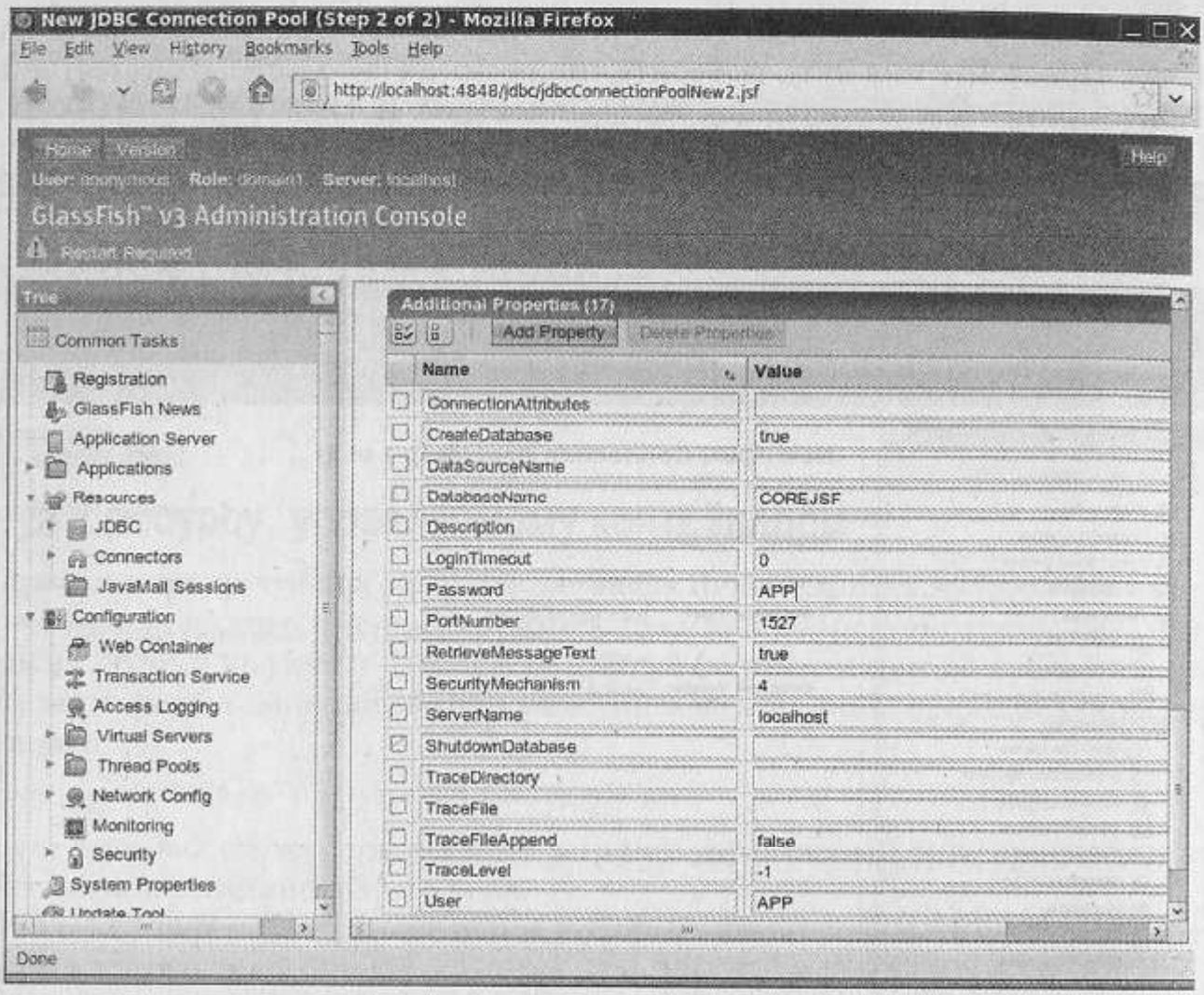


Рис. 12.2. Определение параметров соединения с базой данных

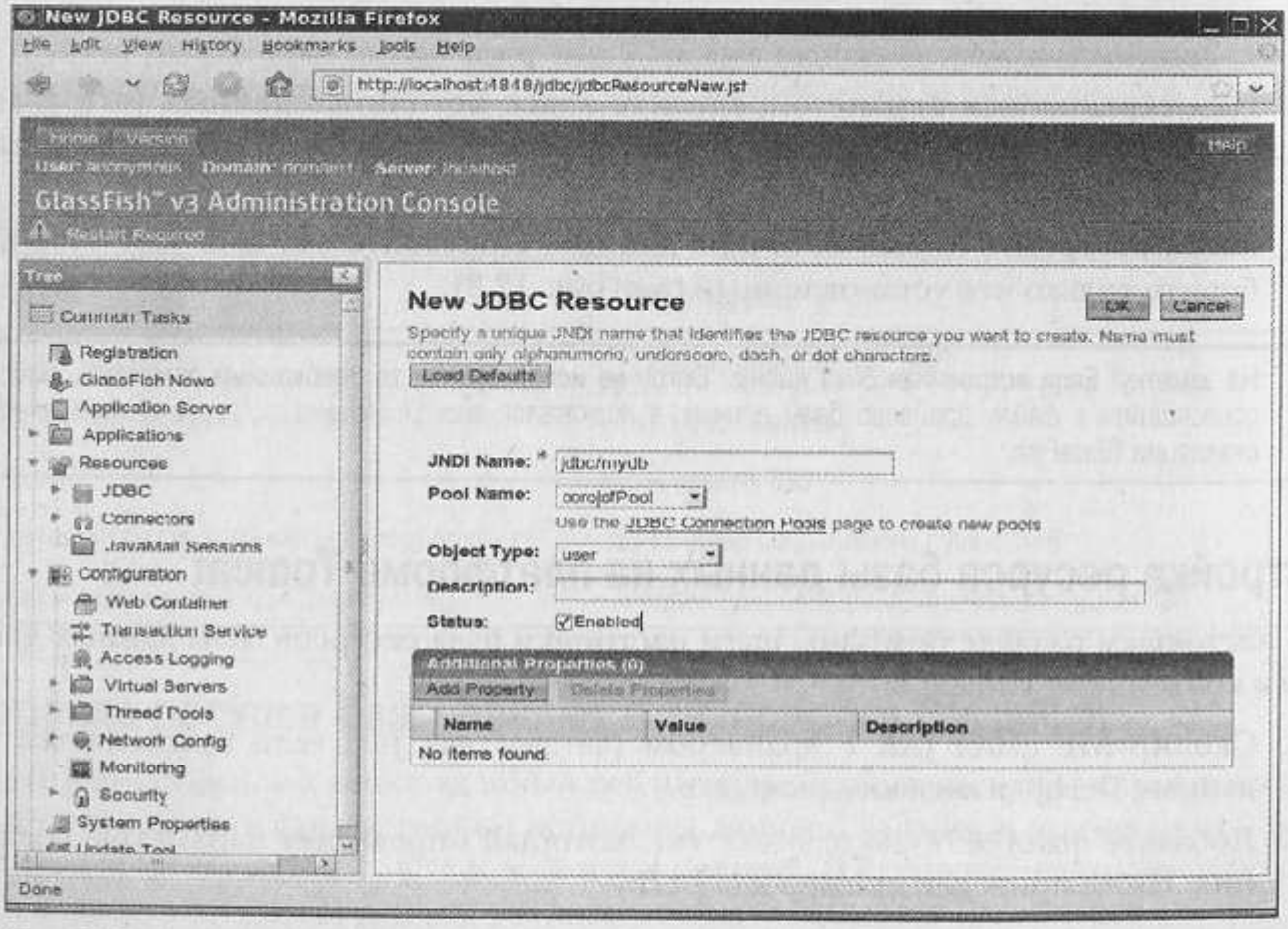


Рис. 12.3. Настройка источника данных

```
<Context>
   <Resource
      name="jdbc/mydb"
      auth="Container"
      type="javax.sql.DataSource"
      username="APP"
      password="APP"
      driverClassName="org.apache.derby.jdbc.ClientDriver"
      url="jdbc:derby://localhost:1527/COREJSF;create=true"/>
</Context>
```

В этом примере выполняется настройка ресурса для конкретного приложения.

3. Добавьте следующую запись в применяемый файл `web.xml`:

```
<resource-ref>
   <res-ref-name>jdbc/mydb</res-ref-name>
   <res-type>javax.sql.DataSource</res-t ype>
   <res-auth>Container</res-auth>
</resource-ref>
```

Совет. Подробные рекомендации по настройке для многих широко применяемых баз данных можно найти по адресу `http://jakarta.apache.org/tomcat/tomcat-6.0-doc/jndi-datasource-examples-howto.html`.

Полный пример применения базы данных

В этом примере будет показано, как проверять комбинации "имя пользователя–пароль". Как и в примере программы в главе 1, начнем с простого окна входа в систему (рис. 12.4). Если комбинация "имя пользователя–пароль" задана правильно, отображается окно с приветствием (рис. 12.5). В противном случае пользователю предлагается сделать еще одну попытку (рис. 12.6). Наконец, если произошла ошибка базы данных, отображается окно с сообщением об ошибке (рис. 12.7).

Таким образом, предусмотрены четыре страницы JSF, которые показаны в листингах 12.1–12.4. В листинге 12.5 приведен код бина `UserBean`.

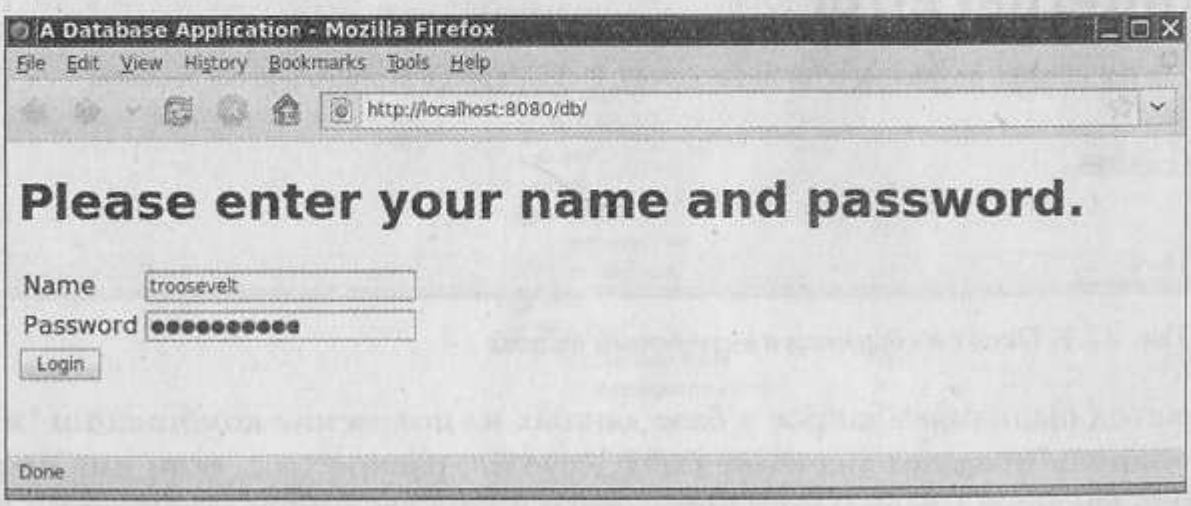


Рис. 12.4. Окно входа в систему

В этом простом примере код доступа к базе данных приведен непосредственно в классе `UserBean`. Код доступа к базе данных помещен в отдельный метод:

```
public void doLogin() throws SQLException
```

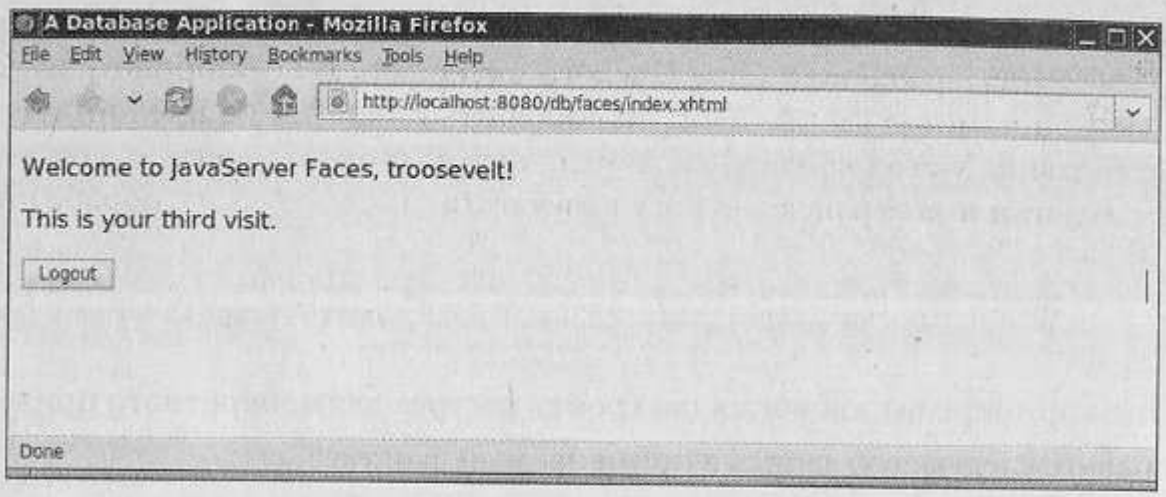


Рис. 12.5. Окно приветствия

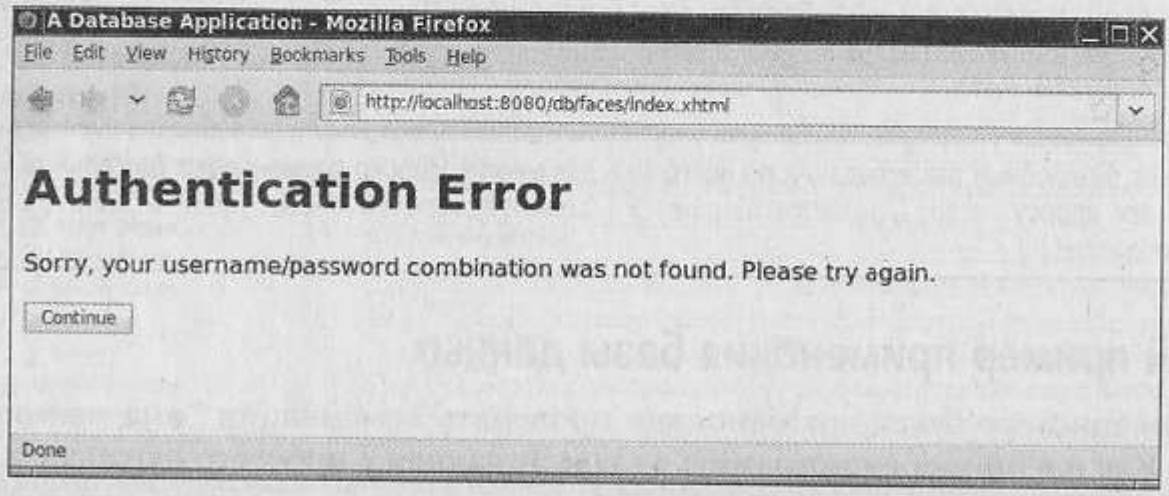


Рис. 12.6. Окно с сообщением об ошибке аутентификации

Рис. 12.7. Окно с сообщением о внутренней ошибке

Этот метод выполняет запрос к базе данных на получение комбинации "имя пользователя–пароль" и задает значение поля `loggedIn`, равное `true`, если имя пользователя и пароль введены правильно. После этого метод увеличивает значение счетчика входа в систему, который отображается на начальной странице.

Кнопка на странице `index.xhtml` ссылается на метод входа в систему `login` бина `user`. Данный метод вызывает метод `doLogin` и возвращает строку результата для обработчика навигации. В методе входа в систему обрабатываются также исключительные ситуации, о которых сообщает метод `doLogin`.

Предполагается, что основное назначение метода doLogin состоит в работе с базой данных, а не в поддержке пользовательского интерфейса. Если происходит исключительная ситуация, метод doLogin должен сообщить о ней, но не предпринимать дальнейших действий. Метод входа в систему, с другой стороны, регистрирует исключительные ситуации и возвращает строку результата "internalError" обработчику навигации.

```
public String login() {
   try {
      doLogin();
   }
   catch (SQLException ex) {
      logger.log(Level.SEVERE, "login failed", ex);
      return "internalError";
   }
   if (loggedIn)
      return "loginSuccess";
   else
      return "loginFailure";
}
```

До начала выполнения этого примера читатель должен запустить применяемую базу данных, создать таблицу Credentials и добавить одну или несколько записей с именем пользователя и паролем:

```
CREATE TABLE Credentials (username VARCHAR(20), passwd VARCHAR(20),
   logincount INTEGER)
INSERT INTO Credentials VALUES ('troosevelt', 'jabberwock', 0)
INSERT INTO Credentials VALUES ('tjefferson', 'mockturtle', 0)
```

Затем можно развернуть и проверить это приложение.

На рис. 12.8 показана структура каталогов для рассматриваемого приложения, а на рис. 12.9 приведена карта навигации. Упомянутые выше файлы приложения находятся в листингах 12.1–12.5.

Рис. 12.8. Структура каталогов приложения базы данных

На заметку! В процессе настройки конфигурации базы данных многое может пойти неправильно. Если в приложении возникла ошибка, ознакомьтесь с записями журнала. На платформе GlassFish журналом по умолчанию является domains/domain1/logs/server.log. На платформе Tomcat таковым служит logs/catalina.out.

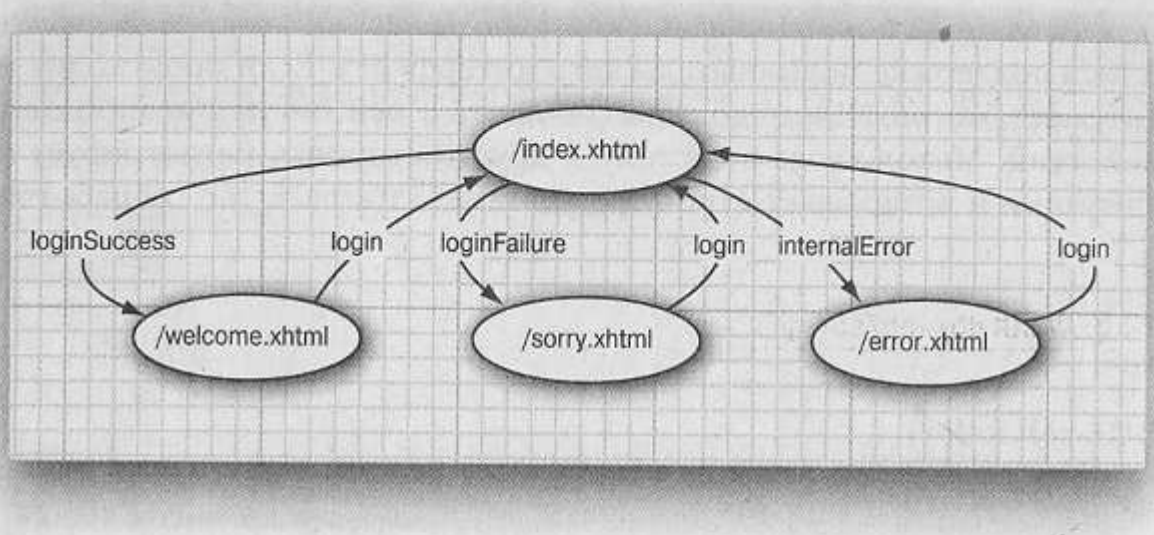


Рис. 12.9. Карта навигации приложения базы данных

Листинг 12.1. Файл db/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <h1>#{msgs.enterNameAndPassword}</h1>
         <h:panelGrid columns="2">
            #{msgs.name}
            <h:inputText value="#{user.name}"/>

            #{msgs.password}
            <h:inputSecret value="#{user.password}"/>
         </h:panelGrid>
         <h:commandButton value="#{msgs.login}" action="#{user.login}"/>
      </h:form>
   </h:body>
</html>
```

Листинг 12.2. Файл db/web/welcome.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <p>#{msgs.welcome} <h:outputText value="#{user.name}"/>!</p>
         <p>
            <h:outputFormat value="#{msgs.visits}">
               <f:param value="#{user.count}"/>
            </h:outputFormat>
```

```
16.          </p>
17.
18.          <p><h:commandButton value="#{msgs.logout}" action="#{user.logout}"/>
19.          </p>
20.       </h:form>
21.    </h:body>
22. </html>
```

Листинг 12.3. Файл db/web/sorry.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
6.    <h:head>
7.       <title>#{msgs.title}</title>
8.    </h:head>
9.    <h:body>
10.      <h:form>
11.         <h1>#{msgs.authError}</h1>
12.         <p>#{msgs.authError_detail}</p>
13.         <p><h:commandButton value="#{msgs.continue}" action="login"/></p>
14.      </h:form>
15.   </h:body>
16. </html>
```

Листинг 12.4. Файл db/web/error.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
6.    <h:head>
7.       <title>#{msgs.title}</title>
8.    </h:head>
9.    <h:body>
10.      <h:form>
11.         <h1>#{msgs.internalError}</h1>
12.         <p>#{msgs.internalError_detail}</p>
13.         <p><h:commandButton value="#{msgs.continue}" action="index"/></p>
14.      </h:form>
15.   </h:body>
16. </html>
```

Листинг 12.5. Файл db/src/java/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.sql.Connection;
5. import java.sql.PreparedStatement;
6. import java.sql.ResultSet;
7. import java.sql.SQLException;
8. import java.util.logging.Level;
9. import java.util.logging.Logger;
10.
11. import javax.annotation.Resource;
12. import javax.inject.Named;
```

```
13.    // или import javax.faces.bean.ManagedBean;
14. import javax.enterprise.context.SessionScoped;
15.    // или import javax.faces.bean.SessionScoped;
16. import javax.sql.DataSource;
17.
18. @Named("user") // или @ManagedBean(name="user")
19. @SessionScoped
20. public class UserBean implements Serializable {
21.    private String name;
22.    private String password;
23.    private int count;
24.    private boolean loggedIn;
25.    private Logger logger = Logger.getLogger("com.corejsf");
26.
27.    @Resource(name="jdbc/mydb")
28.    private DataSource ds;
29.
30.    /*
31.    Если этот веб-контейнер не поддерживает внедрение ресурсов, добавьте конструктор:
32.    public UserBean()
33.    {
34.       try {
35.          Context ctx = new InitialContext();
36.          ds = (DataSource) ctx.lookup("java:comp/env/jdbc/mydb");
37.       } catch (NamingException ex) {
38.          logger.log(Level.SEVERE, "DataSource lookup failed", ex);
39.       }
40.    }
41.    */
42.
43.    public String getName() { return name; }
44.    public void setName(String newValue) { name = newValue; }
45.
46.    public String getPassword() { return password; }
47.    public void setPassword(String newValue) { password = newValue; }
48.
49.    public int getCount() { return count; }
50.
51.    public String login() {
52.       try {
53.          doLogin();
54.       }
55.       catch (SQLException ex) {
56.          logger.log(Level.SEVERE, "login failed", ex);
57.          return "internalError";
58.       }
59.       if (loggedIn)
60.          return "loginSuccess";
61.        else
62.           return "loginFailure";
63.    }
64.
65.    public String logout() {
66.       loggedIn = false;
67.       return "login";
68.    }
69.
70.    public void doLogin() throws SQLException {
71.       if (ds == null) throw new SQLException("No data source");
72.       Connection conn = ds.getConnection();
73.       if (conn == null) throw new SQLException("No connection");
74.
```

```
75.          try {
76.             conn.setAutoCommit(false);
77.             boolean committed = false;
78.             try
79.             {
80.                PreparedStatement passwordQuery = conn.prepareStatement(
81.                   "SELECT passwd, logincount from Credentials WHERE username = ?");
82.                passwordQuery.setString(1, name);
83.
84.                ResultSet result = passwordQuery.executeQuery();
85.
86.                if (!result.next()) return;
87.                String storedPassword = result.getString("passwd");
88.                loggedIn = password.equals(storedPassword.trim());
89.                count = result.getInt("logincount");
90.
91.                PreparedStatement updateCounterStat = conn.prepareStatement(
92.                   "UPDATE Credentials SET logincount = logincount + 1"
93.                   + " WHERE USERNAME = ?");
94.                updateCounterStat.setString(1, name);
95.                updateCounterStat.executeUpdate();
96.
97.                conn.commit();
98.                committed = true;
99.             } finally {
100.               if (!committed) conn.rollback();
101.            }
102.         }
103.         finally {
104.            conn.close();
105.         }
106.      }
107. }
```

Использование спецификации JPA

В предыдущем разделе было показано, как обращаться к базе данных с помощью JDBC. Но в настоящее время многие программисты предпочитают использовать объектно-реляционные (Object/Relational – O/R) средства отображения, а не обычные команды SQL. Архитектура JPA (Java Persistence Architecture) предоставляет стандартное объектно-реляционное средство отображения для стека технологий Java EE. В следующих разделах будет показано, как обеспечить доступ к базе данных через JPA в приложениях JSF.

Краткое описание архитектуры JPA

Для обмена данными между таблицами базы данных и объектами Java, которыми разработчик манипулирует в своей программе, может применяться объектно-реляционное средство отображения. При этом в конкретной программе никогда не происходит доступ непосредственно к самой базе данных. В архитектуре JPA для обозначения классов, которые должны быть сохранены в базе данных, используются аннотации. (Эти классы называют *сущностями*.) В качестве примера ниже приведен класс `Credentials` с необходимыми аннотациями.

```
@Entity public class Credentials {
   @Id private String username;
```

```
    private String passwd;
    private int loginCount;

    public Credentials() {}
    ...
}
```

Класс может применяться в качестве сущности при условии соблюдения трех требований.

- Класс должен быть обозначен аннотацией `@Entity`.
- Каждый объект должен иметь единственный первичный ключ, обозначенный аннотацией `@Id`.
- Класс должен иметь конструктор по умолчанию.

Для обозначения связей между сущностями применяются дополнительные аннотации. В качестве примера ниже показано, как выразить факт, что каждый объект `Person` имеет связанную с ним сущность `Credentials` и от нуля и больше ассоциированных ролей.

```
@Entity public class Person {
   @OneToOne private Credentials creds;
   @OneToMany private Set<Role> roles;
   ...
}
```

Объектно-реляционное средство отображения переводит эти аннотации во внешние ключи или соединенные таблицы.

Для создания, чтения, обновления и удаления объектов сущностей используется диспетчер сущностей. Следующий вызов добавляет новую сущность к базе данных:

```
EntityManager em = ...; // Описание инициализации объекта em
                        // приведено в следующих разделах
em.persist(entity);
```

Чтобы изменить имеющуюся сущность, необходимо модифицировать объект Java и зафиксировать текущую транзакцию. Изменения сохраняются автоматически. Для удаления сущности необходимо вызвать метод `em.remove(entity)`.

Для чтения данных выполняется запрос на языке JPQL, объектно-ориентированном языке запросов, аналогичном языку SQL. Например, следующий запрос может применяться для поиска объектов `Credentials`, соответствующих заданному имени пользователя:

```
SELECT c FROM Credentials c WHERE c.username = :username
```

Двоеточия указывают на параметры, которые должны быть заданы при выполнении запроса. Ниже показано, как возвратить результаты запроса.

```
Query query = em.createQuery(
   "SELECT c FROM Credentials c WHERE c.username = :username")
   .setParameter("username", name);
@SuppressWarnings("unchecked")
List<Credentials> result = query.getResultList();
```

При этом используется аннотация `@SuppressWarnings`, поскольку метод `getResultList` возвращает в виде списка бесформатные данные типа `List`, а этот список присваивается параметризованному объекту `List<Credentials>`. Но должен быть получен список с соответствующим параметром типа, чтобы можно было получить элементы списка как объекты `Credentials`.

Незаметно для пользователя диспетчер сущностей выполняет запрос SQL, создает объекты Credentials на основе результатов (или находит их в своем кэше) и возвращает объекты в виде списка.

В данной главе дополнительные сведения об аннотациях отображения, диспетчере сущностей или языке JPQL не приведены; см. главы 19 и 21 учебника по Java EE 6 на сайте http://java.sun.com/javaee/6/docs/tutorial/doc, чтобы получить более подробную информацию.

Использование архитектуры JPA в веб-приложении

При использовании архитектуры JPA в веб-приложении приходится сталкиваться с двумя проблемами.

1. Получение диспетчера сущностей.
2. Обработка транзакций.

Как будет показано в следующем разделе, обе эти проблемы становятся намного проще, если используется бин сеанса EJB, но для этого необходимо эксплуатировать полноценный сервер приложений EE, а не просто веб-контейнер.

Чтобы выполнить возврат диспетчера сущностей, необходимо вначале получить фабрику диспетчера сущностей, которую реализация JSF внедрит в управляемый бин. Поместите следующее аннотированное поле в класс управляемого бина:

```
@PersistenceUnit(unitName="default") private EntityManagerFactory emf;
```

Настройка так называемых "единиц сохраняемости" осуществляется в файле jpa/src/java/META-INF/persistence.xml на языке XML, как показано в листинге 12.8. Каждая единица имеет имя (в данном случае default), источник данных и различные параметры конфигурации.

Для того чтобы получить диспетчер сущностей, выполните следующий вызов:

```
EntityManager em = emf.createEntityManager();
```

Завершив работу с диспетчером сущностей, вызовите команду

```
em.close();
```

На заметку! В управляемом бине с заданной областью действия запроса можно внедрить диспетчер сущностей непосредственно (как показано в следующем разделе). Такая операция невыполнима применительно к другим областям действия, поскольку диспетчер сущностей не потокобезопасен. Но сама фабрика диспетчера сущностей потокобезопасна.

Любую работу с диспетчером сущностей следует заключать в транзакцию. Необходимо вначале получить диспетчер транзакций с помощью такого внедрения:

```
@Resource private UserTransaction utx;
```

А затем воспользоваться этой схемой:

```
utx.begin();
em.joinTransaction();
boolean committed = false;
try {
   carry out work with em
   utx.commit();
   committed = true;
} finally {
   if (!committed) utx.rollback();
}
```

В рассматриваемом примере программы выполняется такая же работа, как и в предыдущей программе.

Но при этом для доступа к базе данных используется архитектура JPA, а не язык SQL. В листинге 12.6 показана сущность `Credentials`. Бин `UserBean` находится в листинге 12.7 (обратите внимание на то, что метод `doLogin` изменился). В листинге 12.8 приведен файл `persistence.xml`. В файле WAR файл `persistence.xml` находится в каталоге `WEB-INF/classes/META-INF`, как показано на рис. 12.10.

На заметку! Безусловно, возможно добавить поставщик JPA к платформе Tomcat, но этот процесс является сложным, и мы не будем обсуждать его в настоящей книге. Этот пример должен быть выполнен на сервере GlassFish или другом сервере приложений Java EE.

Рис. 12.10. Структура каталогов приложения демонстрационной версии JPA

Листинг 12.6. Файл `jpa/src/java/com/corejsf/Credentials.java`

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import javax.persistence.Entity;
5. import javax.persistence.Id;
6.
7. @Entity
8. public class Credentials implements Serializable {
9.    @Id
10.   private String username;
11.   private String passwd;
12.   private int loginCount;
13.
14.   public Credentials() {} // Требуется согласно спецификации JPA
15.
16.   public Credentials(String username, String password) {
17.        this.username = username;
18.        this.passwd = password;
19.    }
20.    public String getPasswd() { return passwd; }
21.    public String getUsername() { return username; }
22.    public int incrementLoginCount() { loginCount++; return loginCount; }
23. }
```

Листинг 12.7. Файл jpa/src/java/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.util.List;
5. import java.util.logging.Level;
6. import java.util.logging.Logger;
7.
8. import javax.annotation.Resource;
9. import javax.inject.Named;
10.    // или import javax.faces.bean.ManagedBean;
11. import javax.enterprise.context.SessionScoped;
12.    // или import javax.faces.bean.SessionScoped;
13. import javax.persistence.EntityManager;
14. import javax.persistence.EntityManagerFactory;
15. import javax.persistence.PersistenceUnit;
16. import javax.persistence.Query;
17. import javax.transaction.HeuristicMixedException;
18. import javax.transaction.HeuristicRollbackException;
19. import javax.transaction.NotSupportedException;
20. import javax.transaction.RollbackException;
21. import javax.transaction.SystemException;
22. import javax.transaction.UserTransaction;
23.
24. @Named("user") // или @ManagedBean(name="user")
25. @SessionScoped
26. public class UserBean implements Serializable {
27.    private String name;
28.    private String password;
29.    private int count;
30.    private boolean loggedIn;
31.
32.    @PersistenceUnit(unitName="default")
33.    private EntityManagerFactory emf;
34.
35.    @Resource
36.    private UserTransaction utx;
37.
38.     public String getName() { return name; }
39.     public void setName(String newValue) { name = newValue; }
40.
41.     public String getPassword() { return password; }
42.     public void setPassword(String newValue) { password = newValue; }
43.
44.     public int getCount() { return count; }
45.
46.     public String login() {
47.        try {
48.           doLogin();
49.        } catch (Exception ex) {
50.           Logger.getLogger("com.corejsf").log(Level.SEVERE, "login failed", ex);
51.           return "internalError";
52.        }
53.        if (loggedIn)
54.           return "loginSuccess";
55.        else
56.           return "loginFailure";
57.     }
58.
59.     public String logout() {
60.        loggedIn = false;
```

```
61.         return "login";
62.     }
63.
64.     public void doLogin() throws NotSupportedException, SystemException,
65.           RollbackException, HeuristicMixedException, HeuristicRollbackException {
66.        EntityManager em = emf.createEntityManager();
67.        try {
68.           utx.begin();
69.           em.joinTransaction();
70.           boolean committed = false;
71.           try {
72.              Query query = em.createQuery(
73.                    "SELECT c FROM Credentials c WHERE c.username = :username")
74.                    .setParameter("username", name);
75.              @SuppressWarnings("unchecked")
76.              List<Credentials> result = query.getResultList();
77.
78.                if (result.size() == 1) {
79.                   Credentials c = result.get(0);
80.                   if (c.getPasswd().trim().equals(password)) {
81.                      loggedIn = true;
82.                      count = c.incrementLoginCount();
83.                   }
84.           }
85.           utx.commit();
86.           committed = true;
87.        } finally {
88.           if (!committed) utx.rollback();
89.        }
90.     } finally {
91.        em.close();
92.     }
93.   }
94. }
```

Листинг 12.8. Файл jpa/src/java/META-INF/persistence.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <persistence version="1.0"
3.       xmlns="http://java.sun.com/xml/ns/persistence"
4.       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.       xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
6.          http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
7. <persistence-unit name="default" transaction-type="JTA">
8.     <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
9.     <jta-data-source>jdbc/mydb</jta-data-source>
10.    <exclude-unlisted-classes>false</exclude-unlisted-classes>
11.    <properties>
12.      <property name="eclipselink.ddl-generation" value="create-tables"/>
13.    </properties>
14. </persistence-unit>
15. </persistence>
```

Использование управляемых бинов и бинов сеанса, не поддерживающих состояние

Теперь перейдем к рассмотрению полной архитектуры EJB, в которой управляемые бины JSF взаимодействуют с бинами сеанса, не поддерживающими состояния, — объектами, которые управляются сервером приложений. Они обладают существен-

ным преимуществом: по умолчанию методы бина сеанса автоматически становятся транзакционными. При каждом вызове метода бина сеанса сервер приложений берет на себя обработку соответствующей транзакции. Аналогичным образом, сервер приложений обеспечивает поточную организацию обработки, для чего просто предоставляется пул бинов и в ответ на каждый запрос вызывается отдельный бин. Эти бины называются не поддерживающими состояние, поскольку не должны сохранять состояние от одного запроса к другому. Такая особенность позволяет серверу приложений выбирать любой доступный бин для обработки конкретного запроса или создавать новые по мере необходимости.

На заметку! В настоящем разделе рассматривается упрощенная, "не имеющая интерфейса" версия бинов сеанса, которая впервые введена в версии Java EE 6.

Для обозначения бинов сеанса, не поддерживающих состояние, применяется аннотация `@Stateless`. В процессе работы просто происходит внедрение диспетчера сущностей и его использование, без объявления каких-либо транзакций:

```
@Stateless public class CredentialsManager {
   @PersistenceContext(unitName="default") private EntityManager em;

    public int checkCredentials(String name, String password) {
       Query query = em.createQuery(...).setParameter("username", name);
       @SuppressWarnings("unchecked") List<Credentials> result
          = query.getResultList();
       ...
    }
}
```

После этого бин сеанса, не поддерживающий состояние, внедряется в один или несколько управляемых бинов с помощью аннотации `@EJB`:

```
@ManagedBean(name="user") @RequestScoped public class UserBean {
   @EJB private CredentialsManager cm;
   ...
   public String login() {
      count = cm.checkCredentials(name, password);
      ...
   }
}
```

Эта модель программирования является очень простой. Логика приложения размещается в управляемых бинах, а бизнес-логика — в бинах сеанса, не поддерживающих состояние. Единственный недостаток состоит в том, что приходится передавать значительное количество данных между бинами этих двух типов. По традиции для решения этой задачи применялись так называемые "объекты доступа к данным", единственным назначением которых была транспортировка данных между архитектурными уровнями. Естественно, что реализация и поддержка этих объектов весьма трудоемки. С появлением версии EJB 3 появилась возможность вместо этого использовать объекты сущности, но при этом приходится учитывать существенные ограничения.

После возврата объекта сущности в управляемый бин JSF из бина сеанса, не поддерживающего состояние, объект сущности становится отделенным. Диспетчер сущностей больше не имеет никаких сведений об этом объекте. Если управляемый бин вносит изменения в сущность, то должен снова ее объединить с диспетчером сущностей. Это обычно достигается с помощью вызова еще одного метода бина сеанса, который вызывает метод `em.merge(entity)`.

В связи с отделением сущностей возникает также другая проблема. Если сущность хранит коллекцию других сущностей, эта коллекция представляет собой не просто хешированное множество или список массивов, но по умолчанию является коллекцией с отложенным доступом, которая обеспечивает выборку элементов только по запросу. Если возникает необходимость передать такую сущность управляемому бину JSF, то приходится обеспечивать предварительную выборку всех зависимых сущностей, обычно путем добавления команд выборки в запрос JPQL.

Чтобы иметь возможность использовать бины сеанса, не поддерживающие состояние, в конкретном приложении JSF, необходимо обладать достаточными знаниями о сущностях EJB 3, без чего невозможно решать эти проблемы, или вместо этого применять объекты доступа к данным. В следующем разделе показано, как избавиться от необходимости прибегать к указанным подходам, используя бины сеанса, поддерживающие состояние.

Продолжим рассмотрение того же примера приложения, но на этот раз реализуя те же функциональные средства с помощью управляемого бина (листинг 12.9) и бина сеанса, не поддерживающего состояние (листинг 12.10).

Листинг 12.9. Файл `slsb/src/java/com/corejsf/UserBean.java`

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.util.logging.Level;
5. import java.util.logging.Logger;
6.
7. import javax.ejb.EJB;
8. import javax.inject.Named;
9.    // или import javax.faces.bean.ManagedBean;
10. import javax.enterprise.context.SessionScoped;
11.    // или import javax.faces.bean.SessionScoped;
12.
13. @Named("user") // или @ManagedBean(name="user")
14. @SessionScoped
15. public class UserBean implements Serializable {
16.    private String name;
17.    private String password;
18.    private boolean loggedIn;
19.    private int count;
20.    private Logger logger = Logger.getLogger("com.corejsf");
21.
22.    @EJB private CredentialsManager cm;
23.
24.    public String getName() { return name; }
25.    public void setName(String newValue) { name = newValue; }
26.
27.    public String getPassword() { return password; }
28.    public void setPassword(String newValue) { password = newValue; }
29.
30.    public int getCount() { return count; }
31.
32.    public String login() {
33.       try {
34.          count = cm.checkCredentials(name, password);
35.          loggedIn = count > 0;
36.       } catch (Exception ex) {
37.          logger.log(Level.SEVERE, "login failed", ex);
38.          return "internalError";
```

```
39.       }
40.       if (loggedIn)
41.          return "loginSuccess";
42.       else
43.          return "loginFailure";
44.    }
45.
46.    public String logout() {
47.       loggedIn = false;
48.       return "login";
49.    }
50. }
```

Листинг 12.10. Файл `slsb/src/java/com/corejsf/CredentialsManager.java`

```
1. package com.corejsf;
2.
3. import java.util.List;
4.
5. import javax.ejb.Stateless;
6. import javax.persistence.EntityManager;
7. import javax.persistence.PersistenceContext;
8. import javax.persistence.Query;
9.
10. @Stateless
11. public class CredentialsManager {
12.    @PersistenceContext(unitName="default")
13.    private EntityManager em;
14.
15.    public int checkCredentials(String name, String password) {
16.       Query query = em.createQuery("SELECT c FROM Credentials c
17.          WHERE c.username = :username")
18.          .setParameter("username", name);
19.       @SuppressWarnings("unchecked")
20.       List<Credentials> result = query.getResultList();
21.       if (result.size() != 1) return 0;
22.       Credentials c = result.get(0);
23.       String storedPassword = c.getPasswd();
24.       if (password.equals(storedPassword.trim()))
25.          return c.incrementLoginCount();
26.       else
27.          return 0;
28.    }
29. }
```

Бины сеанса, поддерживающие состояние CDI

Бин сеанса, не поддерживающий состояние, по существу представляет собой лишь хранилище для размещения одного или нескольких методов, т.е. это — довольно упрощенный объект. Архитектура EJB определяет также бины сеанса, поддерживающие состояние, которые позволяют хранить состояние, как и обычные объекты Java. Разумеется, бины сеанса, поддерживающие состояние, намного сложнее по своему устройству. Ими управляет контейнер EJB, что может быть связано с необходимостью кэшировать их или перемещать на другой сервер для уравновешивания нагрузки. Контейнер также обеспечивает управление доступом и поддержку транзакций при вызове методов.

Возможность использовать бины сеанса, поддерживающие состояние, вместо управляемых бинов JSF впервые была предусмотрена на платформе Seam (http://seamframework.org). С выходом спецификации CDI (Contexts and Dependency Injection) такая возможность была предусмотрена и в версии Java EE 6.

При использовании спецификации CDI приложение может состоять из бинов сеанса, поддерживающих состояние, и бинов сущности, причем для управления теми и другими применяется контейнер EJB. Страницы JSF непосредственно ссылаются на бины сеанса, поддерживающие состояние. В таком случае проблема отсоединенных сущностей теряет свою значимость, а задача обращения к базе данных из веб-приложения становится очень простой. Безусловно, теперь приходится полагаться на контейнер EJB для обеспечения управления всеми конкретными бинами. Если приложение представляет собой простое односерверное приложение, то в связи с использованием контейнеров EJB возникают более значительные издержки, несмотря на то, что эти контейнеры стали намного менее громоздкими и более быстродействующими по сравнению со всеми предыдущими версиями. Но по мере того как приложение растет, кластеризация приложения EJB становится проще. Что же касается производительности, то разработчики платформы Seam утверждают, что она не снижается.

Ниже показано, как использовать бин сеанса, поддерживающий состояние, на платформе JSF.

```
@Named("user") @SessionScoped @Stateful
public class UserBean {
   private String name;
   @PersistenceContext(unitName="default") private EntityManager em;
   ...
   public String getName() { return name; } // Доступ на странице JSF
   public void setName(String newValue) { name = newValue; }

   public String login() { // Вызов со страницы JSF
      doLogin();
      if (loggedIn) return "loginSuccess";
      else return "loginFailure";
   }

   public void doLogin() { // Доступ к базе данных
      Query query = em.createQuery(...).setParameter("username", name);
      @SuppressWarnings("unchecked") List<Credentials> result
         = query.getResultList();
      ...
   }
}
```

На странице JSF применяются выражения значения #{user.name} и #{user.login} обычным образом. Теперь рассматриваемый пример приложения становится чрезвычайно простым. Бин сеанса, поддерживающий состояние, взаимодействует со страницами JSF и с базой данных, как показано в листинге 12.11. А в связи с использованием архитектуры JPA устраняется громоздкий код SQL. Обработка транзакции становится автоматической.

Листинг 12.11. Файл sfsb/src/java/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.util.List;
4. import java.util.logging.Level;
```

```
5. import java.util.logging.Logger;
6.
7. import javax.ejb.Stateful;
8. import javax.enterprise.context.SessionScoped;
9. import javax.inject.Named;
10. import javax.persistence.EntityManager;
11. import javax.persistence.PersistenceContext;
12. import javax.persistence.Query;
13.
14. @Named("user")
15. @SessionScoped
16. @Stateful
17. public class UserBean {
18.    private String name;
19.    private String password;
20.    private boolean loggedIn;
21.    private int count;
22.
23.    @PersistenceContext(unitName="default")
24.    private EntityManager em;
25.
26.    public String getName() { return name; }
27.    public void setName(String newValue) { name = newValue; }
28.
29.    public String getPassword() { return password; }
30.    public void setPassword(String newValue) { password = newValue; }
31.
32.    public int getCount() { return count; }
33.
34.    public String login() {
35.       try {
36.          doLogin();
37.       }
38.       catch (Exception ex) {
39.            Logger.getLogger("com.corejsf").log(Level.SEVERE, "login failed", ex);
40.            return "internalError";
41.        }
42.        if (loggedIn)
43.           return "loginSuccess";
44.        else
45.           return "loginFailure";
46.    }
47.
48.    public String logout() {
49.       loggedIn = false;
50.       return "login";
51.    }
52.
53.    public void doLogin() {
54.       Query query = em.createQuery(
55.          "SELECT c FROM Credentials c WHERE c.username = :username")
56.          .setParameter("username", name);
57.       @SuppressWarnings("unchecked")
58.       List<Credentials> result = query.getResultList();
59.       if (result.size() == 1) {
60.          Credentials c = result.get(0);
61.          String storedPassword = c.getPasswd();
62.          loggedIn = password.equals(storedPassword.trim());
63.          count = c.incrementLoginCount();
64.       }
65.    }
66. }
```

Аутентификация и авторизация, управляемые контейнером

В предыдущих разделах было показано, как использовать базу данных в веб-приложении для поиска сведений о пользователе. А задача использования этих сведений должным образом для допуска или запрета доступа пользователей к определенным ресурсам возлагается на приложение. В данном разделе обсуждается альтернативный подход: аутентификация, управляемая контейнером. Этот механизм позволяет возложить бремя аутентификации пользователей на сервер приложений.

Если аутентификацией и авторизацией управляет контейнер, то намного проще обеспечить единообразное соблюдение требований по безопасности для всего веб-приложения. В таком случае прикладной программист может сосредоточиться на организации функционирования веб-приложения, не задумываясь о решении задач учета прав пользователей.

На заметку! Большая часть подробных сведений о настройке, приведенных в этой главе, характерна для платформ GlassFish и Tomcat, но аналогичные механизмы применяются и на других серверах приложений.

Для защиты набора страниц необходимо задать сведения об управлении доступом в файле `web.xml`. Например, следующее ограничение защиты регламентирует то, что все страницы в защищенном подкаталоге должны быть доступными только пользователям, которые прошли аутентификацию, имеющим роль `registereduser` или `invitedguest`:

```
<security-constraint>
   <web-resource-collection>
      <url-pattern>/faces/protected/*</url-pattern>
      <url-pattern>/protected/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>registereduser</role-name>
      <role-name>invitedguest</role-name>
    </auth-constraint>
</security-constraint>
```

На заметку! Если используется отображение расширения, то не нужно добавлять префикс `/faces` к шаблону URL.

Роль назначается пользователю во время аутентификации. Роли, имена пользователей и пароли хранятся в пользовательском каталоге, который может представлять собой каталог LDAP, базу данных или просто текстовый файл.

После этого необходимо определить, как пользователи должны проходить аутентификацию. Наиболее гибкий подход состоит в применении аутентификации на основе формы. Для этого необходимо добавить следующую запись в файл `web.xml`:

```
<login-config>
   <auth-method>FORM</auth-method>
   <form-login-config>
      <form-login-page>/login.html</form-login-page>
      <form-error-page>/noauth.html</form-error-page>
   </form-login-config>
</login-config>
```

Эта конфигурация формы входа в систему определяет веб-страницу, на которой пользователь вводит имя пользователя и пароль. Разработчик вправе придавать странице входа в своем проекте любой желаемый внешний вид, но обязан включить механизм отправки запроса `j_security_check` с параметрами запроса `j_username` и `j_password`. Эта задача решается с помощью следующей формы:

```
<form method="POST" action="j_security_check">
   User name: <input type="text" name="j_username"/>
   Password: <input type="password" name="j_password"/>
   <input type="submit" value="Login"/>
</form>
```

В качестве страницы с сообщением об ошибке, вообще говоря, может применяться любая страница.

После того как пользователь запрашивает защищенный ресурс, отображается страница входа (рис. 12.11). Если пользователь вводит допустимые значения имени пользователя и пароля, появляется запрашиваемая страница. В противном случае отображается страница с сообщением об ошибке.

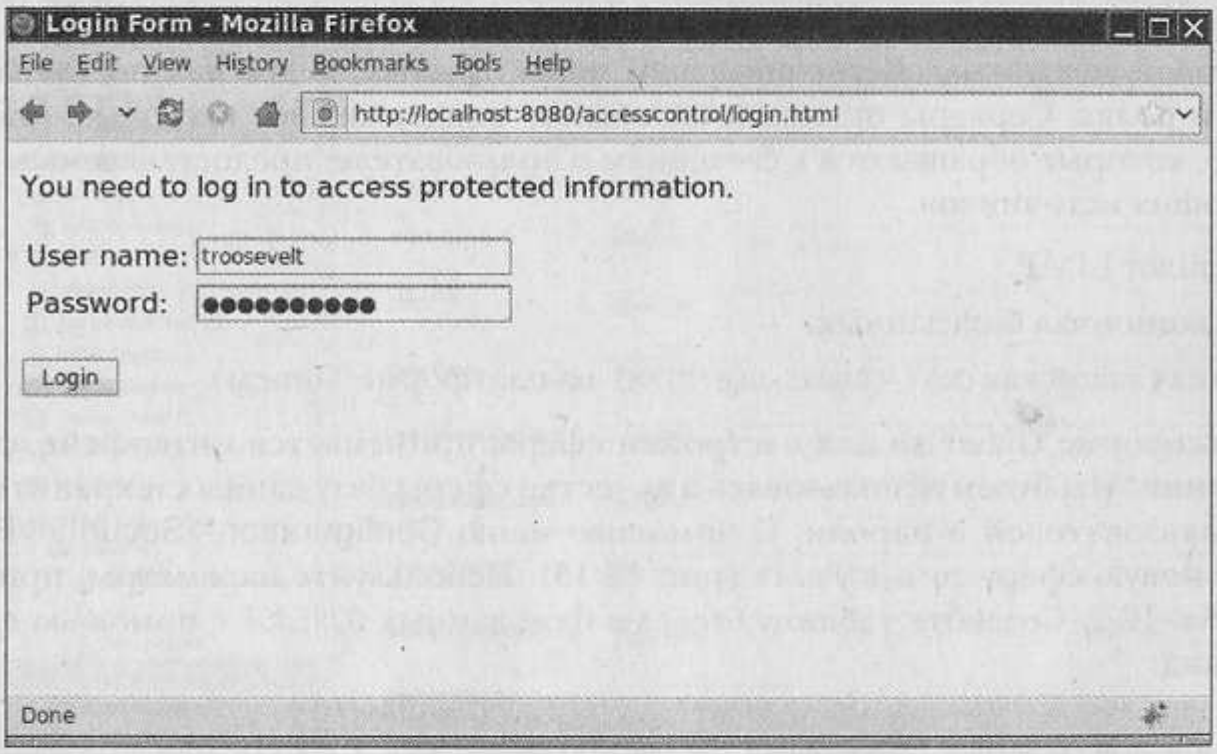


Рис. 12.11. Запрос защищенного ресурса

На заметку! Для безопасной передачи сведений для входа в систему с клиента на сервер необходимо использовать протокол SSL (Secure Sockets Layer). Описание настройки сервера для работы по протоколу SSL выходит за рамки настоящей книги. За дополнительной информацией обратитесь по адресу `http://java.sun.com/developer/technicalArticles/WebServices/appserv8-1.html` (GlassFish) или `http://jakarta.apache.org/tomcat/tomcat-6.0-doc/ssl-howto.html` (Tomcat).

Можно также задать "базовую" аутентификацию путем размещения следующих параметров конфигурации входа в систему в файле `web.xml`:

```
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>This string shows up in the dialog</realm-name>
</login-config>
```

В этом случае браузер выводит всплывающее диалоговое окно для ввода пароля (рис. 12.12). Но с наибольшей вероятностью на профессионально спроектированном веб-сайте будет использоваться аутентификация на основе формы.

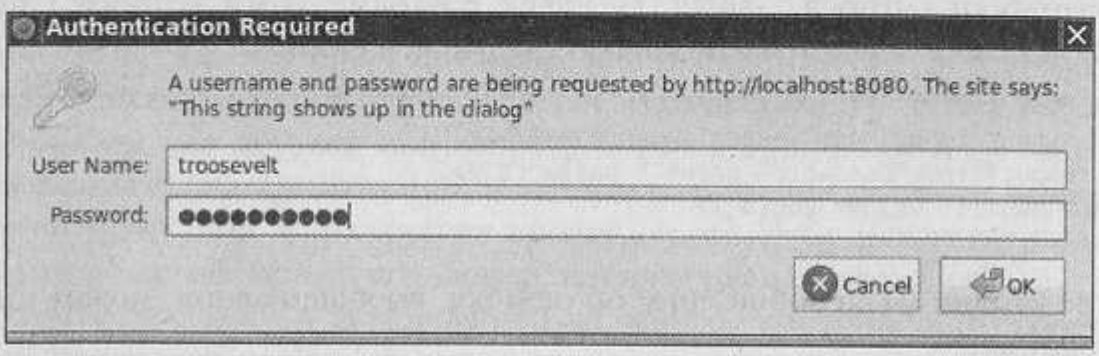


Рис. 12.12. Базовая аутентификация

Файл `web.xml` описывает только ресурсы, для которых определяются ограничения доступа, и роли, которым предоставляется доступ. В нем ничего не говорится о том, как хранятся сведения о пользователях, паролях и ролях. Для настройки этих сведений определяется сфера аутентификации для веб-приложения. Как сфера рассматривается любой механизм, обеспечивающий поиск данных об именах пользователей, паролях и ролях. Серверы приложений обычно поддерживают несколько стандартных сфер, которые обращаются к сведениям о пользователе, предоставляемым одним из следующих источников.

- Каталог LDAP.
- Реляционная база данных.
- Файл (такой как `conf/tomcat-users.xml` на платформе Tomcat).

На платформе GlassFish для настройки сферы применяется интерфейс администрирования. Мы будем использовать в качестве сферы базу данных и хранить в ней имена пользователей и пароли. С помощью меню Configuration⇨Security⇨Realms создайте новую сферу `corejsfRealm` (рис. 12.13). Используйте параметры, приведенные в табл. 12.2. Создайте таблицу `Groups` в базе данных `COREJSF` с помощью следующих команд:

```
CREATE  TABLE Groups (username VARCHAR(20), groupname VARCHAR(20));
INSERT  INTO Groups VALUES ('troosevelt', 'registereduser');
INSERT  INTO Groups VALUES ('troosevelt', 'invitedguest');
INSERT  INTO Groups VALUES ('tjefferson', 'invitedguest');
```

По умолчанию платформа GlassFish обеспечивает отображение имен групп на имена ролей. Такое отображение можно отменить на консоли администрирования GlassFish. Выберите Security и установите параметр Default Principal to Role Mapping. Еще один вариант состоит в том, что можно предусмотреть отображение по умолчанию в файле `WEB-INF/sun-web.xml` с помощью следующего содержимого файла:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Application Server 9.0 Servlet 2.5//EN"
   "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app>
   <security-role-mapping>
      <role-name>registereduser</role-name>
      <group-name>registereduser</group-name>
   </security-role-mapping>
```

```
    <security-role-mapping>
        <role-name>invitedguest</role-name>
        <group-name>invitedguest</group-name>
    </security-role-mapping>
</sun-web-app>
```

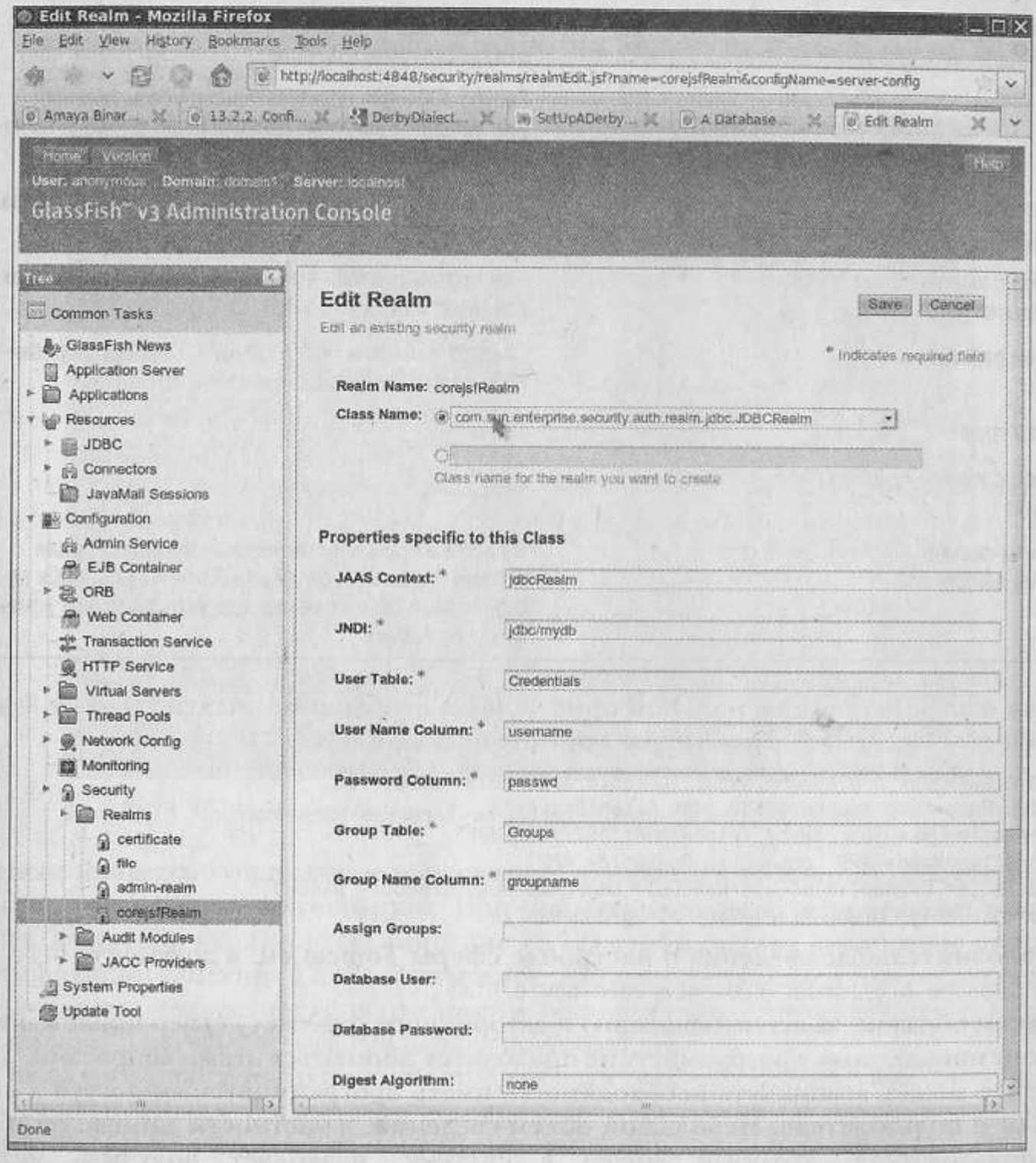


Рис. 12.13. Определение сферы на платформе GlassFish

На заметку! Задача настройки сферы может стать весьма трудоемкой, поскольку при этом необходимо задать правильно слишком много параметров, а по умолчанию какие-либо сообщения об ошибках, кроме "`login failure`", отсутствуют. Для того чтобы получить более содержательные сообщения об ошибках, задайте значение параметра `javax.enterprise.system.core.security.com.sun.enterprise.security.auth.realm.level` равным `FINE` в файле `glassfish/domains/domain1/config/logging.properties`.

Таблица 12.2. Параметры сферы для базы данных

Имя свойства	Значение	Примечания
Имя класса	`com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm`	
Контекст JAAS	`jdbcRealm`	Этот контекст определен в файле `glassfish/domains/domain1/config/login.conf`
JNDI	`jdbc/mydb`	Ресурс JDBC для пользовательской базы данных (см. раздел "Настройка ресурса базы данных в технологии GlassFish" на стр. 428)
Пользовательская таблица	`Credentials`	Используется та же таблица, что и в примерах с базами данных
Столбец с именами пользователей	`username`	Имя столбца должно быть таким же, как и в таблицах для групп и пользователей
Столбец паролей	`passwd`	Следует учитывать, что `password` — зарезервированное слово в PostgreSQL
Таблица групп	`Groups`	
Столбец с именами групп	`groupname`	
Алгоритм формирования дайджеста	`none`	В системе производственного назначения пароли должны быть хешированными. Если используется алгоритм MD5, то хеш можно получить, выполнив команду `echo password | md5sum`

Для настройки сферы на платформе Tomcat необходимо задать элемент `Realm` в файле `conf/server.xml`. Ниже приведен типичный пример.

```
<Realm className="org.apache.catalina.realm.JDBCRealm" debug="99"
   driverName="org.apache.derby.jdbc.ClientDriver"
   connectionURL="jdbc:derby://localhost:1527/COREJSF"
   connectionName="APP" connectionPassword="APP"
   userTable="Credentials" userNameCol="username" userCredCol="passwd"
   userRoleTable="Groups" roleNameCol="groupname"/>
```

Дополнительные сведения о настройке сферы Tomcat см. в документе `http://tomcat.apache.org/tomcat-6.0-doc/realm-howto.html`.

Таким образом, за аутентификацию и авторизацию отвечает сервер приложений, поэтому прикладному программисту не приходится заниматься этими вопросами. Тем не менее может возникнуть необходимость иметь программируемый доступ к сведениям о пользователе. Небольшой объем сведений, в частности данные об имени пользователя, который вошел в систему, позволяет получить метод `HttpServletRequest`. Для возврата объекта запроса применяется внешний контекст:

```
ExternalContext external
   = FacesContext.getCurrentInstance().getExternalContext();
HttpServletRequest request
   = (HttpServletRequest) external.getRequest();
String user = request.getRemoteUser();
```

Предусмотрена также возможность проверить, принадлежит ли текущий пользователь к указанной роли. Например:

```
String role = "admin";
boolean isAdmin = request.isUserInRole(role);
```

На заметку! В настоящее время такая спецификация, которая регламентировала бы выход из системы или переключение удостоверений при использовании защиты, управляемой контейнером, отсутствует. В связи с этим возникают определенные проблемы, особенно при тестировании веб-приложений. На платформах GlassFish и Tomcat для представления текущего пользователя служат cookie-файлы. Если же приходится переключаться на другое удостоверение, то возникает необходимость каждый раз выходить и перезапускать применяемый браузер (или, по крайней мере, удалять персональные данные). Мы решили использовать для тестирования текстовый браузер Lynx, поскольку он запускается намного быстрее по сравнению с графическим веб-браузером (рис. 12.14).

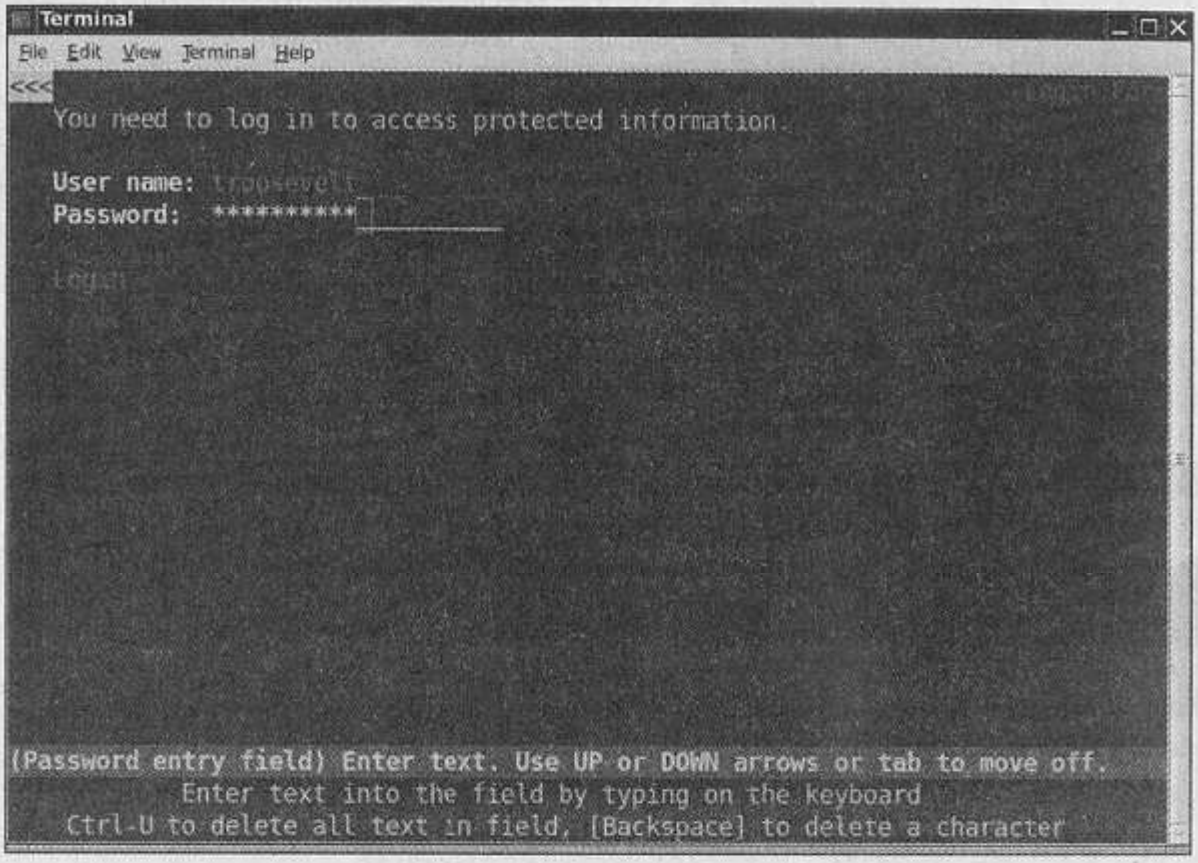


Рис. 12.14. Использование браузера Lynx для тестирования веб-приложения

Ниже приведено упрощенное приложение, которое демонстрирует работу средств защиты, управляемых контейнером. При попытке доступа к защищенному ресурсу `/faces/protected/welcome.xhtml` (листинг 12.12) отображается диалоговое окно аутентификации (листинг 12.13). Возможность продолжить работу предоставляется пользователю только после ввода имени пользователя, принадлежащего к роли `invitedguest` или `registereduser`, и пароля пользователя.

После успешной аутентификации отображается страница приветствия (рис. 12.15), на которой отображается имя зарегистрированного пользователя, чтобы можно было проверить, принадлежит ли пользователь к одной из указанных ролей.

Листинг 12.12. Файл `accesscontrol/web/protected/welcome.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>#{msgs.title}</title>
</h:head>
<h:body>
```

```
    <h:form>
      <p>#{msgs.youHaveAccess}</p>
      <h:panelGrid columns="2">
        #{msgs.yourUserName}
        <h:outputText value="#{user.name}" />

        <h:panelGroup>
          #{msgs.memberOf}
          <h:selectOneMenu onchange="submit()" value="#{user.role}">
            <f:selectItem itemValue="" itemLabel="Select a role" />
            <f:selectItem itemValue="admin" itemLabel="admin" />
            <f:selectItem itemValue="manager" itemLabel="manager" />
            <f:selectItem itemValue="registereduser"
              itemLabel="registereduser" />
            <f:selectItem itemValue="invitedguest" itemLabel="invitedguest" />
          </h:selectOneMenu>
        </h:panelGroup>
        <h:outputText value="#{user.inRole}" />
      </h:panelGrid>
    </h:form>
</h:body>
</html>
```

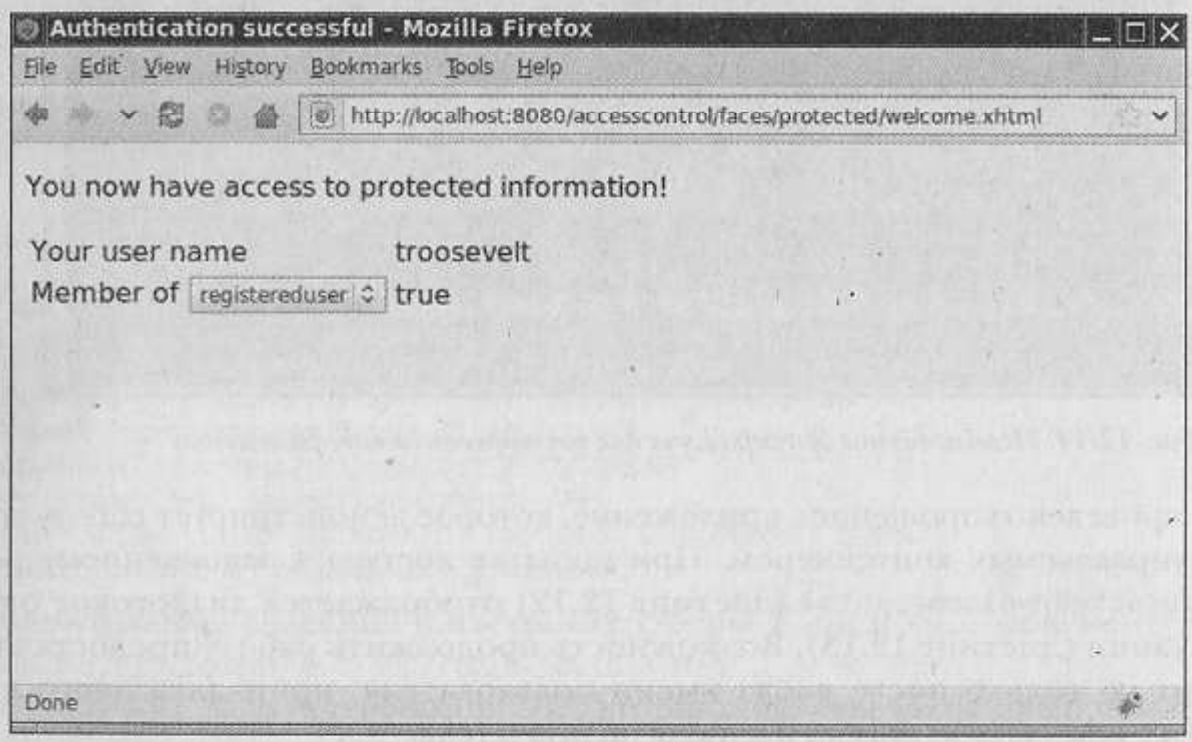


Рис. 12.15. Страница приветствия в приложении для тестирования аутентификации

Листинг 12.13. Файл accesscontrol/web/login.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Login Form</title>
</head>

<body>
<form method="post" action="j_security_check">
<p>You need to log in to access protected information.</p>
<table>
   <tr>
```

```
13.       <td>User name:</td>
14.       <td><input type="text" name="j_username" /></td>
15.    </tr>
16.    <tr>
17.       <td>Password:</td>
18.       <td><input type="password" name="j_password" /></td>
19.    </tr>
20. </table>
21. <p><input type="submit" value="Login" /></p>
22. </form>
23. </body>
24. </html>
```

На рис. 12.16 приведена структура каталогов приложения. Файл web.xml, показанный в листинге 12.14, служит для ограничения доступа к защищенному каталогу. В листинге 12.15 содержится страница, отображаемая в том случае, если авторизация оканчивается неудачей. В листинге 12.12 приведена защищенная страница. Код бина user находится в листинге 12.16, а строки сообщений — в листинге 12.17.

Рис. 12.16. Структура каталогов приложения управления доступом

Листинг 12.14. Файл accesscontrol/web/WEB-INF/web.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.       xmlns="http://java.sun.com/xml/ns/javaee"
4.       xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5.       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6.          http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
7.       version="2.5">
8.    <servlet>
9.       <servlet-name>Faces Servlet</servlet-name>
10.      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
11.   </servlet>
12.   <servlet-mapping>
13.      <servlet-name>Faces Servlet</servlet-name>
14.      <url-pattern>/faces/*</url-pattern>
15.   </servlet-mapping>
16.   <welcome-file-list>
17.      <welcome-file>index.xhtml</welcome-file>
18.   </welcome-file-list>
19.   <context-param>
```

```
      <param-name>javax.faces.PROJECT_STAGE</param-name>
      <param-value>Development</param-value>
   </context-param>

   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Protected Pages</web-resource-name>
         <url-pattern>/faces/protected/*</url-pattern>
         <url-pattern>/protected/*</url-pattern>
      </web-resource-collection>
      <auth-constraint>
         <role-name>registereduser</role-name>
         <role-name>invitedguest</role-name>
      </auth-constraint>
   </security-constraint>

   <login-config>
      <auth-method>FORM</auth-method>
      <realm-name>corejsfRealm</realm-name>
      <form-login-config>
         <form-login-page>/login.html</form-login-page>
         <form-error-page>/noauth.html</form-error-page>
      </form-login-config>
   </login-config>

   <security-role>
      <role-name>registereduser</role-name>
   </security-role>
   <security-role>
      <role-name>invitedguest</role-name>
   </security-role>
</web-app>
```

истинг 12.15. Файл accesscontrol/web/noauth.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Authentication failed</title>
</head>

<body>
<p>Sorry--authentication failed. Please try again.</p>
</body>
</html>
```

истинг 12.16. Файл accesscontrol/src/java/com/corejsf/UserBean.java

```
package com.corejsf;

import java.io.Serializable;
import java.util.logging.Logger;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
```

```
12. import javax.servlet.http.HttpServletRequest;
13.
14. @Named("user") // или @ManagedBean(name="user")
15. @SessionScoped
16. public class UserBean implements Serializable {
17.    private String name;
18.    private String role;
19.    private static Logger logger = Logger.getLogger("com.corejsf");
20.
21.    public String getName() {
22.       if (name == null) getUserData();
23.          return name == null ? "" : name;
24.    }
25.
26.    public String getRole() { return role == null ? "" : role; }
27.    public void setRole(String newValue) { role = newValue; }
28.
29.    public boolean isInRole() {
30.       ExternalContext context
31.          = FacesContext.getCurrentInstance().getExternalContext();
32.       Object requestObject = context.getRequest();
33.       if (!(requestObject instanceof HttpServletRequest)) {
34.          logger.severe("request object has type " + requestObject.getClass());
35.          return false;
36.       }
37.       HttpServletRequest request = (HttpServletRequest) requestObject;
38.       return request.isUserInRole(role);
39.    }
40.
41.    private void getUserData() {
42.       ExternalContext context
43.          = FacesContext.getCurrentInstance().getExternalContext();
44.       Object requestObject = context.getRequest();
45.       if (!(requestObject instanceof HttpServletRequest)) {
46.          logger.severe("request object has type " + requestObject.getClass());
47.          return;
48.       }
49.       HttpServletRequest request = (HttpServletRequest) requestObject;
50.       name = request.getRemoteUser();
51.    }
52. }
```

Листинг 12.17. Файл accesscontrol/src/java/com/corejsf/messages.properties

```
1. title=Authentication successful
2. youHaveAccess=You now have access to protected information!
3. yourUserName=Your user name
4. memberOf=Member of
```

javax.servlet.HttpServletRequest

- `String getRemoteUser()`

  Возвращает имя пользователя, который в настоящее время вошел в систему, или `null`, если таковой пользователь отсутствует.

- `boolean isUserInRole(String role)`

  Проверяет, принадлежит ли текущий пользователь к указанной роли.

Отправка почты

Задача отправки почты в веб-приложении возникает довольно часто. В качестве типичного примера можно указать подтверждение регистрации или сообщение для восстановления пароля. В этом разделе представлено краткое руководство по использованию API-интерфейса JavaMail в приложениях JSF.

Основной процесс является таким же, как при использовании пула базы данных. Настройка ресурса почты осуществляется на сервере приложений или в веб-контейнере. На рис. 12.17 показано, как применять веб-интерфейс GlassFish.

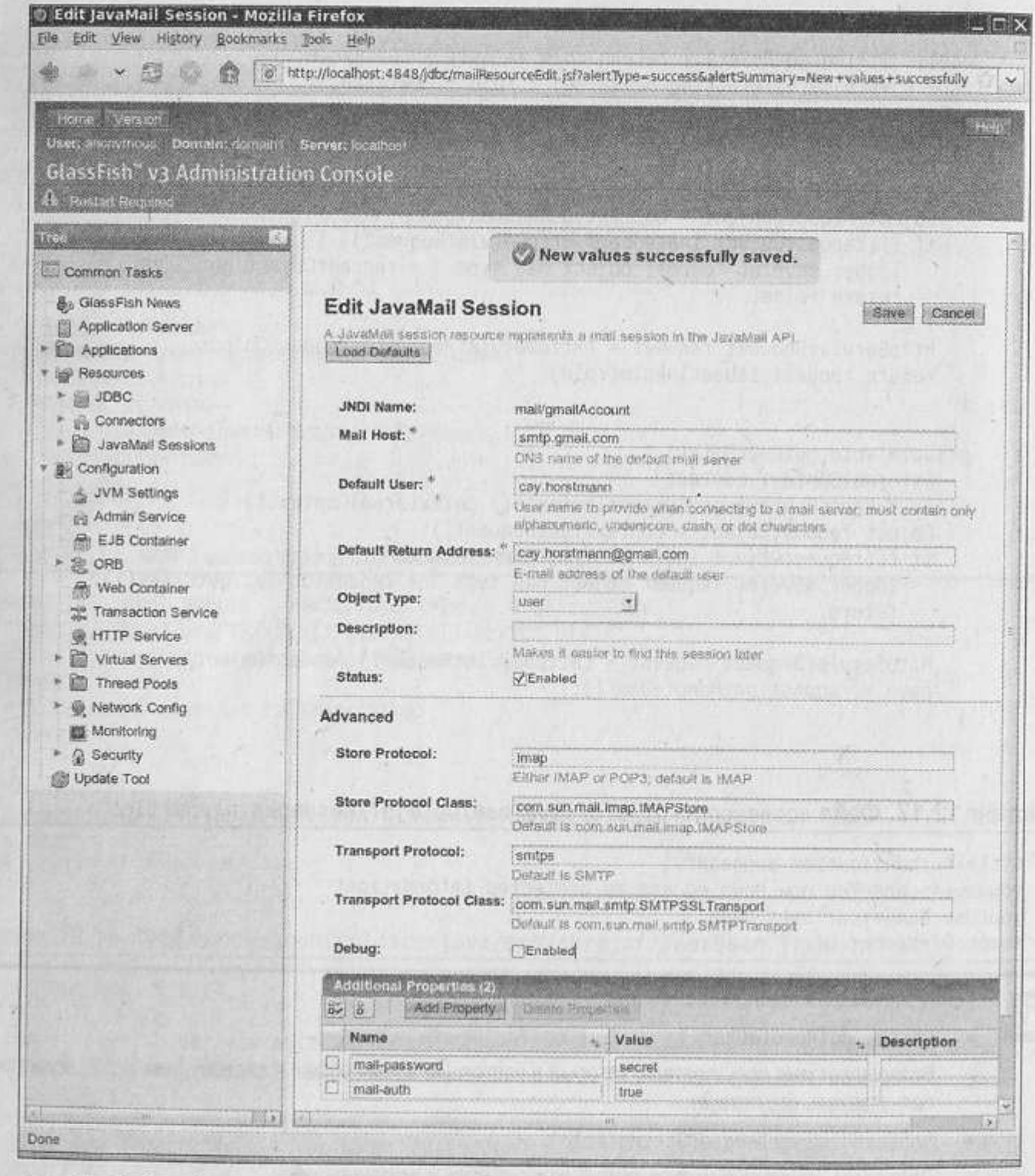


Рис. 12.17. Определение параметров почтового сеанса на платформе GlassFish

Параметры учетной записи GMail приведены в табл. 12.3. (Прежде чем предпринимать попытку использовать рассматриваемый пример программы с конкретной учет-

ной записью GMail, необходимо тщательно проверить, принимает ли сервер GMail соединения SMTP.)

На заметку! Для получения инструкций по работе с платформой Tomcat см. раздел "Сеансы JavaMail" документа http://tomcat.apache.org/tomcat-6.0-doc/jndi-resources-howto.html .

Внимание! При определении параметров конфигурации почты на платформе GlassFish необходимо заменять точки дефисами. Например, mail.auth преобразуется в mail-auth. Кроме того, в список свойств почтового сеанса помещаются только параметры, начинающиеся с префикса mail-.

Таблица 12.3. Параметры почтового сеанса

Параметр	Значение	Примечания
Имя JNDI	mail/gmailAccount	Это имя используется в аннотации @Resource
Почтовый узел	smtp.gmail.com	В качестве такового должен быть указан сервер SMTP
Пользователь по умолчанию	your.account	Имя применяемой учетной записи
Обратный адрес по умолчанию	your.account@gmail.com	Обратный адрес для сообщений
Транспортный протокол	smtps	Если на сервере не используется SSL/TLS, следует указывать smtp
Класс транспортного протокола	com.sun.mail.smtp.SMTPSSLTransport	Если на сервере не используется протокол протокол SSL/TLS, следует указывать com.sun.mail.smtp.SMTPTransport
Дополнительный параметр mail-password	secret	Необходимо подставить собственный пароль, учитывая то, что в имени параметра есть дефис
Дополнительный параметр mail-auth	true	Обратите внимание на наличие дефиса в имени параметра

Выберите имя JNDI для конкретного ресурса почты и воспользуйтесь им для внедрения объекта javax.mail.Session:

```
@Resource(name = "mail/gmailAccount")
private Session mailSession;
```

После того как все будет готово для отправки сообщения, примените следующую схему кода:

```
MimeMessage message = new MimeMessage(mailSession);

Address toAddress = new InternetAddress(email address);
message.setRecipient(RecipientType.TO, toAddress);
message.setSubject(subject);
message.setText(message text);
message.saveChanges();

Transport tr = mailSession.getTransport();
String serverPassword = mailSession.getProperty("mail.password");
tr.connect(null, serverPassword);
tr.sendMessage(message, message.getAllRecipients());
tr.close();
```

Заслуживает внимания то, как происходит поиск пароля учетной записи, который помещен в список дополнительных параметров при настройке параметров почты.

На заметку! Общеизвестно, что при решении задачи отправки почты часто приходится сталкиваться с ошибками, поскольку большинство почтовых серверов налагает ограничения на то, кто может к ним обращаться, чтобы бороться со спамом. Может оказаться, что читатель не сможет отправлять почту через свой корпоративный или университетский сервер из дома, а большинство бесплатных почтовых веб-служб больше не поддерживает соединения SMTP. Еще одна сложность при использовании JavaMail состоит в получении правильных параметров почты. При возникновении проблем получите дистрибутив JavaMail по адресу `http://java.sun.com/products/javamail/downloads` и вызовите на выполнение программу проверки `smtpsend`. После получения с помощью этой программы проверки правильных параметров для почтового сервера снова возвратитесь к настройке сервера приложений.

В приведенном нами примере программы предусмотрена служба регистрации, типичная для веб-приложений. Пользователь выбирает желаемое имя пользователя и задает адрес электронной почты. Система вырабатывает случайный пароль и передает по электронной почте сведения для входа в систему (рис. 12.18). Для простоты мы пропускаем шаг проверки того, доступно ли имя пользователя, и сохраняем параметры доступа в базе данных.

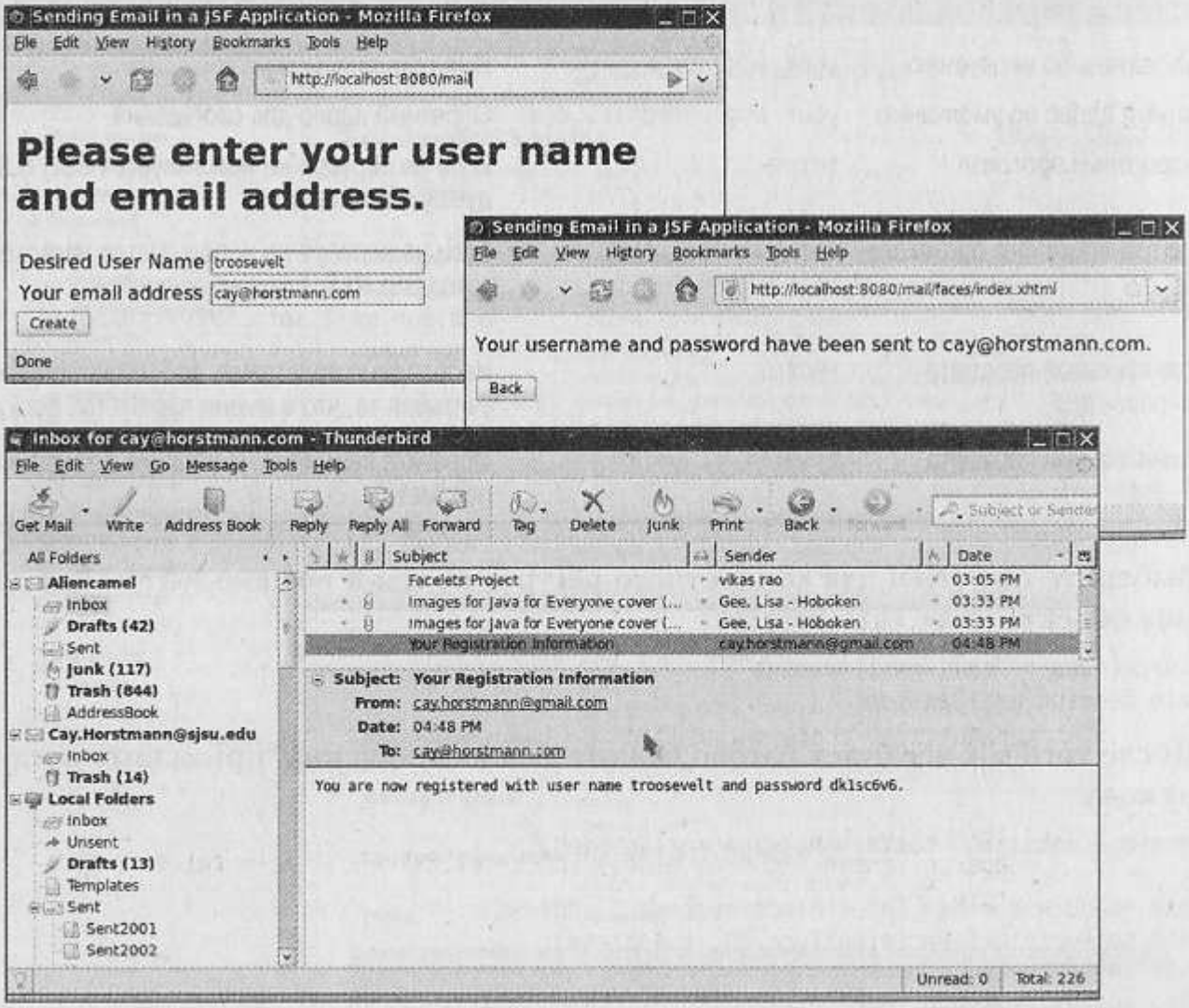


Рис. 12.18. Приложение JSF, которое отправляет сообщение электронной почты

В листинге 12.18 показан класс, с помощью которого осуществляется отправка почтовых сообщений. На странице `index.xhtml` просто задаются свойства `name` и `emailAddress`; эта страница здесь не показана. В листинге 12.19 приведена связка сообщений.

Листинг 12.18. Файл `mail/src/java/com/corejsf/NewAccount.java`

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.text.MessageFormat;
5. import java.util.ResourceBundle;
6. import java.util.logging.Level;
7. import java.util.logging.Logger;
8.
9. import javax.annotation.Resource;
10. import javax.inject.Named;
11.    // или import javax.faces.bean.ManagedBean;
12. import javax.enterprise.context.SessionScoped;
13.    // или import javax.faces.bean.SessionScoped;
14. import javax.mail.Address;
15. import javax.mail.MessagingException;
16. import javax.mail.Session;
17. import javax.mail.Transport;
18. import javax.mail.Message.RecipientType;
19. import javax.mail.internet.InternetAddress;
20. import javax.mail.internet.MimeMessage;
21.
22. @Named // или @ManagedBean
23. @SessionScoped
24. public class NewAccount implements Serializable {
25.    private String name;
26.    private String emailAddress;
27.    private String password;
28.
29.    @Resource(name="mail/gmailAccount")
30.    private Session mailSession;
31.
32.    public String getName() { return name; }
33.    public void setName(String newValue) { name = newValue; }
34.
35.    public String getEmailAddress() { return emailAddress; }
36.    public void setEmailAddress(String newValue) { emailAddress = newValue; }
37.
38.    public String create() {
39.       try {
40.          createAccount();
41.          sendNotification();
42.          return "done";
43.       }
44.       catch (Exception ex) {
45.          Logger.getLogger("com.corejsf").log(Level.SEVERE, "login failed", ex);
46.          return "error";
47.       }
48.    }
49.
50.    private void createAccount() {
51.       // Создание случайного пароля; 8-знаковое число по основанию 36
52.       int BASE = 36;
53.       int LENGTH = 8;
54.       password = Long.toString((long)(Math.pow(BASE, LENGTH) * Math.random()), BASE);
55.       /*
56.        * В реальном приложении необходимо убедиться в том, что имя пользователя еще
57.        * никем не используется и сохранить имя пользователя и пароль в базе данных.
58.        */
59.    }
60.
```

```
61.    private void sendNotification() throws MessagingException {
62.       ResourceBundle bundle = ResourceBundle.getBundle("com.corejsf.messages");
63.         String subject = bundle.getString("subject");
64.         String body = bundle.getString("body");
65.         String messageText = MessageFormat.format(body, name, password);
66.         mailSession.setDebug(true);
67.         MimeMessage message = new MimeMessage(mailSession);
68.
69.         Address toAddress = new InternetAddress(emailAddress);
70.         message.setRecipient(RecipientType.TO, toAddress);
71.         message.setSubject(subject);
72.         message.setText(messageText);
73.         message.saveChanges();
74.
75.         Transport tr = mailSession.getTransport();
76.         String serverPassword = mailSession.getProperty("mail.password");
77.         tr.connect(null, serverPassword);
78.         tr.sendMessage(message, message.getAllRecipients());
79.         tr.close();
80.    }
81. }
```

Листинг 12.19. Файл mail/src/java/com/corejsf/messages.properties

```
1. title=Sending Email in a JSF Application
2. enterNameAndEmail=Please enter your user name and email address.
3. name=Desired User Name
4. email=Your email address
5. emailSent=Your username and password have been sent to {0}.
6. internalError=Internal Error
7. internalError_detail=To our chagrin, an internal error has occurred. \
8.    Please report this problem to our technical staff.
9. create=Create
10. back=Back
11. subject=Your Registration Information
12. body=You are now registered with user name {0} and password {1}.
```

Использование веб-служб

В веб-приложении для получения информации из внешнего источника обычно используется механизм дистанционного вызова. К числу технологий, широко применяемых с этой целью, в частности, относятся веб-службы.

В настоящее время имеются два различных теоретических подхода к тому, как должны быть реализованы веб-службы. В подходе с условным названием *WS-** используется код XML для оформления запросов и ответов, а также тщательно разработанный (хотя и сложный) механизм для представления произвольных данных в коде XML. В подходе, основанном на поддержке метода REST, запросы кодируются в виде URL, а для представления ответов применяется ряд форматов (XML, JSON, простой текст и так далее, в соответствии с тем, что указано в заголовке HTTP). Задача чтения ответа возлагается на клиента. Оба эти подхода обладают определенными преимуществами и недостатками. На платформе JSF предусмотрены встроенные средства поддержки, позволяющие работать с веб-службой WS-*, которые будут описаны в данном разделе.

Для этого рассмотрим простую службу получения прогнозов погоды, которая описана на странице http://wiki.cdyne.com/wiki/index.php?title=CDYNE_Weather. Наиболее

привлекательной особенностью веб-служб является их независимость от языка. Мы будем обращаться к требуемой службе с помощью языка программирования Java, а другие разработчики могут столь же легко воспользоваться языком C++ или PHP. Для описания служб в формате, независимом от языка, применяется файл описания на языке WSDL (Web Services Definition Language). Например, файл WSDL для службы прогнозов погоды CDYNE (расположенный по адресу `http://ws.cdyne.com/WeatherWS/Weather.asmx?wsdl`) описывает операцию `GetCityForecastByZIP` следующим образом:

```
<wsdl:operation name="GetCityForecastByZIP">
    <wsdl:input message="tns:GetCityForecastByZIPSoapIn"/>
    <wsdl:output message="tns:GetCityForecastByZIPSoapOut"/>
</wsdl:operation>
...
<wsdl:message name="GetCityForecastByZIPSoapIn">
    <wsdl:part name="parameters" element="tns:GetCityForecastByZIP"/>
</wsdl:message>
<wsdl:message name="GetCityForecastByZIPSoapOut">
    <wsdl:part name="parameters" element="tns:GetCityForecastByZIPResponse"/>
</wsdl:message>
```

Ниже приведены определения типов `GetCityForecastByZIPResponse` и `GetCityForecastByZIP`.

```
<s:element name="GetCityForecastByZIP">
   <s:complexType>
      <s:sequence>
         <s:element minOccurs="0" maxOccurs="1" name="ZIP" type="s:string"/>
      </s:sequence>
   </s:complexType>
</s:element>
```

Это просто означает, что для службы в качестве параметра требуется необязательная строка с именем `ZIP`. Технология JAX-WS обеспечивает отображение описания типа WSDL в тип Java, в данном случае `String`.

Возвращаемый тип является более сложным:

```
<s:element name="GetCityForecastByZIPResponse">
   <s:complexType>
      <s:sequence>
         <s:element minOccurs="0" maxOccurs="1" name="GetCityForecastByZIPResult"
            type="tns:ForecastReturn"/>
      </s:sequence>
   </s:complexType>
</s:element>

<s:complexType name="ForecastReturn">
   <s:sequence>
      <s:element minOccurs="1" maxOccurs="1"      name="Success" type="s:boolean"/>
      <s:element minOccurs="0" maxOccurs="1"      name="ResponseText" type="s:string"/>
      <s:element minOccurs="0" maxOccurs="1"      name="State" type="s:string"/>
      <s:element minOccurs="0" maxOccurs="1"      name="City" type="s:string"/>
      <s:element minOccurs="0" maxOccurs="1"      name="WeatherStationCity"
         type="s:string"/>
      <s:element minOccurs="0" maxOccurs="1"      name="ForecastResult"
         type="tns:ArrayOfForecast"/>
   </s:sequence>
</s:complexType>
```

Средства JAX-WS отображают возвращаемый тип в класс `ForecastReturn` на языке Java с помощью следующих методов:

```
public   boolean getSuccess()
public   String getResponseText()
public   String getState()
public   String getCity()
public   String getWeatherStationCity()
public   ArrayOfForecast getForecastResult()
```

Тип `ArrayOfForecast` также определен в файле WSDL. Соответствующий класс Java имеет такой метод:

```
List<Forecast> getForecast()
```

Наконец, класс `Forecast` имеет методы `getDesciption` (действительно так), `getTemperatures` и т.д. Для нас не представляют интерес подробные сведения об организации данной конкретной службы; гораздо важнее то, содержит ли файл WSDL все сведения, необходимые для обработки параметров и возвращаемых значений службы.

Чтобы узнать, как вызывается служба поиска, необходимо найти элемент `service` в файле WSDL:

```
<wsdl:service name="Weather">
    <wsdl:port name="WeatherSoap" binding="tns:WeatherSoap">
       <soap:address location="http://ws.cdyne.com/WeatherWS/Weather.asmx"/>
    </wsdl:port>
...
</wsdl:service>
```

Эти данные говорят о том, что мы должны сделать вызов через объект `"port"` типа `WeatherSoap`, полученный от объекта `"service"` типа `Weather`.

Для получения этого объекта службы применяется внедрение зависимости. Для аннотирования поля задается аннотация `@WebServiceRef`:

```
@WebServiceRef(wsdlLocation="http://ws.cdyne.com/WeatherWS/Weather.asmx?wsdl")
private Weather service;
```

После этого осуществляется следующий вызов:

```
ForecastReturn ret = service.getWeatherSoap().getCityForecastByZIP(zip);
```

С точки зрения программиста задача вызова веб-службы WS-* является чрезвычайно простой. Для этого достаточно лишь вызвать метод Java. После этого незаметно для постороннего взгляда платформа JAX-WS преобразует объекты параметров из формата Java в формат XML, передает код XML на сервер (с использованием протокола SOAP, о котором пользователь вообще не обязан знать), получает необходимый результат и преобразует его из формата XML в формат Java.

При организации работы с веб-службой сложнее всего правильно определить параметры и возвращаемые типы, но решение этой задачи намного упрощается, если сведения о них хорошо задокументированы.

Безусловно, разработчик обязан сформировать классы, используемые службой. Для автоматического формирования классов на основе файла WSDL может применяться инструментальное средство wsimport (которое включено в новейшие версии JDK и GlassFish). К сожалению, при формировании классов, которые реализуют интерфейс `Serializable`, приходится сталкиваться с более значительными сложностями. Поместите следующие магические строки в файл `jaxb-bindings.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
   xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
   elementFormDefault="qualified" attributeFormDefault="unqualified"
```

```
    jaxb:extensionBindingPrefixes="xjc" jaxb:version="2.1">
    <xs:annotation>
       <xs:appinfo>
          <jaxb:globalBindings>
             <xjc:serializable />
          </jaxb:globalBindings>
       </xs:appinfo>
    </xs:annotation>
</xs:schema>
```

После этого выполните такие команды:

```
wsimport -b jaxb-bindings.xml -p com.corejsf.ws http://ws.cdyne.com/
↳WeatherWS/Weather.asmx?wsdl
jar cvf weather-ws.jar com/corejsf/ws/*.class
```

(Первая команда должна быть набрана в одной строке.) Сохраните полученный файл JAR в каталоге `WEB-INF/lib` конкретного приложения JSF.

Рассматриваемый нами пример приложения является несложным. Пользователь определяет почтовый индекс и щелкает на кнопке Search (рис. 12.19).

Рис. 12.19. Получение прогноза погоды

Служба возвращает прогноз погоды на семь дней (рис. 12.20). Это может служить указанием на то, что веб-служба работает успешно. Оставляем это приложение в качестве основы упражнения для читателя, который сможет расширить его функциональные возможности.

Рис. 12.20. Прогноз погоды

Класс бина, приведенный в листинге 12.20, содержит вызов веб-службы. Мы убираем сведения о городе и список объектов Forecast, чтобы на странице success.xhtml можно было отобразить только результат.

Страницы JSF приведены в листингах 12.21-12.22. На странице success.xhtml выполняется обработка в цикле объектов Forecast, полученных от службы погоды.

На заметку! Класс Forecast действительно имеет свойство desciption. Просим не отправлять нам сообщения об ошибке по этому поводу.

Листинг 12.20. Файл webservice/src/java/com/corejsf/WeatherBean.java

```
1. package com.corejsf;
2.
3. import java.util.List;
4.
5. import javax.inject.Named;
6.    // или import javax.faces.bean.ManagedBean;
7. import javax.enterprise.context.SessionScoped;
8.    // или import javax.faces.bean.SessionScoped;
9. import javax.xml.ws.WebServiceRef;
10.
11. import com.corejsf.ws.Forecast;
12. import com.corejsf.ws.ForecastReturn;
13. import com.corejsf.ws.Weather;
14. import java.io.Serializable;
15. import java.util.logging.Level;
16. import java.util.logging.Logger;
17.
18. @Named // или @ManagedBean
19. @SessionScoped
20. public class WeatherBean implements Serializable {
21.    @WebServiceRef(wsdlLocation="http://ws.cdyne.com/WeatherWS/Weather.asmx?wsdl")
22.    private Weather service;
23.
24.    private String zip;
25.    private String city;
26.    private List<Forecast> response;
27.
28.    public String getZip() { return zip; }
29.    public void setZip(String newValue) { zip = newValue; }
30.
31.    public List<Forecast> getResponse() { return response; }
32.    public String getCity() { return city; }
33.
34.    public String search() {
35.       try {
36.          ForecastReturn ret = service.getWeatherSoap().getCityForecastByZIP(zip);
37.          response = ret.getForecastResult().getForecast();
38.          for (Forecast f : response)
39.             if (f.getDesciption() == null || f.getDesciption().length() == 0)
40.                f.setDesciption("Not Available");
41.          city = ret.getCity();
42.          return "success";
43.       } catch(Exception e) {
44.          Logger.getLogger("com.corejsf").log(Level.SEVERE, "Remote call failed", e);
45.          return "error";
46.       }
47.    }
48. }
```

Листинг 12.21. Файл webservices/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <link href="styles.css" rel="stylesheet" type="text/css"/>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <h1>#{msgs.weatherSearch}</h1>
         #{msgs.zip}
         <h:inputText value="#{weatherBean.zip}"/>
         <h:commandButton value="#{msgs.search}" action="#{weatherBean.search}"/>
      </h:form>
   </h:body>
</html>
```

Листинг 12.22. Файл webservices/web/success.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions">
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <h1>
            <h:outputFormat value="#{msgs.searchResult}">
               <f:param value="#{weatherBean.city}"/>
            </h:outputFormat>
         </h1>
         <table>
            <tr>
               <ui:repeat value="#{weatherBean.response}" var="item">
                  <td>
                     <h:panelGrid columns="1"
                                  style="width: 8em; border: thin dotted;">
                        <h:outputText value="#{item.desciption}"/>
                        <h:graphicImage library="images"
                              name="#{fn:replace(item.desciption, ' ', '')}.gif"/>
                        <h:outputText
                              value=
                              "#{msgs.low}: #{item.temperatures.morningLow}"/>
                        <h:outputText value=
                              "#{msgs.high}: #{item.temperatures.daytimeHigh}"/>
                     </h:panelGrid>
                  </td>
               </ui:repeat>
            </tr>
         </table>
         <h:commandButton value="#{msgs.back}" action="index"/>
```

```
39.         </h:form>
40.     </h:body>
41. </html>
```

Резюме

В настоящей главе было описано, как обеспечить подключение веб-приложений к внешним службам, таким как базы данных, электронная почта и веб-службы. Серверы приложений предоставляют общее обслуживание в связи с поддержкой пулов соединений с базами данных, сфер аутентификации и т.д. Внедрение зависимостей служит основой удобного и переносимого механизма поиска классов, необходимых для доступа к этим службам.

Дополнительные рекомендации

В этой главе...

- Поиск дополнительных компонентов
- Поддержка выгрузки файлов
- Отображение гиперкарты
- Формирование двоичных данных на странице JSF
- Одновременное отображение крупного набора данных по одной странице
- Формирование всплывающего окна
- Выборочное отображение и сокрытие частей страницы
- Настройка страниц с сообщениями об ошибках
- Написание собственного клиентского тега проверки правильности
- Настройка собственного приложения
- Расширение языка выражений JSF
- Добавление функций к языку выражений JSF JSF 2.0
- Мониторинг трафика между браузером и сервером
- Отладка застывшей страницы
- Использование инструментальных средств тестирования при разработке приложения JSF
- Использование технологии Scala с JSF
- Использование технологии Groovy с JSF

Глава 13

В предыдущих главах было приведено систематическое описание технологии JSF, сосредоточенное вокруг основных понятий. Но каждой технологии присущи свои особенности, которые не могут стать предметом систематического изложения, и JSF не составляет исключения. Время от времени разработчик на JSF сталкивается с вопросом: "Как сделать то-то и то-то" и не находит ответа, возможно, в связи с фактическим отсутствием поддержки в JSF требуемой функции или же по той причине, что искомое решение не очевидно на первый взгляд. Настоящая глава была задумана как пособие по выходу из подобных ситуаций. Мы приводим в довольно произвольном порядке ответы на часто возникающие вопросы, которые мы выслушивали на семинарах или встречали в письмах, полученных от читателей.

Поиск дополнительных компонентов

Стандарт JSF определяет минимальный набор компонентов. Единственным стандартным компонентом, который выходит за пределы базовых средств HTML, является таблица данных. Узнав об этом, любой разработчик, который поверил лозунгу "JSF – это Swing для веб", испытывает разочарование.

У читателя может возникнуть вопрос, почему разработчики спецификации JSF не включили набор профессионально спроектированных компонентов, таких как деревья, средства выбора даты и времени и т.д. Но для проектирования таких компонентов требуется колоссальный уровень мастерства, а сами востребованные навыки не имеют ничего общего с теми, которые позволяют составить технологическую спецификацию.

Ниже перечислено несколько библиотек компонентов, с которыми имеет смысл ознакомиться.

- ICEfaces (`http://icefaces.org`) – библиотека компонентов с поддержкой Ajax с открытым исходным кодом. Предусмотрено дополнение к программе Netbeans для ICEfaces. Версия 2 совместима с JSF 2.0.
- RichFaces (`http://www.jboss.org/jbossrichfaces/`) – еще одна библиотека компонентов с открытым исходным кодом. Она входит в состав сервера приложений JBoss, но может также использоваться отдельно. Версия 4 совместима с JSF 2.0.

- PrimeFaces (http://primefaces.org) и OpenFaces (http://openfaces.org) — две перспективные библиотеки с открытым исходным кодом, имеющие очень богатый выбор компонентов.
- Набор компонентов ADF Faces, предлагаемый корпорацией Oracle (http://oracle.com/technology/products/adf/adffaces), представляет собой высококачественный комплект тщательно спроектированных компонентов; кроме того, он включает функциональные средства поддержки Ajax и позволяет изменять внешний вид пользовательского интерфейса с помощью скинов.
- Библиотека Apache Trinidad (http://myfaces.apache.org/trinidad) с открытым исходным кодом, разработанная на основе ранней версии библиотеки ADF Faces, которую корпорация Oracle бесплатно передала фонду Apache.
- Библиотека Apache Tomahawk (http://myfaces.apache.org/tomahawk) содержит множество невизуальных компонентов, которые могут оказаться полезными для решения определенных задач JSF, но эти невизуальные компоненты не очень привлекательны.
- В рамках проекта Java BluePrints разработан набор компонентов Ajax (https://blueprints.dev.java.net/ajaxcomponents.html). Он включает средства автозавершения, интерфейсы карт Google, всплывающие подсказки, средства выгрузки файлов с индикатором хода работы, а также несколько других привлекательных и полезных компонентов.

Список дополнительных компонентов можно найти по адресу http://jsfcentral.com/products/components.

Поддержка выгрузки файлов

В каждом конкретном приложении пользователям может потребоваться выгружать файлы, такие как фотографии или документы (рис. 13.1-13.2).

К сожалению, в технологии JSF не предусмотрен стандартный компонент выгрузки файла. Однако, как оказалось, решить эту задачу довольно несложно. Самая сложная часть этой работы уже была проделана специалистами из организации Apache при создании библиотеки выгрузки файлов Commons (см. http://jakarta.apache.org/commons/fileupload). Ниже будет показано, как включить эту библиотеку в компонент JSF.

На заметку! Набор компонентов Tomahawk содержит компонент выгрузки файлов с атрибутами, немного отличными от применяемых нами (см. http://myfaces.apache.org/tomahawk). В статье, которая доступна по адресу http://today.java.net/pub/a/today/2006/02/09/file-uploads-with-ajax-and-jsf.html, показано, как добавить индикатор выполнения Ajax к компоненту выгрузки файлов.

Выгрузка файлов отличается от всех других запросов формы. Данные формы (включая выгружаемый файл) при отправке с клиента на сервер кодируются с помощью кодировки "multipart/form-data", а не обычной кодировки "application/x-www-form-urlencoded".

К сожалению, в технологии JSF вообще не предусмотрена обработка этой кодировки. В целях преодоления этой проблемы мы устанавливаем фильтр сервлетов, который перехватывает запрос на выгрузку файла и преобразует выгружаемые файлы в атрибуты запроса, а все прочие данные формы — в параметры запроса. (Для выполнения неблагодарной работы по декодированию запроса multipart/form-data используется сервисный метод из библиотеки выгрузки файлов Commons.)

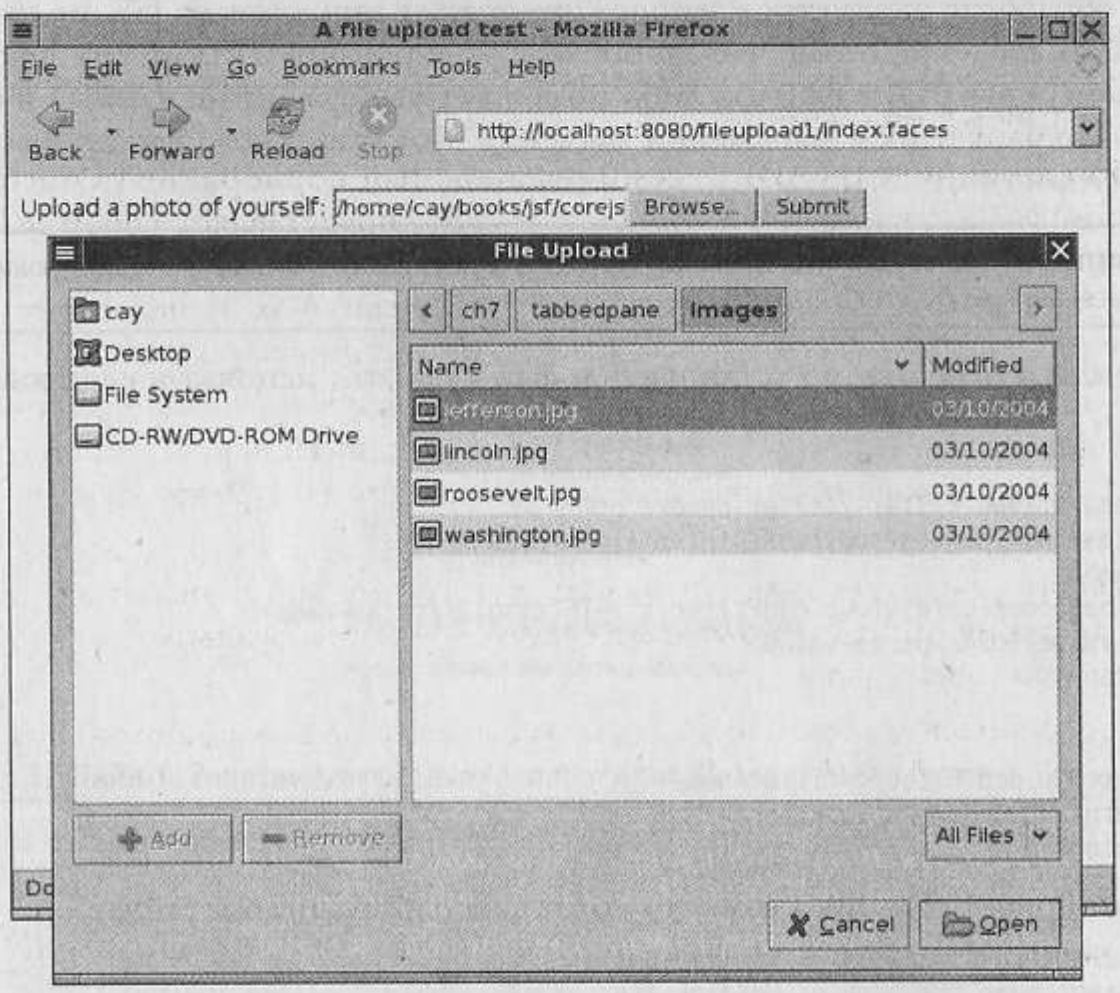


Рис. 13.1. Выгрузка файла с изображением

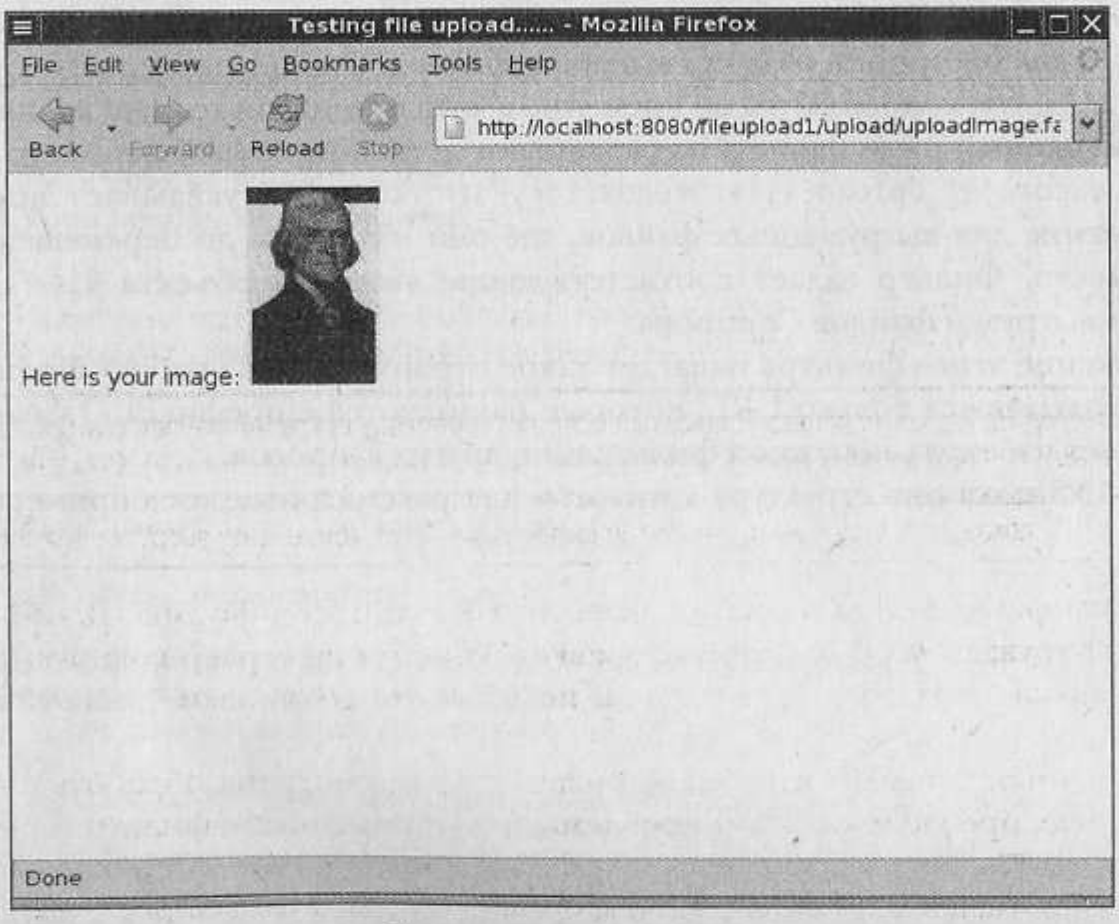


Рис. 13.2. Выгруженное изображение

Затем к обработке параметров запроса приступает приложение JSF, не имея абсолютно никаких сведений о том, что запрос не закодирован в URL. Метод декодирования компонента выгрузки файлов либо помещает выгруженные данные в файл на диске, либо сохраняет их в выражении значения. Код указанного фильтра сервлетов содержится в листинге 13.1.

На заметку! Общие сведения о фильтрах сервлетов можно найти по адресу `http://java.sun.com/products/servlet/Filters.html`.

Такой фильтр необходимо установить в файл `web.xml`, используя следующий синтаксис:

```
<filter>
   <filter-name>Upload Filter</filter-name>
   <filter-class>com.corejsf.UploadFilter</filter-class>
   <init-param>
      <param-name>com.corejsf.UploadFilter.sizeThreshold</param-name>
      <param-value>1024</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>Upload Filter</filter-name>
   <url-pattern>/upload/*</url-pattern>
</filter-mapping>
```

Еще один вариант состоит в том, что можно использовать аннотацию:

```
@WebFilter(urlPatterns="/upload/*", initParams={
   @WebInitParam(name="com.corejsf.UploadFilter.sizeThreshold", value="1024")
})
public class UploadFilter
```

В фильтре применяется параметр инициализации `com.corejsf.UploadFilter.sizeThreshold` для настройки объекта выгрузки файлов. Файлы, размеры которых превышают 1024 байта, сохраняются во временном каталоге на диске, а не хранятся в памяти. Применяемый нами фильтр поддерживает дополнительный параметр инициализации `com.corejsf.UploadFilter.repositoryPath`, который указывает временное местоположение для выгруженных файлов, где они находятся до перемещения в постоянное место. Фильтр задает соответствующие свойства объекта `DiskFileUpload` библиотеки выгрузки файлов Commons.

Отображение этого фильтра налагает такое ограничение на работу фильтра, что в нем обрабатываются только URL, которые начинаются с префикса `/faces/upload/`. Это позволяет избежать ненужной фильтрации других запросов.

На рис. 13.3 показана структура каталогов для рассматриваемого примера приложения.

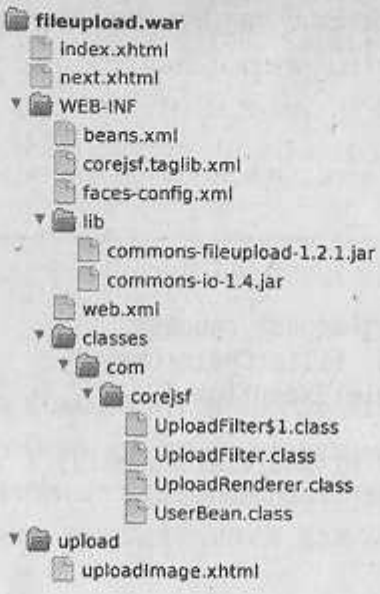


Рис. 13.3. Структура каталогов приложения выгрузки файлов

Листинг 13.1. Файл `fileupload/src/java/com/corejsf/UploadFilter.java`

```
1. package com.corejsf;
2.
3. import java.io.File;
4. import java.io.IOException;
5. import java.util.Collections;
6. import java.util.Enumeration;
7. import java.util.HashMap;
8. import java.util.List;
9. import java.util.Map;
10. import javax.servlet.Filter;
11. import javax.servlet.FilterChain;
12. import javax.servlet.FilterConfig;
13. import javax.servlet.ServletException;
14. import javax.servlet.ServletRequest;
15. import javax.servlet.ServletResponse;
16.
17. import javax.servlet.http.HttpServletRequest;
18. import javax.servlet.http.HttpServletRequestWrapper;
19. import org.apache.commons.fileupload.FileItem;
20. import org.apache.commons.fileupload.FileUploadException;
21. import org.apache.commons.fileupload.disk.DiskFileItemFactory;
22. import org.apache.commons.fileupload.servlet.ServletFileUpload;
23.
24. public class UploadFilter implements Filter {
25.    private int sizeThreshold = -1;
26.    private String repositoryPath;
27.
28.    public void init(FilterConfig config) throws ServletException {
29.       repositoryPath = config.getInitParameter(
30.          "com.corejsf.UploadFilter.repositoryPath");
31.       try {
32.          String paramValue = config.getInitParameter(
33.             "com.corejsf.UploadFilter.sizeThreshold");
34.          if (paramValue != null)
35.             sizeThreshold = Integer.parseInt(paramValue);
36.       }
37.       catch (NumberFormatException ex) {
```

```
38.          ServletException servletEx = new ServletException();
39.          servletEx.initCause(ex);
40.          throw servletEx;
41.       }
42.    }
43.
44.    public void destroy() {
45.    }
46.
47.    public void doFilter(ServletRequest request,
48.       ServletResponse response, FilterChain chain)
49.       throws IOException, ServletException {
50.
51.       if (!(request instanceof HttpServletRequest)) {
52.          chain.doFilter(request, response);
53.          return;
54.       }
55.
56.       HttpServletRequest httpRequest = (HttpServletRequest) request;
57.
58.       boolean isMultipartContent
59.          = ServletFileUpload.isMultipartContent(httpRequest);
60.       if (!isMultipartContent) {
61.          chain.doFilter(request, response);
62.          return;
63.       }
64.
65.       DiskFileItemFactory factory = new DiskFileItemFactory();
66.       if (sizeThreshold >= 0)
67.          factory.setSizeThreshold(sizeThreshold);
68.       if (repositoryPath != null)
69.          factory.setRepository(new File(repositoryPath));
70.       ServletFileUpload upload = new ServletFileUpload(factory);
71.
72.       try {
73.          @SuppressWarnings("unchecked") List<FileItem> items
74.             = (List<FileItem>) upload.parseRequest(httpRequest);
75.          final Map<String, String[]> map = new HashMap<String, String[]>();
76.          for (FileItem item : items) {
77.             String str = item.getString();
78.             if (item.isFormField())
79.                map.put(item.getFieldName(), new String[] { str });
80.             else
81.                httpRequest.setAttribute(item.getFieldName(), item);
82.          }
83.
84.          chain.doFilter(new
85.             HttpServletRequestWrapper(httpRequest) {
86.                public Map<String, String[]> getParameterMap() {
87.                   return map;
88.                }
89.                // Трудоемкие операции... Их следует сделать частью оболочки
90.                public String[] getParameterValues(String name) {
91.                   Map<String, String[]> map = getParameterMap();
92.                   return (String[]) map.get(name);
93.          }
94.          public String getParameter(String name) {
95.             String[] params = getParameterValues(name);
96.             if (params == null) return null;
97.             return params[0];
98.          }
99.          public Enumeration<String> getParameterNames() {
```

```
100.                Map<String, String[]> map = getParameterMap();
101.                return Collections.enumeration(map.keySet());
102.             }
103.          }, response);
104.    } catch (FileUploadException ex) {
105.       ServletException servletEx = new ServletException();
106.       servletEx.initCause(ex);
107.       throw servletEx;
108.    }
109. }
110.}
```

Теперь перейдем к описанию компонента выгрузки. Он поддерживает два атрибута: `value` и `target`. Атрибут `value` обозначает выражение значения, в котором сохраняется содержимое файла. Применение указанного атрибута имеет смысл для коротких файлов. Атрибут `target` чаще всего используется для определения целевого местоположения файла.

Реализация класса `FileUploadRenderer` в листинге 13.2 является несложной. Метод `encodeBegin` подготавливает к отображению элемент HTML. Метод декодирования отыскивает элементы файла, помещаемые фильтром сервлетов в атрибуты запроса, и обрабатывает их в соответствии с тем, что указано в атрибутах тега. Атрибут `target` обозначает имя файла относительно каталога сервера, содержащего корневой каталог веб-приложения.

Наконец, при использовании тега выгрузки файла следует помнить, что в качестве кодировки формы должно быть задано `"multipart/form-data"` (листинг 13.3).

Листинг 13.2. Файл `fileupload/src/java/com/corejsf/UploadRenderer.java`

```
1. package com.corejsf;
2.
3. import java.io.File;
4. import java.io.IOException;
5. import java.io.InputStream;
6. import java.io.UnsupportedEncodingException;
7. import javax.el.ValueExpression;
8. import javax.faces.FacesException;
9. import javax.faces.component.EditableValueHolder;
10. import javax.faces.component.UIComponent;
11. import javax.faces.context.ExternalContext;
12. import javax.faces.context.FacesContext;
13. import javax.faces.context.ResponseWriter;
14. import javax.faces.render.FacesRenderer;
15. import javax.faces.render.Renderer;
16. import javax.servlet.ServletContext;
17. import javax.servlet.http.HttpServletRequest;
18. import org.apache.commons.fileupload.FileItem;
19.
20. @FacesRenderer(componentFamily="javax.faces.Input",
21.    rendererType="com.corejsf.Upload")
22. public class UploadRenderer extends Renderer {
23.    public void encodeBegin(FacesContext context, UIComponent component)
24.       throws IOException {
25.       if (!component.isRendered()) return;
26.       ResponseWriter writer = context.getResponseWriter();
27.
28.       String clientId = component.getClientId(context);
29.
30.       writer.startElement("input", component);
31.       writer.writeAttribute("type", "file", "type");
```

```
32.       writer.writeAttribute("name", clientId, "clientId");
33.       writer.endElement("input");
34.       writer.flush();
35.    }
36.
37.    public void decode(FacesContext context, UIComponent component) {
38.       ExternalContext external = context.getExternalContext();
39.       HttpServletRequest request = (HttpServletRequest) external.getRequest();
40.       String clientId = component.getClientId(context);
41.       FileItem item = (FileItem) request.getAttribute(clientId);
42.
43.       Object newValue;
44.       ValueExpression valueExpr = component.getValueExpression("value");
45.       if (valueExpr != null) {
46.          Class<?> valueType = valueExpr.getType(context.getELContext());
47.          if (valueType == byte[].class) {
48.             newValue = item.get();
49.          }
50.          else if (valueType == InputStream.class) {
51.             try {
52.                newValue = item.getInputStream();
53.             } catch (IOException ex) {
54.                throw new FacesException(ex);
55.             }
56.          }
57.          else {
58.             String encoding = request.getCharacterEncoding();
59.             if (encoding != null)
60.                try {
61.                   newValue = item.getString(encoding);
62.                } catch (UnsupportedEncodingException ex) {
63.                   newValue = item.getString();
64.                }
65.             else
66.                newValue = item.getString();
67.          }
68.          ((EditableValueHolder) component).setSubmittedValue(newValue);
69.          ((EditableValueHolder) component).setValid(true);
70.       }
71.
72.       Object target = component.getAttributes().get("target");
73.
74.       if (target != null) {
75.          File file;
76.          if (target instanceof File)
77.             file = (File) target;
78.          else {
79.             ServletContext servletContext
80.                = (ServletContext) external.getContext();
81.             String realPath = servletContext.getRealPath(target.toString());
82.             file = new File(realPath);
83.          }
84.
85.          try { // Ого! Метод write объявлен с ключевыми словами "throws Exception"
86.             item.write(file);
87.          } catch (Exception ex) {
88.             throw new FacesException(ex);
89.          }
90.       }
91.    }
92. }
```

Листинг 13.3. Файл `fileupload/web/upload/uploadImage.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:corejsf="http://corejsf.com">
<h:head>
   <title>A file upload test</title>
</h:head>
<h:body>
   <h:form enctype="multipart/form-data">
      Upload a photo of yourself:
      <corejsf:upload target="upload/#{user.id}_image.jpg" />
      <h:commandButton value="Submit" action="/next" />
   </h:form>
</h:body>
</html>
```

Отображение гиперкарты

Для реализации клиентской гиперкарты необходимо задать атрибут `usemap` в элементе `h:outputImage`:

```
<h:outputImage value="image location" usemap="#aLabel"/>
```

После этого можно определить карту в коде HTML на странице JSF:

```
<map name="aLabel">
   <area shape="polygon" coords="..." href="...">
   <area shape="rect" coords="..." href="...">
   ...
</map>
```

Однако этот подход не обеспечивает достаточно хорошую интеграцию со средствами навигации JSF. Было бы лучше, если бы области карты действовали как кнопки или ссылки.

В главе 13 учебника по Java EE 5 (`http://java.sun.com/javaee/5/docs/tutorial/doc`) включены пример карты и теги области, позволяющие преодолеть это ограничение.

Для ознакомления с гиперкартой в действии загрузите веб-приложение bookstore6, которое прилагается к указанному учебнику (рис. 13.4). Ниже показано, как используются теги в учебном приложении.

```
<h:graphicImage id="mapImage" url="/template/world.jpg" alt="#{bundle.ChooseLocale}"
      usemap="#worldMap" />
<b:map id="worldMap" current="NAmericas" immediate="true" action="bookstore"
      actionListener="#{localeBean.chooseLocaleFromMap}" >
   <b:area id="NAmerica" value="#{NA}" onmouseover="/template/world_namer.jpg"
      onmouseout="/template/world.jpg" targetImage="mapImage" />
   <b:area id="SAmerica" value="#{SA}" onmouseover="/template/world_samer.jpg"
      onmouseout="/template/world.jpg" targetImage="mapImage" />
      ...
</b:map>
```

Значения области определяются в файле `faces-config.xml`:

```
<managed-bean>
   <managed-bean-name> NA </managed-bean-name>
   <managed-bean-class> com.sun.bookstore6.model.ImageArea </managed-bean-class>
```

```
<managed-bean-scope> application </managed-bean-scope>
    <managed-property>
      <property-name>coords</property-name>
      <value>
53,109,1,110,2,167,19,168,52,149,67,164,67,165,68,167,70,168,72,170,74,172,75,174,77,
175,79,177,81,179,80,179,77,179,81,179,81,178,80,178,82,211,28,238,15,233,15,242,31,
252,36,247,36,246,32,239,89,209,92,216,93,216,100,216,103,218,113,217,116,224,124,221,
128,230,163,234,185,189,178,177,162,188,143,173,79,173,73,163,79,157,64,142,54,139,53,
109
      </value>
    </managed-property>
</managed-bean>
```

Еще один вариант состоит в том, что можно использовать способ, показанный в главе 7. Разместите изображение на кнопке и обрабатывайте координаты x и y на сервере:

```
<h:commandButton image="..." actionListener="..."/>
```

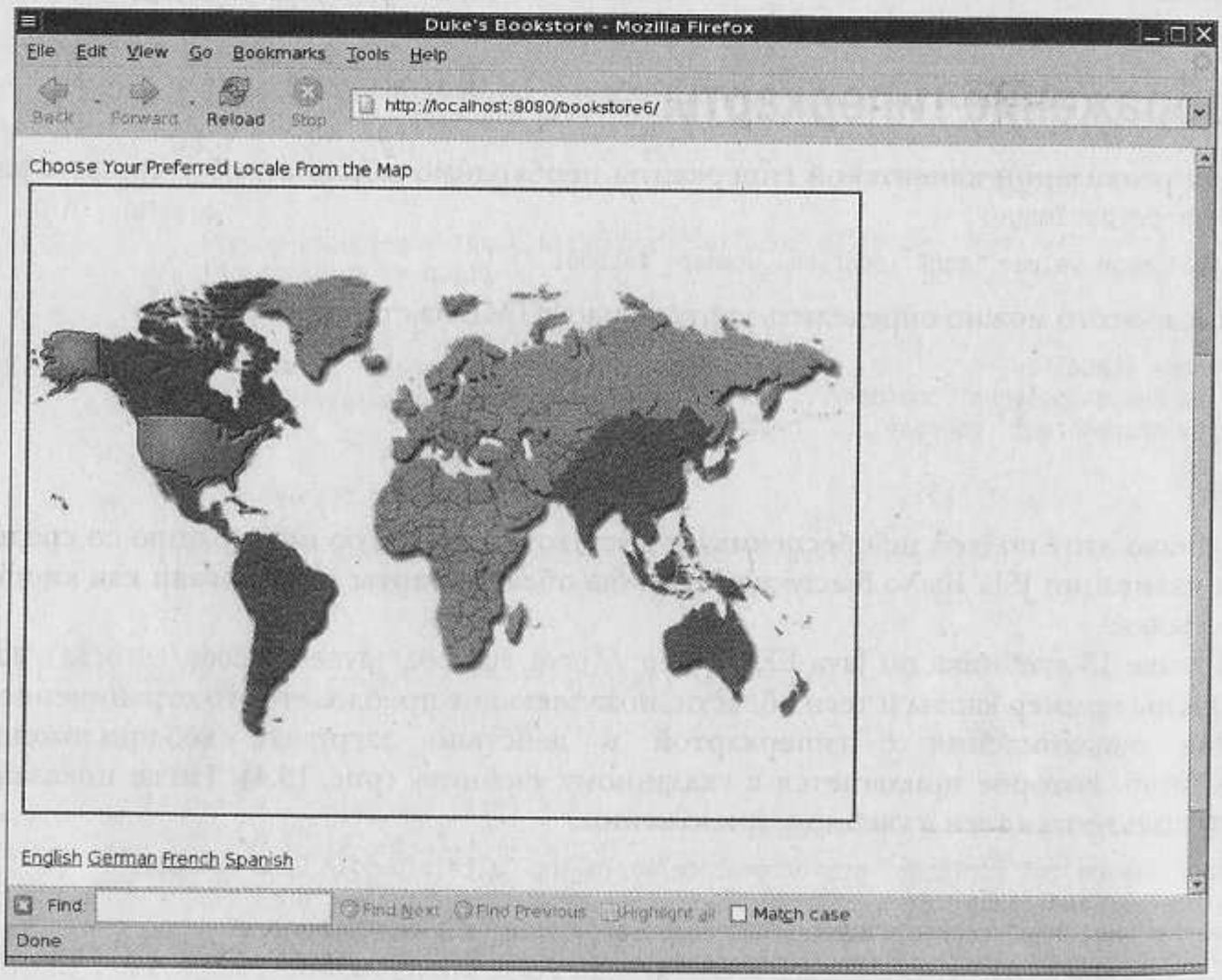


Рис. 13.4. Компонент для примера гиперкарты

Закрепите прослушиватель действий, который возвращает клиентский идентификатор кнопки, присоедините суффиксы .x и .y, после чего обеспечьте поиск значений координат в карте запроса. Эти значения могут обрабатываться любым желаемым способом. При использовании данного подхода необходимо предоставить серверному приложению сведения о геометрии изображения.

Формирование двоичных данных на странице JSF

Иногда возникает необходимость динамически вырабатывать двоичные данные, такие как изображение или файл PDF. В технологии JSF эта задача с трудом поддается решению, поскольку по умолчанию обработчик представлений отправляет текстовый вывод в средство записи, а не в поток. Теоретически было бы возможно заменить обработчик представлений, но гораздо проще воспользоваться вспомогательным сервлетом для выработки двоичных данных. Безусловно, для настройки вывода все еще остается возможность использовать удобные средства JSF, в частности выражения значения. Поэтому необходимо предусмотреть тег JSF, который собирает данные настройки и отправляет их сервлету.

В качестве примера реализуем тег JSF, который создает изображение диаграммы (рис. 13.5). Это изображение содержит данные в формате PNG, которые были сформированы динамически.

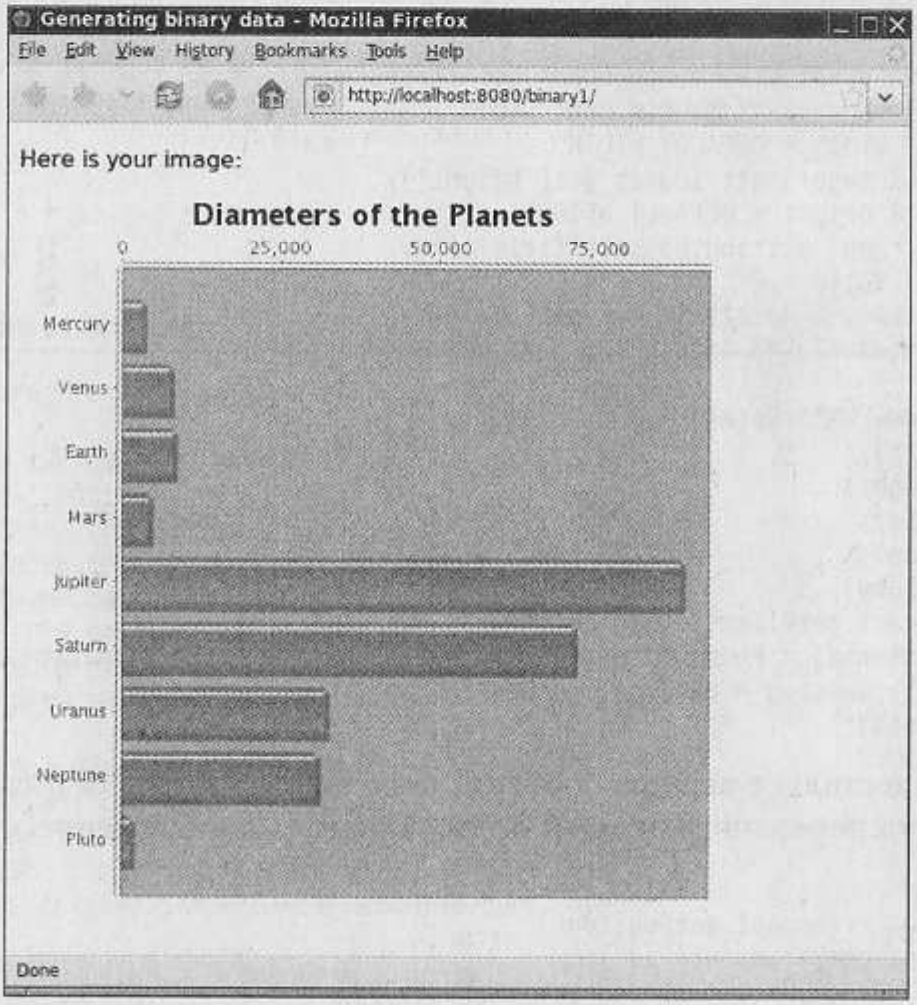


Рис. 13.5. Создание двоичных данных

В листинге 13.4 приведена диаграмма со следующим тегом:

```
<corejsf:chart width="500" height="500"
   title="Diameters of the Planets"
   names="#{planets.names}" values="#{planets.values}"/>
```

где `names` и `values` — выражение значения типов `String[]` и `double[]`. Средство подготовки к отображению, код которого показан в листинге 13.5, формирует тег изображения:

```
<img width="500" height="500" src="/binary/BinaryServlet?id=уникальный идентификатор" />
```

Изображение создается сервлетом BinaryServlet (листинг 13.6). Для настройки сервлета необходимо добавить следующие строки в файл web.xml:

```
<servlet>
   <servlet-name>BinaryServlet</servlet-name>
   <servlet-class>com.corejsf.BinaryServlet</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>BinaryServlet</servlet-name>
   <url-pattern>/BinaryServlet</url-pattern>
</servlet-mapping>
```

Еще один вариант состоит в том, что можно использовать аннотацию:

```
@WebServlet("/BinaryServlet") public class BinaryServlet
```

Безусловно, сервлету должны быть предоставлены данные настройки. Средство подготовки к отображению собирает эти данные из атрибутов компонентов обычным образом, связывает их в виде объекта передачи (листинг 13.8) и помещает объект передачи в карту сеанса:

```
Map<String, Object> attributes = component.getAttributes();
Integer width = (Integer) attributes.get("width");
if (width == null) width = DEFAULT_WIDTH;
Integer height = (Integer) attributes.get("height");
if (height == null) height = DEFAULT_HEIGHT;
String title = (String) attributes.get("title");
if (title == null) title = "";
String[] names = (String[]) attributes.get("names");
double[] values = (double[]) attributes.get("values");

ChartData data = new ChartData();
data.setWidth(width);
data.setHeight(height);
data.setTitle(title);
data.setNames(names);
data.setValues(values);
String id = component.getClientId(context);
ExternalContext external = FacesContext.getCurrentInstance().getExternalContext();
Map<String, Object> session = external.getSessionMap();
session.put(id, data);
```

Сервлет осуществляет выборку объекта передачи из карты сеанса и вызывает метод write объекта передачи, который подготавливает изображение к выводу в поток ответа:

```
HttpSession session = request.getSession();
String id = request.getParameter("id");
BinaryData data = (BinaryData) session.getAttribute(id);

response.setContentType(data.getContentType());
OutputStream out = response.getOutputStream();
data.write(out);
out.close();
```

Чтобы код сервлета оставался достаточно общим, необходимо обеспечить реализацию интерфейса BinaryData классом передачи (листинг 13.7).

На заметку! В целях сокращения объема кода, обеспечивающего формирование двоичных данных, для получения изображения используется библиотека JFreeChart (http://jfree.org/jfreechart). Для этого необходимо добавить файлы jfreechart-*версия*.jar и jcommon-*версия*.jar в каталог WEB-

INF/lib. Если потребуется испытать этот код без библиотеки JFreeChart, то можно воспользоваться классом ChartData из следующего примера.

Тот же подход может быть применен для выработки двоичных данных любого вида. Единственное различие заключается в том, что изменяется код записи данных в выходной поток.

Листинг 13.4. Файл binary1/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html" xmlns:corejsf="http://corejsf.com">
   <h:head>
      <title>Generating binary data</title>
   </h:head>
   <h:body>
      <h:form>
         <p>Here is your image:</p>
         <corejsf:chart width="500" height="500" title="Diameters of the Planets"
                        names="#{planets.names}" values="#{planets.values}"/>
      </h:form>
   </h:body>
</html>
```

Листинг 13.5. Файл binary1/src/java/com/corejsf/ChartRenderer.java

```
package com.corejsf;

import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIComponent;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.FacesRenderer;
import javax.faces.render.Renderer;

@FacesRenderer(componentFamily="javax.faces.Output",
      rendererType="com.corejsf.Chart")
public class ChartRenderer extends Renderer {
   private static final int DEFAULT_WIDTH = 200;
   private static final int DEFAULT_HEIGHT = 200;

   public void encodeBegin(FacesContext context, UIComponent component)
      throws IOException {
      if (!component.isRendered()) return;

      Map<String, Object> attributes = component.getAttributes();
      Integer width = toInteger(attributes.get("width"));
      if (width == null) width = DEFAULT_WIDTH;
      Integer height = toInteger(attributes.get("height"));
      if (height == null) height = DEFAULT_HEIGHT;
      String title = (String) attributes.get("title");
      if (title == null) title = "";
      String[] names = (String[]) attributes.get("names");
      double[] values = (double[]) attributes.get("values");
      if (names == null || values == null) return;

```

```
33.       ChartData data = new ChartData();
34.       data.setWidth(width);
35.       data.setHeight(height);
36.       data.setTitle(title);
37.       data.setNames(names);
38.       data.setValues(values);
39.
40.       String id = component.getClientId(context);
41.       ExternalContext external
42.             = FacesContext.getCurrentInstance().getExternalContext();
43.       Map<String, Object> session = external.getSessionMap();
44.       session.put(id, data);
45.
46.       ResponseWriter writer = context.getResponseWriter();
47.       writer.startElement("img", component);
48.
49.       writer.writeAttribute("width", width, null);
50.       writer.writeAttribute("height", height, null);
51.       String path = external.getRequestContextPath();
52.       writer.writeAttribute("src", path + "/BinaryServlet?id=" + id, null);
53.       writer.endElement("img");
54.
55.       context.responseComplete();
56.    }
57.
58.    private static Integer toInteger(Object value) {
59.       if (value == null) return null;
60.       if (value instanceof Number) return ((Number) value).intValue();
61.       if (value instanceof String) return Integer.parseInt((String) value);
62.       throw new IllegalArgumentException("Cannot convert " + value);
63.    }
64. }
```

Листинг 13.6. Файл binary1/src/java/com/corejsf/BinaryServlet.java

```
1.  package com.corejsf;
2.
3.  import java.io.IOException;
4.  import java.io.OutputStream;
5.
6.  import javax.servlet.ServletException;
7.  import javax.servlet.http.HttpServlet;
8.  import javax.servlet.http.HttpServletRequest;
9.  import javax.servlet.http.HttpServletResponse;
10. import javax.servlet.http.HttpSession;
11.
12. public class BinaryServlet extends HttpServlet {
13.    protected void processRequest(HttpServletRequest request,
14.          HttpServletResponse response)
15.          throws ServletException, IOException {
16.       HttpSession session = request.getSession();
17.       String id = request.getParameter("id");
18.       BinaryData data = (BinaryData) session.getAttribute(id);
19.
20.       response.setContentType(data.getContentType());
21.       OutputStream out = response.getOutputStream();
22.       data.write(out);
23.       out.close();
24.    }
25.
26.    protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
27.             throws ServletException, IOException {
28.         processRequest(request, response);
29.     }
30.
31.     protected void doPost(HttpServletRequest request, HttpServletResponse response)
32.             throws ServletException, IOException {
33.         processRequest(request, response);
34.     }
35. }
```

Листинг 13.7. Файл binary1/src/java/com/corejsf/BinaryData.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.io.OutputStream;
5.
6. public interface BinaryData {
7.    String getContentType();
8.    void write(OutputStream out) throws IOException;
9. }
```

Листинг 13.8. Файл binary1/src/java/com/corejsf/ChartData.java

```
 1. package com.corejsf;
 2.
 3. import java.io.IOException;
 4. import java.io.OutputStream;
 5.
 6. import org.jfree.chart.ChartFactory;
 7. import org.jfree.chart.ChartUtilities;
 8. import org.jfree.chart.JFreeChart;
 9. import org.jfree.chart.plot.PlotOrientation;
10. import org.jfree.data.category.DefaultCategoryDataset;
11.
12. public class ChartData implements BinaryData {
13.    private int width, height;
14.    private String title;
15.    private String[] names;
16.    private double[] values;
17.
18.    private static final int DEFAULT_WIDTH = 200;
19.    private static final int DEFAULT_HEIGHT = 200;
20.
21.     public ChartData() {
22.        width = DEFAULT_WIDTH;
23.        height = DEFAULT_HEIGHT;
24.     }
25.
26.     public void setWidth(int width) {
27.        this.width = width;
28.     }
29.
30.     public void setHeight(int height) {
31.        this.height = height;
32.     }
33.
34.     public void setTitle(String title) {
35.        this.title = title;
36.     }
```

```
37.
38.     public void setNames(String[] names) {
39.        this.names = names;
40.     }
41.
42.     public void setValues(double[] values) {
43.        this.values = values;
44.     }
45.
46.     public String getContentType() {
47.        return "image/png";
48.     }
49.
50.     public void write(OutputStream out) throws IOException {
51.        DefaultCategoryDataset dataset = new DefaultCategoryDataset();
52.        for (int i = 0; i < names.length; i++)
53.           dataset.addValue(values[i], "", names[i]);
54.        JFreeChart chart = ChartFactory.createBarChart(
55.           title, // Заголовок
56.           "", // Метка оси доменов
57.           "", // Метка оси диапазонов
58.           dataset,
59.           PlotOrientation.HORIZONTAL,
60.           false, // Условные обозначения
61.           false, // Всплывающие подсказки
62.           false // URL
63.        );
64.
65.        ChartUtilities.writeChartAsPNG(out, chart, width, height);
66.     }
67. }
```

Возможна также выработка двоичных данных непосредственно из кода JSF (без сервлета). Но при этом приходится соблюдать чрезвычайные меры предосторожности в отношении синхронизации процессов и захватывать выходной поток сервлета до того, как реализация JSF начнет записывать ответ. Захват выходного потока сервлета не может происходить в средстве подготовки к отображению компонента. Компонент JSF вносит свой вклад в вывод страницы, но не замещает весь вывод.

Поэтому взамен мы устанавливаем прослушиватель фаз, который активизируется после фазы восстановления представления. Он записывает двоичные данные. а затем вызывает метод responseComplete для пропуска других фаз:

```
public class BinaryPhaseListener implements PhaseListener {
   public PhaseId getPhaseId() {
      return PhaseId.RESTORE_VIEW;
   }
   ...
   public void afterPhase(PhaseEvent event) {
      if (!event.getFacesContext().getViewRoot().getViewId()
            .startsWith("/binary")) return;
      HttpServletResponse servletResponse
         = (HttpServletResponse) external.getResponse();
      servletResponse.setContentType(data.getContentType());
      OutputStream out = servletResponse.getOutputStream();
      писать данные в out
      context.responseComplete();
   }
}
```

Действие фильтра осуществляется только применительно к идентификаторам представления, которые начинаются с префикса /binary. Как и в решении на основе сервлета, ключ для объекта передачи данных включен в качестве параметра GET.

Чтобы произошел запуск фильтра, URL изображения должен представлять собой допустимый URL JSF, такой как *приложение*/binary.faces?id=key или *приложение*/faces/binary?id=key. Какой именно тип должен быть задан, зависит от отображения сервлета Faces. Средство подготовки к отображению получает правильный формат от метода getActionURL обработчика представлений:

```
ViewHandler handler = context.getApplication().getViewHandler();
String url = handler.getActionURL(context, "/binary");
```

Прослушиватель фаз показан в листинге 13.9. Для установки прослушивателя требуется следующий элемент в файле faces-config.xml:

```
<lifecycle>
   <phase-listener>com.corejsf.BinaryPhaseListener</phase-listener>
</lifecycle>
```

На заметку! В рассматриваемом примере кода диаграмма формируется с нуля, без использования библиотеки JFreeChart, а это может служить иллюстрацией того, что сам способ выработки двоичных данных не имеет значения.

Листинг 13.9. Файл binary2/src/java/com/corejsf/BinaryPhaseListener.java

```
 1. package com.corejsf;
 2.
 3. import java.io.IOException;
 4. import java.io.OutputStream;
 5. import java.util.Map;
 6. import javax.faces.FacesException;
 7. import javax.faces.context.ExternalContext;
 8. import javax.faces.context.FacesContext;
 9. import javax.faces.event.PhaseEvent;
10. import javax.faces.event.PhaseId;
11. import javax.faces.event.PhaseListener;
12. import javax.servlet.http.HttpServletResponse;
13.
14. public class BinaryPhaseListener implements PhaseListener {
15.    public static final String BINARY_PREFIX = "/binary";
16.
17.    public static final String DATA_ID_PARAM = "id";
18.
19.    public PhaseId getPhaseId() {
20.       return PhaseId.RESTORE_VIEW;
21.    }
22.
23.    public void beforePhase(PhaseEvent event) {
24.    }
25.
26.    public void afterPhase(PhaseEvent event) {
27.       if (!event.getFacesContext().getViewRoot().getViewId().startsWith(
28.             BINARY_PREFIX))
29.          return;
30.
31.       FacesContext context = event.getFacesContext();
32.       ExternalContext external = context.getExternalContext();
33.
34.       String id = (String) external.getRequestParameterMap().get(DATA_ID_PARAM);
```

```
35.       HttpServletResponse servletResponse =
36.             (HttpServletResponse) external.getResponse();
37.       try {
38.          Map<String, Object> session = external.getSessionMap();
39.        BinaryData data = (BinaryData) session.get(id);
40.        if (data != null) {
41.           servletResponse.setContentType(data.getContentType());
42.           OutputStream out = servletResponse.getOutputStream();
43.           data.write(out);
44.        }
45.     } catch (IOException ex) {
46.        throw new FacesException(ex);
47.     }
48.     context.responseComplete();
49.   }
50. }
```

Одновременное отображение крупного набора данных по одной странице

Как было показано в главе 6, предусмотрена возможность добавлять полосы прокрутки к таблице. Но если таблица очень большая, то нежелательно отправлять ее клиенту полностью. Загрузка таблицы потребует много времени, и велика вероятность того, что пользователь приложения захочет ознакомиться только с первыми несколькими строками.

Стандартным пользовательским интерфейсом для перемещения по строкам крупной таблицы является средство постраничного просмотра, которое определяет набор ссылок на каждую страницу таблицы, на следующую и предыдущую страницы, а если количество страниц велико — на следующий и предыдущий пакеты страниц. На рис. 13.6 показано средство постраничного просмотра, которое позволяет выполнять прокрутку по крупному набору данных — по данным о предопределенных часовых поясах, полученных с помощью метода `java.util.TimeZone.getAvailableIDs()`.

К сожалению, компонент средства постраничного просмотра не предусмотрен в технологии JSF. Однако написание такого компонента не вызывает сложностей, и ниже приведен код, который читатель может использовать непосредственно или модифицировать для включения в свои собственные приложения.

Средства постраничного просмотра воспринимаются как неразрывно связанные с таблицами данных. Для их использования задаются идентификатор таблицы данных, число страниц, отображаемых средством постраничного просмотра, и стили для выбранных и невыбранных ссылок. Например:

```
<h:dataTable id="timezones" value="#{bb.data}" var="row" rows="10">
   ...
</h:dataTable>
<corejsf:pager dataTableId="timezones" showpages="20"
selectedStyleClass="currentPage"/>
```

Предположим, пользователь щелкает на ссылке ">", чтобы перейти на следующую страницу. Средство постраничного просмотра определяет местонахождение таблицы данных и обновляет ее свойство `first`, складывая его значение со значением свойства `rows`. Этот код можно найти в коде метода декодирования компонента `PagerRenderer` в листинге 13.10.

Метод кодирования немного более сложный: он формирует набор ссылок. По аналогии с commandLink, щелчок на ссылке активизирует код JavaScript, который задает значение в скрытом поле и производит отправку формы.

В листинге 13.11 показана страница index.xhtml, с помощью которой формируется таблица и средство постраничного просмотра. В листинге 13.12 приведен код простейшего вспомогательного бина.

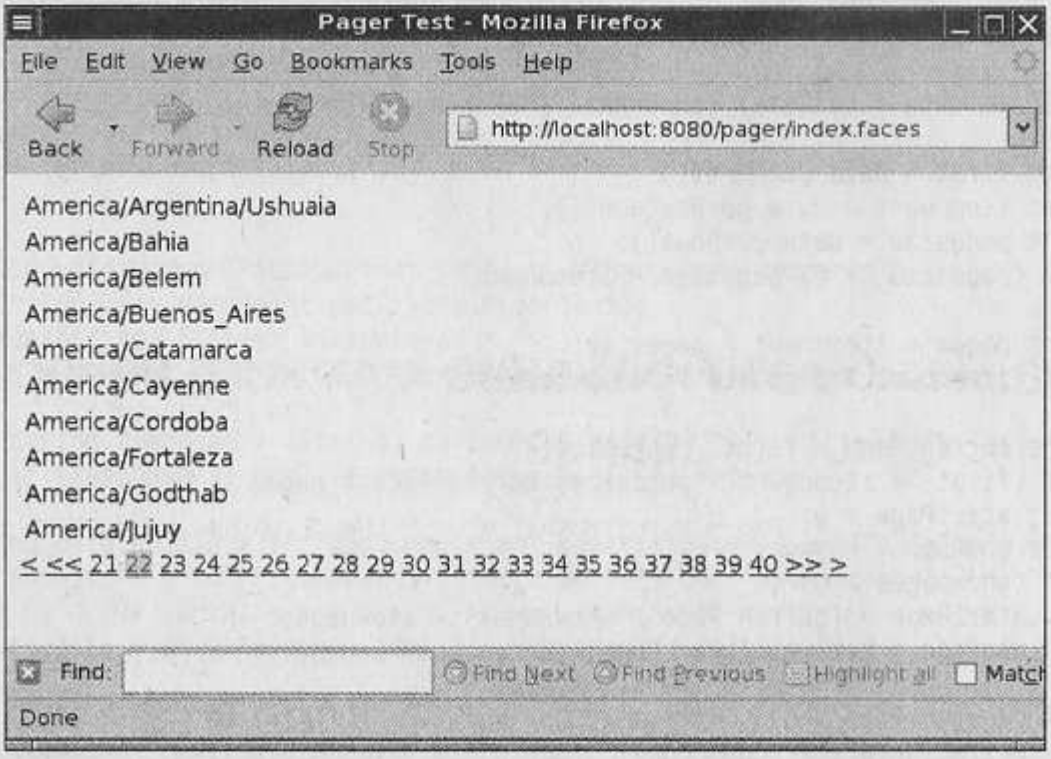


Рис. 13.6. Таблица со средством постраничного просмотра

На заметку! Подобные функциональные возможности предоставляет компонент dataScroller из библиотеки Apache Tomahawk.

Листинг 13.10. Файл pager/src/java/com/corejsf/PagerRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Map;
5. import javax.faces.component.UIComponent;
6. import javax.faces.component.UIData;
7. import javax.faces.component.UIForm;
8. import javax.faces.context.FacesContext;
9. import javax.faces.context.ResponseWriter;
10. import javax.faces.render.FacesRenderer;
11. import javax.faces.render.Renderer;
12.
13. @FacesRenderer(componentFamily="javax.faces.Command",
14.    rendererType="com.corejsf.Pager")
15. public class PagerRenderer extends Renderer {
16.    public void encodeBegin(FacesContext context, UIComponent component)
17.       throws IOException {
18.       String id = component.getClientId(context);
19.       UIComponent parent = component;
20.       while (!(parent instanceof UIForm)) parent = parent.getParent();
21.       String formId = parent.getClientId(context);
```

```
22.
23.       ResponseWriter writer = context.getResponseWriter();
24.
25.       String styleClass = (String) component.getAttributes().get("styleClass");
26.       String selectedStyleClass
27.          = (String) component.getAttributes().get("selectedStyleClass");
28.       String dataTableId = (String) component.getAttributes().get("dataTableId");
29.       int showpages = toInt(component.getAttributes().get("showpages"));
30.
31.       // Найти компонент с указанным идентификатором
32.
33.       UIData data = (UIData) component.findComponent(dataTableId);
34.
35.       int first = data.getFirst();
36.       int itemcount = data.getRowCount();
37.       int pagesize = data.getRows();
38.       if (pagesize <= 0) pagesize = itemcount;
39.
40.       int pages = itemcount / pagesize;
41.       if (itemcount % pagesize != 0) pages++;
42.
43.       int currentPage = first / pagesize;
44.       if (first >= itemcount - pagesize) currentPage = pages - 1;
45.       int startPage = 0;
46.       int endPage = pages;
47.       if (showpages > 0) {
48.          startPage = (currentPage / showpages) * showpages;
49.          endPage = Math.min(startPage + showpages, pages);
50.       }
51.       if (currentPage > 0)
52.          writeLink(writer, component, formId, id, "<", styleClass);
53.
54.       if (startPage > 0)
55.          writeLink(writer, component, formId, id, "<<", styleClass);
56.
57.       for (int i = startPage; i < endPage; i++) {
58.          writeLink(writer, component, formId, id, "" + (i + 1),
59.             i == currentPage ? selectedStyleClass : styleClass);
60.       }
61.
62.       if (endPage < pages)
63.          writeLink(writer, component, formId, id, ">>", styleClass);
64.
65.       if (first < itemcount - pagesize)
66.          writeLink(writer, component, formId, id, ">", styleClass);
67.
68.       // Скрытое поле для хранения результата
69.       writeHiddenField(writer, component, id);
70.    }
71.
72.    private void writeLink(ResponseWriter writer, UIComponent component,
73.       String formId, String id, String value, String styleClass)
74.       throws IOException {
75.       writer.writeText(" ", null);
76.       writer.startElement("a", component);
77.       writer.writeAttribute("href", "#", null);
78.       writer.writeAttribute("onclick", onclickCode(formId, id, value), null);
79.       if (styleClass != null)
80.          writer.writeAttribute("class", styleClass, "styleClass");
81.       writer.writeText(value, null);
82.       writer.endElement("a");
83.    }
```

```
84.
85.    private String onclickCode(String formId, String id, String value) {
86.       return new StringBuilder().append("document.forms['")
87.          .append(formId).append("']['")
88.          .append(id).append("'].value='").append(value).append("'; document.forms['")
89.          .append(formId).append("'].submit(); return false;").toString();
90.    }
91.
92.    private void writeHiddenField(ResponseWriter writer, UIComponent component,
93.       String id) throws IOException {
94.       writer.startElement("input", component);
95.       writer.writeAttribute("type", "hidden", null);
96.       writer.writeAttribute("name", id, null);
97.       writer.endElement("input");
98.    }
99.
100.   public void decode(FacesContext context, UIComponent component) {
101.      String id = component.getClientId(context);
102.      Map<String, String> parameters
103.         = context.getExternalContext().getRequestParameterMap();
104.
105.       String response = (String) parameters.get(id);
106.       if (response == null || response.equals("")) return;
107.
108.   String dataTableId = (String) component.getAttributes().get("dataTableId");
109.   int showpages = toInt(component.getAttributes().get("showpages"));
110.
111.   UIData data = (UIData) component.findComponent(dataTableId);
112.
113.   int first = data.getFirst();
114.   int itemcount = data.getRowCount();
115.   int pagesize = data.getRows();
116.   if (pagesize <= 0) pagesize = itemcount;
117.
118.   if (response.equals("<")) first -= pagesize;
119.   else if (response.equals(">")) first += pagesize;
120.   else if (response.equals("<<")) first -= pagesize * showpages;
121.   else if (response.equals(">>")) first += pagesize * showpages;
122.   else {
123.      int page = Integer.parseInt(response);
124.      first = (page - 1) * pagesize;
125.   }
126.   if (first + pagesize > itemcount) first = itemcount - pagesize;
127.   if (first < 0) first = 0;
128.   data.setFirst(first);
129. }
130.
131. private static int toInt(Object value) {
132.    if (value == null) return 0;
133.    if (value instanceof Number) return ((Number) value).intValue();
134.    if (value instanceof String) return Integer.parseInt((String) value);
135.    throw new IllegalArgumentException("Cannot convert " + value);
136. }
137.}
```

Листинг 13.11. Файл pager/web/index.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
```

```
5.       xmlns:h="http://java.sun.com/jsf/html"
6.       xmlns:ui="http://java.sun.com/jsf/facelets"
7.       xmlns:corejsf="http://corejsf.com">
8.    <h:head>
9.       <h:outputStylesheet library="css" name="styles.css"/>
10.      <title>Pager Test</title>
11.   </h:head>
12.   <h:body>
13.      <ui:debug/>
14.      <h:form>
15.         <h:dataTable id="timezones" value="#{tz.data}" var="row" rows="10">
16.            <h:column>#{row}</h:column>
17.         </h:dataTable>
18.         <corejsf:pager dataTableId="timezones" showpages="20"
19.                        selectedStyleClass="currentPage"/>
20.      </h:form>
21.   </h:body>
22. </html>
```

Листинг 13.12. Файл pager/src/java/com/corejsf/TimeZoneBean.java

```
1. package com.corejsf;
2.
3. import javax.inject.Named;
4.    // или import javax.faces.bean.ManagedBean;
5. import javax.enterprise.context.RequestScoped;
6.    // или import javax.faces.bean.RequestScoped;
7.
8. @Named("tz") // или @ManagedBean(name="tz")
9. @RequestScoped
10. public class TimeZoneBean {
11.    private String[] data = java.util.TimeZone.getAvailableIDs();
12.    public String[] getData() { return data; }
13. }
```

Формирование всплывающего окна

Базовый метод для всплывающего окна является несложным. Используются следующие вызовы JavaScript:

```
popup = window.open(url, name, features);
popup.focus();
```

Параметр `features` представляет собой строку, такую как

```
"height=300,width=200,toolbar=no,menubar=no"
```

Всплывающее окно должно отображаться после того, как пользователь щелкнет на кнопке или ссылке.

Необходимо закрепить функцию за обработчиком `onclick` кнопки или ссылки и обеспечить возврат из этой функции значения `false`, чтобы браузер не отправлял форму и не переходил по ссылке. Например:

```
<h:commandButton value="..." onclick="doPopup(this); return false;"/>
```

Функция `doPopup` содержит команды JavaScript для формирования всплывающего окна. Она содержится в теге сценария в заголовке страницы.

Но, если возникает необходимость передавать данные между главным и всплывающим окном, обнаруживаются определенные сложности.

Ниже рассматривается конкретный пример. На рис. 13.7 показана страница со всплывающим окном, в котором отображается список штатов США или провинций Канады, в зависимости от установки переключателей. Этот список формируется с помощью вспомогательного бина на сервере.

При этом возникает необходимость передать вспомогательному бину сведения о том, какой штат был выбран. Но дело в том, что ко времени запроса пользователем всплывающего окна форма еще не отправлена обратно на сервер. Ниже показаны два решения этой задачи; каждое из них представляет интерес само по себе и может подсказать идеи по решению аналогичных проблем.

Первое решение предусматривает передачу параметров выбора для URL всплывающего окна:

```
window.open("popup.faces?country=" + country[i].value, "popup", features);
```

На странице `popup.faces` выборка значения параметра запроса страны осуществляется как `param.country`:

```
<h:dataTable value="#{bb.states[param.country]}" var="state">
```

В данном случае использование свойства `states` вспомогательного бина `bb` приводит к получению карты, индексом которой служит код страны.

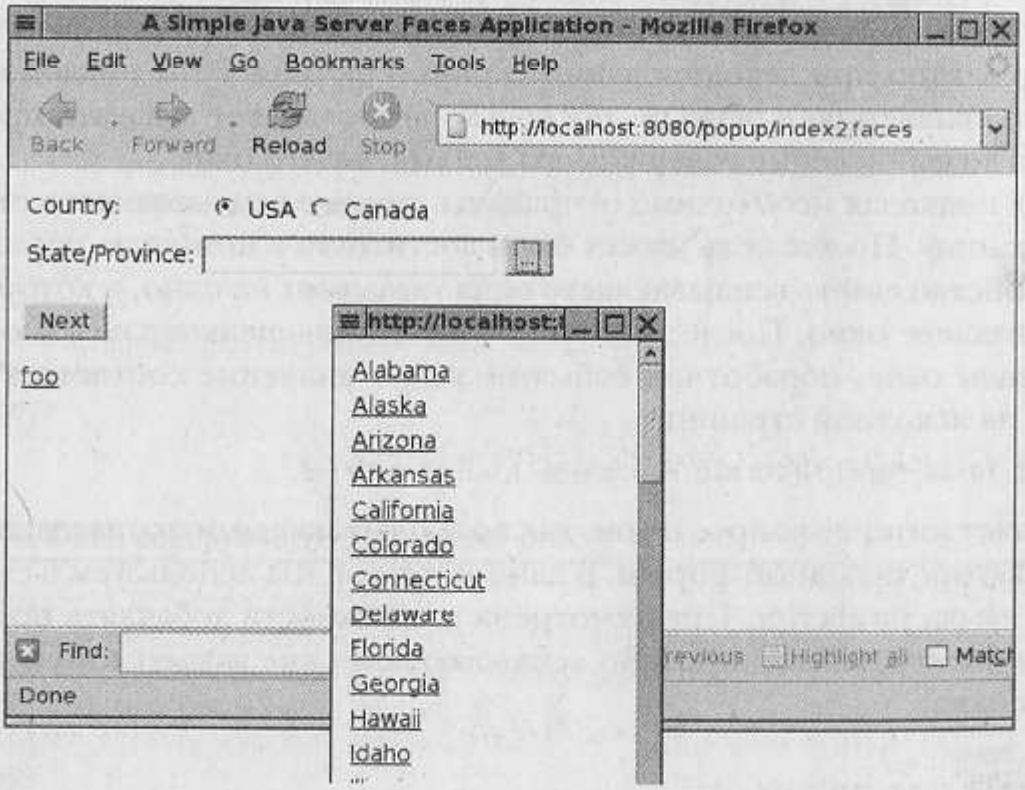


Рис. 13.7. Формирование всплывающего окна для выбора штата или провинции

Второе решение (предложенное Мэрион Басс (Marion Bass) и Сергеем Смирновым (Sergey Smirnov)) является более сложным, но вместе с тем более мощным. Этот способ предусматривает вначале создание всплывающего окна как пустого, а затем заполнение его ответом на команду JSF.

Команда JSF выдается формой, содержащей скрытое поле и невидимую ссылку:

```
<h:form id="hidden" target="popup">
   <h:inputHidden id="country" value="#{bb.country}"/>
   <h:commandLink id="go" action="showStates"/>
</h:form>
```

Следует отметить несколько деталей.

- Адресат формы имеет то же имя, что и всплывающее окно. Поэтому браузер показывает результаты действия в этом окне.
- Скрытое поле страны заполняется перед отправкой формы. При этом задается выражение значения `bb.country`. Это позволяет возвратить с помощью вспомогательного бина соответствующий список штатов или провинций.
- Атрибут `action` командной ссылки используется обработчиком навигации для выбора страницы JSF, которая формирует содержимое всплывающего окна.

Функция `doPopup` инициализирует скрытое поле и активизирует действие ссылки:

```
document.getElementById("hidden:country").value = country[i].value;
document.getElementById("hidden:go").onclick(null);
```

Значение выбранного штата или провинции передается в скрытое поле. После отправки скрытой формы это значение сохраняется во вспомогательном бине.

В рассматриваемом решении страница JSF для всплывающего окна является более простой. Таблица штатов или провинций заполняется с помощью вызова свойства бина:

```
<h:dataTable value="#{bb.statesForCountry}" var="state">
```

При обработке свойства `statesForCountry` учитывается значение свойства `country`, которое было задано при декодировании скрытой формы. Этот подход является более гибким по сравнению с первым, поскольку он позволяет задавать произвольные свойства бина до вычисления содержимого всплывающего окна.

При обоих подходах необходимо отправлять данные всплывающего окна назад на исходную страницу. Но эта цель может быть достигнута с помощью несложного кода JavaScript. Свойство `opener` всплывающего окна указывает на окно, в котором было открыто всплывающее окно. После того как пользователь щелкнет на одной из ссылок во всплывающем окне, обработчик событий задает значение соответствующего текстового поля на исходной странице:

```
opener.document.forms[formId][formId + ":state"].value = value;
```

Представляет интерес вопрос о том, как во всплывающее окно поступают сведения об идентификаторе исходной формы. В данном случае мы используем в своих интересах гибкость языка JavaScript. Предусмотрена возможность добавлять поля экземпляра к любому объекту динамически. Во всплывающем окне задается поле `openerFormId` при создании окна:

```
popup = window.open(...);
popup.openerFormId = source.form.id;
```

После того как появляется возможность модифицировать переменные формы, осуществляется их выборка из всплывающего окна:

```
var formId = window.openerFormId;
```

Все эти приемы необходимо знать тем, кому приходится работать со всплывающими окнами. Два описанных выше подхода демонстрируются в следующем примере. В файлах `technique1.xhtml`, `popup1.xhtml` и `popup1.js` из листингов 13.13–13.15 показан первый подход, в котором используется параметр запроса для настройки всплывающей страницы.

Файлы technique2.xhtml, popup2.xhtml и popup2.js из листингов 13.16–13.18 демонстрируют второй подход, который предусматривает заполнение всплывающей страницы результатами действия JSF. В листинге 13.19 показан вспомогательный бин.

Листинг 13.13. Файл popup/web/technique1.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
   <h:head>
      <h:outputScript library="javascript" name="popup1.js"/>
      <title>Popup window technique 1</title>
   </h:head>
   <h:body>
      <h:form>
         <table>
            <tr>
               <td>Country:</td>
               <td><h:selectOneRadio id="country" value="#{bb.country}">
                  <f:selectItem itemLabel="USA" itemValue="USA"/>
                  <f:selectItem itemLabel="Canada" itemValue="Canada"/>
               </h:selectOneRadio></td>
            </tr>
            <tr>
               <td>State/Province:</td>
               <td><h:inputText id="state" value="#{bb.state}"/></td>
               <td><h:commandButton value="..."
               onclick="doPopup(this); return false;"/></td>
            </tr>
         </table>
         <p><h:commandButton value="Next" action="index"/></p>
      </h:form>
   </h:body>
</html>
```

Листинг 13.14. Файл popup/web/popup1.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputScript library="javascript" name="popup1.js"/>
      <title>Select a state/province</title>
   </h:head>
   <h:body>
      <h:form>
         <h:dataTable value="#{bb.states[param.country]}" var="state">
            <h:column>
               <h:outputLink value="#" onclick="doSave('#{state}');">
                  #{state}
               </h:outputLink>
            </h:column>
         </h:dataTable>
      </h:form>
```

```
20.    </h:body>
21. </html>
```

Листинг 13.15. Файл popup/web/resources/javascript/popup1.js

```
 1. function doPopup(source) {
 2.    country = source.form[source.form.id + ":country"];
 3.    for ( var i = 0; i < country.length; i++) {
 4.       if (country[i].checked) {
 5.          popup = window.open("popup1.xhtml?country="
 6.             + country[i].value, "popup",
 7.             "height=300,width=200,toolbar=no,menubar=no,"
 8.             + "scrollbars=yes");
 9.          popup.openerFormId = source.form.id;
10.          popup.focus();
11.       }
12.    }
13. }
14.
15. function doSave(value) {
16.    var formId = window.openerFormId;
17.    opener.document.forms[formId][formId + ":state"].value = value;
18.    window.close();
19. }
```

Листинг 13.16. Файл popup/web/technique2.xhtml

```
 1. <?xml version="1.0" encoding="UTF-8"?>
 2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4. <html xmlns="http://www.w3.org/1999/xhtml"
 5.       xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://java.sun.com/jsf/core">
 6.    <h:head>
 7.       <h:outputScript library="javascript" name="popup1.js"/>
 8.       <title>Popup window technique 2</title>
 9.    </h:head>
10.    <h:body>
11.       <h:form>
12.          <table>
13.             <tr>
14.                <td>Country:</td>
15.                <td><h:selectOneRadio id="country" value="#{bb.country}">
16.                   <f:selectItem itemLabel="USA" itemValue="USA"/>
17.                   <f:selectItem itemLabel="Canada" itemValue="Canada"/>
18.                </h:selectOneRadio></td>
19.             </tr>
20.             <tr>
21.                <td>State/Province:</td>
22.                <td><h:inputText id="state" value="#{bb.state}"/></td>
23.                <td><h:commandButton value="..."
24.                onclick="doPopup(this); return false;"/></td>
25.             </tr>
26.          </table>
27.          <p><h:commandButton value="Next" action="index"/></p>
28.       </h:form>
29.
30.       <!-- Эта скрытая форма передает запрос во всплывающее окно -->
31.       <h:form id="hidden" target="popup">
32.          <h:inputHidden id="country" value="#{bb.country}"/>
33.          <h:commandLink id="go" action="popup2"/>
```

```
34.       </h:form>
35.    </h:body>
36. </html>
```

Листинг 13.17. Файл popup/web/popup2.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:h="http://java.sun.com/jsf/html">
6.    <h:head>
7.       <h:outputScript library="javascript" name="popup1.js"/>
8.       <title>Select a state/province</title>
9.    </h:head>
10.   <h:body>
11.      <h:form>
12.         <h:dataTable value="#{bb.statesForCountry}" var="state">
13.            <h:column>
14.               <h:outputLink value="#" onclick="doSave('#{state}');">
15.                  #{state}
16.               </h:outputLink>
17.            </h:column>
18.         </h:dataTable>
19.      </h:form>
20.   </h:body>
21. </html>
```

Листинг 13.18. Файл popup/web/resources/javascript/popup2.js

```
1. function doPopup(source) {
2.    country = source.form[source.form.id + ":country"];
3.    for (var i = 0; i < country.length; i++) {
4.       if (country[i].checked) {
5.       popup = window.open("", "/faces/popup2.xhtml",
6.          "height=300,width=200,toolbar=no,menubar=no,scrollbars=yes");
7.          popup.openerFormId = source.form.id;
8.          popup.focus();
9.          document.getElementById("hidden:country").value = country[i].value;
10.         document.getElementById("hidden:go").onclick(null);
11.      }
12.   }
13. }
14.
15. function doSave(value) {
16.    var formId = window.openerFormId;
17.    opener.document.forms[formId][formId + ":state"].value = value;
18.    window.close();
19. }
```

Листинг 13.19. Файл popup/src/java/com/corejsf/BackingBean.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.util.HashMap;
5. import java.util.Map;
6.
7. import javax.inject.Named;
8.    // или import javax.faces.bean.ManagedBean;
```

```
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;

@Named("bb") // или @ManagedBean(name="bb")
@SessionScoped
public class BackingBean implements Serializable {
   private String country = "USA";
   private String state = "California";
   private static Map<String, String[]> states;

   public String getCountry() { return country; }
   public void setCountry(String newValue) { country = newValue; }

   public String getState() { return state; }
   public void setState(String newValue) { state = newValue; }

   public Map<String, String[]> getStates() { return states; }

   public String[] getStatesForCountry() { return (String[]) states.get(country); }

   static {
      states = new HashMap<String, String[]>();
      states.put("USA",
         new String[] {
            "Alabama", "Alaska", "Arizona", "Arkansas", "California",
            "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
            "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa", "Kansas",
            "Kentucky", "Louisiana", "Maine", "Maryland", "Massachusetts",
            "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana",
            "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico",
            "New York", "North Carolina", "North Dakota", "Ohio", "Oklahoma",
            "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
            "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
            "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
         });

      states.put("Canada",
         new String[] {
            "Alberta", "British Columbia", "Manitoba", "New Brunswick",
            "Newfoundland and Labrador", "Northwest Territories",
            "Nova Scotia", "Nunavut", "Ontario", "Prince Edward Island",
            "Quebec", "Saskatchewan", "Yukon"
         });
   }
}
```

Выборочное отображение и сокрытие частей страницы

Задача отображения или сокрытия частей страницы в зависимости от некоторого условия возникает очень часто. В качестве примера можно указать, что если пользователь не вошел в систему, то должны быть показаны поля ввода для имени пользователя и пароля. Но если пользователь уже зарегистрирован в системе, то необходимо отображать имя пользователя и кнопку выхода из системы.

Было бы расточительным проектирование двух отдельных страниц, которые различаются лишь этими незначительными деталями. Вместо этого имеет смысл включить все компоненты на одну страницу и отображать их выборочно.

Эту задачу можно решить с помощью конструкции c:if или c:choose библиотеки JSTL. Те, кто предпочитает не смешивать теги JSF и JSTL, могут без особых сложностей достичь того же эффекта с помощью исключительно технологии JSF.

Если требуется разрешать или запрещать отображение одного компонента (или такого контейнера, как группа панели), то можно воспользоваться свойством rendered:

```
<h:panelGroup rendered="#{userBean.loggedIn}">...</h:panelGroup>
```

А для переключения между двумя наборами компонентов можно использовать взаимно дополняющие атрибуты rendered:

```
<h:panelGroup rendered="#{!userBean.loggedIn}">...</h:panelGroup>
<h:panelGroup rendered="#{userBean.loggedIn}">...</h:panelGroup>
```

Для применения более двух выборов лучше использовать такой компонент, как стек панелей panelStack из библиотеки компонентов MyFaces Apache (http://myfaces.apache.org/tomahawk). Стек панелей аналогичен области окна с вкладками, которая была описана в главе 11, за исключением того, что в нем отсутствуют вкладки. Вместо этого он обеспечивает выбор одного из дочерних компонентов программным путем.

При использовании компонента panelStack каждый дочерний компонент должен иметь идентификатор. Атрибут selectedPanel определяет идентификатор дочернего компонента, который подготавливается к отображению:

```
<t:panelStack selectedPanel="#{userBean.status}">
   <h:panelGroup id="new">...</h:panelGroup>
   <h:panelGroup id="loggedIn">...</h:panelGroup>
   <h:panelGroup id="loggedOut">...</h:panelGroup>
</t:panelStack>
```

Метод getStatus бина user должен возвращать строку "new", "loggedIn" или "loggedOut".

Если необходимо осуществлять выбор между двумя полностью различными страницами, то можно использовать теги ui:include в теге c:choose:

```
<c:choose>
   <c:when test="#{user.loggedIn}">
      <ui:include src="main.xhtml" />
   </c:when>
   <c:otherwise>
      <ui:include src="login.xhtml" />
   </c:otherwise>
</c:choose>
```

Настройка страниц с сообщениями об ошибках

Если при выполнении приложения на стадии проектирования "development" (разработка) обнаруживается ошибка, то появляется сообщение об ошибке, аналогичное показанному на рис. 13.8.

Но появление подобного сообщения об ошибке перед глазами пользователей в производственном приложении вряд ли допустимо. Тем не менее после задания "production" в качестве стадии проектирования в файле web.xml положение становится еще хуже, как показано на рис. 13.9.

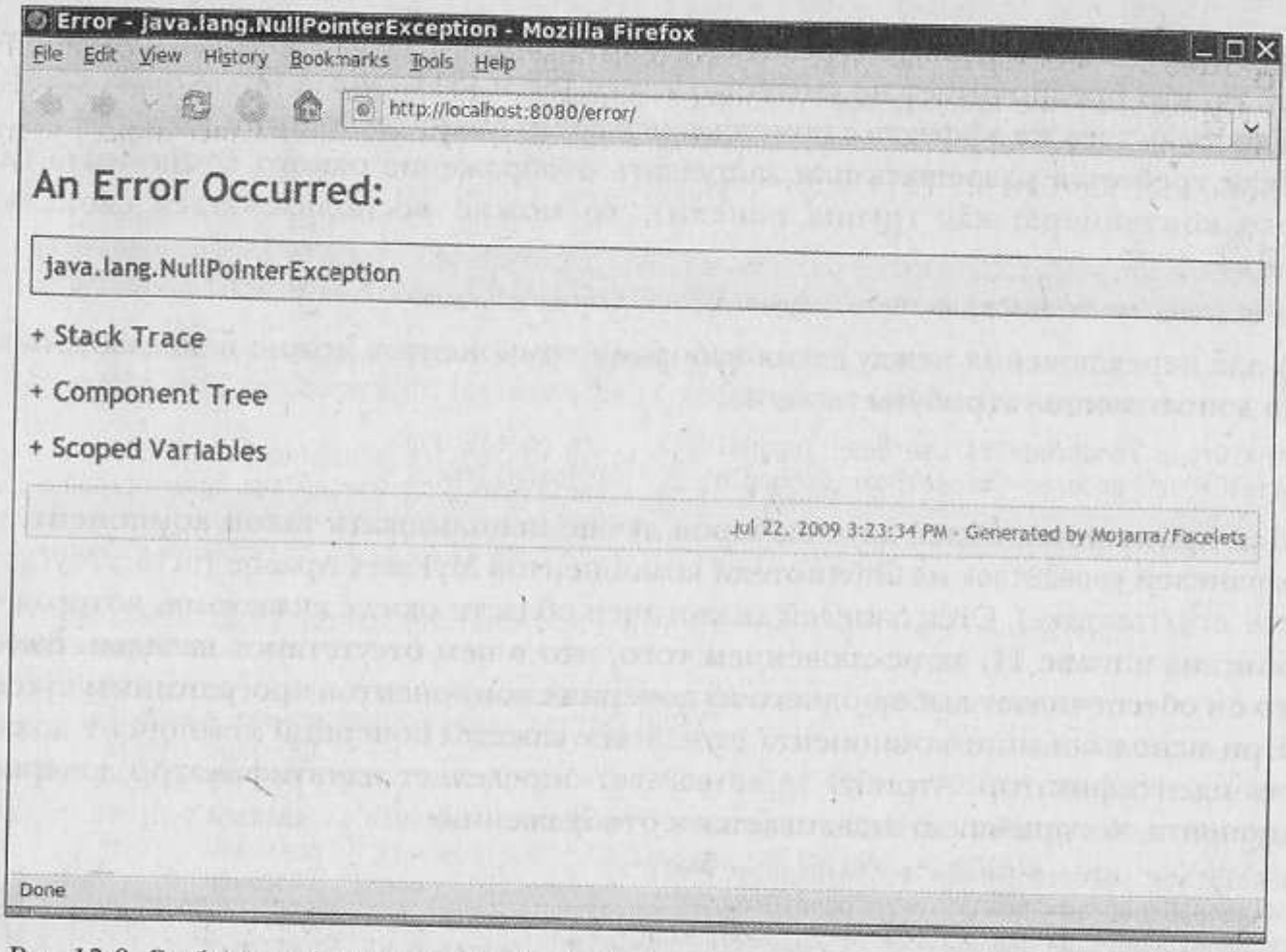


Рис. 13.8. Сообщение об ошибке на стадии разработки

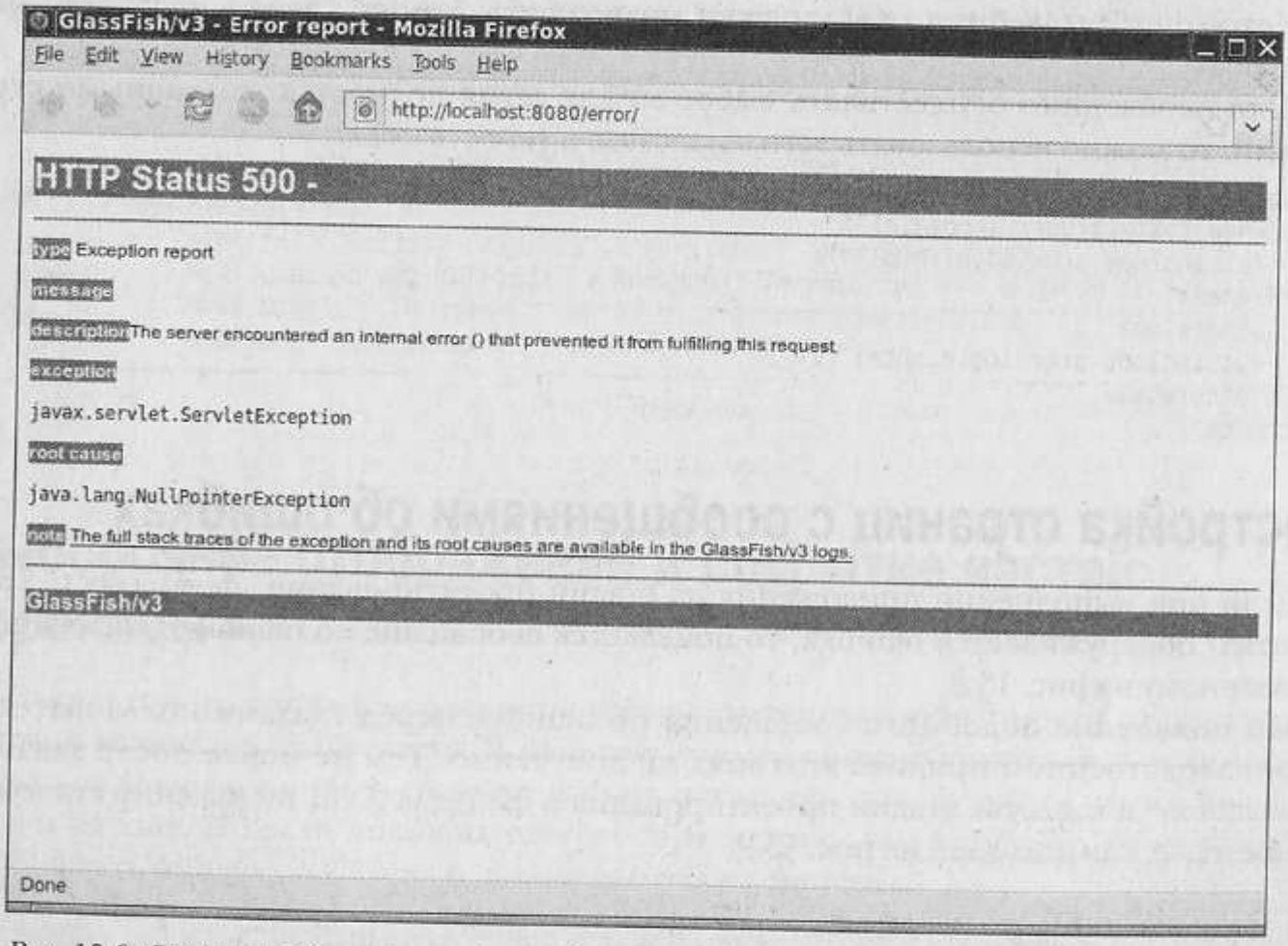


Рис. 13.9. Сообщение об ошибке на производственной стадии

Необходимо действительно избегать того, чтобы пользователи видели эту страницу.

Чтобы вместо нее выводилась лучшая страница с сообщением об ошибке, необходимо использовать тег `error-page` в файле `web.xml`. Задайте либо класс исключения Java, либо код ошибки HTTP. Например:

```
<error-page>
   <exception-type>java.lang.Exception</exception-type>
   <location>/faces/exception.xhtml</location>
</error-page>
<error-page>
   <error-code>500</error-code>
   <location>/faces/error.xhtml</location>
</error-page>
<error-page>
   <error-code>404</error-code>
   <location>/faces/notfound.xhtml</location>
</error-page>
```

Если возникает исключительная ситуация и одна из страниц с сообщением об ошибке соответствует ее типу, то отображается отвечающая этому страница с сообщением об ошибке. В противном случае вырабатывается ошибка HTTP 500.

Если возникает ошибка HTTP и обнаруживается соответствующая ей страница с сообщением об ошибке, то она отображается. В противном случае выводится страница по умолчанию с сообщением об ошибке.

Внимание! Если при попытке отобразить в приложении пользовательскую страницу с сообщением об ошибке возникает сбой, то вместо пользовательской страницы выводится страница по умолчанию с сообщением об ошибке. Если любые усилия не позволяют добиться вывода пользовательской страницы с сообщением об ошибке, проверьте журналы на наличие сообщений, касающихся требуемой страницы с сообщением об ошибке.

Если используется механизм `error-page`, то в карту запроса помещается несколько объектов, связанных с ошибкой (табл. 13.1). Эти значения можно применять для отображения сведений, которые описывают ошибку.

Таблица 13.1. Атрибуты исключительной ситуации сервлета

Ключ	Значение	Тип
`javax.servlet.error.status_code`	Код ошибки HTTP	`Integer`
`javax.servlet.error.message`	Описание ошибки	`String`
`javax.servlet.error.exception_type`	Класс исключительной ситуации	`Class`
`javax.servlet.error.exception`	Объект исключительной ситуации	`Throwable`
`javax.servlet.error.request_uri`	Путь к ресурсу приложения, в котором обнаружена ошибка	`String`
`javax.servlet.error.servlet_name`	Имя сервлета, в котором обнаружена ошибка	`String`

Описанный подход используется в следующем примере приложения. Мы намеренно активизируем исключительную ситуацию в связи с наличием указателя `null` в свойстве пароля `UserBean`, что приводит к получению отчета об ошибке, показанного на рис. 13.10. В листинге 13.20 приведен файл `web.xml`, который задает страницу с сообщением об ошибке в файле `errorDisplay.xhtml` (листинг 13.21).

В листинге 13.22 приведен класс `ErrorBean`. В методе `getStackTrace` этого класса осуществляется сборка полной трассировки стека, которая содержит все вложенные исключительные ситуации.

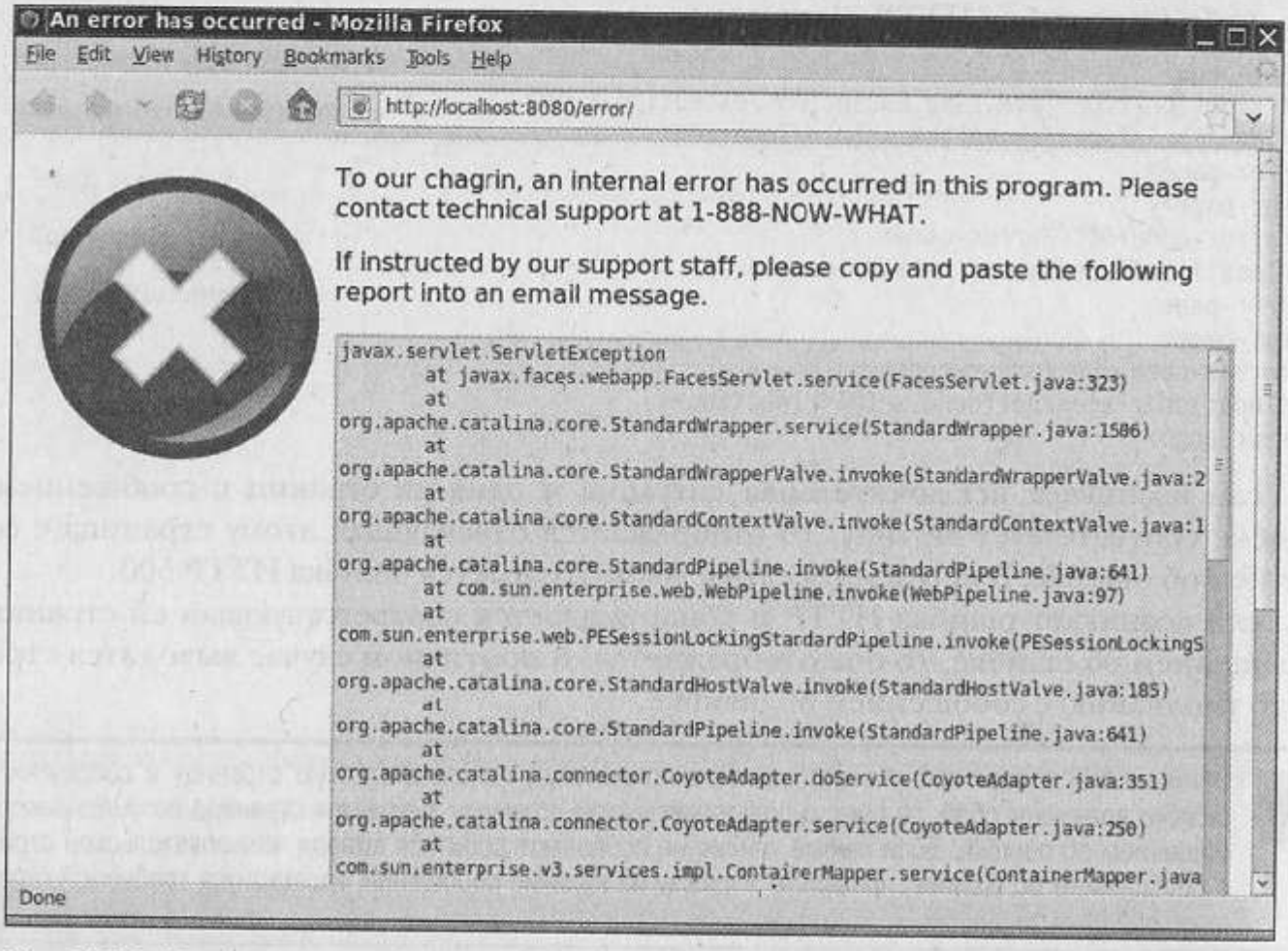


Рис. 13.10. Отображение сообщения об ошибке, определяемого пользователем

На заметку! Даже если применяется пользовательская страница с сообщением об ошибке, все еще остается возможность включить отображение стандартного сообщения об ошибке Facelets на стадии разработки. Для этого достаточно добавить следующую строку:

```
<ui:include src="javax.faces.error.xhtml"/>
```

На производственной стадии отображение ошибок подавляется.

Листинг 13.20. Файл `error/web/WEB-INF/web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   version="2.5">
   <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
   </servlet-mapping>
```

```
    <welcome-file-list>
       <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
    <context-param>
       <param-name>javax.faces.PROJECT_STAGE</param-name>
       <param-value>Production</param-value>
    </context-param>
    <error-page>
       <error-code>500</error-code>
       <location>/faces/errorDisplay.xhtml</location>
    </error-page>
</web-app>
```

Листинг 13.21. Файл error/web/errorDisplay.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.errorTitle}</title>
   </h:head>
   <h:body>
      <h:form>
         <h:graphicImage library="images" name="error.png" style="float: left;"/>
         <p>#{msgs.errorOccurred}</p>
         <p>#{msgs.copyReport}</p>
         <h:inputTextarea value="#{error.stackTrace}" rows="40" cols="80"
                          readonly="true"/>
      </h:form>
   </h:body>
</html>
```

Листинг 13.22. Файл error/src/java/com/corejsf/ErrorBean.java

```
package com.corejsf;

import java.io.PrintWriter;
import java.io.StringWriter;
import java.sql.SQLException;
import java.util.Map;

import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.RequestScoped;
   // или import javax.faces.bean.RequestScoped;
import javax.faces.context.FacesContext;
import javax.servlet.ServletException;

@Named("error") // или @ManagedBean(name="error")
@RequestScoped
public class ErrorBean {
   public String getStackTrace() {
      FacesContext context = FacesContext.getCurrentInstance();
      Map<String, Object> request
         = context.getExternalContext().getRequestMap();
      Throwable ex = (Throwable) request.get("javax.servlet.error.exception");
      StringWriter sw = new StringWriter();
      PrintWriter pw = new PrintWriter(sw);
```

```
25.       fillStackTrace(ex, pw);
26.       return sw.toString();
27.    }
28.
29.    private static void fillStackTrace(Throwable t, PrintWriter w) {
30.       if (t == null) return;
31.       t.printStackTrace(w);
32.       if (t instanceof ServletException) {
33.          Throwable cause = ((ServletException) t).getRootCause();
34.          if (cause != null) {
35.             w.println("Root cause:");
36.             fillStackTrace(cause, w);
37.          }
38.       } else if (t instanceof SQLException) {
39.          Throwable cause = ((SQLException) t).getNextException();
40.          if (cause != null) {
41.             w.println("Next exception:");
42.             fillStackTrace(cause, w);
43.          }
44.       } else {
45.          Throwable cause = t.getCause();
46.          if (cause != null) {
47.             w.println("Cause:");
48.             fillStackTrace(cause, w);
49.          }
50.       }
51.    }
52. }
```

Написание собственного клиентского тега проверки правильности

Предположим, что разработана функция JavaScript для проверки правильности и проведено ее тестирование на браузерах нескольких типов. Теперь настало время использовать эту функцию в конкретных приложениях JSF.

Для этого необходимы два тега.

1. Тег средства проверки, который закрепляется за каждым компонентом, требующим проверки правильности.
2. Тег компонента, формирующий код JavaScript для проверки всех компонентов в форме. Тег компонента должен быть добавлен в конце формы. Следует учитывать, что с этой целью нельзя использовать тег средства проверки. Только компоненты могут подготовливать вывод к отображению.

В качестве примера ниже показано, как использовать код проверки правильности номера кредитной карточки в проекте Apache Commons Validator. Этот код можно загрузить с сайта `http://jakarta.apache.org/commons/validator`.

Прежде всего необходимо подготовить два тега: тег `creditCardValidator`, который может быть добавлен к любому компоненту ввода JSF, и тег компонента `validatorScript`, формирующий требуемый код JavaScript.

Тег `creditCardValidator` имеет два атрибута. Атрибут `message` определяет шаблон сообщения об ошибке:

```
{0} is not a valid credit card number
```

Атрибут arg представляет собой значение, которое должно быть заполнено для {0}, обычно имя поля. Например:

```
<corejsf:creditCardValidator
   message="#{msgs.invalidCard}" arg="#{msgs.primaryCard}"/>
```

Код средства проверки приведен в листинге 13.23. Класс средства проверки validator имеет два не связанных друг с другом назначения: проверка правильности и форматирование сообщения об ошибке.

Этот класс осуществляет традиционную серверную проверку правильности, независимо от клиентского кода JavaScript. Ведь в конечном итоге не следует полагаться исключительно на клиентскую проверку правильности. Дело в том, что пользователи могут запретить использование сценариев JavaScript в своих браузерах. Кроме того, может оказаться, что в веб-приложение поступают непроверенные запросы HTTP, отправленные хакерами, хорошо знающими веб, которые могут даже применять для этого автоматизированные сценарии.

Метод getErrorMessage форматирует сообщение об ошибке, которое должно быть включено в клиентский код JavaScript. Сообщение об ошибке формируется на основе атрибутов message и arg.

Компонент validatorScript представляет намного больший интерес (листинг 13.24). В методе encodeBegin этого компонента вызывается рекурсивный метод findCreditCardValidators, который проходит по дереву компонентов, обнаруживает все компоненты, перечисляет относящиеся к ним средства проверки, выясняет, какие из них являются средствами проверки номеров кредитных карточек, и собирает их в объекте карты. Метод writeValidationFunctions записывает код JavaScript, который вызывает функцию проверки правильности для всех полей, к которым относятся средства проверки номеров кредитных карточек.

Тег validatorScript необходимо поместить в форму таким образом:

```
<h:form id="paymentForm" onsubmit="return validatePaymentForm(this);">
   ...
   <corejsf:validatorScript functionName="validatePaymentForm"/>
</h:form>
```

В листинге 13.25 показан пример страницы JSF. На рис. 13.11 приведено сообщение об ошибке, формируемое при попытке пользователя отправить недопустимый номер кредитной карточки.

Подробности реализации метода writeValidationFunctions во многом определяются нюансами применения кода JavaScript в проекте Commons Validator.

Прежде всего, метод writeValidationFunctions формирует функцию проверки правильности, которая вызывается в обработчике onsubmit формы:

```
var bCancel = false;
function functionName(form) { return bCancel || validateCreditCard(form); }
```

Если форма содержит кнопки Cancel (Отмена) или Back (Назад), то в обработчиках onclick этих кнопок значение переменной bCancel должно быть задано равным true, что позволяет обойти проверку правильности.

Функция validateCreditCard представляет собой точку входа в код Commons Validator. Работа этой функции основана на том, что может быть успешно найдена функция formName_creditCard, которая создает объект конфигурации. Метод writeValidationFunctions формирует код для функции creditCard.

К сожалению, подробности реализации этих методов достаточно запутанны. Функция formName_creditCard возвращает объект с одной переменной экземпляра для каж-

дого проверенного элемента формы. Каждое поле экземпляра содержит массив с тремя значениями: идентификатор элемента формы, сообщение об ошибке, отображаемое при неудачном завершении проверки правильности, и характерное для средства проверки значение параметра настройки. Это значение не используется в средстве проверки номера кредитной карточки; вместо него предусмотрено применение пустой строки.

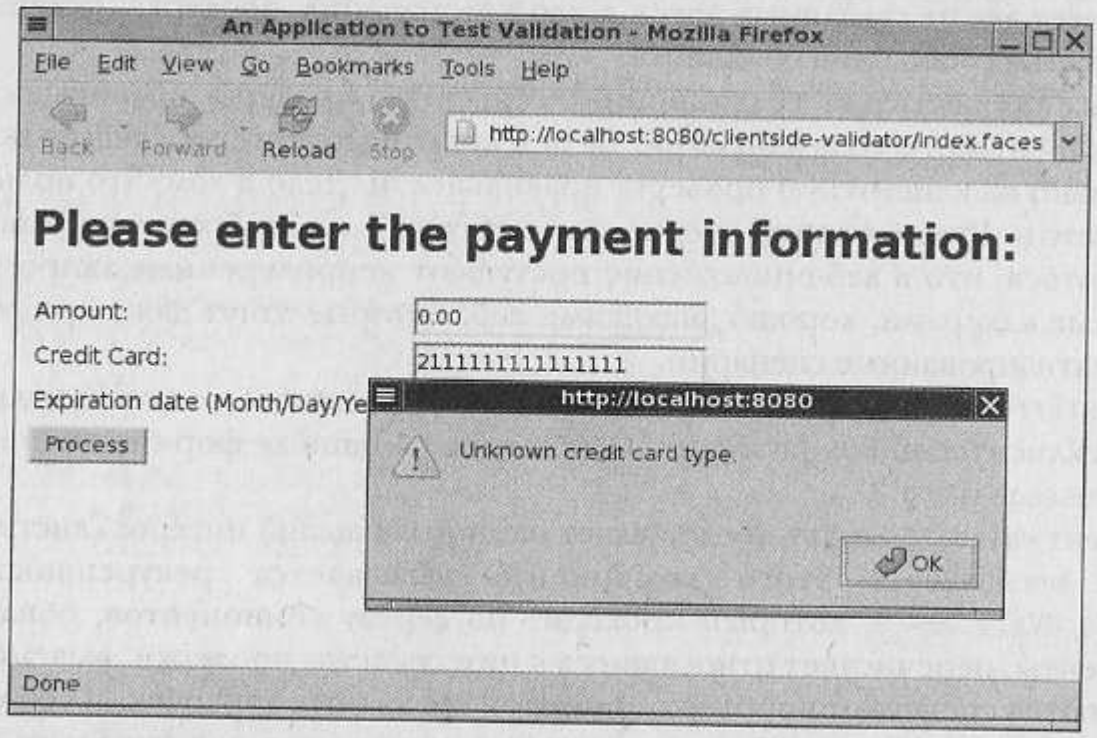


Рис. 13.11. Клиентская проверка правильности номера кредитной карточки

Имена полей экземпляра не имеют значения. В методе `writeValidationFunctions` мы используем в своих интересах гибкость языка JavaScript и вызываем поля 0, 1, 2 и т.д. Например:

```
function paymentForm_creditCard() {
   this[0] = new Array("paymentForm:primary",
      "Primary Credit Card is not a valid card number", "");
   this[1] = new Array("paymentForm:backup",
      "Backup Credit Card is not a valid card number", "");
}
```

Проектируя собственные функции JavaScript, читатель может предусмотреть более надежный механизм оформления набора параметров.

Листинг 13.23. Файл `clientside-validator/src/java/com/corejsf/CreditCardValidator.java`

```
 1. package com.corejsf;
 2.
 3. import java.io.Serializable;
 4. import java.text.MessageFormat;
 5. import java.util.Locale;
 6. import javax.faces.application.FacesMessage;
 7. import javax.faces.component.UIComponent;
 8. import javax.faces.context.FacesContext;
 9. import javax.faces.validator.FacesValidator;
10. import javax.faces.validator.Validator;
11. import javax.faces.validator.ValidatorException;
12.
13. @FacesValidator("com.corejsf.CreditCard")
```

```
14. public class CreditCardValidator implements Validator, Serializable {
15.    private String message;
16.    private String arg;
17.
18.    public void setMessage(String newValue) { message = newValue; }
19.
20.    public void setArg(String newValue) { arg = newValue; }
21.     public String getArg() { return arg; }
22.
23.     public void validate(FacesContext context, UIComponent component,
24.           Object value) {
25.        if (value == null) return;
26.        String cardNumber;
27.        if (value instanceof CreditCard)
28.           cardNumber = value.toString();
29.        else
30.           cardNumber = getDigitsOnly(value.toString());
31.        if (!luhnCheck(cardNumber)) {
32.           FacesMessage message = new FacesMessage(FacesMessage.SEVERITY_ERROR,
33.              getErrorMessage(value, context), null);
34.           throw new ValidatorException(message);
35.        }
36.     }
37.
38.     public String getErrorMessage(Object value, FacesContext context) {
39.        Object[] params = new Object[] { value };
40.        if (message == null)
41.           return com.corejsf.util.Messages.getString(
42.                 "com.corejsf.messages", "badLuhnCheck", params);
43.        else {
44.           Locale locale = context.getViewRoot().getLocale();
45.           MessageFormat formatter = new MessageFormat(message, locale);
46.           return formatter.format(params);
47.        }
48.     }
49.
50.     private static boolean luhnCheck(String cardNumber) {
51.        int sum = 0;
52.
53.         for (int i = cardNumber.length() - 1; i >= 0; i -= 2) {
54.            sum += Integer.parseInt(cardNumber.substring(i, i + 1));
55.            if(i > 0) {
56.               int d = 2 * Integer.parseInt(cardNumber.substring(i - 1, i));
57.               if(d > 9) d -= 9;
58.               sum += d;
59.            }
60.         }
61.
62.         return sum % 10 == 0;
63.     }
64.
65.     private static String getDigitsOnly(String s) {
66.        StringBuilder digitsOnly = new StringBuilder ();
67.        char c;
68.     for (int i = 0; i < s.length (); i++) {
69.        c = s.charAt (i);
70.        if (Character.isDigit(c)) {
71.           digitsOnly.append(c);
72.        }
73.     }
74.     return digitsOnly.toString ();
```

```
75.    }
76. }
```

Листинг 13.24. Файл clientside-validator/src/java/com/corejsf/UIValidatorScript.java

```
 1. package com.corejsf;
 2.
 3. import java.io.IOException;
 4. import java.util.Map;
 5. import java.util.LinkedHashMap;
 6.
 7. import javax.faces.application.ResourceDependency;
 8. import javax.faces.component.EditableValueHolder;
 9. import javax.faces.component.FacesComponent;
10. import javax.faces.component.UIComponent;
11. import javax.faces.component.UIComponentBase;
12. import javax.faces.context.FacesContext;
13. import javax.faces.context.ResponseWriter;
14. import javax.faces.validator.Validator;
15.
16. @FacesComponent("com.corejsf.ValidatorScript")
17. @ResourceDependency(library="javascript", name="validateCreditCard.js",
18.    target="head")
19. public class UIValidatorScript extends UIComponentBase {
20.    private Map<String, Validator> validators
21.       = new LinkedHashMap<String, Validator>();
22.
23.    public String getRendererType() { return null; }
24.    public String getFamily() { return null; }
25.
26.    private void findCreditCardValidators(UIComponent c, FacesContext context) {
27.       if (c instanceof EditableValueHolder) {
28.          EditableValueHolder h = (EditableValueHolder) c;
29.          for (Validator v : h.getValidators()) {
30.             if (v instanceof CreditCardValidator) {
31.                String id = c.getClientId(context);
32.                validators.put(id, v);
33.             }
34.          }
35.       }
36.
37.       for (UIComponent child : c.getChildren())
38.          findCreditCardValidators(child, context);
39.    }
40.
41.    private void writeScriptStart(ResponseWriter writer) throws IOException {
42.       writer.startElement("script", this);
43.       writer.writeAttribute("type", "text/javascript", null);
44.       writer.writeAttribute("language", "Javascript1.1", null);
45.       writer.write("\n<!--\n");
46.    }
47.
48.    private void writeScriptEnd(ResponseWriter writer) throws IOException {
49.       writer.write("\n-->\n");
50.       writer.endElement("script");
51.    }
52.
53.    private void writeValidationFunctions(ResponseWriter writer,
54.       FacesContext context) throws IOException {
55.       writer.write("var bCancel = false;\n");
56.       writer.write("function " );
```

```
57.         writer.write(getAttributes().get("functionName").toString());
58.         writer.write("(form) { return bCancel || validateCreditCard(form); }\n");
59.
60.         writer.write("function ");
61.         String formId = getParent().getClientId(context);
62.         writer.write(formId);
63.         writer.write("_creditCard() { \n");
64.         // Для каждого проверяемого поля этого типа добавить объект конфигурации
65.         int k = 0;
66.         for (String id : validators.keySet()) {
67.            CreditCardValidator v = (CreditCardValidator) validators.get(id);
68.            writer.write("this[" + k + "] = ");
69.            k++;
70.
71.             writer.write("new Array('");
72.             writer.write(id);
73.             writer.write("', '");
74.             writer.write(v.getErrorMessage(v.getArg(), context));
75.             writer.write("', '');\n"); // Третий элемент, не используемый в средстве
76.         }                              // проверки кредитных карт
77.         writer.write("}\n");
78.     }
79.
80.     public void encodeBegin(FacesContext context) throws IOException {
81.        ResponseWriter writer = context.getResponseWriter();
82.
83.         validators.clear();
84.        findCreditCardValidators(context.getViewRoot(), context);
85.
86.        writeScriptStart(writer);
87.        writeValidationFunctions(writer, context);
88.        writeScriptEnd(writer);
89.    }
90. }
```

Листинг 13.25. Файл clientside-validator/web/index.xhtml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://java.sun.com/jsf/core"
6.       xmlns:corejsf="http://corejsf.com">
7.    <h:head>
8.       <h:outputStylesheet library="css" name="styles.css"/>
9.       <title>#{msgs.title}</title>
10.   </h:head>
11.   <h:body>
12.      <h:form id="paymentForm" onsubmit="return validatePaymentForm(this);">
13.         <corejsf:validatorScript functionName="validatePaymentForm"/>
14.         <h1>#{msgs.enterPayment}</h1>
15.         <h:panelGrid columns="3">
16.            #{msgs.amount}
17.            <h:inputText id="amount" value="#{payment.amount}">
18.               <f:convertNumber minFractionDigits="2"/>
19.            </h:inputText>
20.            <h:message for="amount" styleClass="errorMessage"/>
21.
22.            #{msgs.creditCard}
23.            <h:inputText id="card" value="#{payment.card}" required="true">
24.               <corejsf:creditCardValidator message="#{msgs.unknownType}"
```

```
                                                    arg="#{msgs.creditCard}"/>
            </h:inputText>
            <h:message for="card" styleClass="errorMessage"/>

            #{msgs.expirationDate}
            <h:inputText id="date" value="#{payment.date}">
                <f:convertDateTime pattern="MM/dd/yyyy"/>
            </h:inputText>
            <h:message for="date" styleClass="errorMessage"/>
        </h:panelGrid>
        <h:commandButton value="Process" action="result"/>
    </h:form>
  </h:body>
</html>
```

На заметку! К сожалению, реализация Commons Validator предусматривает отображение всплывающего окна при обнаружении ошибки проверки правильности. Но было бы лучше, если бы сообщение об ошибке размещалось рядом с полем, имеющим недопустимое значение. Такая возможность поддерживается в клиентском пакете проверки правильности Cagatay Civici, который можно найти по адресу `http://jsf-comp.sourceforge.net/components/clientvalidators/index.html`.

Настройка собственного приложения

Для многих приложений требуются определенные параметры конфигурации, такие как пути к внешним каталогам, имена учетных записей по умолчанию и т.д. Эти параметры должны обновляться средствами развертывания приложения, поэтому не рекомендуется помещать их в код приложения.

Удобным местом для хранения параметров конфигурации является файл `web.xml`. При этом следует предоставлять набор элементов `context-param` в элементе `web-app`, как в следующем примере:

```
<web-app>
  <context-param>
    <param-name>URL</param-name>
    <param-value>ldap://localhost:389</param-value>
  </context-param>
  ...
</web-app>
```

Для чтения параметра необходимо получить объект внешнего контекста. Этот объект описывает среду выполнения, в которой было запущено конкретное приложение JSF. Если используется контейнер сервлетов, то внешним контекстом является оболочка вокруг объекта `ServletContext`. В классе `ExternalContext` предусмотрен целый ряд вспомогательных методов, позволяющих обращаться к свойствам основного контекста сервлета. Метод `getInitParameter` осуществляет выборку значения параметра контекста с указанным именем:

```
ExternalContext external = FacesContext.getCurrentInstance().getExternalContext();
String url = external.getInitParameter("URL");
```

Внимание! Не следует путать теги `context-param` и `init-param`. Последний тег используется для параметров, которые сервлет может обрабатывать при запуске. К сожалению, метод чтения параметров контекста носит имя `getInitParameter`.

Разработчики некоторых приложений предпочитают обрабатывать собственные файлы конфигурации, а не использовать файл web.xml. Но при этом возникают сложности при определении местонахождения собственного файла конфигурации, поскольку заранее не известно, где веб-контейнер будет хранить файлы конкретного веб-приложения. В действительности веб-контейнеру вообще не требуется хранить пользовательские файлы физически, поскольку может быть принято решение читать их из файла WAR.

Вместо этого следует использовать метод getResourceAsStream класса ExternalContext. Например, предположим, что разработчик хочет организовать чтение файла app.properties из каталога WEB-INF своего приложения. Ниже показан код, который для этого требуется.

```
FacesContext context = FacesContext.getCurrentInstance();
ExternalContext external = context.getExternalContext();
InputStream in = external.getResourceAsStream("/WEB-INF/app.properties");
```

Расширение языка выражений JSF

Иногда возникает необходимость расширить язык выражений. В качестве примера рассмотрим расширение, которое позволяет искать формы и компоненты по идентификатору, допустим, таким образом:

```
view.loginForm
```

Это достигается путем добавления средства разрешения, которое обрабатывает выражение base.property (или, что эквивалентно, base[property]), где base представляет собой строку "view", а property — идентификатор формы.

Чтобы реализовать такое средство разрешения, необходимо расширить класс ELResolver. Ключевым методом является следующий:

```
public Object getValue(ELContext context, Object base, Object property)
```

Если применяемое средство разрешения способно правильно разрешать выражение base.property, то достаточно вызвать

```
context.setPropertyResolved(true);
```

и возвратить значение выражения.

Предусмотрено также несколько других способов запроса типа и поддержки инструментальных средств написания программ; подробные сведения приведены в документации API.

Ниже описано, как реализовать требуемое средство разрешения для идентификаторов форм и компонентов. Рассмотрим, например, следующее выражение:

```
view.loginForm.password.value
```

Нам требуется найти компонент с идентификатором loginForm в корневом каталоге представления, затем идентификатор password в форме, после чего вызвать метод getValue компонента. Рассматриваемое средство разрешения должно обрабатывать выражения, имеющие вид component.name:

```
public class ComponentIdResolver extends ELResolver {
   public Object getValue(ELContext context, Object base, Object property) {
      if (base instanceof UIComponent && property instanceof String) {
         UIComponent r = ((UIComponent) base).findComponent((String) property);
         if (r != null) {
            context.setPropertyResolved(true);
```

```
            return r;
        }
      }
      return null;
    }
    ...
}
```

Следует учитывать, что это средство разрешения вызывается для разрешения первых двух подвыражений (`view.loginForm` и `view.loginForm.password`). Последнее выражение разрешается средством разрешения управляемого бина, которое входит в состав реализации JSF.

Начальное представление выражения рассматривается как частный случай. Средства разрешения вызываются со значением `base`, заданным равным `null`, и значением `property`, в качестве которого задана начальная строка выражения. Это выражение разрешает неявно заданное объектное средство разрешения JSF, которое возвращает объект `UIViewRoot` страницы.

В качестве еще одного примера рассмотрим создание средства разрешения для системных свойств. Например, выражение

```
sysprop['java.version']
```

должно возвратить результат вызова

```
System.getProperty("java.version");
```

Задача станет более интересной, если мы потребуем, чтобы выражение

```
sysprop.java.version
```

также работало. Это пользовательское средство разрешения должно правильно действовать и в частном случае, в котором значение `base` равно `null`, а значение `property` равно `"sysprop"`. Оно также должно справляться с частично завершенными подвыражениями, такими как `sysprop.java`.

Прежде всего осуществляется сборка списка выражений во вложенном классе `SystemPropertyResolver.PartialResolution`. Рассматриваемое средство разрешения различает два случая.

1. Если значение `base` равно `null`, а значение `property` — `"sysprop"`, возвращается пустой объект `PartialResolution`.
2. Если значение `base` является объектом `PartialResolution`, а значение `property` представляет собой строку, то свойство `property` добавляется к концу списка. После этого предпринимается попытка поиска системного свойства, ключом которого служит результат конкатенации разделенных точками записей списка. Если искомое системное свойство существует, происходит его возврат. В противном случае возвращается дополненный список.

Эти случаи иллюстрируются в приведенном ниже фрагменте кода.

```
public class SystemPropertyResolver extends ELResolver {
   public Object getValue(ELContext context, Object base, Object property) {
      if (base == null && "sysprop".equals(property)) {
         context.setPropertyResolved(true);
         return new PartialResolution();
      }
      if (base instanceof PartialResolution && property instanceof String) {
         ((PartialResolution) base).add((String) property);
         Object r = System.getProperty(base.toString());
         context.setPropertyResolved(true);
```

```
            if (r == null) return base;
            else return r;
         }
         return null;
      }
      ...
      public static class PartialResolution extends ArrayList<String> {
         public String toString() {
            StringBuilder r = new StringBuilder();
            for (String s : this) {
               if (r.length() > 0) r.append('.');
               r.append(s);
            }
            return r.toString();
         }
      }
}
```

Чтобы добавить это пользовательское средство разрешения к конкретному приложению JSF, необходимо дополнительно ввести элементы, подобные следующим, в файл `faces-config.xml` (или в другой файл конфигурации приложения):

```
<application>
   <el-resolver>com.corejsf.ComponentIdResolver</el-resolver>
   ...
</application>
```

Полную реализацию для этих двух примеров средств разрешения можно найти в примере кода `ch13/extending-el` на сопровождающем веб-сайте к этой книге.

На заметку! В версии JSF 1.1 задача модификации языка выражений была довольно трудоемкой. В реализации JSF 1.1 предусмотрены конкретные подклассы абстрактных классов `VariableResolver` и `PropertyResolver`. Класс `VariableResolver` разрешает начальное подвыражение, а класс `PropertyResolver` отвечает за вычисление операторов с точками или квадратными скобками.

Разработчик, которому требуется ввести свои переменные, предоставляет собственное средство разрешения переменных и указывает его в файле конфигурации приложения:

```
<application>
         <variable-resolver>
            com.corejsf.CustomVariableResolver
         </variable-resolver>
         ...
      </application>
```

В конкретном классе средства разрешения должен быть предусмотрен конструктор с единственным параметром типа `VariableResolver`. После этого реализация JSF передает в конкретную программу свое средство разрешения переменных по умолчанию. Благодаря этому становится несложным применение шаблона декоратора. Ниже приведен пример кода средства разрешения переменных, которое распознает имя переменной sysprop.

```
public class CustomVariableResolver extends VariableResolver {
         private VariableResolver original;

         public CustomVariableResolver(VariableResolver original) {
            this.original = original;
         }

         public Object resolveVariable(FacesContext context, String name) {
            if (name.equals("sysprop")) return System.getProperties();
            return original.resolveVariable(context, name);
         }
      }
```

Реализация класса `PropertyResolver` аналогична.

Добавление функций к языку выражений JSF JSF 2.0

Разработчик может добавить собственную функцию к языку выражений JSF, выполнив следующие шаги.

1. Реализация функции как статического метода.
2. Отображение имени функции на реализацию в файле библиотеки тегов Facelets.

В качестве примера предположим, что необходимо определить функцию, которая читает файл и возвращает его содержимое. Типичным применением такой функции может быть следующее:

```
<p>Page source:</p><pre>#{corejsf:getFile("/index.xhtml")}</pre>
```

В листинге 13.26 класс реализует функцию как статический метод `ELFunctions.getFile`.

Листинг 13.26. Файл extending-el/src/java/com/corejsf/ELFunctions.java

```
1. package com.corejsf;
2.
3. import java.io.InputStream;
4. import java.util.Scanner;
5. import javax.faces.context.FacesContext;
6.
7. public class ELFunctions {
8.    public static String getFile(String filename) {
9.       FacesContext context = FacesContext.getCurrentInstance();
10.      java.util.logging.Logger.getLogger("com.corejsf").info("context=" + context);
11.      InputStream stream = context.getExternalContext().getResourceAsStream(filename);
12.      Scanner in = new Scanner(stream);
13.      java.util.logging.Logger.getLogger("com.corejsf").info("context=" + context);
14.      StringBuilder builder = new StringBuilder();
15.      while (in.hasNextLine()) { builder.append(in.nextLine()); builder.append('\n'); }
16.      return builder.toString();
17.   }
18. }
```

В листинге 13.27 показан файл библиотеки тегов.

Листинг 13.27. Файл extending-el/web/WEB-INF/corejsf.taglib.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <facelet-taglib version="2.0"
3.    xmlns="http://java.sun.com/xml/ns/javaee"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6.    http://java.sun.com/xml/ns/javaee/web-facelettaglibary_2_0.xsd">
7.    <namespace>http://corejsf.com</namespace>
8.    <function>
9.       <function-name>getFile</function-name>
10.      <function-class>com.corejsf.ELFunctions</function-class>
11.      <function-signature>
12.         java.lang.String getFile(java.lang.String)
13.      </function-signature>
14.   </function>
15. </facelet-taglib>
```

Следуя той же процедуре, разработчик может добавлять другие необходимые методы к языку выражений JSF.

Мониторинг трафика между браузером и сервером

Часто бывает необходимо знать, какие параметры клиент передал обратно на сервер при отправке формы. Безусловно, можно просто внедрить выражение

```
#{param}
```

в код страницы и получить список имен параметров и значений.

Тем не менее, особенно когда задача состоит в отладке приложений Ajax, лучше отслеживать весь трафик между клиентом и сервером. Такое отслеживание поддерживается на обеих платформах, Eclipse и Netbeans.

В версии Eclipse 3.5 многое зависит от адаптера сервера. Ниже приведены инструкции для работы с платформой Glassfish.

1. Щелкните правой кнопкой мыши на сервере во вкладке Servers и выберите Monitoring⇨Properties.
2. Щелкните на кнопке Add и выберите порт 8080 (рис. 13.12).
3. Выберите элемент Start.

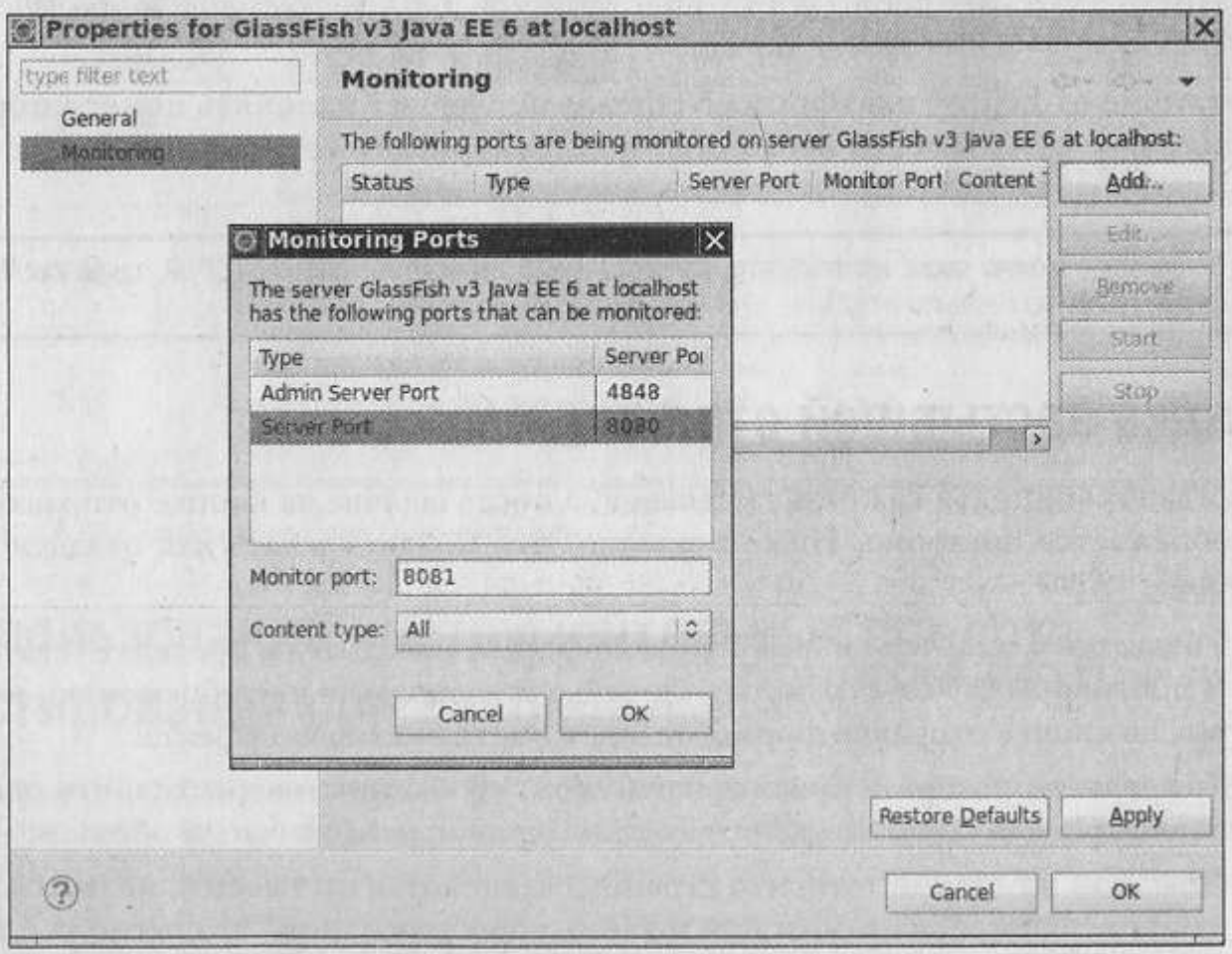


Рис. 13.12. Активизация отслеживания трафика HTTP на платформе Eclipse

4. Задайте в браузере адрес `http://localhost:8081/contextRoot`. Платформа Eclipse перехватывает трафик через порт 8081 и отправляет его в порт 8080. Что еще более важно, на этой платформе предусмотрено отображение декодированных запросов и ответов (рис. 13.13).

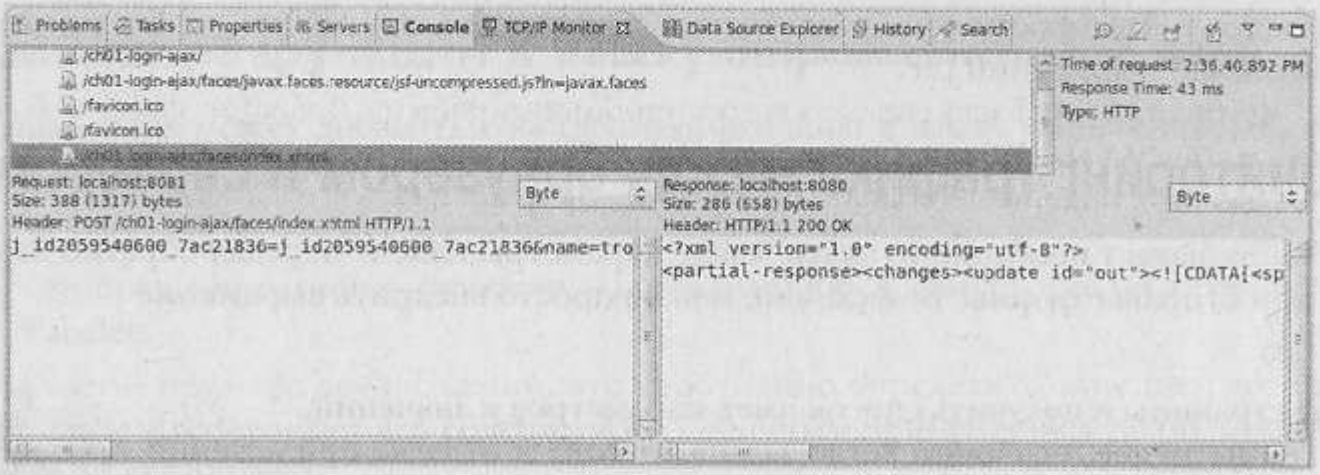


Рис. 13.13. Отслеживание трафика HTTP в действии

Если используются программные средства NetBeans 6.8 и Glassfish, выполните следующие действия.

1. Откройте вкладку Services и разверните узел Servers.
2. Щелкните правой кнопкой мыши на записи GlassFish и выберите Properties.
3. Установите флажок Use HTTP monitor (Использовать отслеживание трафика HTTP).
4. После запуска конкретного приложения выберите элемент меню Window⇨ Debugging⇨HTTP Server Monitor.

В отличие от Eclipse платформа Netbeans не требует изменять номера портов в применяемых URL. Вместо этого она предусматривает установку фильтра в веб-приложении.

На заметку! Можно также использовать универсальный анализатор трафика TCP/IP, такой как Wireshark (`http://www.wireshark.org/`).

Отладка застывшей страницы

Иногда страница JSF кажется застывшей, а после щелчка на кнопке отправки формы отображается повторно. Ниже показано, что можно сделать для отладки такой страницы.

- Убедитесь в том, что кнопка отправки формы находится в пределах тега `h:form`. В противном случае страница будет подготавливаться к отображению, но щелчок на кнопке отправки формы не будет иметь никакого эффекта.
- Тщательно проверьте правила навигации, чтобы удостовериться, что средства навигации для страницы действительно установлены должным образом.
- Обычной причиной того, что страница становится застывшей, является ошибка проверки правильности или ошибка преобразования. Это несложно проверить, поместив тег `<h:messages/>` на страницу.
- Если причины ошибки все еще не удается обнаружить, установите средство отслеживания фаз. Простая реализация применяемого при этом способа была показана в главе 7, но, чтобы заняться "профессиональным шпионажем", попробуйте применить приложение FacesTrace, которое можно получить на сай-

те http://code.google.com/p/primefaces-ext/. Это приложение позволяет визуально отслеживать смену фаз (рис. 13.14).

Приложение FacesTrace является удобным в работе. Добавьте файл JAR этого приложения и файл Commons Logging JAR в каталог WEB-INF/lib конкретного веб-приложения. Добавьте объявление пространства имен в тег html:

```
xmlns:ft="http://primefaces.prime.com.tr/facestrace"
```

Затем введите тег в конце страницы JSF:

```
<ft:trace/>
```

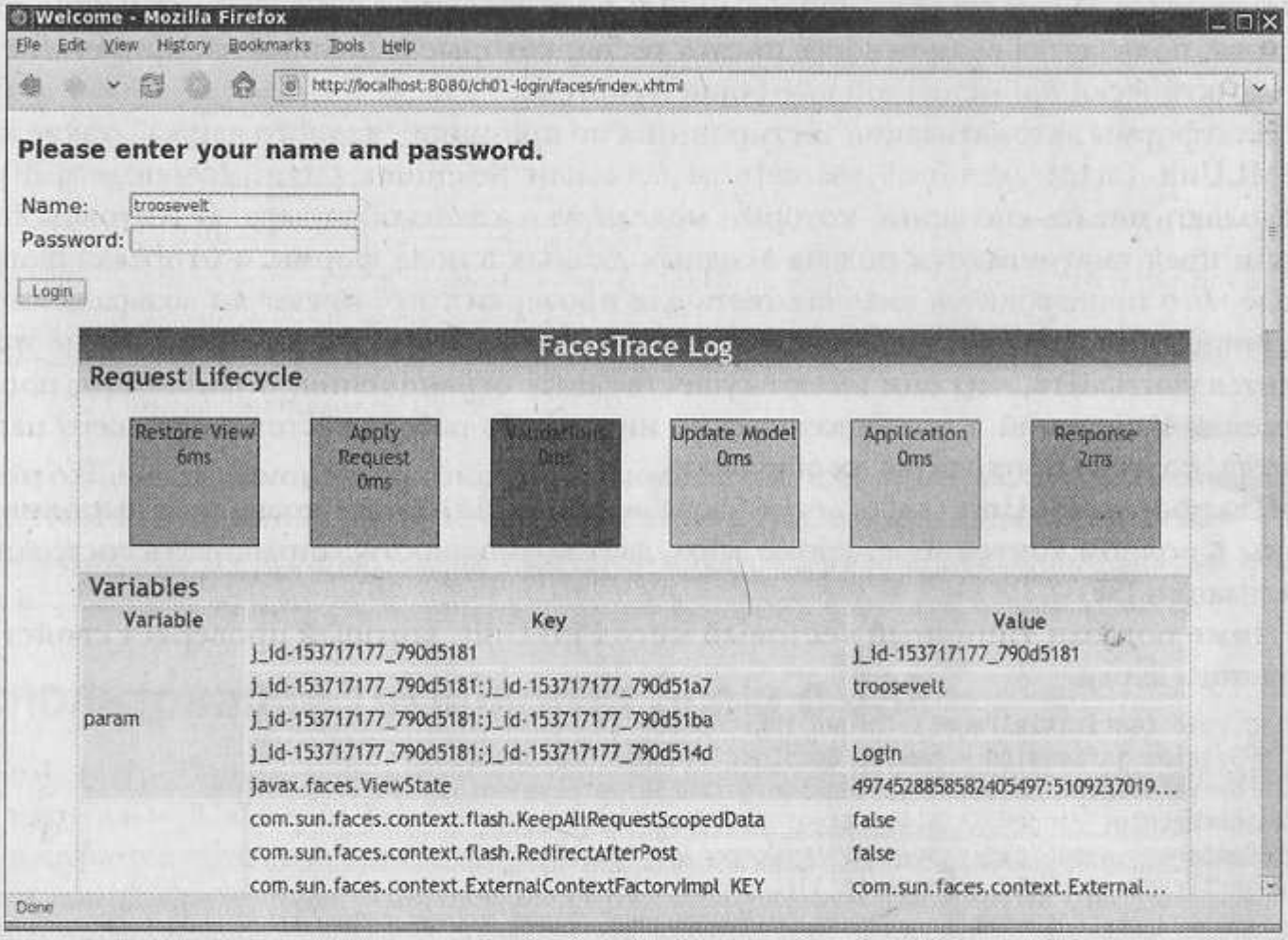


Рис. 13.14. Приложение FacesTrace в действии

Использование инструментальных средств тестирования при разработке приложения JSF

Предусмотрена возможность тестировать управляемые бины в изоляции от остального приложения, вызывая их методы, которые в обычных условиях вызываются реализацией JSF. Например, в качестве единичного теста можно вызвать методы setName и setPassword, чтобы смоделировать декодирование значений полей. Затем следует вызвать метод действия входа в систему login и проверить его возвращаемое значение. Такие действия вполне соответствуют самому духу единичного тестирования.

Но, если рассматриваемый управляемый бин связан с внутренними системами, например с базой данных, возникают проблемы. Поэтому рекомендуется отделять логику работы с базой данных от прикладной логики управляемых бинов. Такое разделение возникает естественным образом, если для доступа к базе данных используются

бины сеанса EJB. В таком случае можно заменить бины сеанса EJB фиктивными классами, которые моделируют функционирование базы данных. Безусловно, применяемый тестовый набор должен обеспечивать связывание управляемых бинов с фиктивными классами.

Задача предоставления классов, которые реализуют фиктивные средства функционирования внутренних систем, может оказаться трудоемкой. Большинство серверов приложений может эксплуатироваться в режиме "внедрения", при котором сервер приложений работает на той же виртуальной машине, что и средство прогона тестов. При этом быстродействие намного повышается по сравнению с тем, когда производятся запуск сервера приложений и подключение к нему дистанционно, кроме того, появляется возможность писать тесты, которые выполняются применительно к фактически применяемой внутренней системе.

Платформы автоматизации тестирования по принципу "черного ящика", такие как HTMLUnit (`http://htmlunit.sourceforge.net`) или Selenium (`http://seleniumhq.org`), позволяют писать сценарии, которые моделируют сеансы браузера. В тестовом сценарии предусматриваются подача входных данных в поля формы и отправка форм, после чего производится анализ ответа для проверки того, имеют ли возвращенные страницы определенные свойства. Эти платформы находят свое применение, но приходится учитывать, что они имеют существенные ограничения. В частности, после внесения изменений в пользовательский интерфейс работа тестов чаще всего нарушается, поэтому приходится их обновлять.

Платформа JSFUnit (`http://www.jboss.org/jsfunit`) также позволяет выполнять тесты в полном контейнере, кроме того, дает возможность запрашивать состояние реализации JSF.

Ниже показан типичный тестовый метод JSFUnit, который проверяет свойства страницы входа.

```
public void testInitialPage() throws IOException {
   JSFSession jsfSession = new JSFSession("/faces/index.xhtml");
   JSFServerSession server = jsfSession.getJSFServerSession();
   assertEquals("/index.xhtml", server.getCurrentViewID());
   UIComponent nameField = server.findComponent("name");
   assertTrue(nameField.isRendered());
   assertEquals("troosevelt", server.getManagedBeanValue("#{user.name}"));
}
```

Тесты JSFUnit развертываются в ходе развертывания веб-приложения вместе с файлами JAR для программных средств JSFUnit, Apache Cactus и необходимыми для них библиотеками.

Тест JSFUnit может вызываться на выполнение с помощью сервлета, который отображает результаты тестирования (рис. 13.15). Еще один вариант состоит в том, что тест можно вызвать на выполнение из командной строки. В таком случае необходимо вначале запустить сервер приложений. Файлы JAR, требуемые для программных средств JSFUnit, должны находиться в пути к классам, а в качестве системного свойства `cactus.contextURL` необходимо задать контекстный URL конкретного приложения. Средство прогона теста можно вызвать с помощью следующей команды:

```
-Dcactus.contextURL=http://localhost:8080/contextRoot
```

или добавить статический инициализатор

```
static {
   System.setProperty("cactus.contextURL", "http://localhost:8080/contextRoot");
}
```

к применяемому тестовому классу JUnit.

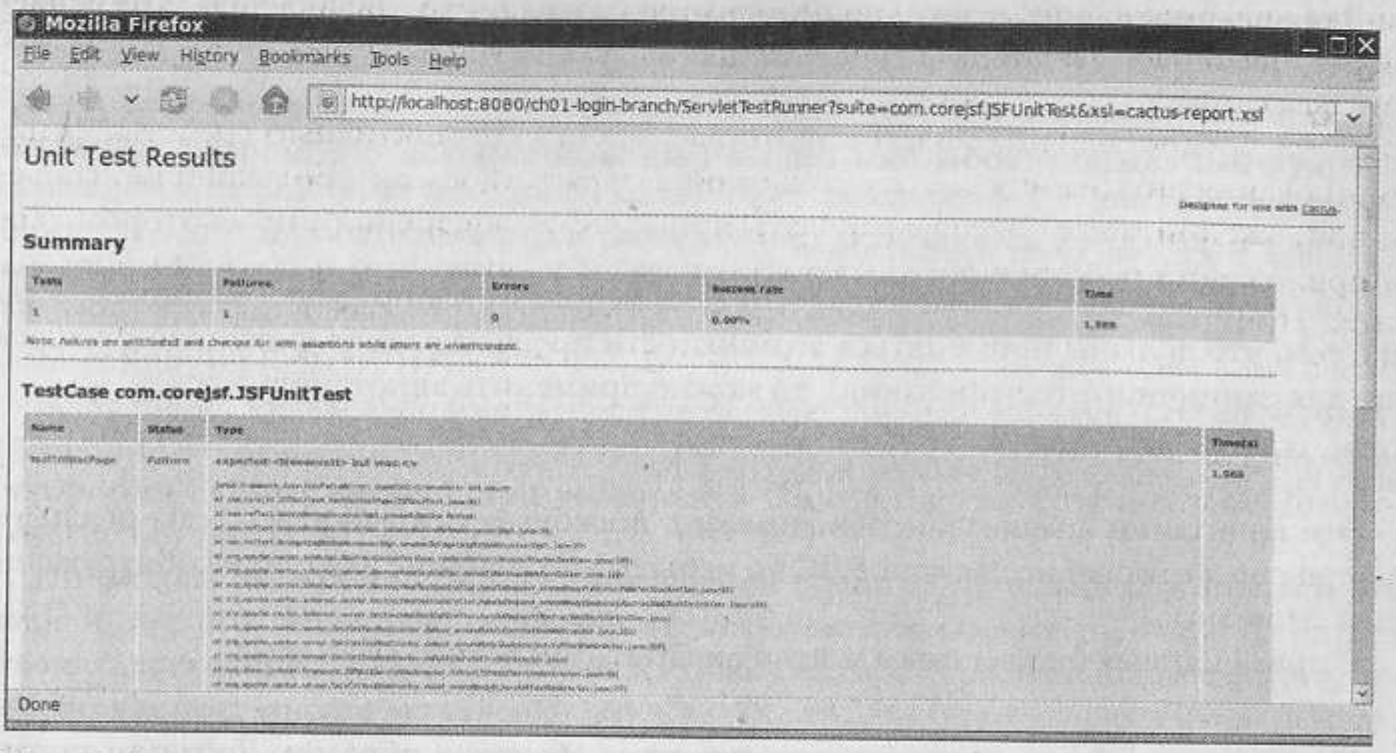


Рис. 13.15. Вывод программы JSFUnit

Ни один из указанных выше подходов не является "единственным и незаменимым" средством тестирования приложений JSF. Разработчик должен быть готов к тому, что в применяемом им наборе тестов придется предусматривать несколько разных подходов.

Использование технологии Scala с JSF

Scala (http://scala-lang.org) — это широко применяемый язык программирования для виртуальной машины Java. Как и Java, он предусматривает строгий контроль типов и является объектно-ориентированным, но поддерживает также функциональное программирование. Язык Scala оказался привлекательным для многих программистов на Java, поскольку не требует задавать столь значительный объем кода для общих конструкций, таких как свойства.

Безусловно, в приложении на языке Scala можно вызывать любой код из библиотеки Java.

Приведенные ниже рекомендации относятся к версии Java EE 6.

Чтобы реализовать управляемый бин на языке Scala, достаточно задать для него аннотацию, как и при использовании Java, но применяя при этом синтаксис аннотаций Scala. Например:

```
@Named{val value="user"} // или @ManagedBean{val name="user"}
@SessionScoped
class UserBean {
   @BeanProperty var name : String = ""
   @BeanProperty var password : String = ""
}
```

С помощью аннотации @BeanProperty формируются методы получения и методы задания свойств Java. После этого появляется возможность ссылаться на выражения языка выражений, такие как #{user.name}, на страницах JSF обычным образом.

Необходимо также разместить файл scala-library.jar в каталоге WEB-INF/lib.

Для внедрения бина сеанса, не поддерживающего состояние, в управляемом бине можно применить следующую конструкцию:

```
@EJB private[this] var mySessionBean: MySessionBean = _
```

При этом важно, чтобы бин сеанса был аннотирован с помощью аннотации @LocalBean:

```
@Stateless @LocalBean class MySessionBean { ... }
```

Еще один вариант состоит в том, что если используется признак (например, в связи с тем, что должны применяться возможности предоставления фиктивной реализации для единичного тестирования), то можно применить аннотацию @Local:

```
@Local trait MySessionBean { ... }
@Stateless MySessionBeanImpl extends MySessionBean { ... }
```

Эти аннотации являются необходимыми, поскольку каждый класс Scala реализует интерфейс, который не связан с EJB, но испытывает влияние алгоритма обнаружения бина сеанса.

В применяемых бинах сеанса можно внедрить диспетчер сущностей:

```
@PersistenceContext private[this] var em: EntityManager = _
```

Применяемые бины сущности аннотируются обычным образом. Добавьте аннотацию @BeanProperty к аннотациям автоматических методов получения и задания свойств. Например:

```
@Entity public class Credentials {
   @Id @BeanProperty var username : String = "";
   @BeanProperty var password : String = "";
}
```

На платформе Eclipse предусмотрен удобный способ ознакомления с языком Scala и платформой JSF. Для этого достаточно установить дополнение к программе Scala и добавить классы Scala к конкретному веб-проекту обычным способом.

Вполне очевидно, что задача использования языка Scala с платформами JSF и EJB является весьма несложной. Получаемая при этом непосредственная отдача не сводится к тому, что происходит запись методов получения и задания свойств для широко применяемых свойств бина. По мере дальнейшего знакомства с языком Scala разработчики все больше начинают ценить другие возможности этого языка, на которых базируются его краткость и изящество, притом что сохраняются все прочие преимущества строгого контроля типов и совместимости с Java.

Использование технологии Groovy с JSF

Еще одним широко применяемым языком программирования для виртуальной машины Java, на создание которого его разработчиков вдохновили успехи языков Ruby, Smalltalk и Python, является Groovy (http://groovy.codehaus.org). Язык Groovy обеспечивает динамическую типизацию, а код Groovy может оказаться более кратким по сравнению с эквивалентным кодом Java, поскольку объявления типов переменных и параметров могут быть опущены.

Почти любой код Java можно рассматривать как допустимый код Groovy. Благодаря этому можно легко приступить к использованию языка Groovy в конкретных приложениях JSF. Измените расширения файлов с кодом с .java на .groovy и используйте компилятор Groovy для компиляции кода Groovy в файлы .class. Платформа JSF

обеспечивает применение файлов .class, поэтому для нее не требуется знать (или не имеет значения), что при их создании использовался язык Groovy, а не Java.

Справочная реализация JSF поддерживает динамическое развертывание кода Groovy. Подробные сведения о выполняемой при этом настройке конфигурации приведены по адресу http://blogs.sun.com/rlubke/entry/groovy_mojarra. После внесения изменений в любой файл исходного кода Groovy реализация JSF автоматически проводит его повторную компиляцию и развертывает полученный файл класса в веб-приложении. Применяя язык Groovy на платформе JSF, разработчик получает почти такую же немедленную отдачу, на которую всегда рассчитывают разработчики, использующие Ruby on Rails и PHP.

Безусловно, желательно иметь поддержку интегрированной среды разработки. На платформе NetBeans предусмотрена встроенная поддержка Groovy. Если используется платформа Eclipse, необходимо установить Groovy Eclipse Plugin (http://groovy.codehaus.org/Eclipse+Plugin).

По умолчанию в приложении Groovy Eclipse Plugin задан вывод компилятора в каталог /bin/groovy. Но разработчику может потребоваться задать WEB-INF/classes в качестве каталога вывода, для чего можно воспользоваться персональными настройками Eclipse, как показано на рис. 13.16.

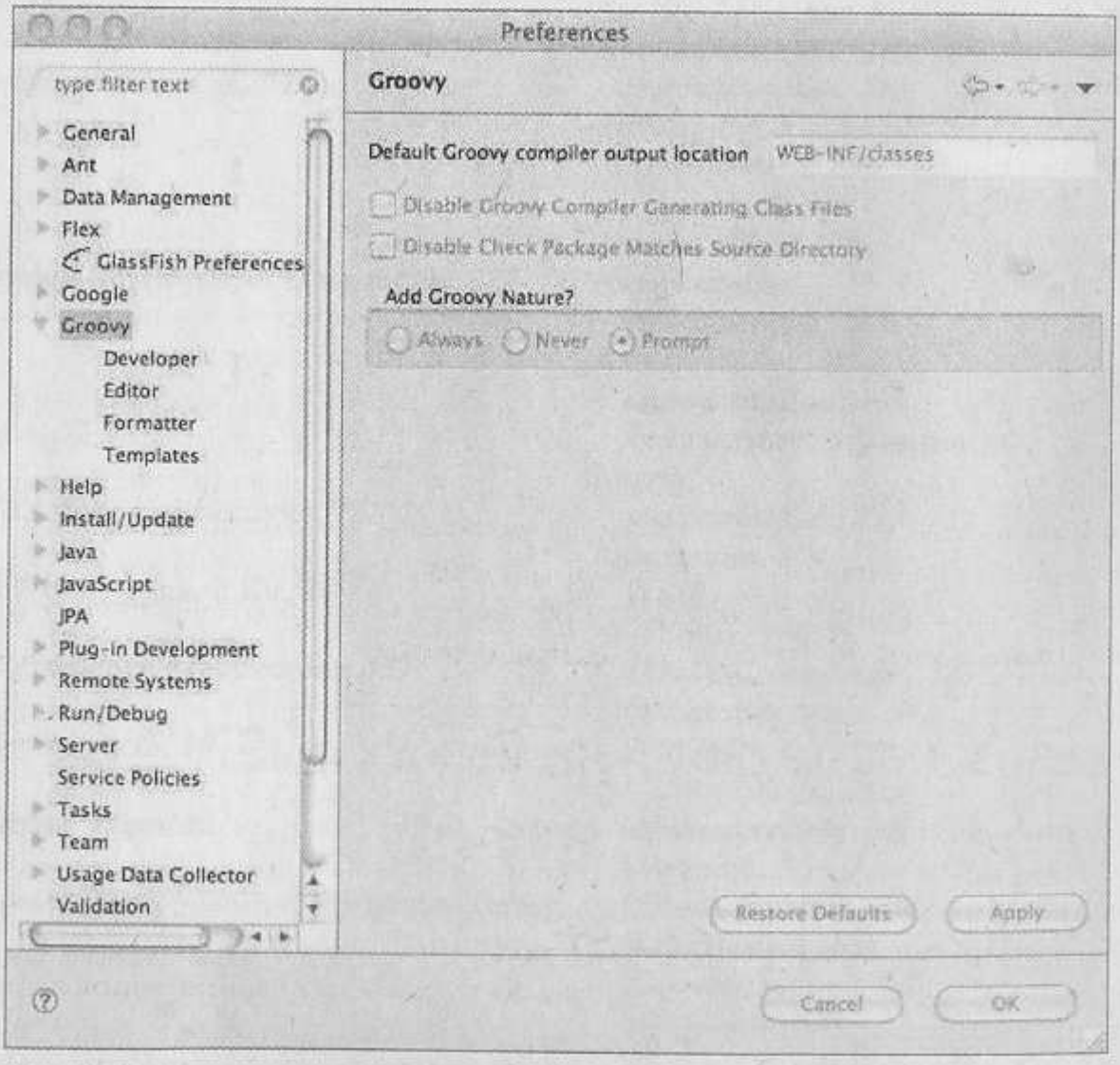


Рис. 13.16. Смена каталога вывода компилятора Groovy

Резюме

На этом изложение материала настоящей книги заканчивается. По ходу этого изложения было показано, по какому принципу платформа JSF обеспечивает разделение дизайна страницы и логики приложения и как можно без особых затруднений реализовать веб-приложения, применяя заранее подготовленные компоненты в сочетании с кодом Java. Было описано, какое место JSF занимает среди общего набора средств создания приложений Java EE и как расширить возможности JSF, если встроенных средств недостаточно для решения конкретных задач.

Предметный указатель

A

Ajax
  Asynchronous JavaScript and XML, *35*; *337*

C

CDI
  Contexts and Dependency Injection, *50*; *446*

D

DOM
  Document Object Model, *337*

E

EL
  Expression Language, *77*

I

IDE
  Integrated Development Environment, *27*

J

JAR-файл, *77*
JDBC
  Java Database Connectivity, *421*
JDK
  Java SE Development Kit, *24*
JPA
  Java Persistence Architecture, *437*
JSF
  JavaServer Faces, *19*
JSP
  JavaServer Pages, *31*
JSTL
  JavaServer Pages Standard Template Library, *49*

R

REST
  Representational State Transfer, *92*

S

SCAM
  Single select item, Collection, Array, Map, *146*
SQL
  Structured Query Language, *421*

W

WAR-файл, *23*
WSDL
  Web Services Definition Language, *463*

A

Автозавершение, *27*
Автоматизация, *518*
  тестирования, *518*
Автор
  компонента, *306*; *384*
  страницы, *306*
Авторизация, *448*
Активизация
  немедленная, *281*
Алгоритм
  сериализации, *405*
  сохранения состояния, *406*
Анкер
  именованный, *133*
Аннотация, *474*
  @EJB, *443*
  @Entity, *438*
  @FacesComponent, *327*
  @FacesRenderer, *381*
  @Id, *438*
  @LuhnCheck, *241*
  @ManagedBean, *47*
  @Named, *50*
  @PostConstruct, *66*
  @PreDestroy, *66*
  @SessionScoped, *50*
  @Stateless, *443*
  @SuppressWarnings, *438*
  отображения, *439*
  проверки правильности, *240*
Архитектура
  "модель–представление–контроллер", *37*

JPA, *437*
REST, *92*
Аспект, *105*; *192*
caption, *193*
составного компонента, *320*
Атака, *128*
на основе сценарной поддержки, *127*
Атрибут
accept-charset, *110*
action, *21*; *79*
actionListener, *281*
anchor, *117*
binding, *108*
border, *141*
collectionType, *152*
columns, *115*; *298*
converter, *108*; *109*; *225*
date, *311*
disabledClass, *141*
enabledClass, *141*
escape, *127*
first, *191*
footerClass, *193*
headerClass, *193*
HTML 4.0, *110*
id, *35*; *108*
immediate, *237*; *282*
include-viewparams, *95*
itemDisabled, *144*
label, *161*
layout, *141*
maximum, *369*
method-signature, *311*
minimum, *369*
onchange, *275*
pattern, *223*
prependId, *36*
rendered, *108*; *180*
required, *108*; *235*
resourceBundle, *398*
rowClasses, *196*
rows, *111*
showDetail, *160*
showSummary, *160*
size, *142*
style, *124*; *228*
styleClass, *126*; *228*
tabindex, *298*
template, *176*
title, *298*
tooltip, *160*
type, *280*; *311*
url, *127*
usemap, *479*
validator, *108*; *235*
value, *108*; *127*
value=, *21*
valueChangeListener, *108*; *280*
var, *188*
varStatus, *197*
xmlns, *21*
базовый, *108*
бина, *47*
события DHTML, *113*
стиля CSS, *111*
тега f:ajax, *342*
тега f:convertDateTime, *225*
тега f:convertNumber, *224*
тега f:selectItem, *145*
тега f:selectItems, *146*
тега h:button, *129*
тега h:column, *192*
тега h:commandButton, *129*
тега h:commandLink, *129*
тега h:dataTable, *191*
тега h:form, *117*
тега h:inputHidden, *121*
тега h:inputSecret, *121*
тега h:inputText, *121*
тега h:link, *129*
тега h:message, *159*
тега h:messages, *159*
тега h:outputFormat, *127*
тега h:outputLink, *132*
тега h:outputText, *127*
тега h:panelGrid, *115*
типа, *311*
Аутентификация, *448*
управляемая контейнером, *448*

Б

База данных
Derby, *427*
JavaDB, *427*
Библиотека, *112*
Apache Tomahawk, *472*
Apache Trinidad, *472*
ICEfaces, *471*
JavaScript, *337*
JSTL, *499*
PrimeFaces, *472*

Prototype, *347*
RichFaces, *471*
выгрузки файлов Commons, *472*
составных тегов, *306*
тегов HTML, *103*
тегов JSF, *103*
тегов основная, *103*
функций JSTL, *72*
Бин, *45*
CDI, *50*
customerBean, *206*
Java, *30*
localeChanger, *288*
user, *36*
вспомогательный, *49*
сеанса, *442*; *445*
сериализуемый, *47*
управляемый, *31*
Блок
try, *424*
try/finally, *422*
Браузер
Internet Explorer, *143*
Netscape, *143*
клиентский, *221*

В

Ввод
текста, *121*
Веб-платформа
JSF, *20*
Ruby on Rails, *91*
Веб-сайт
http://corejsf.com, *15*
сопровождающий, *15*
Веб-служба, *462*
Веб-технология, *13*
Версия
JSF 1.x, *20*
JSF 2.0, *20*
Визуализация
страницы, *39*
Включение
заголовка, *171*
содержимого, *169*
Вложение
тегов f:ajax, *344*
Восстановление
состояния, *405*
Выбор
единственный, *142*
множественный, *138*
пустой, *138*
Вывод
на экран, *126*
стилизованный, *126*
Выгрузка, *472*
файлов, *472*
Вызов
Ajax, *341*
метода, *72*
Выполнение, *338*
Выработка
двоичных данных, *486*
Выражение
значения, *49*; *71*
значения #{''}, *184*
значения #{form.condimentItems}, *146*
левостороннее, *70*
метода, *76*; *80*; *310*
метода answerAction, *84*
правостороннее, *70*
регулярное, *234*
слева, *71*
сложное, *75*
справа, *71*
ссылочного значения, *298*

Г

Гиперкарта, *479*
клиентская, *479*
Гиперссылка, *400*
Граница, *139*
Группа
JSF Expert Group, *13*
компонентов Ajax, *343*
правил навигации, *81*
элементов, *148*
Группирование
элементов, *148*

Д

Данные
бесформатные, *438*
построчные, *190*
приложения, *38*
серверные, *314*
столбца, *207*
строковые, *123*

табличные, *187*
тега h:dataTable, *190*
Действие, *275*
logout, *81*
пользователя, *84*
Декодирование, *40*
запроса, *40*; *372*
Декоратор, *176*
Дерево
компонентов, *39*; *50*
Диаграмма
синтаксическая, *69*
Дизайн
графический, *30*
Дизайнер
графики, *20*
Директива, *298*
include, *298*
Диспетчер, *438*
сущностей, *438*
Дисплей
двухсимвольный, *414*
Добавление
аспекта, *320*
пробела, *184*
средства проверки, *233*
строки, *204*
функций, *514*
Доступ
к данным, *207*
к компонентам, *260*
параллельный, *426*

З

Завершение
работы, *57*
Заголовок, *169*
страницы, *169*
Загрузчик
классов, *249*
Задание
значения, *109*
значения свойства, *67*
свойств бина, *46*
Закрытие
соединения, *422*
физического соединения, *425*
Запись
атрибутов элементов, *370*
разметки, *370*
Запрос
Ajax, *36*; *337*; *344*
GET, *92*
HTTP, *337*
POST, *92*
идемпотентный, *92*
кэшируемый, *92*
на перенаправление, *90*
Запуск
жизненного цикла JSF, *132*
обработчика навигации, *282*
фильтра, *487*
Знак
подстановочный, *99*
Значение, *109*
null, *80*
запроса, *221*
индексированное, *48*
карты, *148*
локальное, *43*; *222*
обязательное, *235*
переданное, *221*
преобразованное, *222*
строковое, *221*
типа Date, *329*
типа Integer, *329*
Значок, *310*

И

Идентификатор
form, *316*
клиента, *132*
компонента, *109*
компонента формы, *36*
представления, *79*
преобразователя, *225*; *247*
средства проверки, *235*
ссылки, *204*
товара, *93*
формы, *36*
целевого представления, *93*
Имя
события, *343*
события JavaScript, *343*
Индекс, *212*
отсортированный, *212*
Индикатор
выполнения, *348*
Инструкция
SQL, *421*

подготовленная, *425*
Интернационализация, *38*
Интерфейс
ActionListener, *280*
ActionSource, *367*
ActionSource2, *367*
ClientBehaviorHolder, *414*
ConstraintValidator, *241*
Converter, *244*
EditableValueHolder, *367*
Serializable, *47*
StateHolder, *405*
ValueHolder, *367*
веб-приложения, *19*
пользовательский, *19*; *165*
Исключение
ConverterException, *244*
SQLException, *424*
ValidatorException, *256*
проверки, *237*
Исполнитель
сервлетов, *24*
сервлетов Tomcat, *24*
Использование
атрибутов, *261*
Исправление
активное, *27*
Источник
данных, *421*
действия, *269*
Итерация, *187*
в коллекции, *188*
по данным, *187*
по подмножеству, *197*

К

Карта
атрибутов, *105*; *367*; *384*
атрибутов компонента, *383*
выражений значения, *367*
запроса, *106*; *501*
интеллектуальная, *384*
компонентов аспекта, *367*
ресурсов, *128*
сеанса, *482*
ссылок на значения, *384*
Карточка
кредитная, *223*
Каталог
LDAP, *448*
META-INF, *377*
resources, *308*
resources/javascript, *348*
resources/util, *308*
WEB-INF/lib, *377*; *465*
исходный, *27*
корневой контекста, *131*
развертывания, *26*
Класс
ActionLogger, *281*
CSS, *138*
DataModel, *209*
ExternalContext, *510*
FacesContext, *284*
FileUploadRenderer, *477*
MessageFormat, *53*
Problem, *82*
ProblemBean, *55*
PropertyEditor, *70*
QuizBean, *83*
ResponseWriter, *370*
ResultSet, *422*
TabbedPaneRenderer, *399*
TableData, *209*
UIComponent, *367*
UIComponentBase, *415*
UITabbedPane, *406*
UserBean, *47*
компонента, *366*
редактора свойств, *70*
столбца, *196*
строки, *196*
Ключ
карты, *148*
первичный, *438*
Кнопка, *130*
декремента, *368*
инкремента, *368*
нажимная, *130*
передачи формы, *221*
счетчика, *372*
Код
JavaScript, *118*; *322*
преобразования, *221*
проверки правильности, *221*
страницы, *32*
языка, *52*; *285*
Кодирование, *40*
Коллекция, *188*
прослушивателей событий, *367*
Команда, *109*

Комплект
JDK, *24*
Композиция, *166*
Компонент, *305*
ADF Faces, *28*
Ajax, *409*
execute, *36*
h:commandlink, *106*
icon, *310*; *311*
JSF, *19*; *193*; *197*
render, *36*
UICommand, *41*
UIData, *188*
UIInput, *41*
UISpinner, *394*
ввода, *50*; *200*
вспомогательный, *326*
вывода, *50*; *200*
выгрузки файлов, *472*
дополнительный, *471*
дочерний, *329*
календаря, *365*
командный, *275*; *284*
контейнерный, *194*
настраиваемый, *307*
немедленно активизируемый, *282*
области окна с вкладками, *396*
пользовательский, *38*
с автоматическим завершением, *358*
составной, *305*
счетчика, *365*; *393*
текстовый, *184*
эхо-повтора, *341*
Конструктор
QuizBean, *56*
UISpinner, *373*
общедоступный, *48*
по умолчанию, *438*
пользовательских интерфейсов, *46*
Контейнер
EJB, *445*
JSF, *221*
сервлетов, *61*
Контекст
faces, *158*
веб-приложения, *127*
внешний, *452*
Контроллер, *38*
Конфигурация
сервлета, *33*
Кэширование, *425*
подготовленных инструкций, *425*

Л

Логика
программы, *45*
Локализация
приложения, *52*
составного компонента, *315*
страницы, *135*
файла связки, *52*

М

Макет
lineDirection, *141*
pageDirection, *141*
бокового меню, *169*
главного содержимого, *169*
заголовка, *169*
списка, *161*
страницы, *166*; *187*
Массив
имен, *211*
Мемориал
Rushmore, *276*
Меню, *142*
боковое, *167*
каскадное, *150*
Местонахождение, *179*
шаблона, *179*
Метаключ, *110*
Метка, *161*
Age, *161*
Метка-заполнитель, *53*; *286*
Метод
close, *423*
decode, *204*
deleteRow, *203*
encodeChildren, *399*
get, *48*
get/set, *30*
getConnection, *423*
getConvertedValue, *329*
getConverterWithType, *382*
getInitParameter, *510*
getNames, *209*
getRowIndex, *209*
HttpServletRequest, *452*
prepareStatement, *425*
renderResponse, *284*

restoreState, *405*
saveState, *405*
set, *48*
setConverter, *373*
setSubmittedValue, *373*
Statement.executeQuery, *206*
TableData.names, *211*
toInteger, *373*
toString, *147*
validateName(), *346*
действия, *85*
декодирования, *477*
задания свойства, *49*
перегруженный, *72*
получения свойства, *49*
составления, *172*
Механизм
исключения проверки, *237*
навигации, *79*
памяти, *105*
Мини-приложение, *217*
постраничного просмотра, *217*
Модель, *38*
DataModel, *211*
данных, *211*
документа объектная, *337*
исходная, *211*
компонентов, *19*
программирования, *19*; *443*
сортировки, *211*
таблицы, *209*
таблицы Swing, *190*
фильтра сортировки, *212*
Модуль
подготовки к отображению, *40*
разрешения, *74*

H

Набор
компонентов ADF Faces, *472*
компонентов Ajax, *472*
компонентов Tomahawk, *472*
преобразователей, *225*
результирующий, *206*
Навигация
динамическая, *80*
с вытеснением, *94*
статическая, *79*
условная, *100*
Настройка
конфигурации, *52*; *66*; *307*
меток, *307*
пула соединений, *423*
ресурса почты, *458*
Номер
кредитной карточки, *244*
строки текста, *209*

O

Область
действия бина, *60*
действия диалога, *60*; *63*
действия запроса, *60*; *62*
действия пользовательская, *60*
действия приложения, *60*; *62*
действия просмотра, *64*
действия сеанса, *60*
окна с вкладками, *296*
содержимого, *174*
текстовая, *124*
Обновление
значений модели, *222*
модели, *221*
Оболочка, *206*
Обработка
аспектов, *399*
введенных данных, *221*
ошибок, *38*
параметров запроса, *132*
платежей, *223*
событий, *270*; *302*
Обработчик
кнопки, *91*
навигации, *79*
представлений, *165*
событий, *269*
тега, *39*
Образец
правильного значения, *227*
Объединение
вызовов Ajax, *356*
событий, *356*
соединений в пул, *423*
Объект, *45*
CachedRowSet, *207*
Connection, *423*
Element, *347*
FacesMessage, *248*
Person, *438*
tableData, *211*

UserBean, *47*
класса UserBean, *47*
компонента, *221*
передачи, *482*
представления, *405*
скалярный, *188*
соединения, *421*
сущности, *443*
флеш-памяти, *91*
Объявление
пространства имен, *31*; *165*; *308*
связки ресурсов, *51*
схемы, *52*
Ограничение
проверки правильности, *239*
Окно
всплывающее, *492*
с вкладками, *296*
списка, *107*; *142*
Оператор
арифметический, *75*
выбора трехзначный, *75*
логический, *75*
сравнения, *75*
Операционная система
Linux, *15*
Mac OS X, *15*
Solaris, *15*
Windows, *15*
Операция
empty, *75*
редактирования, *79*
Описание
конфигурации, *66*
Определение
преобразователя, *225*
Ответ
Ajax, *350*
перенаправления, *90*
Откат, *426*
Отладка
приложений Ajax, *515*
Отображение
номеров строк, *209*
расширения, *81*
Отправка
данных формы, *79*
почты, *458*
Отслеживание
сеансов, *61*
смены фаз, *517*
Отсчет
значений индексов, *72*
Оформление, *176*
модели, *210*
модели таблицы, *210*
содержимого, *176*
Оценка
отсроченная, *49*
Очередь
событий, *270*
Ошибка
HTTP, *501*
преобразования, *226*; *235*
проверки правильности, *235*

П

Пакет
javax.enterprise.context, *50*
javax.faces.bean, *47*; *50*
JSF 2.0, *24*
Палитра
конструктора, *46*
Панель, *114*
Параметр
доступа, *460*
запроса, *93*; *132*
метода, *77*
представления, *93*
средства проверки, *235*
Перебор
элементов, *148*
Передача
параметров, *178*
представительного состояния, *92*
сообщения об ошибке, *227*
формы, *117*
формы принудительная, *271*
Перезагрузка
страницы, *122*
Перезапись
URL-адреса, *61*
Переменная
связки, *62*
системная classpath, *333*
флеш-памяти, *91*
Перенаправление, *90*
Переход
на локаль, *134*
Платформа
Facelets, *165*

GlassFish, *427*
Java EE 6, *47*
JSF, *19*, *305*
Ruby on Rails, *305*
Sitemesh, *176*
Struts, *305*
Tiles, *176*
Tomcat, *429*
компонентно-ориентированная, *19*, *305*
проверки правильности бинов, *240*
Поведение
с ветвлением, *81*
Подготовка
к отображению, *338*
Поддержка
Ajax, *39*
интернационализации, *52*
нескольких диалогов, *63*
свойств, *48*
состояния, *60*
состояния компонента, *366*
состояния сеанса, *62*
Поле
адреса, *90*
аннотированное, *439*
ввода, *109*
ввода пустое, *235*
ввода скрытое, *261*
вывода, *109*
секретное, *122*
скрытое, *117*; *123*; *400*
текстовое, *122*
экземпляра класса, *109*
Полоса
прокрутки, *216*
Получение
соединения, *421*
Порядок
правил, *86*
Построитель
программы визуальный, *28*
Почта
электронная, *460*
Правило
действия, *70*
навигации, *21*; *81*
поведения, *337*
поведения клиентское, *416*
преобразования, *70*
преобразования строки, *70*
синтаксическое, *76*
сопоставления, *33*
Представление, *79*
Преобразование, *221*
данных, *38*
строки, *70*
Преобразователь, *105*; *244*
f:convertNumber, *223*; *228*
встроенный, *223*
логического значения, *229*
пользовательский, *227*
стандартный, *222*
чисел, *227*
Префикс
/faces, *33*
/secure/, *99*
composite:, *306*
form:, *316*
h:, *31*
тега, *21*
Приведение
типов, *381*
Признак, *520*
Приложение
FacesTrace, *516*
JavaQuiz, *82*
planets, *167*
Rushmore, *276*
демонстрационное, *22*
Пример
кода, *24*
преобразователя, *231*
приложения JSF, *20*
проверки правильности, *237*
Присваивание
имен, *370*
Проверка
данных, *38*
локальных значений, *222*
номера кредитной карточки, *241*
по формуле Луна, *241*
правильности, *221*
правильности бинов, *239*
Программное обеспечение
Ajax, *13*
Ajax4jsf, *13*
Eclipse, *15*
Facelets, *13*
GlassFish, *15*
ICEFaces, *13*
JSF Templates, *13*
NetBeans, *15*

Pretty Faces, *13*
REST, *13*
RichFaces, *13*
Seam, *13*
Tomcat, *15*
Прокрутка, *216*
Прослушиватель
action, *104*; *132*
действий, *275*
изменений значений, *269*
фаз, *288*; *486*
Просмотр
исходного кода страницы, *41*
постраничный, *217*; *488*
Пространство
имен, *179*; *308*
имен JavaScript, *349*
Протокол
итеративный, *422*
Пул, *423*
соединений, *423*

P

Развертывание
приложения JSF, *25*
Разграничитель
#{...}, *32*; *49*
${...}, *49*
элементов, *370*
Размер, *139*
Разметка, *172*
HTML, *126*
XHTML, *172*
таблицы, *196*
Расположение
вертикальное, *138*
горизонтальное, *138*
Реализация, *307*
Ajax, *340*
Facelets, *170*
JSF справочная, *24*
Struts, *176*
значка, *310*
составного компонента, *309*
Регистрация
средства проверки, *257*
Редактирование, *200*
строки, *200*
таблицы, *200*
ячейки таблицы, *200*
Результат, *79*
again, *85*
done, *85*
failure, *85*
startOver, *85*
success, *81*; *85*
Резюме, *158*
сообщения, *158*
Рефакторизация, *27*

C

Сведения
о модели, *209*
Свойство, *30*
answer, *56*
current, *56*
editable, *201*; *203*
index, *197*
languages, *152*
opener, *494*
problems, *56*
score, *56*
sequence, *56*
statesForCountry, *494*
бина, *45*
Связка
локализованная, *52*
ресурсов, *229*
сообщений, *51*; *57*; *229*
Семейство, *380*
компонентов, *326*; *381*
Сервер
GlassFish, *24*
приложений, *24*; *423*
приложений Java EE, *19*
Сервлет
Faces, *33*
контроллера, *270*
Сетка, *126*
панели, *126*
Сигнатура
метода, *311*
Символ
подчеркивания, *346*
пробельный, *184*
разделителя, *105*
Система
уведомления, *289*
Ситуация
исключительная, *422*

Скобка
квадратная, *71*
Служба, *421*
внешняя, *421*
Смена
страницы, *36*
Смешивание
разметки и кода, *30*
Событие
blur, *340*
DHTML, *113*; *139*; *191*
keyup, *337*; *340*
активизированное пользователем, *269*
действия, *269*; *275*
изменения значения, *269*; *271*
системное, *269*; *290*
фазы, *269*; *288*
Соглашение
Java Blueprints, *23*
о программировании, *47*
об именовании, *30*
Создание
бина, *46*
приложения JSF, *24*
пула, *423*
сообщения, *158*
Сообщение, *158*
заменяющее, *229*
об ошибке, *227*; *499*
с переменными частями, *52*
составное, *127*
специализированное, *231*
стандартное, *229*
Сортировка, *210*
индексов строк, *211*
Состояние
веб-страницы, *46*
начальное, *406*
несогласованное, *222*
нормальное, *222*
Сохранение
изменений, *200*
состояния, *405*; *406*
состояния частичное, *406*
Спецификация
HTML 4.01, *150*
JavaBeans, *45*; *48*
JSF, *13*; *380*
JSR 299, *50*
бинов, *30*
преобразователя, *253*
Список
дочерних компонентов, *367*
исполнения, *43*
ненумерованный, *161*
подготовки к отображению, *43*
Способ
прокрутки, *216*
редактирования, *202*
Среда
Eclipse, *27*
JBoss Developer Studio, *27*
MyEclipse, *27*
NetBeans, *27*
Oracle JDeveloper, *27*
Rational Application Developer, *27*
разработки интегрированная, *24*; *27*; *170*
Средство
визуализации альтернативное, *39*
инструментальное построителя программы, *27*
интернационализации Java, *52*
навигации, *82*; *129*
отслеживания сеансов, *61*
поддержки состояния, *406*
постраничного просмотра, *217*; *488*
проверки, *43*; *234*
самодостаточное, *414*
связывания, *108*
сериализации, *405*
Ссылка
командная, *134*; *284*
на компонент, *381*
на метод, *81*
регистрационная, *321*
Стадия
проектирования, *34*
разработки, *34*
Стандарт
ISO-639, *52*
Стек технологий
Java EE, *13*
Степень
серьезности, *248*
Стиль
CSS, *111*
REST, *92*
Столбец
алфавитно-цифровой, *192*
Страница
XHTML, *31*
входа, *167*

застывшая, *516*
начальная, *34*; *167*
планеты, *167*
приветствия, *25*
свойств, *46*
Стратегия
перехода на аварийный режим, *61*
Строка
выбранная, *210*
идентификатора, *382*
результата, *81*; *93*
сообщения, *51*; *250*
Структура
каталогов, *23*
Сумма
в валюте, *53*
Суффикс
.faces, *81*
number,currency, *53*
локали, *52*
Сущность, *437*
Credentials, *438*
символьная, *127*
Сфера, *451*
Схема
именования, *380*
Сценарий
построения, *24*
Сцепление
строковых значений, *75*
Счетчик, *365*
размера шрифта, *409*
Считывание
свойств бина, *46*

T

Таблица
HTML, *187*
базы данных, *205*
стилей, *112*; *228*
стилей CSS, *167*
Таймер
JavaScript, *356*
Тег
corejsf:planet, *179*
f:ajax, *337*
f:attribute, *105*; *255*
f:convertDateTime, *225*
f:facet, *105*
f:param, *105*; *132*
f:selectItems, *145*
f:setPropertyActionListener, *287*
f:validator, *235*
f:valueChangeListener, *280*
f:verbatim, *134*
f:view, *289*
Facelets, *165*
graphicImage, *112*
h:column, *188*
h:commandButton, *21*
h:commandLink, *298*
h:dataTable, *187*
h:inputHidden, *123*
h:inputSecret, *21*
h:inputText, *21*
h:link, *132*
h:message, *160*; *228*
h:messages, *228*
h:outputScript, *323*
h:outputText, *126*
h:panelGrid, *298*
h:selectBooleanCheckbox, *138*; *139*
h:selectManyCheckbox, *140*
h:selectManyListbox, *142*
h:selectOneRadio, *140*
HTML, *21*; *106*
outputScript, *112*
script, *183*
selectMany, *138*
selectOne, *138*
ui:composition, *170*
ui:debug, *181*
ui:decorate, *176*
ui:define, *170*; *178*
ui:include, *169*
ui:insert, *169*
ui:param, *178*
ui:remove, *183*
ui:repeat, *196*
validatorScript, *505*
view-param вложенный, *95*
выбора, *137*
дочерний, *166*; *322*
компонента, *233*
окна списка, *142*
основной, *103*
сообщения, *228*
составного компонента, *306*; *322*
специализированный, *178*
средства проверки, *233*
таблицы данных, *187*

Текст
заголовка, *173*
Телефон
сотовый, *366*
Тестирование, *517*
единичное, *520*
Технология
Ajax, *35*
CDI, *50*
Facelets, *165*
JavaServer Faces, *234*
JSF, *46*
Swing, *19*
представления, *165*
Тип, *380*
boolean, *223*
double, *223*
int, *223*
Queue, *152*
Set, *152*
SortedSet, *152*
атрибута, *311*
возвращаемый, *80*
компонента, *327*; *380*
перечислимый, *70*; *151*
Торговля
электронная, *61*
Транзакция, *426*

У

Уникальность, *373*
Уничтожение
бина, *46*
Упаковка
составного компонента, *333*
Управление
соединениями, *422*
Уровень
бизнес-логики, *30*
представления, *28*
Условие
проверки правильности, *222*
Установка
локали, *54*
локали динамическая, *54*
Утечка
ресурсов соединения, *423*

Ф

Фабрика
диспетчера сущностей, *439*
Фаза
восстановления представления, *42*
вызова приложения, *43*
обновления значений модели, *43*
подготовки ответа к отображению, *42*; *43*
применения значений запроса, *42*
проверки правильности процесса, *43*
Файл
faces-config.xml, *81*
styles.css, *161*
WEB-INF/faces-config.xml, *21*
WSDL, *464*
журнала, *27*
конфигурации web.xml, *33*
конфигурации приложения, *85*; *226*
описания, *376*
описания тегов, *262*
Фиксация, *426*
автоматическая, *426*
Фильтр, *474*
сервлетов, *472*; *474*
Фильтрация, *210*
Флаг, *285*
немецкого языка, *285*
Флажок, *153*
выбора языка, *153*
Флеш-память, *91*
Формат
choice, *53*
номера кредитной карточки, *245*
строки шаблона, *223*
Форматирование
строки сообщения, *53*
числа, *53*
Формирование
закладки, *95*
Формула
Луна, *241*
Функция
checkForm(), *323*
checkPassword, *118*
doPopup, *492*
JavaScript, *343*
JSTL, *73*
showProgress, *348*
spin, *393*
декодирования, *366*

дополнительная, *180*
кодирования, *366*
обработки ошибок, *350*
обратного вызова, *348*
сравнения данных, *212*

Х

Хеш-таблица, *41*
Хранилище
централизованное, *51*

Ц

Целостность
базы данных, *426*
данных, *222*
модели, *222*
Цикл
жизненный, *41*
запроса, *92*

Ч

Часть
переменная, *166*

Ш

Шаблон, *166*
именования, *48*

Э

Экземпляр
ArrayDataModel, *211*
SelectItem, *147*
SelectItemGroups, *149*
массива, *189*
Экранирование, *94*
Элемент, *188*
<pre>, *124*
DOM, *351*
from-action, *100*
from-outcome, *100*
from-view-id, *82*
HTML, *110*; *128*
img, *127*
key, *68*
managed-bean, *67*
navigation-case, *100*
null-value, *68*
optgroup, *150*
Realm, *452*
redirect, *91*
span, *126*
to-view-id, *91*
value, *68*
видимый, *142*
выбора единственный, *144*
навигации, *99*
прокручиваемый, *217*
строки, *203*
управления, *216*
управления формой, *119*
формы, *118*
Эхо-повтор, *341*

Я

Язык
Groovy, *520*
JPQL, *438*
Python, *520*
Ruby, *520*
Scala, *519*
Smalltalk, *172*; *520*
SQL, *440*
выражений, *49*; *77*
выражений JSF, *32*
выражений значения, *49*
китайский, *141*

JAVA™ 2
БИБЛИОТЕКА ПРОФЕССИОНАЛА
ТОМ 1. ОСНОВЫ
8-Е ИЗДАНИЕ

Кей Хорстманн
Гари Корнелл

www.williamspublishing.com

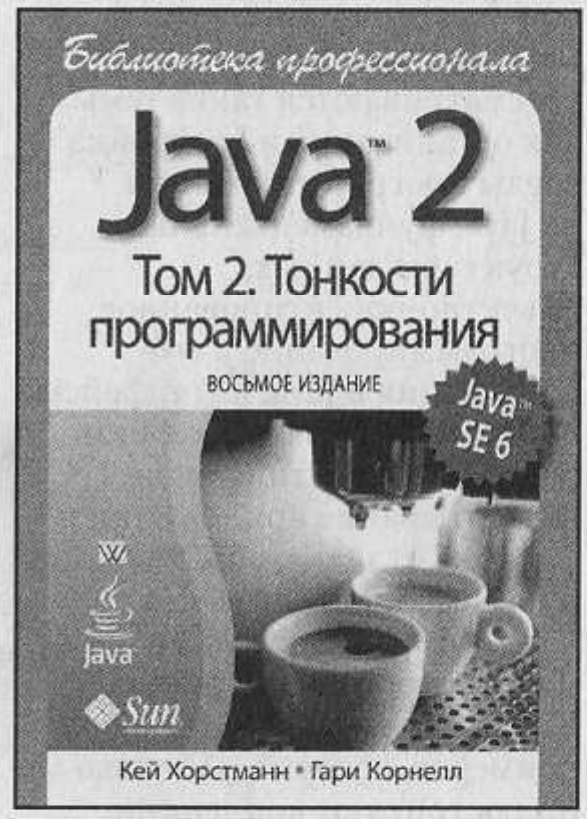
Книга ведущих специалистов по программированию на языке Java представляет собой обновленное издание фундаментального труда, учитывающее всю специфику новой версии платформы Java SE 6. Подробно рассматриваются такие темы, как организация и настройка среды программирования на Java, фундаментальные структуры данных, объектно-ориентированное программирование и его реализация в Java, интерфейсы, программирование графики, обработка событий, Swing, развертывание приложений и аплетов, отладка, обобщенное программирование, коллекции и построение многопоточных приложений. Книга изобилует множеством примеров, которые не только иллюстрируют концепции, но также демонстрируют способы правильной разработки, применяемые в реальных условиях. Книга рассчитана на программистов разной квалификации, а также будет полезна студентам и преподавателям.

ISBN 978-5-8459-1378-4

в продаже

JAVA™ 2
БИБЛИОТЕКА ПРОФЕССИОНАЛА
ТОМ 2. ТОНКОСТИ ПРОГРАММИРОВАНИЯ
ВОСЬМОЕ ИЗДАНИЕ

Кей Хорстманн
Гари Корнелл



www.williamspublishing.com

Книга ведущих специалистов по программированию на языке Java представляет собой обновленное издание фундаментального труда, учитывающее всю специфику новой версии платформы Java SE 6. Подробно рассматриваются такие темы, как новые средства ввода-вывода, использование XML, API-интерфейсы работы в сети и взаимодействия с базами данных (JDBC), интернационализация, построение графических интерфейсов, безопасность, JavaBeans, взаимодействие с распределенными объектами и кодом, написанным на языке сценариев, а также платформенно-ориентированные методы, позволяющие обращаться на низком уровне к функциям операционной системы. Книга изобилует множеством примеров, которые не только иллюстрируют концепции, но также демонстрируют способы правильной разработки, применяемые в реальных условиях.

ISBN 978-5-8459-1482-8 **в продаже**

ПОЛНЫЙ СПРАВОЧНИК ПО JAVA™ 7-Е ИЗДАНИЕ

Герберт Шилдт

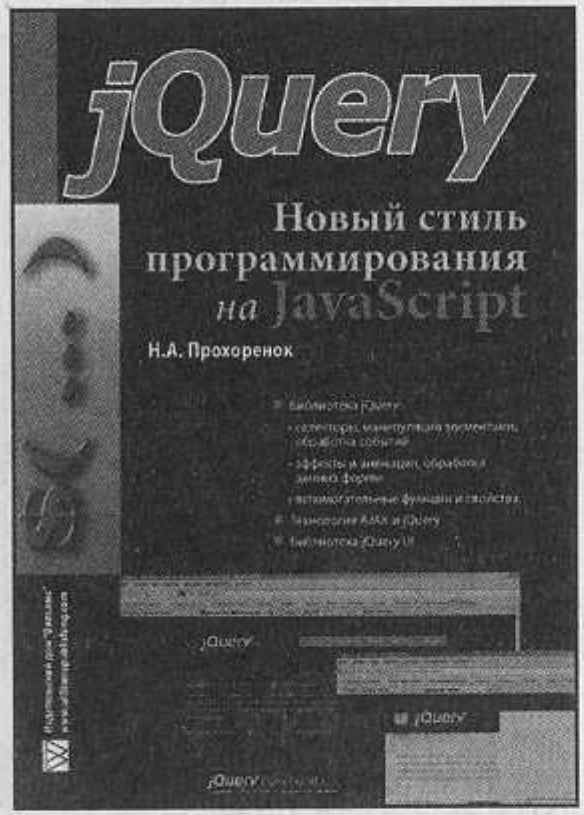
www.williamspublishing.com

Книга известного "гуру" в области программирования посвящена новой версии одного из наиболее популярных и совершенных языков – Java. Построенная в виде учебного и справочного пособия, она является превосходным источником исчерпывающей информации по последней версии платформы Java, Java SE 6, и позволяет практически с нуля научиться разрабатывать приложения и аплеты производственного качества. Помимо синтаксиса самого языка и фундаментальных принципов программирования, в книге подробно рассматриваются такие сложные вопросы, как ключевые библиотеки Java API, каркас коллекций, создание аплетов и сервлетов, AWT, Swing и Java Beans. Немалое внимание уделяется вводу-выводу, работе в сети, регулярным выражениям и обработке строк.

ISBN 978-5-8459-1168-1 **в продаже**

JQUERY
Новый стиль программирования на JavaScript

Н.А. Прохоренок



www.williamspublishing.com

Книга является справочником по JavaScript-библиотеке jQuery. Рассматриваются функциональные возможности библиотеки, полезные для максимально широкого круга задач, включая механизм селекторов, манипулирование параметрами и содержимым элементов DOM-модели документа, обработку событий и данных форм. Продемонстрированы возможности использования технологии AJAX для обмена данными с сервером без перезагрузки страницы. Описаны как базовые свойства и методы объекта XMLHttpRequest, так и интерфейс доступа к AJAX, предоставляемый библиотекой jQuery. Кроме того в книге рассматривается библиотека визуальных компонентов jQuery UI, предоставляющая готовые решения, которые может использовать любой разработчик, даже не владея основами jQuery и JavaScript. Эта библиотека позволяет создавать в документе нестандартные компоненты, панели с вкладками и др.

ISBN 978-5-8459-1603-7

в продаже

JAVA™
РУКОВОДСТВО ДЛЯ НАЧИНАЮЩИХ

Герберт Шилдт

www.williamspublishing.com

Книга разделена на 15 модулей. В начале автор знакомит читателя с основными элементами языка: типами данных, операторами, управляющими выражениями. Здесь же приведена информация об объектных средствах: классах, объектах и методах. Далее следуют вопросы, не владея которыми невозможно написать даже простые программы: доступ к объектам и членам класса, особенности использования ссылок, средства ввода-вывода, поддержка событий, обработка исключений и т.д. Большое внимание уделяется объектно-ориентированному программированию, рассматриваются инкапсуляция, наследование и полиморфизм. Несмотря на то что большая часть книги посвящена Java-приложениям, читатель также найдет в ней сведения об аплетах. Помимо базовых вопросов Java-программирования, в книге рассматриваются элементы, реализованные в последних версиях Java, например универсальные классы и нумерованные типы. Завершается книга рассмотрением Swing — мощного инструмента создания графических пользовательских интерфейсов.

ISBN 978-5-8459-1440-8

в продаже

ГИБКАЯ РАЗРАБОТКА ПРИЛОЖЕНИЙ НА JAVA С ПОМОЩЬЮ SPRING, HIBERNATE И ECLIPSE

Анил Хемраджани

www.williamspublishing.com

В этой книге основное внимание уделено разработке и в меньшей степени инфраструктуре. Другими словами, больше внимания уделено технологиям разработки приложений, таким как Spring, Hibernate и Eclipse, а не программным продуктам, таким как серверы приложений или базы данных. Все, что представлено в этой книге, опробовано в реальных приложениях, которые успешно работают (некоторые в кластеризуемой среде сервера приложений). Одна из задач этой книги заключается в краткости и конкретности, поэтому автор решил практически полностью сосредоточиться на разработке хорошо масштабируемого приложения. В данной книге, кроме технологий Spring, Hibernate и Eclipse, также описаны альтернативные и конкурирующие технологии.

ISBN 978-5-8459-1375-3 **в продаже**

JAVA МЕТОДИКИ ПРОГРАММИРОВАНИЯ ШИЛДТА

Герберт Шилдт

www.williamspublishing.com

Книга общепризнанного авторитета в области языков программирования построена в виде готовых рецептов, иллюстрирующих передовые приемы и подходы в программировании на Java. В каждом рецепте описан набор ключевых ингредиентов, таких как классы, интерфейсы и методы. Затем даются шаги, которые потребуется предпринять, собирая из этих ингредиентов последовательность кода, необходимого для достижения нужного результата. Подробно рассматриваются такие темы, как обработка строк, использование регулярных выражений, работа с файлами, создание аплетов и сервлетов, построение графических интерфейсов с помощью Swing, применение каркаса коллекций, многопоточность и форматирование данных. Книга рассчитана на программистов разной квалификации, а также будет полезна студентам и преподавателям дисциплин, связанных с программированием на языке Java.

ISBN 978-5-8459-1395-1

в продаже

JAVA
КАРМАННЫЙ СПРАВОЧНИК

Тимоти Фишер

www.williamspublishing.com

В данную книгу включено более 100 фрагментов кода, с помощью которых читатель сможет быстро реализовать требуемые ему функциональные возможности на языке Java: обработку текстовых строк, чисел и дат; поиск по образцу с помощью регулярных выражений; работу с базами данных и XML; создание многопоточных приложений; выполнение операций ввода-вывода; создание клиентских и серверных сетевых приложений и др. Приведенные в ней примеры упорядочены по принципу "от простого к сложному" и отражают практически весь спектр задач, с которыми обычно сталкивается программист при создании типичного приложения на Java. Настоящая книга — не учебник по языку Java, не введение в него и, тем более, не полное руководство. В книге описано лишь ограниченное количество классов и API. Эта книга будет полезна как для опытных программистов на Java, так и для новичков.

255921 568193 255921 568193

ISBN 978-5-8459-1392-0

в продаже