Разработка веб-приложений в Yii 2

Марк Сафронов

[PACKT] PUBLISHING

ДМК ИЗДАТЕЛЬСТВО

Mark Safronov, Jeffrey Winesett

Web Application Development with Yii 2 and PHP

[PACKT] PUBLISHING
open source
community experience distilled

Марк Сафронов

Разработка веб-приложений в Yii 2

Москва, 2015

УДК 004.738.5:004.45Yii
ББК 32.973.202-018.2
С21

Сафронов М.
С21 Разработка веб-приложений в Yii 2. – М.: ДМК Пресс, 2015. – 392 с.: ил.

ISBN 978-5-97060-252-2

Yii – это высокопроизводительный фреймворк, используемый для быстрой разработки веб-приложений на PHP. Он хорошо спроектирован, имеет прекрасную поддержку, его легко изучить и легко сопровождать. Эта книга на практических примерах покажет вам самые важные возможности Yii 2. Сквозь всю книгу проходит пример построения реального приложения – каждая глава представляет новую функциональность и показывает приёмы тонкой настройки. Вместо того, чтобы пытаться быть всеобъемлющим справочником по Yii 2, издание является руководством по тем сведениям, которые важно знать практикующему разработчику.

Издание предназначено для веб-разработчиков как уже знакомых с Yii, так и начинающих пользователей фреймворка.

УДК 004.738.5:004.45Yii
ББК 32.973.202-018.2

ISBN 978-1-78398-188-5 (анг.)
ISBN 978-5-97060-252-2 (рус.)

Содержание

Вступительное слово от разработчика Yii

Я слежу за всеми новыми материалами по Yii и был обрадован и удивлён появлению книги по Yii 2.0 ещё до релиза фреймворка. Финальная английская версия ушла в печать, в то время как во фреймворке были сделаны довольно серьёзные изменения, и по этому поводу я был настроен несколько скептически: а вдруг там устаревшая или неточная информация?

Опасения не оправдались: материал был действительно хороший, в книге показаны как возможности фреймворка, так и лучшие практики разработки. Да, были небольшие недочёты, но в общем всё было отлично.

Когда я узнал, что готовится перевод книги на русский, опасения вернулись, но оказались напрасными: перевод делал сам автор, попутно обновляя то, что успело поменяться, и исправляя те неточности, что всё-таки пробрались в англоязычную версию.

В итоге получился действительно хороший материал, который поможет как познакомиться со второй версией фреймворка, так и улучшить свои навыки и знания применительно к разработке в целом.

Единственное, что стоит учесть, – книга не для новичков в программировании в целом. Предполагается, что читатель знаком с командной строкой, системами контроля версий, тестированием и может ориентироваться в исходном коде.

Александр Макаров,
Yii core team

Об авторах

Сафронов Марк – профессиональный разработчик веб-приложений из Российской Федерации, с опытом и интересами в широком спектре языков и технологий программирования. Построил и участвовал в создании различных типов веб-приложений, от чисто вычислительного характера до полноценных интернет-магазинов. Является также сторонником следования современному передовому опыту разработки, основанному на тестировании и принципах чистого, сопровождаемого кода.

В данный момент работает сотрудником компании Clevertech над веб-приложениями, основанными на Yii. Некоторое время являлся сопровождающим популярного расширения Yii под названием YiiBooster.

Ранее, в 2008 году, он перевёл книгу «Visual Prolog 7.1 for Tyros», за авторством Эдуардо Коста (Eduardo Costa), на русский язык с совершенно новым цветным оформлением. В 2013 году, в соавторстве с Джейкобом Маммом (Jacob Mumm), написал книгу «Instant Yii Application Development Starter» от Packt Publishing.

Предисловие

Эта книга — руководство, описывающее процесс постепенной, основанной на тестах разработки веб-приложения с помощью языка PHP и второй версии Yii 2, фреймворка приложений для PHP.

Yii 2 можно найти по адресу http://www.yiiframework.com/. Это фреймворк приложений на языке PHP, основанный на композитном паттерне Модель–Вид–Контроллер. Он подходит для построения как приложений командной строки, так и веб-приложений, однако состав его возможностей делает его наиболее полезным при разработке именно веб-приложений. Он содержит несколько средств для автоматической генерации исходного кода, включая создатель полноценных Create-Read-Update-Delete (CRUD) интерфейсов. Он в значительной степени полагается на соглашения, выраженные в его настройках по умолчанию.

В целом если всё, что вам нужно – это изощрённый интерфейс для нижележащей базы данных, то, возможно, вы не найдёте ничего лучше, чем Yii. Однако, учитывая широкие возможности по настройке, вы можете в конечном счёте построить приложение любого типа.

Вторая версия этого фреймворка опирается на последние улучшения в инфраструктуре PHP, скопившиеся с течением лет. В качестве основного метода установки используется Composer (см. https://getcomposer.org), стандарты PSR уровней 1, 2 и 4 от PHP Framework Interop Group (см. http://www.php-fig.org/) и возможности PHP 5.4 и выше, такие как сокращённый синтаксис массивов и замыкания.

Что включено в эту книгу

Глава 1 «Начинаем» покрывает простейшие методы развёртывания рабочего веб-приложения полностью «из ничего», имея только рабочую станцию, стек LAMP и подключение к Сети.

Глава 2 «Создаём приложение с Yii 2 вручную» показывает, как можно, используя Yii 2, с нуля реализовать веб-приложение с одной работающей, оттестированной функциональной единицей.

Глава 3 «Автоматическая генерация кода» показывает, как можно реализовать работающую оттестированную единицу функциональности в уже существующем веб-приложении, используя только возможности автоматической генерации кода, не написав ни единой строчки кода вручную.

Глава 4 «Рендерер» описывает детали того, как фреймворк конструирует свой вывод для передачи его пользователю, а также показывает некоторые трюки для внесения изменений в этот процесс.

Глава 5 «Аутентификация» обсуждает инструменты аутентификации посетителей, то есть удостоверения их личности.

Глава 6 «Авторизация пользователей и контроль доступа» рассказывает о путях контроля доступа посетителей и в особенности о системе контроля доступа, основанного на ролях (RBAC).

Глава 7 «Модули» возвращается от конкретных возможностей фреймворка к его основам. Здесь мы ясно поймём внутреннюю структуру и логику приложения, основанного на Yii 2.

Глава 8 «Поведение в целом» говорит о некоторых инфраструктурных возможностях, влияющих на всё приложение в целом.

Глава 9 «Создание расширения» говорит, как сделать расширение для Yii и подготовить его так, чтобы его можно было установить так же просто, как и расширения, включённые в базовую поставку самого фреймворка.

Глава 10 «События» исследует подробности системы внутри Yii 2, которая позволяет нам присоединить нестандартное поведение ко множеству действий, которые приложение обычно совершает самостоятельно, такие как извлечение записей из базы данных или рендеринг файла представления.

Глава 11 «Таблица» имеет два назначения. Во-первых, она объясняет мощный виджет GridView, который позволяет нам относительно легко создавать сложные интерфейсы на основе таблиц. Во-вторых, она показывает иной подход к разработке приложений на Yii 2, более обыкновенный в его сообществе, так что вы сможете увидеть как преимущества, так и недостатки обоих подходов.

Глава 12 «Маршрутизация» объясняет верхний уровень фреймворка: то, как он, собственно, откликается на HTTP-запросы от посетителей.

Глава 13 «Совместная работа» завершает книгу, представляя методы, которые помогут вам сопровождать исходный код приложения, основанного на Yii, в условиях, когда над ним работает несколько разработчиков.

Приложение А «Настройка развёртывания с Vagrant» показывает простой способ создания виртуальной машины для локальной разработки, которую вы можете использовать для запуска примеров из этой книги.

Приложение В «Пример использования Active Form» содержит расширение к главе 11, в которой мы используем другой мощный элемент управления, включённый в Yii, а именно виджет ActiveForm. Это было исключено из одиннадцатой главы, потому что напрямую с виджетом GridView данный виджет не связан, но мы также и не могли полностью проигнорировать его. Без ActiveForm тот функционал, который строится в главе 11, останется незавершённым.

Начиная со второй главы и до конца мы будем работать с одним и тем же исходным кодом. Каждая последующая глава продолжает работу, начатую в главе перед ней. По этой причине ожидается, что книга будет прочитана последовательно, без пропуска глав или чтения их в произвольном порядке.

Для кого эта книга

Текст нацелен на квалифицированных разработчиков программного обеспечения, желающих быстро оценить, удовлетворяет ли фреймворк Yii 2 их потребностям и в особенности рабочему процессу. Это не справочник, а скорее путеводитель. Более того, ожидается, что читатель в качестве дополнительного материала всегда будет иметь под рукой исходный код и официальную документацию от этого фреймворка.

Мы ожидаем относительно высокую квалификацию от читателя, так как некоторые базовые концепции разработки, такие как POSIX-совместимая командная строка, система контроля версий, конвейер развёртывания (deployment pipeline), автоматические тесты и способность ориентироваться в исходном коде по полным именам классов, подразумеваются как очевидные и не требующие объяснений.

Что вам понадобится для этой книги

Рабочая станция с полным стеком LAMP. То есть имеющая веб-сервер Apache, систему управления базами данных MySQL и среду PHP, установленные на какой-либо дистрибутив, основанный на Linux. Если читатель достаточно подкован, любая из этих программ, за исключением PHP, очевидно, может быть заменена на аналоги других поставщиков.

Вам придётся использовать PHP версий 5.4 и выше, потому что это требование для самого Yii 2, и в целом больше нет никакой необходимости использовать более старые версии этого языка.

Даже если вы не используете POSIX-совместимую операционную систему, такую как дистрибутив, основанный на Linux, или же Mac OS X, вам необходим интерпретатор командной строки, подобный Bash, так как все примеры команд командной строки в этой книге подразумевают именно его.

Для того чтобы скачать множество необходимых библиотек исходного кода, используемых в этой книге, вам потребуется соединение с Интернетом. Даже если вы ничего не будете обновлять впоследствии, вы скачаете примерно 320 Мб библиотек, так что мобильная связь, возможно, вам не подойдёт.

Соглашения

В этой книге вы найдёте несколько отличительных стилей текста, подчёркивающих информацию разного рода. Вот несколько примеров таких стилей, а также объяснение их значения.

Код в тексте, названия баз данных, названия папок и файлов, расширения файлов, примеры URL, пользовательский ввод и никнеймы в Twitter изображаются следующим образом: «Теперь выполните следующую команду, для того чтобы создать подкаталог под названием `basic` и заполнить его базовым шаблоном приложения».

Блоки кода оформляются следующим образом:

```
require_once(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
new yii\web\Application(
    require(__DIR__ . '/../../config/web.php')
);
```

Когда мы хотим привлечь ваше внимание к определённой части блока кода, соответствующие строчки или символы выделяются жирным шрифтом:

```
require_once(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
new yii\web\Application(
    require(__DIR__ . '/../../config/web.php')
);
```

Любой ввод или вывод командной строки оформляется следующим образом:

```
$ php composer.phar require --prefer-dist yiisoft/yii2-debug "*"
```

Новые термины и **важные слова** выделяются жирным шрифтом. Слова, которые вы видите на экране, в меню или в диалоговых окнах, упоминаются в тексте, выделенные вот так: «вы должны заполнить

имеющиеся поля так, как указано в следующей таблице, а затем нажать кнопку *Preview*».

Предупреждения или важные замечания показаны вот в такой рамке.

Советы и различные трюки показаны в такой рамке.

Отзывы читателей

Отзывы наших читателей всегда приветствуются. Дайте нам знать, что вы думаете об этой книге — что вам понравилось или, возможно, не понравилось. Отзывы важны для нас, чтобы выпускать книги, максимально полезные для вас.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Поддержка пользователей

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе Читателям – Файлы к книгам.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг (возможно, ошибку в тексте или в коде) мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Глава 1

Начинаем

Давайте посмотрим, как мы можем «с нуля» и с минимальными усилиями развернуть веб-сайт, используя Yii 2. Нашей целью будет изучить процесс установки шаблонов приложений, которые разработчики Yii предлагают нам, и начальный набор возможностей, включённых в них.

Вам понадобится компьютер с интерпретатором командной строки, PHP версии не ниже 5.4, веб-браузером и чем-нибудь, способным скачивать файлы с Интернета, например curl.

Базовое приложение

Самый простой и прямолинейный способ начать применять Yii 2 – это воспользоваться шаблоном приложения, опубликованным командой разработчиков Yii 2 на их репозитории исходного кода в GitHub (https://github.com/yiisoft/yii2) и утилитой Composer. В предыдущих версиях Yii вам обычно нужно было вручную скачать и распаковать архив с содержимым фреймворка. В то время как вы можете продолжать поступать так же, используя Yii 2, эта версия специально собрана таким образом, что её особенно просто установить, используя именно Composer.

Установка базового шаблона приложения

Найдите подходящий каталог на вашем жёстком диске и скачайте PHP-архив (PHAR) Composer'а в него любым удобным для вас способом. Например, используя вот такую команду:

```
$ curl -sS http://getcomposer.org/installer | php
```

В этом каталоге появится файл `composer.phar`, который и является исполняемым файлом утилиты Composer. Он запускается командой `php composer.phar`.

Перед тем как устанавливать Yii 2, вам понадобится один плагин к Composer. Yii 2 с момента релиза начал активно использовать его для своих нужд. Выполните следующую команду:

```
$ php composer.phar global \
require "fxp/composer-asset-plugin:1.0.0-beta3"
```

Обратите внимание, что эту команду (несмотря на наличие аргумента под названием `global`) *не нужно выполнять с правами администратора*, иначе позднее вы не сможете установить шаблон приложения из-за недостаточных прав доступа.

Это одна строчка, разделённая на две для улучшения читабельности. Косая черта обозначает переход на следующую строку текста. Командные интерпретаторы в Unix-подобных системах должны понимать это соглашение, так что вы можете просто скопировать и вставить код как есть, и он будет успешно выполнен.

Эта команда установит плагин Composer Asset Plugin, который можно найти по адресу https://github.com/francoispluchino/composer-asset-plugin. Он позволяет через Composer манипулировать пакетами, которые обычно контролируются менеджерами пакетов NPM (Node Package Manager, см. https://www.npmjs.org/) и Bower (см. http://bower.io/). В нашем случае он нужен, потому что Yii 2 зависит от некоторых пакетов Bower, содержащих такие вещи, как jQuery и Twitter Bootstrap. К сожалению, вы не можете установить этот плагин локально, так как ещё нет проекта, для которого можно объявить его в качестве зависимости. Кроме того, Yii 2 не объявляет его в качестве зависимости для себя, поэтому вам всегда придётся устанавливать его самостоятельно.

Пожалуйста, обратите внимание на то, что Yii 2 имеет некоторые зависимости на уровне системы, отсутствие которых не даст вам *даже установить* его, и они достаточно часто меняются. Вам, возможно, понадобится проверить файл `composer.json` в их репозитории GitHub, чтобы узнать о них заранее (см. https://github.com/yiisoft/yii2). Хотя в любом случае Composer вам скажет, что нужно установить для того, чтобы Yii 2 заработал. На момент написания одним из неочевидных требований, кроме вышеупомянутого Composer Asset Plugin, было расширение PHP mcrypt.

Теперь выполните следующую команду, чтобы создать подкаталог под названием `basic` и заполнить его базовым шаблоном приложения:

```
$ php composer.phar create-project --prefer-dist --stability=dev \
yiisoft/yii2-app-basic basic
```

Вам лучше свериться с документацией Composer, для того чтобы точно узнать значение этой команды, однако часть, которая нас интересует, – это `yiisoft/yii2-app-basic basic`, что означает «скопировать

содержимое репозитория, опубликованного по адресу https://github.com/yiisoft/yii2-app-basic, в нашу локальную папку под названием basic». Команда установит скелет проекта, представляющий собой заранее подготовленные подкаталоги, и среди них подкаталог vendor, который будет содержать довольно много других пакетов, установленных Composer, помимо Yii 2. Папка basic будет корневым каталогом нашего приложения.

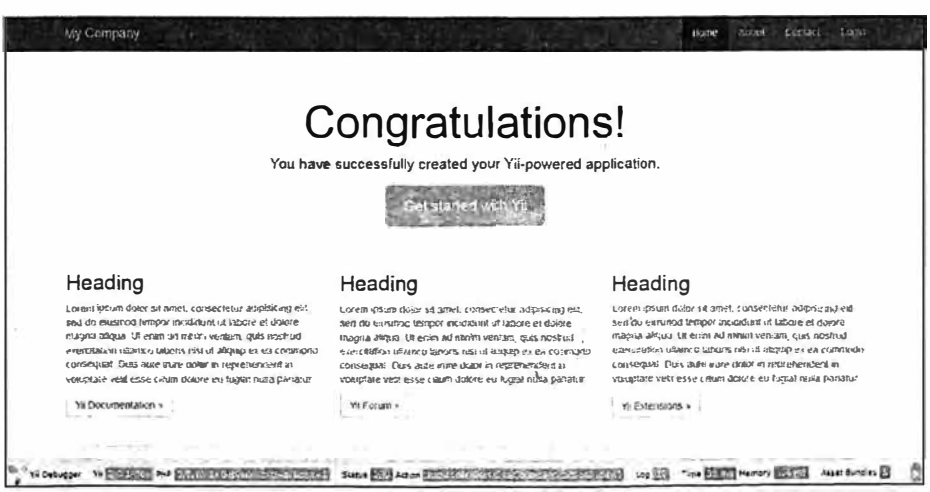
После того как Composer закончит устанавливать требуемые пакеты, вы можете просто выполнить следующую команду:

```
$ php -S localhost:8000 -t basic/web
```

Здесь 8000 – это номер порта, который вы можете сменить на любой по вашему усмотрению (кроме тех, которые уже заняты операционной системой, естественно). Эта команда запустит встроенный в PHP 5.4 веб-сервер.

Конечно же, это не самый лучший способ установки основанных на PHP веб-приложений. Встроенный веб-сервер был использован только как «дымовой тест», чтобы проверить, что в целом всё работает. Он подходит лишь для локальной разработки без серьёзной нагрузки. Следующая глава будет рассматривать развёртывание веб-приложения, основанного на Yii, в реальных условиях.

Посетите в веб-браузере адрес http://localhost:8000/. Вы должны увидеть страницу приветствия приложения Yii 2, что означает, что вы закончили с установкой.



Перенесите файл `composer.phar` из исходного каталога в только что созданный каталог `basic`. Он больше не понадобится там, где лежал, а вот в корне вашего приложения он будет крайне полезен.

Подробности о базовом шаблоне приложения

Вы можете получить подробный обзор различных подкаталогов в базовом шаблоне, прочитав файл README, идущий в комплекте с ним (https://github.com/yiisoft/yii2/blob/master/apps/basic/README.md), или прочитав страницу официальной документации Yii 2, описывающую базовый шаблон (http://www.yiiframework.com/doc-2.0/guide-start-installation.html).

Самая важная вещь, которую следует понять (на данном этапе), – это то, что папка, которую опубликовал веб-сервер, – это всего лишь одна папка из всей базы кода. В нашем базовом шаблоне приложения это папка `web`. Всё остальное находится снаружи этого каталога и, следовательно, за пределами досягаемости веб-сервера.

Как вы уже заметили, при условии что у вас есть PHP (с некоторыми специфичными модулями к нему, впрочем) и, опционально, curl, этот код может быть сразу же использован, нет необходимости в каком-либо дополнительном окружении, таком как система управления базами данных или различные библиотеки PHP. Всеми зависимостями управляет Composer.

Средства тестирования уже включены в базовый шаблон приложения на трёх уровнях. В наличии приёмочные, функциональные и модульные тесты, покрывающие весь код шаблона. Они полезны в том числе как примеры, показывающие, как пользоваться рекомендованным Yii 2 фреймворком тестирования. В нашем случае им является Codeception (см. http://codeception.com/).

Этот шаблон может быть действительно полезен для вас, если всё, что вам нужно, – это что-то вроде новостной ленты или веб-инструмента на пару страниц. Однако отсутствие разделения на подсистемы, такие как административная и публичная части, будет мешать вам в разработке приложений больших размеров; возможно, приложений, имеющих уже хотя бы 10 уникальных страниц/маршрутов.

Если взглянуть на файл `composer.json` у сгенерированного приложения, то можно увидеть несколько важных частей, выделенных в подключаемые пакеты Composer. Вот они:

- Gii, генератор кода, который мы в деталях обсудим в *главе 3 «Автоматически генерируем код для CRUD»*;
- модуль Debug, представляющий собой консоль для отладки приложения прямо в браузере. Вы можете увидеть её на скриншоте внизу. Если у вас в браузере она не видна, то, значит, вы развернули приложение не на той машине, с которой открыли его главную страницу (хитрый вы!). Мы рассмотрим консоль отладки во всех подробностях в *главе 7 «Модули»*;
- обёртку вокруг фреймворка тестирования Codeception и потрясающе удобной библиотеки генерации случайных данных Faker. Мы начнём работать с ними прямо со следующей главы;
- обёртку вокруг библиотеки Swiftmailer (http://swiftmailer.org/), которую можно найти по адресу https://github.com/yiisoft/yii2-swiftmailer. Она будет вкратце упомянута в *главе 8 «Поведение в целом»*;
- библиотеку пользовательского интерфейса Twitter Bootstrap, запакованную в виде пакета материалов (*asset bundle*) Yii 2. Эта библиотека сейчас практически повсеместно известна, но всё равно вот ссылка: http://getbootstrap.com/. Мы посмотрим, что такое пакеты материалов, в *главе 4 «Рендерер»*, но поработаем с Twitter Bootstrap раньше, в *главе 3 «Автоматически генерируем код для CRUD»*.

Первые три пункта настроены таким образом, что вы получите их, только когда развёртываете приложение для разработки на локальной рабочей станции, так как они бесполезны и даже в какой-то степени вредны на продакшене. И вероятнее всего, вам всё это потребуется на любом серьёзном проекте.

Краткий обзор базовой установки приложения:

```
$ curl -s http://getcomposer.org/installer | php
$ php composer.phar global \
require "fxp/composer-asset-plugin:1.0.0-beta3"
$ php composer.phar create-project –prefer-dist \
--stability=dev yiisoft/yii2-app-basic basic
$ mv composer.phar ./basic
$ php -S localhost:8080 -t basic/web
```

Проверка требований к системе

На случай, если запуск приложения не удаётся из-за возникающих сообщений об ошибках или просто заканчивается белоснежной пустой

страницей в веб-браузере, Yii 2 включает в себя специальный скрипт проверки системы на соответствие требованиям. Это файл под названием `requirements.php`, лежащий в корневом каталоге как базового, так и расширенного шаблона приложения.

Вы можете запустить его в консольном режиме очевидной командой:

```
$ php requirements.php
```

И он напечатает на экране все неочевидные требования Yii 2 к системе вместе с пометками, удовлетворены они или нет. Вам останется только установить необходимые компоненты. Практически все они – различные расширения PHP.

Расширенный шаблон приложения

Кроме базового, Yii 2 включает в себя продвинутый шаблон приложения. Он более приспособлен для приложений среднего размера (собственно, приложения, действительно полезные для бизнеса), и его главная особенность – это два раздельных веб-интерфейса: один выделен на управление контентом, и другой – на представление этого контента посетителям. Так что с этим шаблоном вы получаете почти законченный скелет системы управления контентом (CMS).

Установка расширенного шаблона приложения

Первые шаги те же самые, что и для базового шаблона. Вам нужно скачать исполняемый файл Composer, установить плагин Composer Assets Manager и затем установить новый проект через Composer, вот только вместо `basic` нужно написать `advanced`:

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar global \
require "fxp/composer-asset-plugin:1.0.0-beta3"
$ php composer.phar create-project --prefer-dist --stability=dev \
yiisoft/yii2-app-advanced advanced
```

Не забудьте переместить `composer.phar` в свежесозданную папку `advanced` и перейти туда, прежде чем продолжить настройку.

После этого вам нужно сгенерировать необходимые конфигурационные файлы, выполнив следующую команду:

```
$ ./init
```

Да, это просто скрипт под названием `init` в корневом каталоге проекта. Он спросит вас, хотите ли вы режим разработчика или production-режим, и создаст все необходимые дополнительные кусочки конфигурации и точки входа. Если совсем точно, то он просто копирует содержимое каталогов `dev` или `prod` из подкаталога `environments` в зависимости от того, что вы выберете. Просто откройте подкаталог `environments` – и вы поймёте, как он работает.

Затем вам нужно создать базу данных, которая будет использоваться этим приложением. По умолчанию для режима разработки вам необходимо создать базу данных MySQL под названием `yii2advanced`, доступную с адреса `localhost` по порту MySQL по умолчанию, для пользователя под названием `root` без пароля. Детали можно подсмотреть в файле `common/config/main-local.php`, одном из тех, которые создаёт скрипт `init`.

После того как база данных будет создана, вам будет нужно запустить миграции. Мы поговорим о скриптах миграции в следующей главе (и даже несколько напишем самостоятельно), но если сама концепция **миграций баз данных** для вас совершенно чужда, вы можете прочитать о ней в официальной документации Yii 2 здесь: http://www.yiiframework.com/doc-2.0/guide-db-migrations.html. Подробное обсуждение реализации этого механизма в Yii будет в самом конце книги, в *главе 13 «Совместная работа»*.

В любом случае, просто выполните следующую команду:

```
$ ./yii migrate
```

Она покажет вам список, содержащий в точности одну миграцию, и спросит вашего подтверждения, прежде чем выполнить свою работу.

Теперь всё готово. Сделайте обе стороны приложения доступными для вас, выполнив следующие две команды из корневого каталога приложения (только что созданной папки `advanced`):

```
$ php -S localhost:8080 -t frontend/web &
$ php -S localhost:8081 -t backend/web &
```

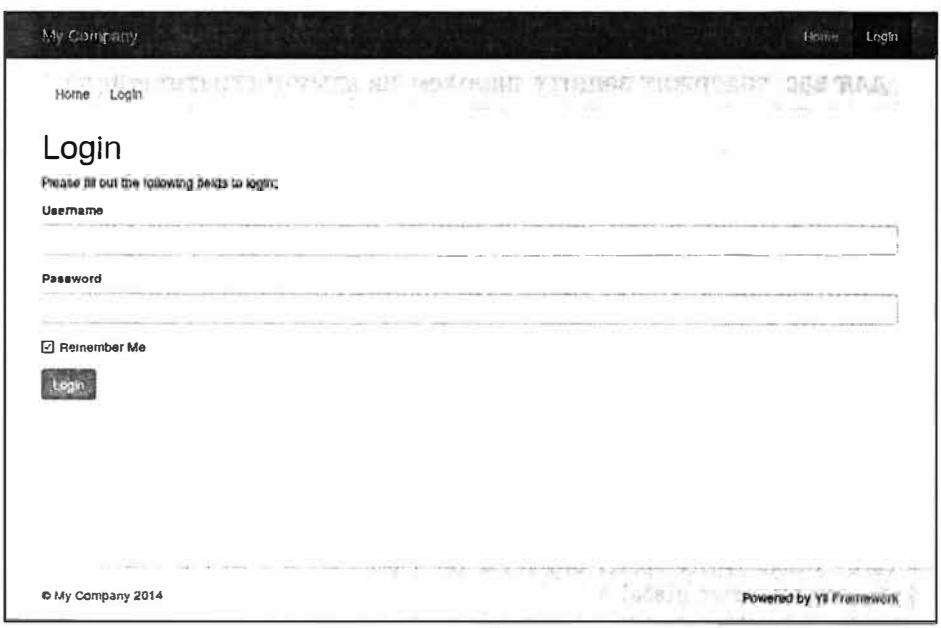
Как и в случае базового шаблона, мы используем встроенный в PHP 5.4 веб-сервер просто потому, что его гораздо проще показать в книге, нежели объяснять то, как устанавливать Apache или какой-нибудь ещё веб-сервер, чтобы опубликовать эти две папки.

Нам пришлось выполнить обе команды в фоновом режиме (обратите внимание на символ «&» в конце каждой строки), потому что обыч-

но встроенный веб-сервер забирает контроль над командной строкой, а нам их нужно запустить два. Просто закройте консоль, когда вам надоест, оба фоновых процесса должны после этого остановиться.

Теперь у вас есть административная часть приложения, предназначенная для использования контент-менеджерами, и публичная часть приложения, предназначенная для просмотра посетителями. Также обратите внимание, что у вас есть полностью контролируемый скрипт запуска консольных команд, который был использован скриптом `yii`, которым вы запускали миграции.

Публичная часть расширенного шаблона приложения выглядит в точности так же, как и у базового шаблона, за исключением возможности авторизации. Вот как выглядит административная часть, когда вы на неё зайдёте в первый раз:



Подробности о расширенном шаблоне приложения

Самое главное в расширенном шаблоне приложения – это то, что он представляет собой *три* базовых, связанных воедино:

- внутри подкаталога `frontend` находится структура приложения для публичной части веб-сайта. Ожидается, что здесь будет находиться та функциональность, ради которой веб-приложение или веб-сайт затевались;

- подкаталог под названием `backend` предназначен для вашей CMS, защищённой от неавторизованного доступа. Ожидается, что вся функциональность по манипуляции с базой данных, доступная для администраторов, будет находиться именно здесь;
- подкаталог `console` в основном для самодельных консольных команд, в надежде что они у вас появятся, и, что гораздо более вероятно, для ваших скриптов миграции;
- подкаталог `common` содержит код, который будет использован всеми тремя точками входа, так как это всё-таки одно цельное приложение.

Конечно же, никто не заставляет использовать публичную и административную части так, как описано. У вас просто есть два веб-сайта, делящих общую базу кода, так что вы можете использовать их так, как пожелаете. Однако пользовательский интерфейс, уже подготовленный для вас, содержит защиту паролем на административной части с самого начала.

Вам стоит знать о возможности аутентификации на свежеустановленном расширенном шаблоне приложения. Изначально никаких пользователей в нём не определено, и вам необходимо создать одного, используя функциональность регистрации на публичной части приложения. После этого вы сможете аутентифицироваться и на публичной, и на административной сторонах, используя одну и ту же учётную запись. Публичная часть идентична базовому шаблону приложения, и административная часть лишена всего, кроме главной страницы и формы входа, так что всё в ваших руках.

Краткий обзор процедуры установки расширенного шаблона приложения:

```
$ curl -s http://getcomposer.org/installer | php
$ php composer.phar global \
require "fxp/composer-asset-plugin:1.0.0-beta3"
$ php composer.phar create-project –prefer-dist \
--stability=dev yiisoft/yii2-app-advanced advanced
$ mv composer.phar advanced/
$ cd advanced
$ ./init
$ mysql -u root -e 'create database yii2advanced'
$ ./yii migrate
$ php -S localhost:8080 -t frontend/web
$ php -S localhost:8081 -t backend/web
```

Итоги

В стандартной поставке Yii 2 присутствуют два аккуратно собранных шаблона приложения, базовый и расширенный, которые могут использоваться для создания, соответственно, маленьких веб-приложений и приложений среднего размера. Оба шаблона крайне просто развернуть. Фактически базовое приложение устанавливается при помощи одной команды (всё остальное – рутина), а расширенное, при наличии готовой базы данных, – при помощи трёх. Учитывая то, что оба эти «шаблона», по сути, представляют собой законченные рудиментарные веб-приложения, они являются крайне удобным способом начать работу с Yii 2.

Однако в следующей главе мы не будем ими пользоваться. В ней мы рассмотрим то, как мы можем основать на Yii 2 пусть небольшой, но реальный проект, начатый полностью «из ничего».

Глава 2

Создаём приложение с Yii 2 вручную

В этой главе мы посмотрим, как именно Yii 2 может нам помочь в построении веб-приложений. Пример будет разумно небольшим, но он будет выполнен как положено, с использованием должных практик инженерии программного обеспечения. Мы пройдём поэтапно весь процесс разработки приложения, и каждый шаг будет подкреплён тем или иным проверенным приёмом, описанным в лучшей книге на соответствующую тему:

- **Формулировка предметной модели**: это объяснено в *Domain-Driven Design: Tackling Complexity in the Heart of Software*, *Eric Evans*, *Addison-Wesley Professional*.
- **Установка средств тестирования:** мы будем следовать практике разработки, основанной на приёмочных тестах (acceptance test-driven development), описанной в *Growing Object-oriented Software, Guided by Tests*, *Steve Freeman and Nat Pryce*, *Addison-Wesley Professional*.
- **Установка конвейера развёртывания:** это объяснено в следующих книгах:
  - *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, *Jez Humble and David Farley*, *Addison-Wesley Professional*;
  - *Continuous Integration: Improving Software Quality and Reducing Risk*, *Paul M. Duvall*, *Steve Matyas*, и *Andrew Glover*, *Addison-Wesley Professional*.
- **Цикл разработки «Красное–Зелёное–Рефакторинг»**: в мельчайших подробностях описан в следующих книгах:

Clean Code: A Handbook of Agile Software Craftsmanship, *Robert Martin*, *Prentice Hall*;
Test-Driven Development by Example, *Kent Beck*, *Addison-Wesley Professional*.

- **Развёртывание и тестирование вручную**: эти шаги так же, как и конвейер развёртывания, соответствуют парадигме безостановочной поставки (*Continuous Delivery*), но они неизбежны в любом случае.

Оставайтесь с нами.

Этап проектирования

Обратите внимание на то, что мы будем использовать данное приложение-пример в течение всей книги. В этом разделе мы определим ландшафт для всего последующего приключения, ожидающего нас впереди.

Поставленная задача

Давайте представим, что мы – небольшой бизнес, предоставляющий некоторые услуги. У нас есть некоторое количество клиентов, с которыми у нас поддерживаются отношения, и это количество настолько велико, что организовывать их записями на бумаге и стопкой визитных карточек становится слишком неудобно. Так что нам нужен какого-то рода автоматический способ поиска полной информации о нужном клиенте.

Для начала нам подойдёт что-то вроде пользовательского интерфейса для создания, просмотра, обновления и удаления простых записей, описывающих самые основные атрибуты наших клиентов.

Очевидно, что пока наше дело будет расти и развиваться, то же самое будет происходить с нашим управлением клиентами, а значит, наше приложение тоже будет расти и развиваться. Мы должны учесть возможность будущих изменений с самого начала.

Так как мы делаем программу, которой будем постоянно пользоваться сами, лучше бы ей быть наилучшего возможного качества.

Проектирование предметной модели

Очевидно, мы будем иметь дело с моделью «клиента» в нашем приложении.

«Клиент» (Customer) – это человек, у которого как минимум есть имя, почтовый адрес, адрес электронной почты и номер телефона.

Клиентам мы предоставляем услуги (Services). Это всё, что мы включим в нашу модель на первой итерации проектирования.

Основное допущение, которое мы сделаем, – это допущение о том, что каждый зарегистрированный «клиент» соответствует одному физическому лицу. В результате мы не будем иметь дела с вещами вроде компаний, у которых есть несколько контактных лиц.

Имя – это крайне сложная конструкция, если мы будем разбираться с деталями её построения, такими как формы вежливости, титулы, прозвища (в том числе сетевые), средние имена, отчества и т. д. Но нас на самом деле не интересует структура имени клиента, нам оно нужно только для его идентификации. Поэтому мы будем представлять его в виде строки текста, что позволит нам записывать имена в произвольной форме.

Адрес – это конструкция такой же сложности, но в его случае нам придётся сохранять структуру вместо использования произвольных строк текста, потому что нам когда-нибудь понадобится делать две вещи с адресами:

- вычислять какие-либо статистические сведения, например сколько клиентов у нас есть в определённом городе;
- корректно генерировать почтовые адреса согласно правилам, принятым в различных культурах.

Поэтому мы остановимся на следующей структуре адреса:

- назначение (например, платёжный адрес, адрес доставки, домашний, рабочий и т. п.);
- страна;
- регион – для стран, разделённых на регионы, как Россия или США;
- город;
- улица;
- строение;
- квартира/офис;
- имя получателя;
- почтовый индекс.

Следует заметить, что адрес может быть для квартиры, абонентского ящика, офиса в офисном здании, сотрудника фирмы или для целого здания. А ещё у клиента может быть несколько адресов.

У номера телефона есть следующие атрибуты:

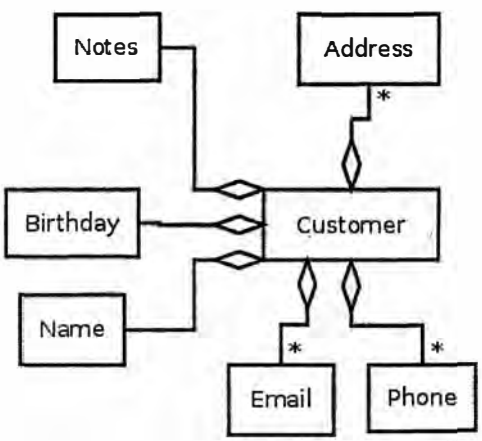
- назначение (личный/рабочий);
- собственно номер.

У клиента может быть несколько телефонных номеров, поэтому и возникает необходимость в поле «назначение».

Кроме сущностей «имя» (name), «адрес» (address) и «телефон» (phone), нашим сотрудникам, скорее всего, понадобится приписывать какое-либо текстовое описание клиента в свободной форме. Его мы назовём просто «заметки» (notes). А ещё будет классно записывать дни рождения клиентов. И ещё адрес электронной почты, конечно же. Между прочим, у одного клиента может быть несколько адресов электронной почты.

Давайте здесь остановимся уже.

Теперь мы можем примерно представить законченный агрегат (aggregate) для модели клиента и нарисовать его следующим образом:



Для простоты мы не будем детализировать то, как мы ведём дела с клиентами, просто потому что мы не в состоянии всё это осветить в рамках данной книги. Однако заметим, что, так как наша воображаемая фирма предоставляет некоторые услуги клиентам, было бы полезно вести записи о них. Эта модель будет использована в следующей главе.

Целевая функциональность

Давайте решим одну конкретную задачу. Когда кто-нибудь нам позвонит по телефону (подразумевается, что номер мы определили), мы хотим получить все подробности о звонящем, которые мы собрали к этому моменту. Если этот номер не связан с кем-либо, зарегистри-

рованным в нашем приложении, то мы знаем, что это не наш клиент (пока что). Если связан, то мы, по крайней мере, можем поприветствовать этого человека по имени, что уже является примером превосходного обслуживания.

Следует понимать, что для того, чтобы иметь возможность делать запросы к базе данных, нам нужно как минимум что-то записывать в неё и, потенциально, редактировать и удалять из неё. Но мы не будем делать всё это в рамках нашей первой итерации. Сократим набор функциональности до следующего:

- возможность записывать информацию о клиенте в базу данных;
- возможность запросить информацию о клиенте из базы данных по его номеру телефона.

Совершение произвольных запросов к базе данных не является нашей целью. Мы будем иметь дело только с запросами, содержащими телефонный номер.

В таком случае давайте начнём.

Начальная подготовка

В течение всей книги мы будем работать с одним и тем же приложением, поэтому подготовку нужно будет провести только один раз.

Скачивание исходного кода примеров

Вы можете скачать файлы исходного кода примеров для всех книг Packt, которые вы купили, из вашей учётной записи на http://www.packtpub.com. Если вы купили эту книгу где-либо ещё, вы можете посетить http://www/packtpub.com/support и зарегистрироваться там. После этого файлы отправят вам по электронной почте.

Настройка управления проектом

Наше приложение-пример фактически представляет собой систему управления взаимоотношениями с клиентами (CRM-систему). Поэтому мы начнём с папки под названием `crmapp`.

Пожалуйста, обратите внимание, что все примеры команд командной строки и все относительные пути *в течение всей книги* предполагают, что ваш текущий рабочий каталог – это папка `crmapp` и нигде больше в файловой системе.

Предпочтительный метод установки Yii 2 – через утилиту Composer, так что мы тоже будем её использовать. Хоть вы и можете прочитать подробности в полной документации Composer, вот краткий курс обращения с ним:

- все пакеты, установленные Composer, хранятся в специальном подкаталоге корневой директории под названием `vendor`;
- все зависимости и остальная информация о вашем приложении, имеющая отношение к Composer, хранятся в манифесте под названием `composer.json` в корневом каталоге проекта. Пока все зависимости объявлены в этом файле, вы можете безо всякого вреда в любое время удалить папку `vendor`, так как она будет заново создана и заполнена после очередного вызова `php composer.phar install` или `php composer.phar update`.

Документация по Composer предлагает милую команду в одну строчку, которая добудет вам исполняемый файл этой утилиты:

```
$ curl -sS https://getcomposer.org/installer | php
```

Конечно же, если у вас нет CURL где-нибудь в вашем PATH, вы можете просто посетить официальный сайт по адресу https://getcomposer.org/ и достать PHAR-архив оттуда.

После этого Composer запускается обращением к файлу `composer.phar`:

```
$ php composer.phar <command>
```

Подразумевается, что вы используете какую-либо систему контроля версий вашего исходного кода. Пакет исходного кода для этой книги использует Git (http://git-scm.com/).

Подготовка коротко:

```
$ mkdir crmapp
$ cd crmapp
$ curl -sS https://getcomposer.org/installer | php
$ git init
```

Установка средств тестирования

Как мы заявили в первом абзаце этой главы, мы будем следовать практике разработки с тестами в первую очередь (test-first development practice), основанной на приёмочных тестах. Причины такого решения следующие:

- у нас будет способ проверить, работает ли приложение так, как задумано, без необходимости полагаться на тягомотину ручного тестирования;
- у нас нет особой надобности в настоящем, глубоком модульном тестировании, так как большая часть того, что мы будем делать, будет заключаться в соединении друг с другом уже существующих компонентов, с рудиментарной логикой между ними. Так что приёмочные тесты сквозь всё приложение, начиная с пользовательского интерфейса, – это самое простое и действенное решение.

Нам в любом случае нужны приёмочные тесты в том или ином виде, если нам вообще есть дело до пользовательской функциональности.

В Yii 2 встроена поддержка фреймворка тестирования под названием Codeception, чьим официальным сайтом является http://codeception.com/. Расширение Yii 2 под названием `yii2-codeception` (см. https://github.com/yiisoft/yii2-codeception) предоставляет набор классов-помощников, для того чтобы более тесно интегрировать ваши тесты с Yii. Мы не будем использовать данное расширение в этой книге, но теперь вы о нём хотя бы знаете.

Давайте объявим, что мы хотим, чтобы в нашем проекте был доступен Codeception. Выполните следующую команду:

```
$ php composer.phar require "codeception/codeception:*"
```

Подождите немного, пока Composer закончит.

Вот каким должно быть содержимое манифеста «`composer.json`» к этому моменту:

```
{
   "require": {
      "codeception/codeception": "*",
   }
}
```

Команда `php composer.phar require <packagename:version>` – это не более чем вспомогательный метод для того, чтобы вставлять строчки внутри блока `require` и вызывать процедуру обновления зависимостей.

Конечно же, в определённый момент нам придётся добавить Yii 2 в качестве зависимости, но пока что давайте делать только одно дело за раз.

Теперь исполняемый файл Codeception доступен для нас в подкаталоге `./vendor/bin/codecept`.

Это довольно долго набирать на клавиатуре, но POSIX-совместимый командный интерпретатор вроде bash позволяет нам сократить это следующим образом:

```
$ alias cept="./vendor/bin/codecept"
```

Так уже лучше. Во всех примерах команд командной строки в этой главе (и на самом деле далее по всей книге) мы подразумеваем, что вы сделали эту подстановку.

Codeception – сложная система, поэтому нам придётся положиться на его возможности самосборки. Чтобы не вдаваться в излишние подробности внутренней работы Codeception, давайте просто пользоваться настройками по умолчанию. Выполните следующую команду:

```
$ cept bootstrap
```

Это создаст подкаталог `tests` и дерево требуемых конфигурационных файлов.

Теперь давайте создадим наш первый приёмочный тест-заглушку, чтобы проверить самый верхний уровень наших средств тестирования:

```
$ cept generate:cept acceptance SmokeTest
```

Эта команда создаст файл `SmokeTestCept.php` в подкаталоге `tests/acceptance`. Когда вы его откроете, то увидите нечто вроде следующих строчек, в зависимости от версии Codeception:

```
$I = new AcceptanceTester($scenario);
$I->wantTo('perform actions and see result');
```

`AcceptanceTester` – это класс объектов, хранящих все методы, которые мы можем использовать, чтобы проверить наше приложение, имитируя реально существующего пользователя позади браузера. В Codeception также есть `UnitTester` для модульных тестов и `FunctionalTester` для функциональных тестов, но об этом в другой раз.

Когда мы говорим `AcceptanceTester.wantTo("do something")`, мы просто делаем заголовок (из того, что находится в кавычках) для действий, следующих за этим вызовом.

Давайте заменим эту заглушку на тест, проверяющий, что наша стартовая страница работает:

```
$I = new AcceptanceTester($scenario);
$I->wantTo('See that landing page is up');
$I->amOnPage('/');
$I->see('Our CRM');
```

Таким образом, мы ожидаем увидеть строчку **Our CRM**, зайдя на стартовую страницу нашего будущего приложения. Сделаем вид, что где-то что-то такое у нас будет написано.

Теперь мы запускаем этот тест:

```
$ cept run
```

Мы видим, что он провален, потому что у нас нет веб-сервера, который бы что-либо отдавал по запросу «/». Так что мы пришли к тому моменту, когда нам нужно писать реальный код, для того чтобы тесты были пройдены успешно. Однако прямо сейчас нам нужен не код, а инфраструктура, чтобы его исполнять. Нам нужна машина, на которую мы будем развёртывать приложение.

Настройка конвейера развёртывания

Проблема в следующем. Приёмочные тесты, которые мы будем писать, имитируют реального пользователя, который открывает веб-приложение в браузере и взаимодействует с ним, используя предоставленный пользовательский интерфейс. Для того чтобы они работали, нам нужно приложение, которое полностью развёрнуто в каком-то месте, доступном с той машины, на которой мы будем запускать приёмочные тесты.

> В большинстве случаев вы решите просто-напросто запускать приложение с той же машины, на которой вы редактируете его исходный код. Неверно! Не делайте этого.

Вероятнее всего, ваша рабочая станция не является точной копией той машины, на которой в конце концов будет запущено ваше приложение. Это уже десятилетиями является насущной проблемой в индустрии. Можете быть уверены: когда продолжительность существования вашего приложения станет исчисляться годами, вы совершенно точно получите проблемы интеграции, если вы тестируете приложение на компьютере с окружением, отличающимся от вашего «боевого» сервера.

Конечно же, это не относится к распространяемому ПО, которое вы продаёте различным пользователям для самостоятельной установки,

и вам требуется переносимость. Так как Yii 2 имеет достаточно удобные средства для разработки консольных приложений, вы можете и с таким столкнуться когда-нибудь. В нашем случае мы подразумеваем стационарное веб-приложение с единственной точкой развёртывания, так что нашей проблемой является не переносимость исходного кода, а воспроизводимость тестов.

В конечном счёте ваше приёмочное тестирование будет состоять из следующих шагов:

1) развернуть приложение на тестовый сервер;
2) запустить приёмочные тесты на вашей машине.

Конечно, вы можете запускать приёмочные тесты на тестовом сервере. Чтобы добиться этого, вам просто нужно настроить тесты так, чтобы они использовали сетевой интерфейс «внутренней петли» (loopback interface), `localhost`. Однако это потребует от вас установки дополнительного программного обеспечения на ваш тестовый сервер, никак не относящегося к самому приложению. Например, если вы решите гонять тесты полного цикла внутри браузера, используя **Selenium**, вам, возможно, потребуется устанавливать веб-браузер, среду исполнения Java и приложение для виртуализации экранного буфера (virtual framebuffer), что приведёт к установке очень серьёзного количества системных библиотек, которые фактически являются просто отходами производства. Гораздо более продуктивно использовать вашу собственную рабочую среду для запуска приёмочных тестов.

Конечно же, это нельзя сказать про модульные и функциональные тесты. Модульные тесты по своей природе выполняются над сырой базой кода, вообще без необходимости развёртывания. Функциональные тесты должны выполняться над развёрнутым приложением, потому что им требуется проверять корректность взаимодействий между уже сконфигурированными и работающими частями конечного приложения (именно для этого они и нужны).

В любом случае, в идеале у вас должна быть одна простая команда, с названием вроде `deploy`, которая должна делать следующее:

1) получить доступ и запустить целевую рабочую станцию (в особенности если это экземпляр виртуальной машины);
2) сделать всё необходимое, чтобы там присутствовало корректное окружение, ожидаемое приложением;
3) скопировать текущее состояние базы кода на целевую машину;

4) настроить скопированную базу кода под окружение на целевой машине;
5) запустить приложение.

Вы должны быть в состоянии сделать всё вышеперечисленное, набрав `deploy` в командной строке и нажав *Enter*. Как Мартин Фаулер (Martin Fowler) сказал в своей основополагающей статье «Непрерывная интеграция» (*Continuous Integration*, последний раз видна по адресу http://martinfowler.com/articles/continuousIntegration.html), это должно стать «несуществующим событием» («non-event», непереводимо). В идеале развёртывание должно происходить автоматически, когда вы запускаете приёмочные тесты.

В этой книге мы занимаемся только двумя последними пунктами в вышеописанной процедуре. Так как мы работаем с приложением PHP, шаг «запустить приложение» обычно выполнен сразу, как только у нас есть веб-сервер, работающий на целевой машине.

Эта книга – о разработке веб-приложений, а не о системном администрировании, и его целевая аудитория – веб-разработчики, а не инженеры обслуживания (operation engineers). Однако в приложении А мы подготовили описание одного варианта установки, основанного на использовании виртуальной машины, который вы можете легко повторить на практически любой рабочей станции. Вам не понадобится отдельный физически существующий компьютер, но тем не менее вы всё равно будете в состоянии имитировать реальную процедуру развёртывания. Если у вас нет других вариантов, мы крайне рекомендуем вам обратить на него внимание. На самом деле весь код в этой книге был подготовлен с использованием описанных в этом приложении техник.

Давайте представим, что у вас есть окружение, подготовленное командой развёртывания, и, для простоты, допустим, что эта команда будет выполняться перед каждым запуском приёмочных тестов. Результатом вашего развёртывания должен быть конкретный URL, доступный с вашей машины, который средства приёмочного тестирования будут использовать как точку входа в ваше приложение.

Теперь давайте отправимся в ту часть конфигурации Codeception, которая относится к приёмочным тестам, в файле `tests/acceptance.suite.yml`, и добавим этот URL в токен `modules.config.PhpBrowser.url`. Этот файл, при условии что вы ничего больше не меняли и в стандартной поставке Codeception ничего не менялось с тех пор, как эта глава была написана, должен выглядеть следующим образом:

```
class_name: AcceptanceTester
modules:
    enabled:
        - PhpBrowser
        - WebHelper
    config:
        PhpBrowser:
            url: 'http://URL.ВАШЕГО.ПРИЛОЖЕНИЯ'
```

Например, если вы настроили целевую машину с веб-сервером Apache, используя технику виртуальных хостов, основанных на IP-адресах (IP-based virtual host technique, как описано на странице https://httpd.apache.org/docs/2.2/vhosts/ip-based.html), значение настройки `modules.config.PhpBrowser.url` может выглядеть как-то вроде `http://127.0.0.1:8000`.

Так как мы внесли изменения в конфигурацию, нам следует пересобрать среду Codeception. Вот команда, которая это делает:

```
$ cept build
```

Не забывайте, что `cept` – это просто сокращение, которое мы сами создали. Настоящий исполняемый файл – это `./vendor/bin/codecept`.

Если вы выполните тесты теперь:

```
$ cept run
```

Вы должны увидеть вот такой вывод:

```
[(detached from 2f2a996)*] $ vendor/codeception/codeception/codecept run acceptance
Codeception PHP Testing Framework v2.0.0-beta
Powered by PHPUnit 4.1-dev by Sebastian Bergmann.

Acceptance Tests (1) ------------------------------------------------------------------------------------------
Trying to See that landing page is up (SmokeTestCept.php)                                              Fail
------------------------------------------------------------------------------------------------------------


Time: 97 ms, Memory: 18.06Mb

There was 1 failure:

---------
1) Failed to see that landing page is up in SmokeTestCept.php (/home/hijarian/projects/crmapp/tests/acceptance/SmokeTestCept.php)
Sorry, I couldn't see "Our CRM":
Failed asserting that
--> /

--> contains "our crm".

Scenario Steps:
2. I see "Our CRM"
1. I am on page "/"
```

Видно, что Codeception что-то наблюдает по маршруту «/», но не то, что нам нужно. Это будет или ошибка 404, или ошибка 403, в зависимости от версии используемого сервера Apache, или что-то совсем другое, если вы используете другой веб-сервер. В любом случае, корень проблемы прост: нам нужен файл `index.php` внутри каталога, доступного из Сети.

Делаем точку входа в веб-приложение видимой

Давайте определимся теперь с одним соглашением: единственная папка, которая будет видна из Сети, будет называться `web` и размещена в корневом каталоге исходного кода. Например, если ваш веб-сервер – это Apache, то именно путь к каталогу `web` вы должны написать в параметре конфигурации `DocumentRoot`.

С учётом этого просто создайте файл `index.php` в подкаталоге `web` со следующим содержимым:

```
Our CRM
```

Да, это просто текстовый файл из семи букв (включая пробел). В конце концов, это всё, что ожидают наши приёмочные тесты, верно?

Затем мы запускаем тесты:

```
$ cept run
```

Получаем следующий вывод:

```
hijarian:hijaria 17:50:12 jobs: 0 (~/projects/crmapp)
└─^ [(detached from 09e8616)*] $ ./cept run acceptance
Codeception PHP Testing Framework v1.9-dev
Powered by PHPUnit 3.7.29-4-g641cd68 by Sebastian Bergmann.

Acceptance Tests (1) ------------------------------------------
Trying to see that landing page is up (SmokeTestCept.php)       Ok
---------------------------------------------------------------


Time: 79 ms, Memory: 10.50Mb

OK (1 test, 1 assertion)
                    crmapp : bash
```

Теперь надо найти повод использовать Yii 2 в нашем проекте. Простым способом сделать это будет написание полноценного теста через всё приложение, который будет описывать нужную нам функциональность.

Добавление фреймворка Yii в наше приложение

Раз у нас есть вся инфраструктура, которая была нужна для начала работы, давайте вернёмся к той функциональности, которую мы определили на этапе проектирования, и напишем на неё приёмочный тест.

Первый тест через всё приложение

Основная особенность с приёмочными тестами через всё приложение – то, что мы взаимодействуем с приложением только через его пользовательский интерфейс. У нас нет никакого способа получить прямой доступ к базе данных (далее БД) или, ещё хуже, файловой системе вокруг приложения. Поэтому, чтобы протестировать какой-либо запрос к данным в БД, эти данные должны быть вначале записаны в неё. И это должно быть сделано через пользовательский интерфейс.

Вот итоговые шаги тестирования:

1. Открыть пользовательский интерфейс добавления данных клиента в БД.
2. Добавить клиента № 1 в БД. Мы должны увидеть пользовательский интерфейс списка клиентов с одной записью в нём.
3. Добавить клиента № 2 в БД. Теперь мы должны увидеть пользовательский интерфейс списка клиентов с двумя записями в нём.
4. Открыть пользовательский интерфейс запроса данных о клиенте по номеру телефона.
5. Совершить запрос, используя номер телефона клиента № 1. Мы должны увидеть пользовательский интерфейс результата запроса с данными для клиента № 1, но не для клиента № 2.

Таким образом, тест вынуждает нас иметь три страницы пользовательского интерфейса: запись нового клиента, список записанных клиентов и интерфейс запроса к БД. Частично именно поэтому он называется тестом «через всё приложение» (end-to-end test).

Переведённая в виде приёмочного теста Codeception только что описанная процедура выглядит вот так:

```
$I = new \AcceptanceTester\CRMOperatorSteps($scenario);
$I->wantTo('add two different customers to database');

$I->amInAddCustomerUi();
```

```
$first_customer = $I->imagineCustomer();
$I->fillCustomerDataForm($first_customer);
$I->submitCustomerDataForm();

$I->seeIAmInListCustomersUi();

$I->amInAddCustomerUi();
$second_customer = $I->imagineCustomer();
$I->fillCustomerDataForm($second_customer);
$I->submitCustomerDataForm();

$I->seeIAmInListCustomersUi();

$I = new \AcceptanceTester\CRMUserSteps($scenario);
$I->wantTo('query the customer info using his phone number');

$I->amInQueryCustomerUi();
$I->fillInPhoneFieldWithDataFrom($first_customer);
$I->clickSearchButton();

$I->seeIAmInListCustomersUi();
$I->seeCustomerInList($first_customer);
$I->dontSeeCustomerInList($second_customer);
```

Вставим этот текст в файл `tests/acceptance/QueryCustomerByPhoneNumberCept.php`. Это будет нашей конечной целью в данной главе.

Давайте обсудим неочевидные вещи в этом тестовом сценарии.

В первую очередь мы разделили сценарий на две логические части и сделали два разных подкласса класса `AcceptanceTester`, чтобы подчеркнуть это различие.

В Codeception есть полезная команда, которая автоматически генерирует подклассы разных классов `Tester`. Вот как мы можем создать класс `\AcceptanceTester\CRMOperatorSteps`, используя её:

```
$ cept generate:stepobject acceptance CRMOperatorSteps
```

Composer спросит у вас названия методов, когда будет создавать класс. Просто нажмите *Enter* в этот момент и не пишите ничего. Так вы покажете, что хотите начать «с чистого листа».

Этот помощник используется для поддержки паттерна «Объект шага» **(StepObject)** (см. http://codeception.com/docs/07-AdvancedUsage#StepObjects), так что он автоматически добавит суф-

фикс «Steps» к названию класса, даже если вы не напишете его сами. Конечно же, намного более естественно рассуждать о подклассах `AcceptanceTester` как о сущностях, имеющих различные *роли*, нежели как об абстрактных контейнерах для шагов тестового скрипта. Однако если мы принудительно переименуем созданные классы, убрав ненужный суффикс, мы потеряем автоподключение классов, которое Codeception нам предоставляет, поэтому придётся терпеть.

Вышеописанная команда помещает файл `CRMOperatorSteps.php` в подкаталог `tests/acceptance/_steps`.

Тем же образом создаётся класс `CRMUserSteps`.

Теперь давайте определим шаги, упомянутые в сценарии теста. Почти все эти высокоуровневые шаги будут не более чем контейнерами для более низкоуровневых шагов, встроенных в стандартную поставку Codeception.

Вначале посмотрим на шаги оператора CRM.

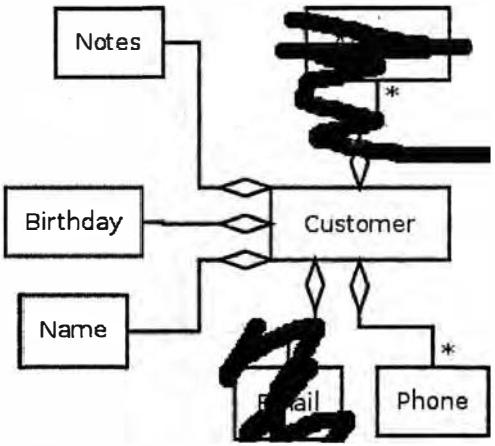
Шаг «I am in Add Customer UI» («Я в пользовательском интерфейсе добавления клиента») – это просто переход по адресу, соответствующему нашему будущему интерфейсу добавления клиента, так что он будет выглядеть вот так:

```
function amInAddCustomerUi()
{
   $I = $this;
   $I->amOnPage('/customers/add');
}
```

Шаг «Imagine Customer» («Вообразить клиента») – это вспомогательный метод для генерации случайных данных о пользователе, которые должны быть введены в интерфейс добавления клиента.

Мы будем использовать восхитительную библиотеку под названием **Faker** (см. https://github.com/fzaninotto/Faker), чтобы получить правдоподобные данные. Однако мы посмотрим на неё позже.

В данный момент необходимость вводить данные в пользовательский интерфейс добавления клиента вынуждает нас принять решение о том, как этот пользовательский интерфейс будет выглядеть. Мы не собираемся делать что-то сложное; это будет простая HTML-форма с кнопкой **Submit**. Но какие поля должны в ней находиться? Давайте вернёмся к нашей модели клиента и посмотрим, какие её части нам по-настоящему необходимы, для того чтобы выполнить наш тестовый сценарий:



Для простоты мы оставим модели **E-mail** и **Address** до *главы 11 «Таблица»*.

Теперь давайте определимся с нашей формой добавления клиента. Пожалуйста, обратите внимание на способ именования полей формы: они названы не произвольным образом, а в соответствии с нашей будущей схемой базы данных и конфигурации моделей Yii 2. Взгляните на следующую таблицу:

Поле	Название в форме
Name (Имя)	`CustomerRecord[name]`
Birth Date (Дата рождения)	`CustomerRecord[birth_date]`
Notes (Заметки)	`CustomerRecord[notes]`
Phone Number (Номер телефона)	`PhoneRecord[number]`

Заметим, что хотя наш проект предполагает, что один клиент может иметь несколько телефонных номеров, здесь мы допускаем только один. Мы не можем реализовывать функциональность, пока у нас нет базового теста на неё, и тот тест, который у нас есть, явным образом не проверяет возможность вводить несколько телефонных номеров.

Теперь мы можем определить метод `CRMOperatorSteps.imagineCustomer()`. В первую очередь давайте включим библиотеку Faker в наш проект:

```
$ php composer.phar require "fzaninotto/faker:*"
```

После этого мы можем «вообразить» клиента следующим образом:

```
public function imagineCustomer()
{
   $faker = \Faker\Factory::create();
   return [
      'CustomerRecord[name]' => $faker->name,
      'CustomerRecord[birth_date]' => $faker->date('Y-m-d'),
      'CustomerRecord[notes]' => $faker->sentence(8),
      'PhoneRecord[number]' => $faker->phoneNumber
   ];
}
```

Наше «воображение» в данном случае собирает для нас структуру данных, и мы можем легко использовать её в методе `fillCustomerDataForm()`:

```
function fillCustomerDataForm($fieldsData)
{
   $I = $this;
   foreach ($fieldsData as $key => $value)
      $I->fillField($key, $value);
}
```

Шаг отправки формы будет достаточно прямолинейным. Просто условимся, что кнопка называется **Submit**:

```
function submitCustomerDataForm()
{
   $I = $this;
   $I->click('Submit');
}
```

Затем у нас остаются только два метода: один для проверки, находимся ли мы в пользовательском интерфейсе списка клиентов, и другой для того, чтобы на самом деле туда отправиться:

```
public function seeIAmInListCustomersUi()
{
   $I = $this;
   $I->seeCurrentUrlMatches('/customers/');
}

function amInListCustomersUi()
{
   $I = $this;
   $I->amOnPage('/customers');
}
```

В идеологии Codeception ожидается, что имена методов проверки (assertion methods) имеют префикс see, так что мы так и сделали.

Мы используем проверку CurrentUrlMatches, которая сравнивает, используя регулярные выражения, вместо более строгой CurrentUrlEquals, потому что мы подразумеваем, что на конце URL будут некоторые параметры запроса.

Со всеми этими методами, определёнными в классе CRMOperatorSteps, у нас теперь есть первая половина нашего теста.

Теперь давайте закончим с шагами теста для пользователя CRM, который будет задавать запросы. В классе CRMUserSteps мы должны выполнить следующие изменения.

В первую очередь самое очевидное изменение:

```
function amInQueryCustomerUi()
{
   $I = $this;
   $I->amOnPage('/customers/query');
}
```

Давайте назовём поле для ввода телефонного номера тем же образом, что и такое же поле в форме добавления клиента:

```
function fillInPhoneFieldWithDataFrom($customer_data)
{
  $I = $this;
  $I->fillField(
    'PhoneRecord[number]',
      $customer_data['PhoneRecord[number]']
   );
}
```

Кнопку для запуска поиска данных о клиенте назовём **Search (Поиск)**:

```
function clickSearchButton()
{
   $I = $this;
   $I->click('Search');
}
```

Затем мы обнаруживаем дублирование метода CRMOperatorSteps.seeIAmInListCustomersUi():

```
function seeIAmInListCustomersUi()
{
   $I = $this;
```

```
        $I->seeCurrentUrlMatches('/customers/');
    }
```

Давайте в данном случае следовать Правилу трёх (**Rule of Three**), предложенному в *Refactoring: Improving the Design of Existing Code, Martin Fowler, Kent Beck, John Brant, William Opdyke, и Don Roberts, Addison-Wesley Professional*, и оставим этот метод, как он есть.

Наконец, вот наши методы проверки:

```
    function seeCustomerInList($customer_data)
    {
        $I = $this;
        $I->see($customer_data['CustomerRecord[name]'], '#search_
results');
    }
    function dontSeeCustomerInList($customer_data)
    {
        $I = $this;
        $I->dontSee($customer_data['CustomerRecord[name]'], '#search_
results');
    }
```

Следует заметить, что это экстремально простая реализация, и она полагается на несколько допущений, которые будут истинны на этом этапе разработки:

- у всех клиентов определено имя;
- нет клиентов с одинаковыми именами;
- результаты поиска рендерятся в элементе HTML, значение атрибута ID которого – `search_results`.

Когда у нас будет более одного результата поиска, нам нужно будет озаботиться тем, как корректно различать их (и, скорее всего, изначальной семантики метода `see` для нас будет недостаточно).

Достаточно важный вопрос: почему мы не проверяем, что данные о клиенте действительно отображаются в интерфейсе списка клиентов после каждого добавления нового клиента? Мы написали это в шагах 2 и 3 сценария в самом начале.

Ну, *пока что* смысл очень простой: наша цель – проверить, что мы можем запросить информацию о пользователе по его номеру телефона. Также существование каких-то проверок на полпути через тестовый сценарий будет нарушать принцип одной проверки (**Single Assertion principle**) (детально описанный в *Clean Code: A Handbook of Agile Software Craftsmanship, Robert Martin, Prentice Hall*). Хотя, конечно, раз это тест полного цикла, возможно, нарушение данного

принципа не повлечёт за собой много вреда. Попробуйте в качестве упражнения реализовать тест самостоятельно, но только после того, как дочитаете главу до конца!

Если вы теперь запустите законченный тестовый сценарий, то должны увидеть что-то наподобие следующего сообщения об ошибке:

```
1) Failed to add two different customers to database in
QueryCustomerByPhoneNumberCept
Sorry, I couldn't fill field "CustomerRecord[first_name]","Cheyanne":
Field by name, label, CSS or XPath 'CustomerRecord[first_name]' was not
found on page.

Scenario Steps:
2. I fill field "CustomerRecord[first_name]","Cheyanne"
1. I am on page "/customers/add"
```

Мы получили эту ошибку, потому что по адресу `/customers/add` ничто не возвращает HTML-форму. Наконец-то, у нас есть повод для установки Yii 2.

Установка Yii 2 на чистую базу кода

Мы собираемся сделать приложение полностью вручную, не используя готовых шаблонов приложений.

В первую очередь нам нужно объявить Yii 2 как зависимость нашего приложения.

Или добавьте необходимую для Yii 2 строчку `require` вручную в файл `composer.json`, или выполните следующую команду:

```
$ php composer.phar require "yiisoft/yii2:*"
```

> Если вы по какой-либо причине пропустили первую главу, не забудьте: вы должны глобально установить плагин Composer под названием Composer Asset Plugin, прежде чем устанавливать Yii 2! Вот команда, которую вам необходимо выполнить:
>
> ```
> $ php composer.phar global \
> require "fxp/composer-asset-plugin:1.0.0-beta3"
> ```

Если вы редактировали манифест вручную, не забудьте вызвать команду установки зависимостей:

$ php composer.phar install

Composer после этого установит вам Yii 2. Он должен оказаться в подкаталоге `vendor/yiisoft/yii2`.

Введение в соглашения Yii

Вот вам очень общее описание того, как всё это работает.

Чтобы ответить на любой запрос, который приходит к приложению, Yii использует единственный физический скрипт PHP, который создаёт экземпляр специального объекта класса `\yii\web\Application`. Этот объект использует композитный паттерн Модель–Вид–Контроллер **(Model View Controller, MVC)** в интерпретации Yii для обработки запроса и отображения результата обратно отправителю. Если вы забыли или не знали про MVC, возможно, вам захочется хотя бы официальную документацию Yii прочитать, чтобы получить полноценное представление о нём.

Модель–Вид–Контроллер в интерпретации Yii заключается в следующем:

- **Представление (View)** – это класс, который занимается рендерингом того, что будет отправлено обратно клиенту. Обычно это HTML-страница, но вы этим не ограничены;
- **Модель (Model)** – это класс, содержащий всю бизнес-логику;
- **Контроллер (Controller)** – это класс, который получает запрос пользователя, решает, что с ним делать, вызывает модели, если нужно выполнить реальную работу, затем вызывает некоторое представление для рендеринга и отправляет результат обратно пользователю.

Самая тонкая часть здесь – это концепция «модели». В зависимости от интерпретации модель – это или то, что контроллер использует, чтобы получить данные, которые нужно передать в представление, или это *и есть* то, что он передаёт в представление. Yii 2 не настаивает ни на каком из этих подходов, но его реализация моделей подразумевает, что модель – это контейнер для некоторых данных, либо временных (transient, хранящихся только в памяти), либо постоянных (persistent, с поддержкой паттерна Active Record).

Итак, запрос проходит через следующие шаги:

1. Веб-сервер получает запрос и передаёт его скрипту `index.php`.
2. Создаётся объект Yii `Application`. Он решает, какой класс контроллера должен быть использован для обработки этого запроса.
3. Создаётся объект `Controller`. Он решает, какое действие он должен выполнить (действия представлены либо отдельными классами `Action`, либо методами класса `Controller`), и выполняет его, передавая в него детали запроса.
4. Действие выполняется, и если оно было корректно написано программистом, оно возвращает что-то, отрендеренное пред-

ставлением. Каркас к этому никак не принуждает; у вас могут быть действия контроллера, которые ничего не отображают.

5. Специальный компонент приложения, ответственный за форматирование данных перед отправкой их пользователю, делает своё дело.
6. Полученные данные, будь то HTML, JSON, XML или пустой ответ, отправляются обратно пользователю.

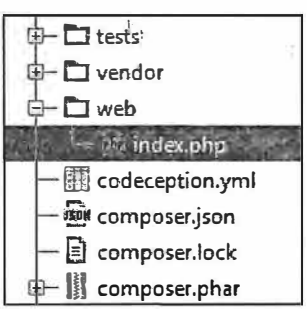
Зная эти шаги, давайте модифицируем наш текущий скрипт точки входа так, что он будет рендерить то же самое, но вместо вывода сырого текста он будет использовать фреймворк Yii.

Вы можете посмотреть на красивую схему со стрелочками в официальной документации Yii. Подробности будут описаны в *главе 12 «Управление маршрутизацией»*.

Само «приложение» состоит из отдельных «компонентов», которые представлены подклассами класса `\yii\base\Component`. В любой момент времени любой из компонентов доступен в виде именованного свойства объекта приложения и предоставляет некоторые методы согласно своей специфике.

Строим фреймворк кода

На данный момент у нас должна быть вот такая структура проекта:



Мы начнём вставлять Yii 2 в проект со скрипта точки входа. Разумный минимум того, что должно быть в файле `index.php`, выглядит так:

```
<?php
// Включаем сам фреймворк Yii (1)
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
// Получаем конфигурацию (2)
$config = require(__DIR__ . '/../config/web.php');
// Создаём и немедленно выполняем приложение (3)
(new yii\web\Application($config))->run();
```

На шаге (1) мы включаем всё, что Yii нужно, в наше окружение.

На шаге (2) мы получаем дерево конфигурации для приложения. Конфигурация приложения Yii – это здоровенный массив PHP, описывающий начальные значения атрибутов как приложения, так и различных его компонентов.

На шаге (3) мы создаём новый экземпляр подкласса `Application`, представляющий веб-приложение, и немедленно вызываем его метод `run`.

На шаге (2) мы загружаем несуществующий файл `config/web.php`. Давайте сделаем его:

```
<?php
return [
   'id' => 'crmapp',
   'basePath' => realpath(__DIR__ . '/../'),
   'components' => [
      'request' => [
         'cookieValidationKey' => 'your secret key here',
      ],
   ]];
```

Мы должны указать три вещи:

- `id`: это обязательный идентификатор для нашего приложения. То, почему он необходим, будет подробно объяснено в *главе 7 «Модули»*;
- `basePath`: этот параметр обязателен, потому что он является практически единственным способом для Yii понять, где он находится в файловой системе. Все относительные пути, разрешённые в других настройках, начинают свой отсчёт отсюда;
- `components.request.cookieValidationKey`: это «протечка» из подсистемы аутентификации пользователя, которую мы обсудим в *главе 6 «Аутентификация»*. Эта настройка – тайный ключ для валидации пользователей, использующих известную возможность *«запомнить меня»* (*remember me*), которая полагается на cookies. В ранних бета-версиях Yii 2 этот ключ генерировался автоматически. Он стал видимым после коммита 4e4e76e8 (https://github.com/yiisoft/yii2/commit/4e4e76e8838cbe097134d6f9c2ea58f20c1deed6). Кроме этой настройки, вы можете установить `components.request.enableCookieValidation` равной `false`, что полностью выключит аутентификацию через cookies. Так ваше приложение тоже заработает.

Заметьте, что мы можем настраивать не только свойства самого приложения, но и свойства его компонентов.

Затем мы добавим некоторые обязательные подкаталоги, потому что Yii просто бросает исключения в случае, если их нет, и не создаёт их самостоятельно. Это подкаталоги `web/assets` и `runtime`. Данные папки используются фреймворком, когда приложение работает.

Добавляем контроллер

Каждый контроллер должен обладать тремя особенностями:

- он должен находиться в пространстве имён, определённом в настройке `controllerNamespace` приложения;
- имя его класса должно иметь суффикс `Controller`;
- он должен быть подклассом класса `\yii\base\Controller`. В случае контроллеров, которые предназначены для использования веб-приложением, а не консольным, мы должны расширять класс `\yii\web\Controller`. Для консольного приложения используйте `\yii\console\Controller`.

Также важно понимать, как Yii 2 будет искать классы контроллеров.

Yii 2 использует автозагрузчик, совместимый с набором правил PSR-4 (см. http://www.php-fig.org/psr/psr-4/). Если коротко, то такой автозагрузчик рассматривает пространства имён как пути в файловой системе, при условии что существует специальное корневое пространство имён, которое явно было отображено на определённый корневой каталог в базе кода.

В нашем случае Yii 2 самостоятельно определяет пространство имён `\app`, которое соответствует корневому каталогу проекта. В результате, например, значение настройки `controllerNamespace` по умолчанию, которым является строка «`\app\controllers`», соответствует подкаталогу `controllers` в корневом каталоге, поэтому все определения классов контроллеров должны находиться там.

Также каждый класс, который должен быть доступен через автозагрузчик Yii 2, должен находиться в отдельном файле, названном так же, как и сам класс.

Давайте создадим наш первый контроллер, чтобы наш «дымовой тест» был пройден. Мы не будем менять значение настройки, устанавливающей пространство имён контроллеров, так что напишем следующий код в файле `controllers/SiteController.php`:

```
namespace app\controllers;
use \yii\web\Controller
```

```
class SiteController extends Controller
{
   public function actionIndex()
   {
      return 'Our CRM';
   }
}
```

Этот код очень сильно полагается на соглашения Yii. Не углубляясь в тему маршрутизации, можно сказать, что без специальных настроек Yii использует метод `actionIndex` контроллера под названием `SiteController`, для того чтобы обрабатывать запрос «/».

Самый простой и прямолинейный способ определять действия контроллеров – это определять их как публичные методы контроллеров, имена которых имеют префикс `action`. Чтобы явно добраться до метода `SiteController.actionIndex`, вам следует сделать запрос `site/index`, то есть, например, в случае локальной установки перейти по URL http://localhost/site/index. Впрочем, чтобы добиться обработки ссылок в таком виде, нам понадобятся дополнительные телодвижения, о чём позже.

Итак, с базовой маршрутизацией Yii наш «дымовой тест» наконец проходит. Давайте добавим некоторые вспомогательные возможности для облегчения отладки.

Облегчение отладки возможных ошибок

На этом этапе разработки вы можете получить множество странных ошибок. Давайте посмотрим, что можно быстро сделать, чтобы получить как можно больше обратной связи.

В первую очередь, если вы действительно серьёзно что-то сломаете, например не определите `id` или `basePath` в конфигурации приложения, вы в результате получите белую страницу в качестве ответа от Yii. Единственное место, куда можно будет посмотреть в данном случае, – это логи веб-сервера. Например, в Apache вы можете использовать директиву `ErrorLog` для указания файла, в который будут записываться отчёты о таких фатальных ошибках. Конечно же, все остальные ошибки тоже там окажутся, вне зависимости, показали вы их в браузере или нет.

Чтобы избавиться от «проблемы белого экрана», вы можете принудительно переопределить настройку PHP `display_errors` в вашем скрипте точки входа `index.php` сразу *после* того, как вы подключили библиотеку Yii, но *до* создания и выполнения объекта класса `Application`:

```
ini_set('display_errors', true);
```

Также вам следует определить одну полезную константу *до* того, как вы подключите библиотеку Yii. Критически важно определить её до подключения Yii, так как фреймворк самостоятельно её определит, если она не была определена заранее. Вот как это делается:

```
define('YII_DEBUG', true);
```

Это переключит приложение в режим отладки, и в случае появления каких-либо исключений вы получите не просто стандартную страницу статуса 500 или пустой экран, но детальный отчёт от Yii с подсветкой наиболее важных строк.

В Yii 2 появился механизм переключения между «окружениями», который основан на использовании константы YII_ENV. В целом это более удобный механизм, чем константа YII_DEBUG, потому что у вас может быть более двух «окружений». Посмотреть, как используется эта константа, можно в коде точек входа расширенного шаблона приложения.

Наконец, вы можете добавить вручную сделанный механизм отчётов, который будет записывать ошибки в файл средствами Yii 2. Конечно же, он не будет работать, если ошибка произошла до подключения самого Yii 2. *Глава 8 «Поведение в целом»* подробно объясняет этот вариант.

Создаём слои данных и приложения

Теперь перейдём к настоящей работе.

Чтобы удовлетворить нашему приёмочному тесту, мы можем начать с различных уровней нашего приложения. Так как мы заранее знаем, что будем работать с Yii, давайте разберёмся, что нам нужно делать в контроллере.

Нам нужно предоставить два маршрута: `/customers/add` и `/customers`. Вот определение контроллера, которое нам нужно для них:

```
namespace app\controllers;
use yii\web\Controller;

class CustomersController extends Controller
```

Согласно настройкам по умолчанию, маршрут `/customers` соответствует маршруту `/customers/index`. Нам нужно предоставить метод под названием `actionIndex`, чтобы включить этот маршрут.

Что мы будем делать в ответ на запрос по этому маршруту? Традиционно в данном случае возвращают список всех записей, имеющихся в БД. У нас нет теста на эту функциональность, поэтому мы избавим себя от данной задачи в рамках текущей главы. Однако будет довольно глупо считать, что нам *никогда* не понадобится такая функциональность. На самом деле нам и так нужно возвращать список записей о клиентах, только не обо всех клиентах, а соответственно запросу по номеру телефона. То есть мы будем ожидать, что нам передадут некоторый параметр запроса, и отсюда следует, что наш метод `actionIndex` должен выглядеть следующим образом:

```
public function actionIndex()
{
   $records = $this->findRecordsByQuery();
   return $this->render('index', compact('records'));
}
```

Встроенная в PHP функция `compact('var_name_1', 'var_name_2', ...)` невероятно полезна при использовании Yii. Есть очень много мест, где вам нужно передавать в функции ассоциативные массивы, и переменные будут иметь те же имена, что и ключи в этих массивах. Если вы не знали об этой функции, мы предлагаем вам проконсультироваться с документацией по функциям PHP и узнать (см. http://php.net/manual/en/function.compact.php).

Теперь нам осталось только разобраться с методом `findByQuery` («получить записи согласно запросу»). Чтобы доделать его, нам сначала нужна база данных. Но до этого давайте определимся с моделью клиента.

Определение модели клиента на слое данных

Модель клиента – это просто класс, хранящий данные, так что нам на него не нужны тесты. Вот как мы его определим:

```
namespace app\models\customer;

class Customer {
   /** @var string */
   public $name;

   /** @var \DateTime */
   public $birth_date;

   /** @var string */
```

```
    public $notes = '';

    /** @var PhoneRecord[] */
    public $phones = [];

    public function __construct($name, $birth_date)
    {
        $this->name = $name;
        $this->birth_date = $birth_date;
    }
}
```

Так как мы поместили его в пространство имён `app\models\customer`, нам нужно поместить файл с этим определением в подкаталог `models/customer`, или его не найдёт автозагрузчик.

Мы представляли себе ещё один объект домена: `Phone`. Вот как он определён:

```
namespace app\models\customer;

class Phone {
    /** @var string */
    public $number;
}
```

Таким образом, наш агрегат `Customer` будет не более чем структурой данных, хранящей другие структуры данных, состоящие из значений примитивных типов. Мы сделали невозможным создание объектов класса `Customer` без указания имени и даты рождения, но, кроме этого, все поля публичные, так что любой может делать всё, что угодно, с агрегатом.

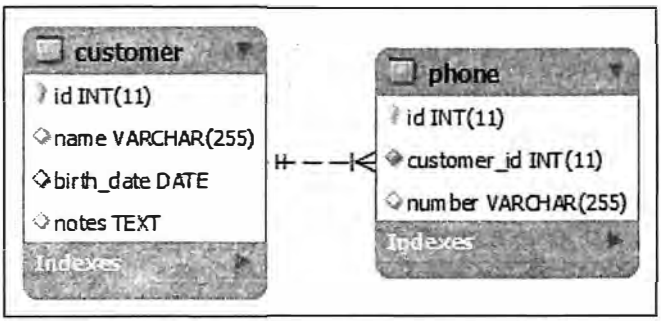
Теперь давайте подумаем о том, как мы будем хранить эту модель в базе данных.

Подготовка базы данных

Допустим, что мы используем СУБД MySQL и выполнили в её консоли следующую команду:

```
create database `crmapp` default character set utf8 default collate
utf8_unicode_ci;
```

Наиболее разумный способ организации данных для хранения нашего агрегата `Customer` изображён на следующей схеме (снято с использованием MySQL Workbench, доступной для скачивания с http://dev.mysql.com/downloads/workbench/):



Так как мы совершенно точно не собираемся создавать эту схему данных при каждом развёртывании, нам нужен некий способ делать это автоматически. Yii включает в себя поддержку концепции «миграций» специально по этой причине.

Однако чтобы использовать её, нам сначала нужны ещё две вещи.

Для начала нам нужна своя версия исполнителя консольных команд Yii. С ним мы сможем в будущем конструировать свои собственные консольные команды, но эту тему мы рассмотрим только в самом конце книги, в *главе 13 «Совместная работа»*. Сейчас нам нужна возможность выполнять команду `migrate`, уже встроенную в фреймворк. Создайте файл под названием `yii` в корневом каталоге приложения и напишите в нём следующее, дословно:

```
#!/usr/bin/env php
<?php
define('YII_DEBUG', true);

require(__DIR__ . '/vendor/autoload.php');
require(__DIR__ . '/vendor/yiisoft/yii2/Yii.php');

$config = require(__DIR__ . '/config/console.php');

$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

Фактически это обрезанная версия скрипта `yii`, который поставляется с базовым шаблоном приложения, который мы устанавливали в *главе 1 «Начинаем»*. Можно увидеть как сходства, так и различия со скриптом точки входа в веб-приложение `index.php`.

Не забудьте сделать этот файл исполняемым!
Если вы на POSIX-совместимой системе, это может быть сделано командой `chmod +x yii`.

Этому скрипту нужен файл конфигурации под названием `config/console.php`. Создадим его с вот таким содержимым:

```
<?php
return [
    'id' => 'crmapp-console',
    'basePath' => dirname(__DIR__),
    'components' => [
        'db' => require(__DIR__ . '/db.php'),
    ],
];
```

Класс `yii\console\Application` в некоторых местах не отличается от `yii\web\Application`, и при инициализации консольного приложения нам всё так же нужно вручную указывать `id` и `basePath`.

Видите строчку, начинающуюся с `db`? Это место, где хранятся настройки подключения к нашей БД (наконец-то). Настройки для компонента `db` выделены в отдельный скрипт, потому что нам нужны в точности те же настройки для нашего веб-приложения. Вот как выглядит файл `db.php`:

```
<?php
return [
    'class' => '\yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=crmapp',
    'username' => 'root',
    'password' => 'cheesy/hamburger'
];
```

Конечно же, ваше имя пользователя и пароль будут другими. База данных с таким названием также уже должна существовать в MySQL к этому моменту, Yii ничего создавать за вас не будет.

Имея все эти три элемента на своих местах: скрипт запуска команд `yii`, файл конфигурации `config/console.php` для него и настройки подключения к БД, выделенные в файл `config/db.php`, – мы теперь можем создать скрипт миграции, используя следующую команду:

```
$ ./yii migrate/create init_customer_table
```

Это автоматически создаст подкаталог `migrations` в корне проекта и класс с по-настоящему длинным названием вроде `m140204_190825_init_customer_table` в отдельном файле внутри этого подкаталога.

Подробно миграции мы рассмотрим в последней главе, *главе 13 «Совместная работа»*. Если коротко, то скрипт миграции в стиле Yii – это класс, содержащий два метода: `up()` и `down()`. В методе `up()` вы описываете изменения, которые хотите совершить в БД. Для описания этих изменений класс предоставляет вам набор удобных методов на все случаи жизни, наподобие `createTable()`, `alterColumn()`, `insert()`, и даже `execute()`, позволяющий выполнить произвольный SQL-код. В методе `down()` вы описываете, как отменить изменения, сделанные методом `up()`, или указываете, вернув из него `false`, что вы сдаётесь; это невозможно, и вы не можете отменить эту конкретную миграцию.

Когда вы запускаете `yii migrate/up` или его сокращение `yii migrate`, Yii проверяет все классы в подкаталоге `migrations`, которые сортированы естественным образом по штампам времени в именах, и выполняет метод `up()` у всех них, от первого до последнего. Затем он записывает имена применённых миграций в таблицу под названием `tbl_migration` (имя настраивается) в целевой БД, так что он не будет выполнять те же самые миграции в следующий раз. Для безопасности команда `yii migrate/down` не имеет никаких сокращений и по умолчанию отменяет только *одну* миграцию за раз. Вы можете указать количество отменяемых миграций вызовом `yii migrate/down <число>`.

Имея скрипты миграции, вы можете просто добавить строчку `yii migrate` в ваш сценарий развёртывания и быть уверены, что все изменения в БД, которые вам нужны, будут сделаны.

Вот содержимое метода `up` в нашем случае, когда нам надо создать таблицу `customer`:

```
$this->createTable(
    'customer',
    [
        'id' => 'pk',
        'name' => 'string',
        'birth_date' => 'date',
        'notes' => 'text',
    ],
    'ENGINE=InnoDB'
);
```

Вы отменяете это изменение вот такой простой строчкой кода в методе down:

```
$this->dropTable('customer');
```

В другом скрипте миграции мы напишем создание таблицы phone и дополнительно объявим внешний ключ к таблице customer:

```
    $this->createTable(
        'phone',
        [
            'id' => 'pk',
            'customer_id' => 'int unique',
            'number' => 'string',
        ],
        'ENGINE=InnoDB'
    );
    $this->addForeignKey('customer_phone_numbers', 'phone', 'customer_
id', 'customer', 'id');
```

Отменять нужно в обратном порядке, или MySQL будет сопротивляться:

```
    $this->dropForeignKey('customer_phone_numbers', 'phone');
    $this->dropTable('phone');
```

Не забудьте на самом деле выполнить эти миграции командой:

```
./yii migrate
```

Теперь, пока мы ещё здесь, давайте настроим объектно-реляционное отображение (**object-relation mapping, ORM)** для этих двух таблиц.

Так как с этого момента нам нужно подключение из веб-приложения к БД, скопируйте из файла config/console.php строчку

```
    'db' => require(__DIR__ . '/db.php'),
```

Вставьте её в раздел components файла настроек config/web.php.

ORM в Yii

ORM в Yii поддерживается паттерном «Активная запись» (**Active Record**). Разработчик определяет классы объектов, которые представляют собой таблицы в БД (один класс на таблицу), и довольно много функциональностей по манипуляциям с этими таблицами становятся уже реализованными за него. Реализация этого паттерна в Yii помогает:

- сохранять активные записи в БД;
- производить валидацию данных, присвоенных полям активной записи, перед сохранением их в БД;
- извлекать активные записи из БД по первичному ключу, значениям атрибутов либо по каким-либо произвольным запросам.

Всё, что вам нужно, – это корректно сформулировать определение активной записи. Давайте начнём с таблицы `customer`.

Так как имя «customer» уже занято нашей моделью предметной области, назовём активную запись `CustomerRecord`, что логично. Определим следующий класс в пространстве имён `app\models\customer`:

```
namespace app\models\customer;

use yii\db\ActiveRecord;

class CustomerRecord extends ActiveRecord
{
   public static function tableName()
   {
      return 'customer';
   }

}
```

Опять же, чтобы автозагрузчик смог найти этот класс, он должен быть сохранён в файле `CustomerRecord.php` в подкаталоге `models/customer`.

Если честно, это всё, что вы должны определить. Если вы работали раньше с Yii 1.1, то, возможно, вы сейчас поражены (как были и мы).

Впрочем, нам не хватает правил валидации данных. Так как у нас уже есть схема таблицы, определённая и загруженная в БД, ничто не останавливает нас от указания нескольких правил в этом же классе:

```
   public function rules()
   {
      return [
         ['id', 'number'],
         ['name', 'required'],
         ['name', 'string', 'max' => 256],
         ['birth_date', 'date', 'format' => 'Y-m-d'],
         ['notes', 'safe']
      ];
   }
```

Вероятно, никаких объяснений не требуется, чтобы в целом понять это определение, так как оно и так читается, словно проза, за исключением правила safe («безопасный»). На данный момент достаточно лишь знать, что это правило означает, что мы можем присвоить что угодно в это поле. Вы можете задать вопрос «почему оно вообще тогда существует?»; потому что в классе \yii\base\Model есть специальный вспомогательный метод под названием setAttributes, который принимает ассоциативный массив с названиями и значениями атрибутов. По умолчанию, если на атрибут не наложено какое-либо правило валидации и он не помечен как «безопасный» правилом safe, он будет этим методом проигнорирован. Эта возможность позволяет нам передавать, не глядя, всё содержимое $_POST методу setAttributes() и оставаться уверенными, что только ожидаемые значения будут присвоены активной записи.

> Вы можете прочитать полный список встроенных правил валидации в документации Yii: http://www.yiiframework.com/doc-2.0/guide-tutorial-core-validators.html.

Класс PhoneRecord немного проще:

```
namespace app\models\customer;
use yii\db\ActiveRecord;

class PhoneRecord extends ActiveRecord
{
   public static function tableName()
   {
      return 'phone';
   }

   public function rules()
   {
      return [
         ['customer_id', 'number'],
         ['number', 'string'],
         [['customer_id', 'number'], 'required'],
      ];
   }
}
```

Мы указали, что поле number должно проходить валидацию как произвольная строка, потому что мы не хотим заставлять пользователя

вводить номер телефона в каком-то определённом формате на этом этапе разработки. Возможно, позже мы что-нибудь бы и придумали.

Атрибуты `CustomerRecord.name`, `PhoneRecord.customer_id` и `PhoneRecord.number` объявлены обязательными, чтобы предотвратить возможность отправки пустых форм.

Оба этих класса должны быть размещены в подкаталоге `models/customer`.

Теперь мы сделали всё, относящееся к подготовке базы данных. Давайте уже использовать её.

Отделяемся от ORM

Так как у нас есть наш крошечный, но вполне самостоятельный слой предметной области, состоящий из моделей `Customer` и `Phone`, будет разумным шагом держать его отделённым от фреймворка. Поэтому нам нужен слой перевода между ORM от Yii 2 и моделями предметной области. Однако описание должной разработки паттерна **Репозиторий** (**Repository**, см. http://martinfowler.com/eaaCatalog/repository.html) займёт слишком много места, и это не соответствует теме книги. Мы остановимся на реализации всего двух методов в классе `CustomersController`, но в целом это всего лишь компромиссное решение ради нашего приложения-примера. В реальном, крупном приложении вам обязательно понадобится правильный слой трансляции по следующим четырём причинам.

- Практически гарантировано то, что структура вашей модели предметной области не будет соответствовать структуре базы данных, которую вы используете. Это называется object-relation impendance mismatch, что грубо можно перевести как «несоответствие объектов реляциям». Только очень простые модели предметной области можно отобразить один к одному на таблицы БД. Этому препятствует в том числе необходимость в некоторых случаях денормализовывать таблицы для повышения производительности, дублируя данные.
- ORM на основе активных записей очень удобна в использовании, но она достаточно дорого обходится, так как делает слишком много запросов к БД. В определённый момент вы захотите заменить использование активных записей в некоторых местах на что-то более низкоуровневое, как, например, механизм **DAO** (https://github.com/yiisoft/yii2/blob/master/docs/guide/db-dao.md) в Yii, или, возможно, полностью обойти фреймворк и вызывать **PDO** напрямую. Без паттерна Репозиторий или подоб-

ного ему решения (типа CQRS), вполне вероятно, вашей единственной возможностью будет только полнотекстовый поиск по всему проекту имён ваших подклассов `ActiveRecord`.

- Возможно, в определённый момент вы захотите заменить нижележащую БД с чего-то, что Yii поддерживает, на что-то, чего Yii не поддерживает. Опять же, чтобы сделать это изменение, вам придётся сделать множество правок в самых разных местах.
- Вполне возможно, что ваше приложение переживёт Yii версии 2 и встретится с версией 3, которая будет иметь произвольные изменения в публичном API. Без отделения от фреймворка, как минимум от его наиболее широко проникающей части, которой является слой доступа к данным, вы будете практически лишены всякой возможности обновиться.

Чтобы не раздувать слишком эту главу, давайте добавим следующие методы в `CustomersController`.

Первым будет метод для сохранения модели клиента в базе данных, следующим образом:

```
private function store(Customer $customer)
{
    $customer_record = new CustomerRecord();
    $customer_record->name = $customer->name;
    $customer_record->birth_date = $customer->birth_date->format('Y-
m-d');
    $customer_record->notes = $customer->notes;

    $customer_record->save();

    foreach ($customer->phones as $phone)
    {
        $phone_record = new PhoneRecord();
        $phone_record->number = $phone->number;
        $phone_record->customer_id = $customer_record->id;
        $phone_record->save();
    }
}
```

Как видно, мы получаем экземпляр `Customer`, но используем активные записи, для того чтобы сохранить данные из него в БД. Обратите внимание, что новый экземпляр `CustomerRecord` магическим образом получает значение в поле `id` после того, как он был сохранён методом `save()`.

Следующий – метод для конвертирования активных записей в экземпляр класса Customer:

```
private function makeCustomer(
    CustomerRecord $customer_record,
    PhoneRecord $phone_record
) {
    $name = $customer_record->name;
    $birth_date = new \DateTime($customer_record->birth_date);

    $customer = new Customer($name, $birth_date);
    $customer->notes = $customer_record->notes;
    $customer->phones[] = new Phone($phone_record->number);

    return $customer;
}
```

Мы принимаем единственный экземпляр PhoneRecord, потому что прямо сейчас мы имеем дело только с одним номером телефона на клиента. Этот метод нужно будет изменить, когда появится необходимость поддерживать несколько телефонных номеров на клиента.

Эти два метода, по сути, являются слоем трансляции между Yii и нашей моделью предметной области.

Теперь мы сделаем пользовательский интерфейс и по пути реализуем запросы по номеру телефона.

Создание пользовательского интерфейса

Имея CustomersController, способный превращать модели предметной области в активные записи, мы, наконец, переходим к страницам пользовательского интерфейса.

Пользовательский интерфейс добавления клиента

Что для нас должен сделать контроллер, когда мы прибудем на URL /customer/add? Ну, он должен просто отрисовать для нас пользовательский интерфейс:

```
public function actionAdd()
{
    return $this->render('add');
}
```

Да-да, это всё (пока что). Этот код полагается на важное соглашение в Yii, описывающее, откуда контроллер должен брать свои

представления. Фактически, если у нас есть класс под названием `CustomersController`, тогда ожидается, что файлы, содержащие код его представлений, находятся в подкаталоге `views/customer`. Так что когда мы рендерим что-то, называемое «`add`», мы обращаемся к файлу `views/customer/add.php`. Давайте создадим его.

Что мы хотим видеть на этой странице? Согласно нашему приёмочному тесту, там должна быть форма для ввода информации о пользователе, включая номер его телефона, а также кнопка **Submit**.

Yii содержит широкий выбор вспомогательных методов, для того чтобы быстро и легко составлять веб-формы. Центральная концепция позади них – это класс `ActiveForm`. Мы инициализируем `ActiveForm` и затем используем его методы, для того чтобы генерировать HTML-код для полей ввода, соответствующих атрибутам моделей. Вот как должен выглядеть файл `views/customers/add.php` с использованием `ActiveForm`:

```
<?php
use app\models\customer\CustomerRecord;
use app\models\customer\PhoneRecord;
use yii\web\View;
use yii\helpers\Html;
use yii\widgets\ActiveForm;

/**
 * Add Customer UI.
 *
 * @var View $this
 * @var CustomerRecord $customer
 * @var PhoneRecord $phone
 */

$form = ActiveForm::begin([
    'id' => 'add-customer-form',
]);

echo $form->errorSummary([$customer, $phone]);
echo $form->field($customer, 'name');
echo $form->field($customer, 'birth_date');
echo $form->field($customer, 'notes');

echo $form->field($phone, 'number');

echo Html::submitButton( Submit, ['class' => 'btn btn-primary']);
ActiveForm::end();
```

Это описание интерфейса безо всяких украшательств; мы не делаем никаких дополнительных HTML-обёрток, помимо тех, которые будут за нас сделаны вспомогательными классами `ActiveForm` и `Html`.

Это единственный дословный пример файла представления в данной книге. Мы больше не будем показывать выражения «`use`» и блоки документации, чтобы сохранить место.

Обратите внимание на метод `field()`, который принимает экземпляр `ActiveRecord` и название атрибута для обработки. Это главный метод, больше всего облегчающий нам жизнь при генерации кода веб-форм. С помощью этого метода мы можем сгенерировать HTML-код поля ввода для произвольной активной записи, так как сам `ActiveForm` никак не связан с каким-то одним классом активных записей.

Метод под названием `errorSummary()` – это очень полезное сокращение, которое покажет нам все ошибки валидации данных в переданных ему моделях, в случае если введённые данные не соответствуют объявленным правилам валидации.

Но для того, чтобы это представление могло работать, у него должен быть доступ к объектам моделей – в нашем случае `$customer` и `$phone`. Это делается с использованием специального механизма передачи данных из класса `View` в `Controller` через второй аргумент метода `render()`. В целом мы, как минимум, должны иметь вот такое содержимое метода `actionAdd()`, чтобы форма была корректно отрисована:

```
public function actionAdd()
{
   $customer = new CustomerRecord;
   $phone = new PhoneRecord;
   return $this->render('add', compact('customer', 'phone'));
}
```

Теперь нам нужно решить две проблемы, до того, как мы на самом деле сможем посмотреть на наш интерфейс.

Вводный курс маршрутизации

По умолчанию ссылки в Yii выглядят вот так:

`http://yourdomain/index.php?r=controller/action`

Поэтому, если у нас есть класс `CustomersController` и в нём метод `actionAdd()`, мы должны обращаться к нему по URL http://yourdomain/index.php?=customers/add.

Это довольно печальная картина, и у объекта приложения Yii есть специальная настройка, для того чтобы изменить способ работы

с URL на более привычный формат. Вам нужно добавить следующее в раздел components конфигурации вашего приложения:

```
'urlManager' => [
   'enablePrettyUrl' => true,
   'showScriptName' => false
]
```

Если enablePrettyUrl установлено так, как показано выше, ссылки станут выглядеть вот так: http://yourdomain/index.php/customers/add.

Если showScriptName установлено так, как показано выше, то ссылки, *создаваемые* Yii, не будут содержать упоминания точки входа index.php. Для того чтобы ваше веб-приложение могло *распознавать* такие ссылки, вы должны настроить соответствующее переписывание URL на вашем веб-сервере. Для примера разработчики, использующие Yii, обыкновенно настраивают Apache следующими строчками в файле .htaccess, который положен в их подкаталог web:

```
RewriteEngine on

# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

Для любого другого веб-сервера логика такая же.

С корректно настроенным переписыванием URL вы сможете использовать ссылки вида http://yourdomain/customers/add, что, конечно же, то, что нужно.

Мы в мельчайших деталях поговорим о маршрутизации в Yii 2 в *главе 12 «Управление маршрутизацией»*.

Шаблоны

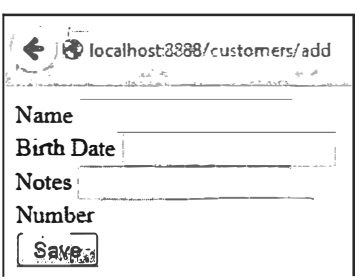
Мы поговорим о рендеринге в Yii в *главе 4 «Рендерер»*. Пока что достаточно знать всего одну деталь. По умолчанию, когда вы вызываете Controller.render(), этот метод подразумевает, что представление, которое вы отрисовываете, – это просто отдельный фрагмент, который нужно вставить в другой скрипт, называющийся шаблоном (layout). По умолчанию все контроллеры ищут шаблон в файле views/layouts/main.php. Результат рендеринга того, ради чего вы вызывали метод render(), передаётся в шаблон в виде строковой переменной под на-

званием `$content`. За исключением этого, шаблон является просто ещё одним файлом представления. Ожидается, что у внешнего вида вашего приложения есть некоторые элементы, представленные на всех страницах, и шаблон – это именно то место, где их надо держать.

Нас вполне устраивают настройки по умолчанию, так что давайте просто создадим базовый фреймворк HTML5-страницы и увидим уже, наконец, нашу форму добавления клиента. Поместите следующий код в файл `views/layouts/main.php`:

```
<!DOCTYPE html>
<html>
<head>
   <title>CRM</title>
</head>
<body>
   <?= $content; ?>
</body>
</html>
```

Итак, теперь вы можете открыть путь `/customers/add` тем или иным методом из описанных в предыдущем разделе «*Вводный курс маршрутизации*».



Что? Вы ожидали какие-то сложные трюки с CSS на этом этапе разработки? Прямо сейчас наша цель – реализовать голую функциональность, так что этого базового HTML более чем достаточно.

Завершение интерфейса добавления клиента

Идиоматический код, основанный на Yii, подразумевает, что формы отправляют данные на тот же маршрут, который использовался, чтобы отрисовать их. Мы будем использовать тот же подход; таким образом, нам нужно обрабатывать возможные входные данные в методе `actionAdd()`. Мы сделаем это следующим образом:

```
    public function actionAdd()
    {
        $customer = new CustomerRecord;
        $phone = new PhoneRecord;

        if ($this->load($customer, $phone, $_POST))
        {
            $this->store($this->makeCustomer($customer, $phone));
            return $this->redirect('/customers');
        }

        // магия состояний: и $customer и $phone прошли валидацию к этому
моменту
        return $this->render('add', compact('customer', 'phone'));
    }
```

Обратите внимание на выделенные строчки. Когда (и если) мы успешно загрузили данные, отправленные нам POST-запросом, мы сохраняем получившуюся модель клиента в базе данных при помощи методов, определённых в нашем слое трансляции. Метод под названием `load()` – это сокращение для следующих четырёх проверок:

```
    private function load(CustomerRecord $customer, PhoneRecord $phone,
array $post)
    {
        return $customer->load($post)
            and $phone->load($post)
            and $customer->validate()
            and $phone->validate(['number']);
    }
```

Вот эти четыре проверки:

- `\yii\base\Model::load()` – это встроенный метод, позволяющий заполнить атрибуты модели данными, переданными POST-запросом, который условно сделал виджет `ActiveForm`;
- `\yii\base\Model::validate()` – это встроенный метод, который проверяет значения атрибутов модели на соответствие правилам валидации, определённым в `\yii\base\Model::rules()`. Заметьте, что мы можем провести валидацию только некоторых избранных атрибутов, а не всех сразу. В нашем случае нам нужно провести валидацию лишь атрибута `number`, потому что в данных формы, переданных нам, ещё нет поля `customer_id`, а оно было объявлено обязательным. Впрочем, есть способы решить эту задачу по-другому;

- метод `\yii\db\BaseActiveRecord::save()` вызывает `validate()`, перед тем как на самом деле сохранить данные в БД.

Идея, лежащая в основе реализации `validate()`, заключается в том, что если есть какие-то ошибочные значения, то этот метод сохраняет сообщение об ошибке в специальном атрибуте `\yii\base\Model::$errors`. Именно это позволяет нам использовать метод под названием `errorSummary` у класса `ActiveForm`. Он проверит сообщения об ошибке одно за другим и красиво их распечатает.

Когда элемент успешно добавлен, мы делаем перенаправление на список клиентов. Это, возможно, пока что самое интересное действие контроллера.

```
public function actionIndex()
{
    $records = $this->findRecordsByQuery();
    return $this->render('index', compact('records'));
}
```

Эта часть уже ранее обсуждалась. Здесь же мы должны выводить список найденных записей. Однако теперь мы на самом деле готовы «найти записи согласно запросу». Обычно мы просто обращаемся к суперглобальной переменной `$_GET`, чтобы получать параметры запроса, но в Yii есть своя обёртка вокруг него, поэтому давайте используем её. Специальный компонент приложения под названием `Request` манипулирует параметрами запроса и предоставляет метод под названием `get`. С этим методом мы можем проверить, был ли некоторый параметр `$_GET` установлен в какое-либо значение.

```
private function findRecordsByQuery()
{
    $number = Yii::$app->request->get('phone_number');
    $records = $this->getRecordsByPhoneNumber($number);
    $dataProvider = $this->wrapIntoDataProvider($records);
    return $dataProvider;
}
```

Здесь мы немного смухлевали: вы ещё не знаете, что такое провайдеры данных (название метода «`wrapIntoDataProvider`» переводится как «обернуть в провайдер данных»). Чтобы понять, зачем нам нужно оборачивать записи, которые наш запрос нам предоставляет, в провайдер данных, нам нужно знать, как мы собираемся отрисовывать результаты.

Виджеты

Сами по себе провайдеры данных не важны. Их важность в том, что практически все виджеты, встроенные в Yii, используют их в качестве источника моделей для отрисовки.

Виджет можно представить себе как вариацию класса представления (View) в MVC модели Yii, с некоторой дополнительной логикой. Мы будем иметь возможность детально обсудить виджеты в *главе 11 «Таблица»*.

Типичными встроенными виджетами являются:

- `ListView` для инкапсуляции отрисовки списка моделей;
- `DetailView` для инкапсуляции отрисовки детальной информации об одной конкретной модели;
- `GridView` для инкапсуляции отрисовки табличного представления набора моделей. Мы обсудим этот мощный элемент пользовательского интерфейса в *главе 11 «Таблица»*.

Виджеты используются в представлениях следующим образом:

```
echo \yii\widgets\DetailView::widget($settings);
```

Настройки передаются в виджеты в виде ассоциативных массивов начальных значений переменных – членов класса виджета. Учитывая объём самодокументации у всех классов Yii, даже без какой-либо пользовательской документации и примеров кода вы можете просто открыть определение класса виджета и разобраться, как настраивать его.

Итак, нам нужны провайдеры данных, потому что они инкапсулируют действие по поиску набора моделей, требуемых для отрисовки в данный момент. Они делают сортировку, разбивку на страницы и фильтрацию за вас. Они наиболее полезны, когда вы используете ActiveRecord'ы как модели предметной области в вашем приложении (то есть когда у вас есть однозначное отображение таблиц БД на модели предметной области). В нашем случае, когда мы делаем всё возможное для того, чтобы отвязаться от ORM, провайдеры данных нам нужны только как обёртки вокруг наших данных, чтобы удовлетворить требованиям виджетов.

Пользовательский интерфейс списка клиентов

Идеальный провайдер данных для наших целей – это `yii\data\ArrayDataProvider`, который просто получает список готовых моделей и оборачивает их, позволяя скармливать их виджетам.

Вот что мы делаем в методе wrapIntoDataProvider():

```
private function wrapIntoDataProvider($data)
{
    return new ArrayDataProvider(
        [
            'allModels' => $data,
            'pagination' => false
        ]
    );
}
```

Установка настройки pagination в значение false означает, что мы хотим отключить возможности разбивки на страницы, так как пока что интерфейс перехода между страницами под списком, если вы посмотрите на него, когда мы всё доделаем, выглядит просто ужасно.

Собственно, данные, для того чтобы вложить их в DataProvider и отправить на рендеринг, находятся следующим образом:

```
private function getRecordsByPhoneNumber($number)
{
    $phone_record = PhoneRecord::findOne(['number' => $number]);
    if (!$phone_record)
        return [];

    $customer_record = CustomerRecord::findOne($phone_record->customer_id);
    if (!$customer_record)
        return [];

    return [$this->makeCustomer($customer_record, $phone_record)];
}
```

Здесь мы впервые встречаемся с методами запросов класса ActiveRecord. Они так многочисленны и API в целом такое обширное, что, возможно, будет лучше, если вы обратитесь к официальной документации вот сюда: http://www.yiiframework.com/doc-2.0/guide-db-active-record.html

Построить пользовательский интерфейс списка клиентов имеет смысл на основе виджета \yii\widgets\ListView. Вот как его нужно настроить в нашем случае:

```
echo \yii\widgets\ListView::widget(
    [
        'options' => [
            'class' => 'list-view',
```

```
            'id' => 'search_results'
        ],
        'itemView' => '_customer',
        'dataProvider' => $records
    ]
);
```

Это точное содержимое файла `views/customers/index.php`.

Нам нужно установить HTML-атрибут `id` для виджета, потому что в нашем приёмочном тесте мы ожидаем, что результаты запроса находятся в элементе `#search_results`.

Элемент `itemView` в настройках виджета содержит название отдельного файла представления, который будет использован для того, чтобы, собственно, отрисовывать каждую отдельную модель из предоставленного провайдера данных. Его имеет смысл определить в терминах виджета `\yii\widgets\DetailView`. Следующий код представляет собой реализацию файла `views/customers/_customer.php`:

```
echo \yii\widgets\DetailView::widget(
    [
        'model' => $model,
        'attributes' => [
            ['attribute' => 'name'],
            ['attribute' => 'birth_date', 'value' => $model->birth_date-
>format('Y-m-d')],
            'notes:text',
            ['label' => 'Phone Number', 'attribute' => 'phones.0.number']
        ]
    ]);
```

Так как у нас не «плоская» активная запись, основанная на единственной таблице в качестве модели, а составной агрегат данных из предметной области, настроить поля для отображения, как вы видите, приходится довольно изощрённым образом. Однако мы можем даже ссылаться на атрибуты-массивы в этом виджете, что, если честно, просто поразительно.

Пользовательский интерфейс запроса к БД

Последняя деталь пользовательского интерфейса смехотворно проста:

```
    public function actionQuery()
    {
        return $this->render('query');
    }
```

Мы просто покажем вручную написанную HTML-форму на этой странице:

```
<?php
use yii\helpers\Html;

echo Html::beginForm(['/customers'], 'get');
echo Html::label('Phone number to search:', 'phone_number');
echo Html::textInput('phone_number');
echo Html::submitButton('Search');
echo Html::endForm();
```

Надеемся, что этот код достаточно прямолинеен, чтобы его можно было понять без каких-либо объяснений. В целом вспомогательный класс `Html` крайне облегчает генерацию HTML-кода, предоставляя человекопонятные названия методов. Как вы, возможно, заметили, мы не написали ни строчки кода HTML напрямую ни в одном из файлов представлений, за исключением файла шаблона, где это было явно проще.

> Не слишком обольщайтесь насчёт этого вспомогательного класса. Используя его методы для генерации HTML-кода, в особенности его самые общие методы `openTag()` и `closeTag()`, которые мы благополучно пропустили, вы теряете самую важную особенность HTML: возможность вложения элементов друг в друга. Метод `tag()` позволяет имитировать это до некоторой степени, но в сложных случаях вам всё равно придётся туго. К тому же ваш HTML-верстальщик точно не обрадуется необходимости учить ещё и список методов класса `Html`, чтобы иметь возможность читать и модифицировать ваш код. Даже если верстальщик – это вы сами.

Использование приложения

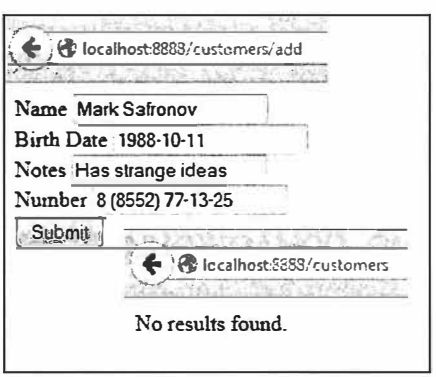
Теперь мы можем вернуться обратно к нашему приёмочному тесту. Мы сделали в нём допущение, что мы можем просто использовать относительные URL наподобие `/customers/add`, но, как мы узнали в разделе «*Вводный курс маршрутизации*», чтобы сделать это, нам нужно произвести некоторые настройки веб-сервера, чтобы он понимал запросы без подстроки «`index.php`». Допустим для простоты, что вы это уже сделали.

Вот что вы получите, когда вычистите вашу машину для развёртывания, пересоздадите её, развернёте на неё приложение и запустите приёмочные тесты со своей рабочей станции, обращаясь к нему:

```
[506]------------------------------------------------------------------------
hijarian:hijaria 17:02:33 jobs: 0 (~/projects/crmapp)
0_0 [master*] $ vendor/bin/codecept run acceptance
Codeception PHP Testing Framework v1.9-dev
Powered by PHPUnit 3.7.29-4-g641cd68 by Sebastian Bergmann.

Acceptance Tests (2) ---------------------------------------------------------
------------
Trying to query the customer info using his phone number (QueryCustomerByPhoneNumberCept.php)
       Ok
Trying to see that landing page is up (SmokeTestCept.php)
       Ok
------------------------------------------------------------------------------
------------

Time: 341 ms, Memory: 14.25Mb

OK (2 tests, 4 assertions)
```

Однако давайте пройдёмся по интерфейсу вручную.

Для конкретности представим, что развёрнутое приложение доступно по адресу `http://localhost:8888/`. Тогда после чистой установки мы открываем пользовательский интерфейс добавления нового клиента по адресу http://localhost:8888/customers/add и заполняем как-нибудь поля ввода. *Запомните введённый вами номер телефона.* Затем щёлкаем по кнопке **Submit** и попадаем в интерфейс списка клиентов, который говорит нам, что результатов найдено не было. Посмотрите на следующий снимок экрана:

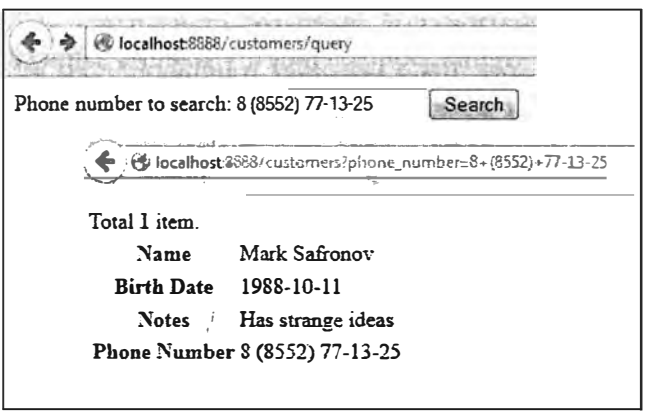


Ясно, что здесь не хватает функциональности: когда мы в пользовательском интерфейсе списка пользователей ожидаем увидеть всех пользователей, записанных в БД. А сейчас этот интерфейс выглядит скорее как «пользовательский интерфейс результатов запроса».

На самом деле проблема решается достаточно тривиальным дополнением в методе `CustomersController.actionIndex()`. Чтобы сделать это дополнение,

вам необходимо знать, что мы можем делать не только `CustomerRecord::findOne()`, но и `CustomerRecord::findAll()` (оба этих метода – методы класса `ActiveRecord`, почитайте о них в документации). Зная это, вы можете самостоятельно написать приёмочный тест на нужную нам функциональность и реализовать её. Это прекрасная возможность на простом, уже понятном примере испытать свои силы в разработке с использованием Yii 2.

Вручную перейдите по адресу http://localhost:8888/customers/query, и вы окажетесь в пользовательском интерфейсе запроса к БД. Вставьте запомненный номер телефона в поле ввода и нажмите **Search**. Вот соответствующий снимок экрана:



Вы должны оказаться в пользовательском интерфейсе списка пользователей, который вам сообщит, что только одна запись найдена, и покажет её детали. На самом деле мы точно знаем из исходного кода, что даже если есть несколько клиентов с одним и тем же номером телефона, система всё равно вернёт нам только одну запись. Это ещё один пример функциональности, которой здесь не хватает. Не говоря уже о том, что мы допускаем только один номер телефона на клиента.

Давайте на этом закончим наш сеанс и уберёмся отсюда, чтобы подвести итоги этой главы.

Итоги

Мы рассмотрели очень многое, реализовывая всего одну функциональную возможность. Не стоит огорчаться по этому поводу, потому что в действительности был выполнен огромный объём подготовительной работы, который больше не придётся выполнять ещё раз.

Сама функциональность заняла достаточно малую часть всего потраченного времени. Мы не стали возиться с внешним видом веб-приложения, сконцентрировавшись только на функциональности. Это будет отложено до следующей главы. Был использован полный стек технологий Yii, от активных записей и провайдеров данных внизу до контроллеров, виджетов и форм наверху.

Ожидалось, что эта глава будет примером того, что именно используется во время разработки с применением Yii. Мы опустили важную часть процесса рендеринга конечного результата: ассеты (assets), – но это материал для будущих глав. Далее мы посмотрим, что ещё есть в рукаве Yii для того, чтобы уменьшить усилия по кодированию для разработчика: автоматический генератор стандартного CRUD-кода (кода для вставки, чтения, обновления и удаления записей в хранилище данных).

Вкратце: вам необходимо знать множество концепций, прежде чем разрабатывать приложение, используя Yii.
Мы надеемся, что после этой сессии разработки у вас появилось ясное представление о них.

Глава 3

Автоматическая генерация кода

Если всё, что нужно какой-либо части вашего приложения, – это интерфейс для манипуляции с данными в БД, то в Yii уже включено всё, что нужно для поддержки такого решения. Используя активные записи, один к одному соответствующие таблицам в базе данных, вы можете очень кратко описывать в вашем контроллере стандартные операции создания, чтения, обновления и удаления записей в этих таблицах.

К сожалению, когда вы манипулируете множеством схожих таблиц, у вас получается повторяющийся код, который ещё и очень скучно писать. Чтобы преодолеть это препятствие, Yii предоставляет инструмент под названием Gii, который автоматизирует генерацию стандартного кода за вас.

В этой главе мы посмотрим, как Gii может вам помочь при разработке приложений, основанных на Yii.

Определение модели данных для работы

Мы продолжим пример, начатый в *главе 2 «Создание приложения с Yii 2»*. Представим, что нам нужно управлять списком «услуг» в нашей базе данных. Это будет простая таблица из двух колонок, описывающая самые важные характеристики предоставляемых нами услуг:

- название;
- оплата в час.

Мы создадим эту таблицу при помощи механики «миграций», которую рассмотрели в предыдущей главе.

- Сначала мы создаём новую миграцию, выполнив следующую команду в командной строке:

```
$ ./yii migrate/create init_services_table
```

- Затем мы пишем следующие методы `up` и `down`:

```
public function up()
{
   $this->createTable(
      'service',
      [
         'id' => 'pk',
         'name' => 'string unique',
         'hourly_rate' => 'integer',
       ]
   );
}

public function down()
{
   $this->dropTable('service');
}
```

- Наконец, мы запускаем созданную миграцию:

```
$ ./yii migrate
```

Теперь у нас есть таблица в БД. С этого момента наша стратегия будет выглядеть следующим образом:

1. Создать класс модели для таблицы `service`, чтобы настроить ORM к ней.
2. Создать CRUD-интерфейс.

Использование Gii

Перед тем как мы сможем использовать Gii, нам нужно его установить.

Установка Gii в приложение

Выполните следующую команду, чтобы скачать файлы, необходимые для использования Gii:

```
$ php composer.phar require --prefer-dist "yiisoft/yii2-gii:*"
```

Аргумент командной строки `"--prefer-dist"` был использован для того, чтобы мы получили только файлы, необходимые для *использования* Gii, а не всё, необходимое для его разработки.

Теперь нам нужно подсоединить Gii к нашему приложению.

Так как мы собираемся использовать что-то, установленное утилитой Composer в нашем приложении, нам нужен её автозагрузчик. Он представлен файлом под названием `autoload.php` в подкаталоге `vendor`. Вызов соответствующего метода `require()` нужно разместить в нашей точке входа, в файле `web/index.php`:

```
<?php
define('YII_DEBUG', true);

require(__DIR__ . '/../vendor/autoload.php');
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

ini_set('display_errors', true);

$config = require(__DIR__ . '/../config/web.php');

(new yii\web\Application($config))->run();
```

Строчка, которую нужно вставить, выделена жирным.

Затем нам нужно добавить в наше приложение пользовательский интерфейс Gii. Это делается объявлением модуля `gii` в настройках приложения в файле `config/web.php`:

```
    'modules' => [
        'gii' => [
            'class' => 'yii\gii\Module',
            'allowedIPs' => ['*']
        ]
],
```

Этот раздел `modules` должен быть на верхнем уровне массива настроек. Раздел `allowedIPs` нужен, если вы работаете с приложением, которое развёрнуто удалённо (как от вас и ожидается). По умолчанию Gii разрешает доступ только с локальной машины (IP-адреса `127.0.0.1` и `::1`).

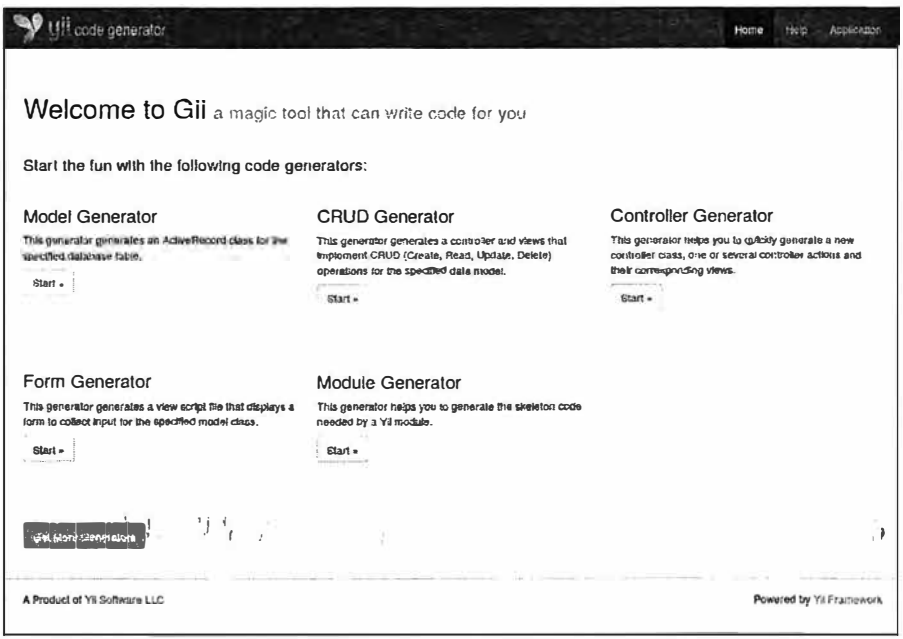
Наконец, вам нужно добавить одну весьма специфическую связь со внутренностями Yii 2 в конфигурацию вашего приложения. Вставьте следующую строчку в конце дерева конфигурации в том же файле `config/web.php`:

```
'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php')
```

Этот раздел `extensions` должен быть также расположен на верхнем уровне. Возможно, в вашем случае путь будет другим, но он должен в конечном счете указывать на файл `extensions.php` в подкаталоге

yiisoft среди каталогов, установленных утилитой Composer. Мы рассмотрим настоящий смысл этого файла в *главе 9 «Создаём расширение»*.

Сделав всё это, мы можем открыть маршрут /gii в нашем приложении и наконец-то увидеть интерфейс Gii, как показано на следующем снимке экрана:



Создаём код для класса модели

Выберем элемент **Model** из главного меню Gii и окажемся в разделе **Model Generator**.

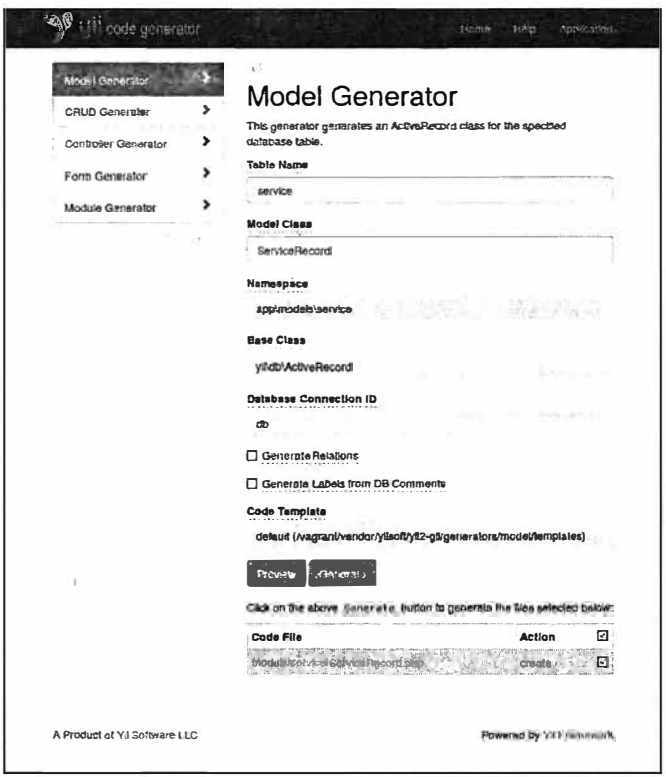
Insert Image: 1885OS_03_02.png

Назначение этого генератора – на основе таблицы в БД сделать класс активной записи, настроенный таким образом, чтобы поддерживать ORM с этой таблицей. Заметно, что поля ввода в интерфейсе генератора довольно просты для понимания, особенно со всплывающими подсказками.

Нам нужно заполнить поля ввода так, как показано в следующей таблице, и нажать кнопку **Preview**.

Название поля ввода	Значение для ввода (в точности)
Table Name (Название таблицы)	`service`
Model Class (Класс модели)	`ServiceRecord`
Namespace (Пространство имён)	`app\models\service`

После этого вы должны получить страницу, похожую на следующий снимок экрана:



Мы назвали класс модели «`ServiceRecord`», чтобы продолжить следовать соглашению, установленному в предыдущей главе. Чтобы сразу прояснить ситуацию: это не стандартное соглашение о наименовании (и Yii 2 не заставляет вас следовать никакому соглашению). Суффикс `Record` добавлен, чтобы явным образом показать, что рассматриваемый класс – наследник класса `ActiveRecord` и не имеет ничего общего с моделью предметной области в нашем приложении.

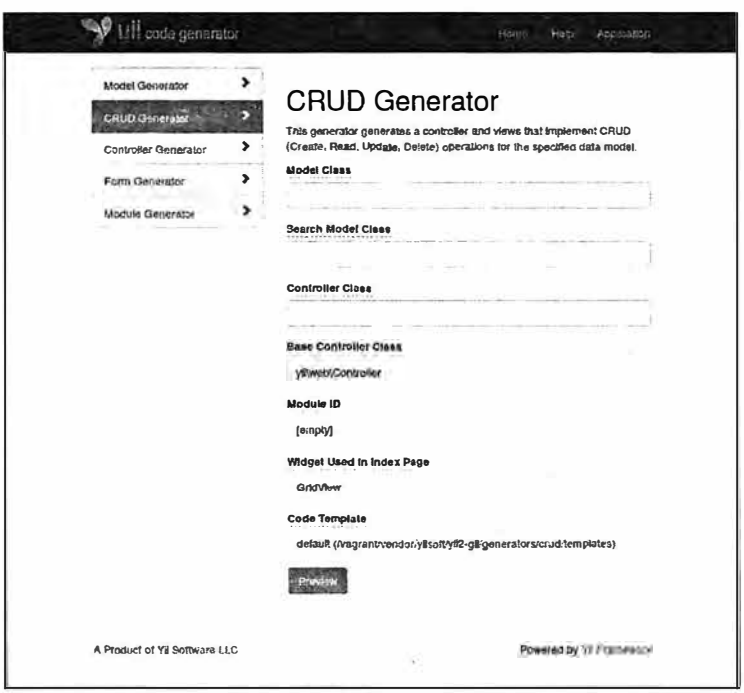
Этот генератор создаёт только наследников класса `ActiveRecord`. Мы объявили желаемое пространство имён по этой же причине. Пространство имён `app\models\service` отображается на подкаталог `models/service` в нашем проекте.

Как только вы нажмёте **Preview**, под интерфейсом появится небольшая область с перечислением файлов, которые будут созданы, и появится кнопка **Generate**. Это стандартное двухшаговое поведение Gii, которое позволяет вам в точности увидеть, что и где будет сгенерировано.

Нажмите **Generate**, и на этом мы закончим. У нас есть определение класса `ServiceRecord` в файле `models/service/ServiceRecord.php`.

Создаём CRUD

Теперь давайте из главного меню откроем раздел **CRUD Generator**. Вы увидите следующую страницу:



Назначение этого генератора – создать контроллер на основе указанной вами модели. Контроллер, который будет создан, будет иметь пять уже подготовленных действий:

- **index**: это действие перечисляет все модели, записанные в БД;
- **view**: это действие показывает нам детальное описание одной модели;
- **create**: это действие позволяет нам записывать новые модели в БД;
- **update**: это действие позволяет нам изменять атрибуты одной модели и обновлять их в БД;
- **delete**: это действие позволяет нам удалить запись об определённой модели из БД.

На действии `index` контроллер отрендерит нам не только список записей, но и функциональность поиска. Для её поддержки Gii сгенерирует дополнительный класс, который называется «модель поиска» (Search Model), которая инкапсулирует механику поиска моделей по значениям их атрибутов.

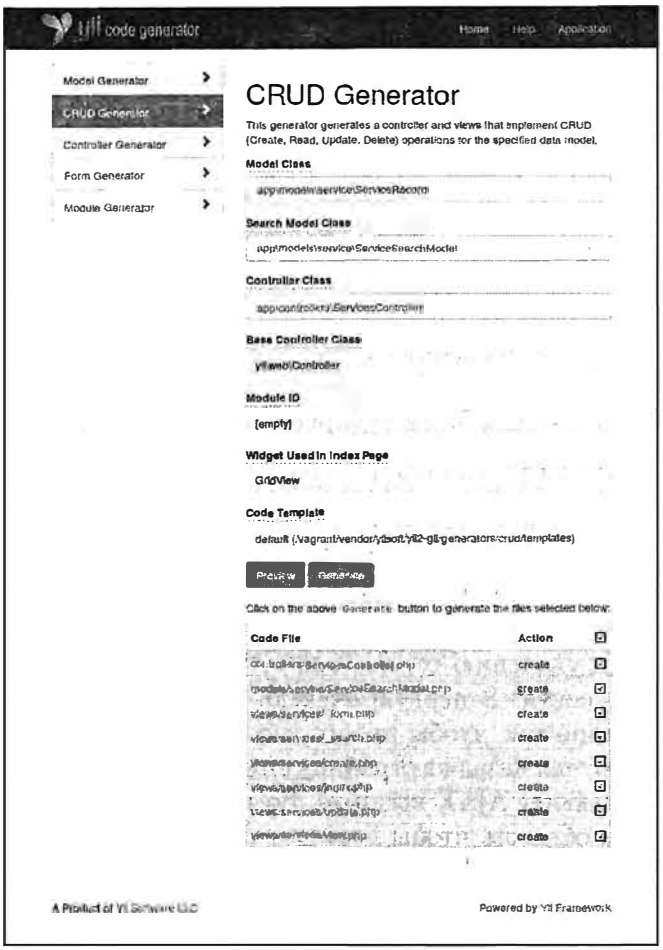
Действия по созданию и обновлению настолько похожи по функциональности, что они даже используют один и тот же сценарий для генерации формы ввода значений атрибутов модели. Единственное различие между ними в том, что при создании контроллер добавит новую запись в БД, а при обновлении – изменит существующую.

Действие по удалению отличается от других из-за своей потенциальной опасности. В первую очередь оно экстремально простое и спроектировано так, чтобы просто сделать свою работу и перенаправить посетителя назад на страницу списка записей. Это делает его идеальной целью для AJAX-вызовов. Во-вторых, Gii генерирует контроллер таким образом, чтобы маршрут **delete** был доступен только через POST-запрос, что фактически делает AJAX единственным способом активировать это действие.

Давайте, наконец, сделаем CRUD. Заполните поля так, как показано в следующей таблице:

Название поля	Значение поля
`Model Class`	`app\models\service\ServiceRecord`
`Search Model Class`	`app\models\service\ServiceSearchModel`
`Controller Class`	`app\controllers\ServicesController`

В поле **Model Class** находится имя существующего класса, который мы только что создавали. В остальных полях – имена, которые мы придумали, чтобы соответствовать структуре нашего проекта Yii. Когда вы нажмёте на кнопку **Preview**, вы должны увидеть результат, такой, как на следующем снимке экрана:



Обратите внимание на то, сколько кода создаёт этот генератор. Мы настоятельно рекомендуем прочитать его. Вы можете использовать более-менее функциональный IDE, с возможностью «перехода на определение», чтобы легче перемещаться между сгенерированным кодом и базовыми классами во фреймворке Yii 2. Мы не будем подробно обсуждать стандартный код в приложениях Yii, или эта книга превратится в хрестоматию исходного кода и, возможно, втрое вырастет в размере.

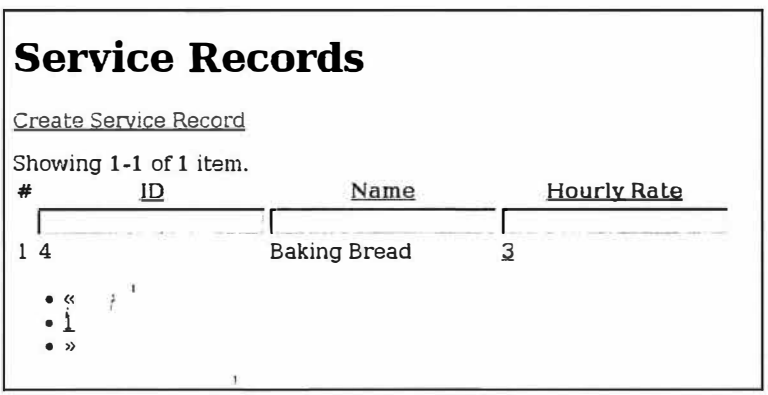
Нажмите на **Generate**, и мы обратимся к некоторым завершающим штрихам, которые осталось сделать.

Завершающие штрихи

Нам нужно совершить кое-какие изменения в нашем пользовательском интерфейсе, чтобы он выглядел корректно.

Создаём новый шаблон для поддержки созданных Gii страниц

Вот как выглядит наш интерфейс, при условии что одна запись уже сохранена в таблице `services`:



Учитывая наш спартанский внешний вид из *главы 2 «Создаём приложение с Yii 2»*, вы, возможно, и так особо многого не ожидали от сгенерированного Gii интерфейса, но здесь явно есть серьёзные проблемы с вёрсткой. Проблема в том, что мы шагнули слишком далеко, так как до этого момента не обращали внимания на внешний вид приложения. Мы будем обсуждать систему отрисовки в Yii 2 в следующей главе, однако сейчас мы можем одним глазком взглянуть в будущее и приготовить самый минимум кода представления, который включит нам стандартный дизайн проекта, основанного на Yii 2. Вы его и так уже видели в базовом шаблоне приложения.

Единственной нашей проблемой является на самом деле слишком ограниченный файл шаблона. Давайте вспомним, как выглядит наш файл: `views/layouts/main.php`:

```
<!DOCTYPE html>
<html>
<head>
```

```
        <title>CRM</title>
    </head>
    <body>
        <?= $content; ?>
    </body>
</html>
```

Это не то, чего Gii от нас ждёт. Не вдаваясь в излишние подробности, которые будут даны в следующей главе, мы просто заменим этот шаблон на то, что используется в базовом шаблоне приложения:

```
<?php
use yii\helpers\Html;

\yii\bootstrap\BootstrapAsset::register($this);
\yii\web\YiiAsset::register($this);
?>
<?php $this->beginPage() ?>
    <!DOCTYPE html>
    <html lang="<?= Yii::$app->language ?>">
    <head>
        <meta charset="<?= Yii::$app->charset ?>"/>
        <title><?= Html::encode($this->title) ?></title>
        <?php $this->head() ?>
        <?= Html::csrfMetaTags() ?>
    </head>
    <body>
    <?php $this->beginBody() ?>
    <div class="container">
        <?= $content ?>
        <footer class="footer"><?= Yii::powered();?></footer>
    </div>
    <?php $this->endBody() ?>
    </body>
    </html>
<?php $this->endPage() ?>
```

Кроме разных полезных вызовов функций вроде вставки атрибута, указывающего язык для данного HTML, метатега с кодировкой и HTML-кодирования заголовка страницы, обратите внимание на выделенные строчки. Они являются фреймворком механизма отрисовки в Yii 2. Мы изучим его в следующей главе.

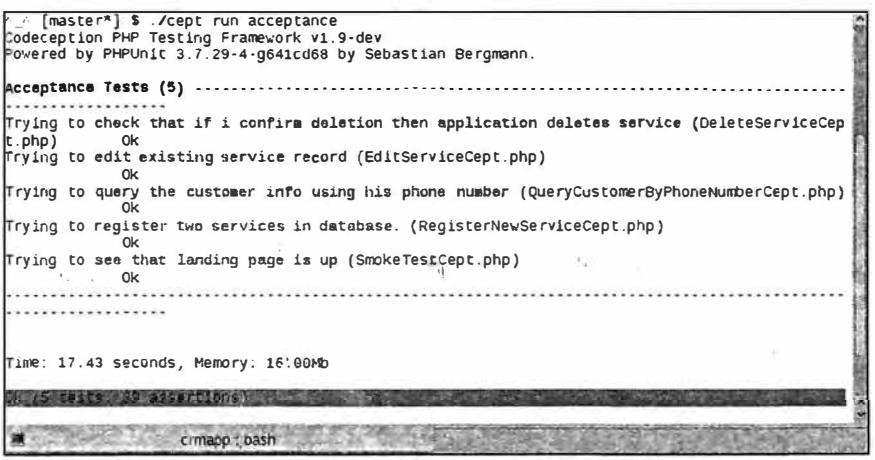
Обзор созданного пользовательского интерфейса

Давайте посмотрим на то, как теперь выглядит этот интерфейс.

Внутри пакета кода, прилагающегося к этой книге, вы найдёте приёмочные тесты на интерфейс, который мы сделали в этой главе. Они не были включены в текст для краткости. Выполните следующую команду:

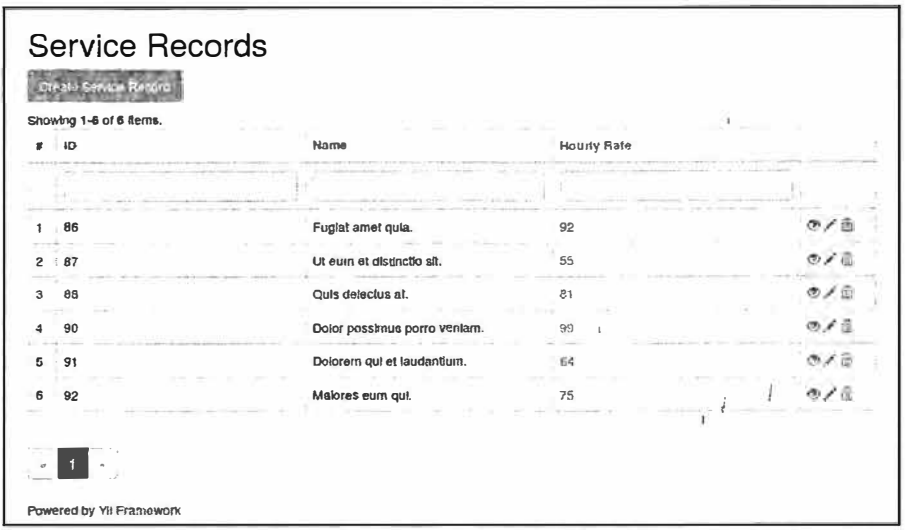
```
$ ./cept run acceptance
```

Эта команда должна вывести что-то, похожее на следующий снимок экрана:

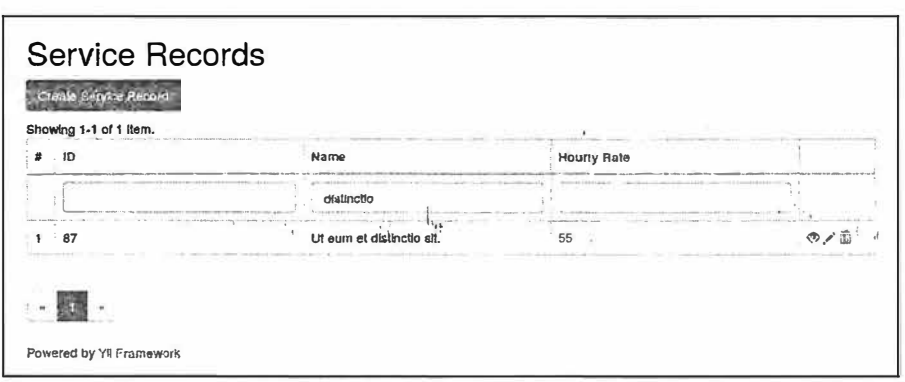


Заметьте, что мы не написали ни строчки кода вручную, чтобы успешно пройти эти тесты (за исключением шаблона). Также, при условии что все возможные приготовления были сделаны, в следующий раз, когда нам понадобится такой же пользовательский интерфейс, нам просто нужно будет заполнить шесть полей ввода и шесть раз щёлкнуть по кнопкам, и он у нас будет.

Перейдя по маршруту `/services`, мы получим приятного вида таблицу, до краёв набитую функциональностью:



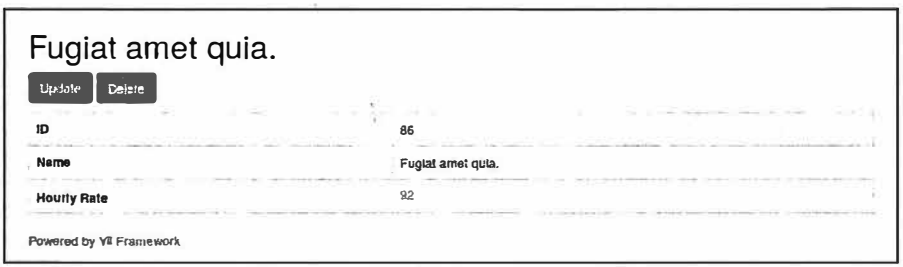
Каждая запись представлена строкой в таблице. Этот табличный пользовательский интерфейс имеет разбивку по страницам, 20 строчек на страницу. Это может стать проблемой, если вы запускаете приёмочные тесты много раз без очистки БД между запусками. Также в верхней части таблицы есть поля ввода для фильтрации таблицы согласно введённым значениям, как показано на следующем снимке экрана:



Наконец, последняя колонка в каждой строчке содержит три кнопки-иконки: одна с глазом, одна с карандашом и одна с мусорным ба-

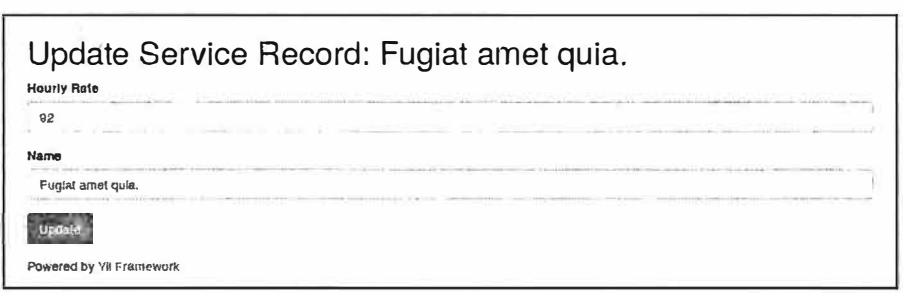
ком. Они соответствуют действиям «просмотра», «редактирования» и «удаления».

Действие «просмотра» отображает список полей и их значений для выбранной записи:



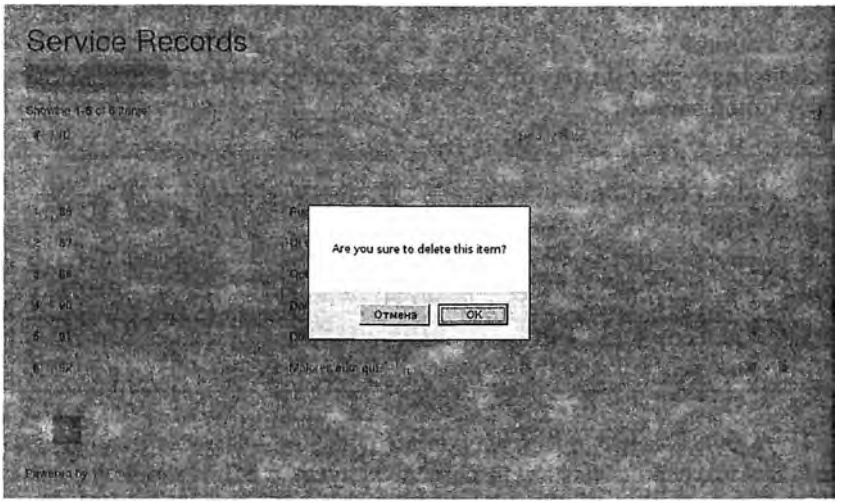
Путь будет выглядеть следующим образом: `/services/view?id=:id`, где `:id` – это первичный ключ записи «услуги» в БД. На этой же странице вы так же, как и на странице списка, увидите кнопки **Update** («Редактировать») и **Delete** («Удалить»).

Если вы щёлкнете по кнопке **Update** из интерфейса списка или из интерфейса детального просмотра, то получите веб-форму для редактирования полей выбранной записи.



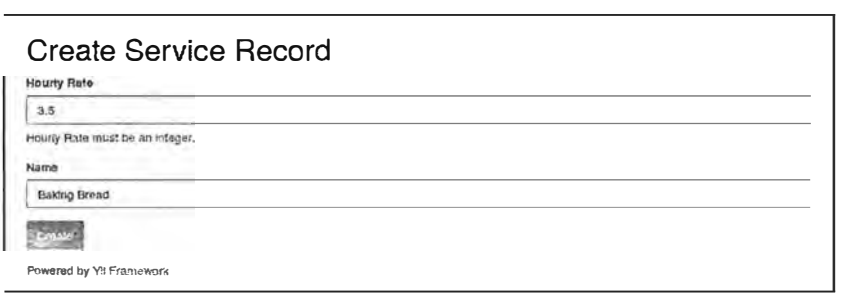
Маршрутом будет `/services/update?id=:id` с теми же правилами, что и маршрут для просмотра. Эта форма полностью идентична форме, где мы создаём новую запись.

Если вы щёлкнете по кнопке **Delete**, что на странице списка, что на странице детального просмотра, то получите запрос на подтверждение действия:



Если вы согласитесь на удаление, таблица будет перезаполнена посредством AJAX. Если у вас отключён Javascript, вы вообще не получите возможности удалять записи, потому что маршрут `/services/delete?id=:id`, как было сказано ранее, сконфигурирован таким образом, что принимает только POST-запросы.

Если вы щёлкнете на большую зелёную кнопку **Create Service Record**, то получите форму для создания новой записи о модели `ServiceRecord`. Маршрут будет выглядеть так: `/services/create`. Мы уже видели эту форму на странице редактирования записи. Стоит отметить напоследок, что эта форма включает в себя валидацию данных на стороне клиента и не даст отправить данные на сервер, если поля заполнены неверно, как показано на следующем снимке экрана:



Весь этот пользовательский интерфейс основан на фреймворке пользовательского интерфейса Twitter Bootstrap плюс немного кода JavaScript, специфичного для Yii.

«За» и «против» автоматической генерации классов

Как и в случае с любым другим генератором исходного кода, Gii не следует считать автоматическим программистом, делающим всю работу за вас.

В первую очередь Gii скрывает механику CRUD. Поэтому, когда вы автоматически генерируете действия контроллеров, для вас будет сложно понять, как они работают, особенно если вы не имеете опыта работы с фреймворком. Это общая проблема всех автоматических генераторов кода; они полезны только для людей, которые уже знают, какой код им нужно написать, и просто не хотят писать его самостоятельно.

Во-вторых, возможно, что вам нужен набор полностью стандартизированных страниц CRUD-интерфейса для множества моделей. В этом случае повторение кода, вызванное Gii в сгенерированных контроллерах, станет совершенно ненужным. В *главе 11 «Таблица»* мы покажем, как три контроллера, созданных Gii, могут быть сокращены до одного базового класса Controller и трёх подклассов, урезанных до буквально *двух строчек* ручного кода на каждый.

Подведем итоги: автоматическая генерация пользовательского интерфейса в Gii особенно полезна только в том случае, когда вам нужна какая-то обобщённая отправная точка для вашей реальной работы и *ожидается, что действия просмотра, списка, редактирования и удаления будут изменены.* В остальных случаях, возможно, будет проще писать контроллеры «с нуля», тем более что это и так несложно.

Обратите внимание, однако, что Gii не ограничен лишь созданием CRUD-интерфейсов. Его генератор моделей, например, практически бесценен, потому что вы определённо не хотите писать шаблонный код для ваших активных записей, и Gii может вывести много информации из схем таблиц БД за вас. Также в него включено несколько других генераторов, включая генератор расширений; впрочем, мы не будем использовать его в *главе 9 «Создаём расширение»*, потому что он помешает нам понять, как Yii 2 работает изнутри.

Итоги

В этой главе мы узнали, как интегрировать Gii в существующий проект. Также по пути мы рассмотрели минимальный объём кода представления, который Yii ожидает от нас, если мы всё делаем «с нуля».

В этой и предыдущей главах мы использовали довольно много кода, относящегося к «представлениям» в идеологии MVC, и необходимость шаблонов пока что является самой загадочной частью. В следующей главе мы поговорим о том, как Yii 2 на самом деле производит рендеринг представления.

Глава 4

Рендерер

Ближе к концу предыдущей главы нам пришлось прыгнуть выше головы и покопаться в коде шаблона. Эта глава фактически будет объяснением того, что мы тогда сделали.

Несмотря на название главы, в Yii нет отдельного объекта «рендерера». Так как Yii основан на парадигме MVC, в нём используется целый набор процессов, совершающих отрисовку. Эти процессы распределены по всей базе кода.

Анатомия отрисовки в Yii

Когда запрос посетителя веб-приложения обрабатывается, ваши данные проходят через несколько этапов, прежде чем их отправляют обратно в браузер посетителя:

1. Запускается действие контроллера (controller action, см. главу 2). Оно будет использовать метод `render()`, чтобы обработать некоторый PHP-скрипт (если нужно, передав в него некоторые данные) и получить от него данные, которые нужно послать браузеру. Заметьте, что это действие добровольное, а не обязательное. У вас могут быть действия контроллеров, которые вообще не вызывают `render()`.
2. Метод `render()` компонента приложения `View` («Представление») вызывается, и в него передаются аргументы `$view` и `$params`.
3. Компонент `View`, согласно псевдоимени, переданному через аргумент `$view`, определяет, какой именно файл представления нужно использовать.
4. Компонент `View` проверяет, есть ли какие-либо отрисовщики представления (View Renderers), ассоциированные с расширением файла, который он только что нашёл.

   Если такой есть, вызывается его метод `render()`. Ему передаются путь к файлу представления, экземпляр компонента `View` и параметры, переданные изначально через аргумент `$params`.

   - Если такого нет, то файл представления обрабатывается обычным встроенным в PHP механизмом `require()`.
5. Если результат рендеринга – не экземпляр класса `yii\web\Response`, такой экземпляр создаётся, и результат рендеринга передаётся в него как значение свойства `data`.
6. Экземпляр `Response` смотрит на значение своего атрибута `format` и проверяет, соответствуют ли этому значению какие-нибудь классы – реализации интерфейса `ResponseFormatterInterface`.
   - Если такой «компоновщик отклика» (прямой перевод термина Response Formatter) действительно есть, вызывается его метод `format()`, и весь экземпляр `Response`, вызвавший его, передаётся в этот метод, так что `format()` сможет по своему усмотрению перенастроить заголовки и содержимое `Response`.
   - Если `Response` в состоянии обработать переданный `format` самостоятельно, он так и делает, модифицируя свои заголовки и содержимое.
7. Наконец, HTTP-заголовки, сохранённые в `Response`, отправляются клиенту, используя стандартную для PHP механику вызовов `header()`, а затем туда же отправляется содержимое.

Концепция отрисовщика представлений реализована в Yii 2 абстрактным классом под названием `\yii\base\ViewRenderer`, что совершенно неудивительно.

В самом простейшем случае, когда у вас нет никаких тем, преднастроенных отрисовщиков представлений или компоновщиков отклика и вы делаете следующее в вашем контроллере:

```
$this->render('index', ['dataProvider' => $dataProvider]);
```

тогда происходит следующее:

1. Проверяется, существует ли файл `index.php` в каталоге `<корневой_каталог>/<каталог_представлений>/<id_контроллера>/`
2. Внутри компонента `View` происходит `extract(['dataProvider' => $dataProvider])`. В результате этого значение, сохранённое по ключу `'dataProvider'`, становится доступным как переменная `$dataProvider` в текущей области видимости. Это делает её доступной в том числе в файле `index.php`.
3. Метод `require("<корневой_каталог>/<каталог_представлений>/<id_контроллера>/index.php")` выполняется, окружённый парой вызовов `ob_start()` ... `ob_get_clean()`, так что весь его вывод сохраняется как строка текста.

4. Отправляется HTTP-заголовок `Content-Type: text/html; charset=<кодировка приложения Yii>`.
5. Результат, полученный на шаге 3, отправляется клиенту дословно, без дальнейших изменений.

Подробности того, как `render()` делает своё дело, можно посмотреть в документации и в исходном коде метода `yii\base\Controller.render()`. Нам на самом деле интересны четыре простых вопроса:

1. Как сформулировать значение первого аргумента метода `render()` так, что Yii найдёт файл представления?
2. Как Yii определяет, какой шаблон использовать, и как мы можем указать его явно?
3. Можем ли мы использовать другие (кроме сценариев PHP) виды файлов представлений?
4. Можем ли мы отправлять результаты отрисовки клиенту каким-либо другим образом? В качестве самого очевидного примера можно вспомнить JSON.

Перед тем как мы поговорим об ответах на эти вопросы, будет полезно получить объяснение концепции «компонентов приложения» Yii.

Компоненты приложения

Взглянем на инициализацию приложения, которая происходит в файле точки входа `index.php`:

```
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
$config = require(__DIR__ . '/../config/web.php');
(new yii\web\Application($config))->run();
```

Согласно идеологии Yii, эти три строчки означают следующее:

1. Определить класс `Yii` и подключить, таким образом, автозагрузчик классов.
2. Создать экземпляр класса `yii\web\Application`.
3. Создать различные компоненты, объявленные в разделе `components` массива, содержащегося в `$config`, и присоединить их к экземпляру приложения из п. 2.
4. Произвести всю остальную обработку `$config` вроде настройки атрибутов самого приложения.
5. Присоединить заряженный экземпляр `yii\web\Application` к классу `Yii` в виде статической переменной `Yii::$app`.

На шаге 3 компоненты создаются согласно следующему правилу: каждый массив внутри раздела `components` конфигурации приложения

превращается в экземпляр класса, упомянутого в ключе class этого массива, а остальные пары ключ–значение определяют значения, которые должны быть присвоены свойствам этих объектов. Это правило рекурсивно. Все эти компоненты будут присоединены к приложению в виде свойств, чьи имена соответствуют ключам массива components.

Например, рассмотрим следующий фрагмент настройки из раздела components, идеально подходящий в качестве примера:

```
'log' => [
   'traceLevel' => 3,
   'targets' => [
      [
         'class' => 'yii\log\FileTarget',
         'levels' => ['error', 'warning'],
      ],
   ],
],
```

Такая конфигурация означает, что при создании приложения будет выполнен следующий код:

```
$log = new yii\log\Dispatcher;
$log->traceLevel = 3;
$fileTarget = new yii\log\FileTarget;
$fileTarget->levels = ['error', 'warning'];
$log->targets = [$fileTarget];
Yii::$app->log = $log;
```

Есть некоторое количество компонентов, которые присоединяются к приложению по умолчанию. Их идентификаторы, по которым они должны упоминаться в конфигурации, известны заранее, так что Yii 2 знает, какие классы использовать для этих компонентов, даже если вы их не укажете.

Фактически каждый массив, у которого есть корректное имя класса по ключу class, будет расцениваться как настройки начальных значений экземпляра этого класса.

Эти правила инкапсулированы в вызове Yii::createObject(). Вы всегда можете проконсультироваться с исходным кодом, если желаете захватывающее путешествие в реализацию контейнеров внедрения зависимостей (**Dependency Injection Containers**), которое мы пропустим в этой книге, как из-за того, что детали реализации в данном случае не имеют для нас значения, так и для того, чтобы в целом сохранить наш рассудок.

Как только компонент `log` будет создан, он присоединяется к приложению в виде свойства под названием `log`. В результате вы сможете добраться до этого свежесозданного экземпляра класса `yii\log\Dispatcher` следующим образом:

```
Yii::$app->log
```

В данном случае этот вызов не имеет особой пользы, так как компонент журналирования обычно используется опосредованно, через вызовы методов `Yii::error()`, `Yii::warning()`, `Yii::info()` и `Yii::trace()` (вы, возможно, уже поняли, что каждый из них делает, однако мы всё равно подробно рассмотрим журналирование в Yii 2 позднее).

Вот список компонентов, которые по умолчанию присоединяются как к экземплярам `yii\console\Application`, так и к экземплярам `yii\web\Application`, вне зависимости от того, укажете вы для них конфигурацию или нет:

Идентификатор компонента	Класс компонента
`log`	`yii\log\Dispatcher`
`formater`	`yii\base\Formatter`
`i18n`	`yii\i18n\I18N`
`mailer`	`yii\swiftmailer\Mailer`
`urlManager`	`yii\web\UrlManager`
`view`	`yii\web\View`
`assetManager`	`yii\web\AssetManager`
`security`	`yii\base\Security`

Следующие компоненты присоединяются только к консольным приложениям:

Идентификатор компонента	Класс компонента
`request`	`yii\console\Request`
`response`	`yii\console\Response`
`errorHandler`	`Yii\console\ErrorHandler`

Следующие компоненты присоединяются только к веб-приложениям:

Идентификатор компонента	Класс компонента
`request`	`yii\web\Request`
`response`	`yii\web\Response`
`session`	`yii\web\Session`
`user`	`yii\web\User`
`errorHandler`	`yii\web\ErrorHandler`

Заметьте, что как консольные, так и веб-приложения имеют компоненты `response`, `request` и `errorHandler`, но они на самом деле разных классов. Это сделано для того, чтобы использовать одни и те же идиомы в действиях контроллеров.

Если Yii 2 обновится тем или иным образом, вы всегда можете посмотреть актуальный список компонентов в определении метода `Application.coreComponents()`.

Самая важная особенность этой системы компонентов – то, что ничто не запрещает вам создать и зарегистрировать свой собственный компонент. Он будет неотличим от встроенных компонентов приложения.

Компонент представления

Компонент `View` («Представление»), ответственный за создание вывода, который отправится клиентам приложения, – это тоже один из компонентов, и он может быть настроен по ключу `components.view` в конфигурации приложения. Чтобы узнать полный список параметров, вы можете либо посмотреть в документацию класса `yii\web\View` (по следующему URL: http://www.yiiframework.com/doc-2.0/yii-base-view.html), либо просто посмотреть на его исходный код, так как эта документация всё равно автоматически создаётся из комментариев в исходном коде.

Будучи компонентом приложения, `View` позволяет вам делать следующее:

```
Yii::$app->view->render($viewAlias, $params);
```

Класс `Controller` содержит метод-обёртку, который немного абстрагирует нас от этого компонента:

```
$this->render($viewAlias, $params);
```

Однако контроллер не просто оборачивает вызов компонента представления. Он также рендерит специальный файл представления, называемый в терминологии Yii шаблоном (**layout**), затем рендерит то, что было запрошено изначально, и только в конце собирает итоговый вывод на основе шаблона и запрошенного представления. До того как мы начнём говорить про шаблоны, давайте сначала посмотрим, чем является аргумент `$viewAlias`.

Алгоритм поиска файлов представлений

Концепция модулей, в некотором смысле являющаяся ключевой для Yii в целом, несколько усложняет ситуацию. Мы обсудим моду-

ли позже, в *главе 7 «Модули»*, где углубимся в теоретические детали, дабы внести ясность в то, как Yii на самом деле устроен. До того момента давайте сделаем вид, что мы имеем дело с приложением без подмодулей.

Свойство basePath приложения, как вы помните из *главы 2 «Создание приложения Yii 2»*, должно иметь какое-то значение. Этот «базовый путь» (прямой перевод фразы «base path») есть абсолютный путь до корневого каталога проекта.

На самом деле у приложения есть ещё одно свойство, называемое viewPath. Оно указывает на каталог, в котором должны находиться все файлы представлений этого приложения. По умолчанию viewPath содержит относительный путь к каталогу под названием view. Будучи относительным, этот путь, очевидно, разрешается в подкаталог view под basePath приложения.

Свойство viewPath, в свою очередь, является основой для определения относительных путей, которые вы упоминаете в качестве первого аргумента вызова метода Controller.render($view, $params). К этому viewPath присоединяется идентификатор контроллера, у которого мы вызываем метод render().

Это всё. Допустим, все настройки приложения установлены в значения по умолчанию, и вы вызываете следующий метод actionIndex():

```
class CustomersController
{
   public function actionIndex()
   {
      return $this->render('index');
   }
}
```

Будет использован следующий файл представления:

```
Yii::$app->basePath . "/views/customers/index.php"
```

Выделенное слово customers в предыдущей строчке – это идентификатор контроллера, выведенный из названия класса CustomersController.

Yii 2 делает идентификаторы контроллеров достаточно простым способом. От имени класса контроллера отрезается суффикс Controller, и оставшаяся часть имени превращается из строки в «ВерхнемРегистре» в строку «разделённую-дефисами».

Ранее мы уже упоминали, что расширение, по умолчанию ожидаемое от файла представления, – это php.

Однако вы можете использовать не только полностью относительные пути. Если путь начинается с символа «@», он будет считаться **псевдонимом пути** Yii (**Path Alias**).

Псевдонимы пути были и в Yii 1.x, но для Yii 2 синтаксис серьёзно поменялся. Теперь раскрывается только первый токен. Это работает так, как показано в следующем фрагменте кода (нет необходимости выделять целый раздел только для псевдонимов пути):

```
Yii::setAlias("@token", "some/filesystem/path/to/application");
Yii::$app->view->render("@token/subfolder/view");
```

В результате Yii 2 будет искать в каталоге some/filesystem/path/to/application/subfolder/ файл представления под названием view.php.

Конечно же, вызов setAlias() обычно совершается гораздо раньше в процессе жизни приложения. Yii 2 настолько добр к нам, что определяет пять самых важных псевдонимов за нас:

- @webroot – это абсолютный путь к каталогу, содержащему сценарий точки входа index.php;
- @web – это относительный путь к каталогу, содержащему сценарий точки входа index.php относительно корневого каталога, определённого веб-сервером;
- @app указывает на каталог, установленный в настройке basePath экземпляра приложения (ожидается, что это абсолютный путь до корневого каталога приложения);
- @runtime и @vendor устанавливаются как подкаталоги runtime и vendor относительно @app.

Все эти псевдонимы раскрываются *без* косой черты на конце.

Для полной определённости, @webroot – это фактически dirname($_SERVER['SCRIPT_FILENAME']). С другой стороны, @web – это путь от $_SERVER['DOCUMENT_ROOT'] до сценария index.php (Yii при этом очень старается стандартизировать значение DOCUMENT_ROOT между различными платформами). В итоге в настройках приложения по умолчанию (например, в базовом шаблоне приложения) @webroot равен корневому каталогу, установленному в веб-сервере, а @web пуст.

Вы можете использовать псевдонимы не только при указании путей к файлам представлений. Почти все настройки встроенных компонентов Yii, которые принимают что-либо, напоминающее путь в файловой системе, принимают также и псевдонимы путей.

Кроме псевдонимов путей, вы также можете указывать «абсолютные» пути до файлов представлений. «Абсолютность» в данном случае определяется не файловой системой, а каталогом приложения. Если начать путь к файлу представления с символов **//** (двух левых косых чёрт), Yii начнёт искать его со значения настройки `viewPath` приложения.

Когда ваше приложение вырастет и вы разделите его на модули, то сможете детально прописывать пути до файлов представлений, начиная их с символа **/** (одинарной левой косой черты). В этом случае Yii начнёт искать от значения настройки `viewPath` текущего *модуля. Если приложение не разбито на модули, оно само работает как единственный модуль*, поэтому одинарная левая косая черта будет работать так же, как и две. Больше про модули – в *главе 7*. На самом деле полностью относительный путь к файлу представления тоже будет раскрыт не от `viewPath` приложения, а от `viewPath` текущего модуля.

В следующей таблице представлена краткая сводка различных видов спецификаций путей, доступных в Yii 2:

Спецификация	Смысл
`$filename`	`Yii::$app->viewPath . '/' . Yii::$app->controller->id . '/' . $filename`
`"@aliasname" . $filepath`	`Yii::getAlias("@aliasname") . '/' . $filepath`
`"//" . $filepath`	`Yii::$app->viewPath . '/' . $filepath`
`"/" . $filepath`	`Yii::$app->controller->module->viewPath . '/' . $filepath`

Во всех случаях, если вы опустите расширение файла в конце `$filename` или `$filepath`, по умолчанию будет подразумеваться `php`.

Алгоритм поиска файла шаблона

Ранее уже было сказано, что метод `Controller.render()` не только отрисовывает файл представления, который ему было сказано обработать, но и помещает результат внутрь другого файла представления, который называется **шаблоном**.

Реализация этого механизма на самом деле выглядит следующим образом:

```
public function render($view, $params = [])
{
    $output = $this->getView()->render($view, $params, $this);
    $layoutFile = $this->findLayoutFile($this->getView());
    if ($layoutFile !== false) {
```

```
            return $this->getView()->renderFile($layoutFile, ['content' =>
$output], $this);
        } else {
            return $output;
        }
    }
```

Этот код взят без изменений из исходников класса `yii\base\Controller`. Метод под названием `getView()` извлекает компонент `View` из текущего экземпляра приложения идентично вызову `Yii::$app->view`. Методы `View.render()` и `View.renderFile()` в нашем примере практически идентичны (`renderFile()` не занимается собственно поиском файла представления, подробно описанным в предыдущем разделе).

Из этого исходного кода можно сделать следующие выводы.

1. Шаблон – это просто ещё один файл представления, обрабатывается он по тем же правилам.
2. В шаблоне доступна переменная `$content` (и больше никаких переменных в него передать нельзя, за единственным исключением в виде `$this`). Это строка текста, содержащая в себе результат рендеринга того файла представления, с которого всё и началось.
3. Контроллеру нужно каким-то способом найти файл шаблона, и этот способ не зависит от пути до файла представления, который мы передаём в метод `render()`.

На самом деле у класса `Controller` есть свойство `layout`. В нём хранится псевдоним пути до файла шаблона, который должен использовать метод `render()`. Если по какой-то причине вы присвоите этому свойству значение `false`, система шаблонов будет полностью выключена, и контроллер отрендерит только запрошенный вами файл представления. Тех же результатов можно добиться, если напрямую вызвать метод `\yii\base\View.renderFile()`. Если свойству `layout` присвоено значение `null`, будет использовано значение свойства `layout` родительского модуля. Как уже было сказано в главе 2, модуль самого верхнего уровня, являющийся родительским для всех остальных, – это само приложение.

После того как значение свойства `layout` было получено и оно не равно ни `null`, ни `false` и не является пустым, применяются правила, перечисленные в следующей таблице.

Спецификация	Смысл
"@alias"	`Yii::getAlias("@alias")`
"/" + $filepath	`Yii::$app->layoutPath . '/' . $filepath`
$filepath	`$module->layoutPath . '/' . $filepath.`

Как видно, у каждого модуля есть свойство layoutPath, которое содержит путь до каталога, где хранятся все шаблоны этого модуля. По умолчанию layoutPath указывает на подкаталог под названием layout под каталогом, на который указывает свойство viewPath. Используя псевдонимы путей с символом @, вы можете указать на любой каталог в проекте. Начав значение свойства layout с символа /, вы укажете Yii искать шаблон под layoutPath самого приложения. Во всех остальных случаях, как отмечено в таблице символом $module, будет использовано свойство layoutPath модуля, чьё свойство layout мы использовали.

По умолчанию Controller.layout содержит null. Это означает, что Yii должен использовать свойство layout модуля, содержащего этот контроллер. Даже если вы используете систему модулей, по умолчанию всё равно каждый модуль создаётся со свойством layout, которому присвоено значение null. Это означает, что будет использовано значение свойства layout самого приложения, которое обычно настраивается в конфигурации приложения. Наконец, значение по умолчанию этого свойства – это строка main, что означает файл main.php в каталоге, на который указывает свойство layoutPath. Если явно указать расширение файла, то расширение php автоматически добавлено не будет.

Таким образом, если вы не меняете все значения по умолчанию, не используете модули и выполняете следующее:

```
class CustomersController
{
   public function actionIndex()
   {
      $this->render('index');
   }
}
```

тогда будет использован следующий файл шаблона:

```
Yii::$app->basePath . '/views/layouts/main.php'
```

Это поразительно простой механизм, для того чтобы сделать некоторые фрагменты HTML общими для множества (обычно для всех) страниц вашего веб-приложения.

Внутренности процесса отрисовки файла представления

Хоть мы и определённо не хотим далеко лезть в детали работы компонента View, Yii вынуждает нас сделать это из-за своих соглашений о структуре шаблонов. Мы посмотрим на вручную настроенные от-

рисовщики в следующем разделе. Пока что давайте вспомним, как организован типичный PHP-шаблон в Yii, такой, какой мы использовали в предыдущей главе. Мы использовали следующие методы компонента View:

Метод	Объяснение
`beginPage()`	Начинает буферизацию последующего содержимого. Всё после этого вызова вплоть до вызова `endPage()` подлежит постобработке. Этот метод также инициирует событие `View::EVENT BEGIN PAGE`
`head()`	Отмечает место, в которое нужно разместить элементы, регистрируемые динамически. Эти элементы включают в себя элементы `<meta>`, самодельные элементы `<link>` и CSS- и Javascript-файлы, зарегистрированные в позиции `View::POS_HEAD`
`beginBody()`	Отмечает место, в которое нужно разместить Javascript, зарегистрированный в позиции `View::POS_BEGIN`. Инициирует событие `View::EVENT BEGIN BODY`
`endBody()`	Отмечает место, в которое нужно разместить Javascript, зарегистрированный в позициях `View::POS_END`, `View::POS_READY` и `View::POS_LOAD`. Здесь же пакеты материалов регистрируют свои CSS- и Javascript-файлы. Инициирует событие `View::EVENT END BODY`
`endPage()`	Инициирует событие `View::EVENT_END_PAGE`. Заканчивает буферизацию. *На самом деле* вставляет ранее зарегистрированные элементы `meta`, `link`, `style` и `script` в конечную HTML-страницу

Вызов `View.endPage()` очищает всё, что было зарегистрировано в соответствующем компоненте `View`, так что если вы окончательно сошли с ума, то можете даже начать следующую последовательность отрисовки HTML.

Цель этой сложной структуры методов – в том, чтобы предоставить вам полный контроль над смыслом различных спецификаторов позиций `View::POS_*`. Скажем, подразумевается, что всё, что вы регистрируете в позиции `View::POS_END`, должно быть размещено сразу перед закрывающим тегом элемента `body`. Однако благодаря тому, что вы контролируете, где находится вызов метода `View.endBody()`, итоговые элементы `script` (и даже, Боже упаси, `style`) могут быть где угодно.

Ручная настройка отрисовщиков

Теперь вы знаете, вероятно, самую изощрённую часть соглашений Yii. Давайте сейчас поговорим про ручную настройку отрисовщиков (renderers).

Для отрисовки результата, который будет отправлен в браузер пользователя, Yii использует специальный класс объектов, представленных абстрактным классом `\yii\base\ViewRenderer`. Конкретная реализация отрисовщика, если она известна компоненту `View`, выбирается на основе расширения имени файла представления. Если компонент `View` не может найти отрисовщик для данного файла представления, он считает этот файл сценарием на языке PHP и выполняет следующий метод под названием `renderPhpFile`:

```
/**
 * @param string $_file_ the view file.
 * @param array $_params_ the parameters (name-value pairs) that
will be extracted and made available in the view file.
 * @return string the rendering result
 */
public function renderPhpFile($_file_, $_params_ = [])
{
    ob_start();
    ob_implicit_flush(false);
    extract($_params_, EXTR_OVERWRITE);
    require($_file_);
    return ob_get_clean();
}
```

Как можно видеть, он просто выполняет стандартный `require()` на файле представления, буферизуя вывод. Важным эффектом в данном случае является то, что вы можете делать всё, что угодно, в ваших файлах представления, что может быть опасно, если вы *на самом деле* будете делать всё, что угодно. В идеале представления, основанные на PHP, нужно рассматривать как написанные для некоего шаблонизатора, только лишь передавая им данные в виде ассоциативных массивов или атомарных данных.

В стандартной поставке Yii 2 нет никаких дополнительных преднастроенных отрисовщиков. Давайте подумаем, как мы можем использовать данную функциональность для своего удобства.

Изначальная цель разработчиков Yii, очевидно, заключалась в том, что вы напишете парсер для какой-то определённой системы шаблонов и затем получите возможность составлять файлы представлений не прямо на PHP, но в виде этих (предположительно более ограниченных и связанных с предметной областью) шаблонов. Задавая именам файлов особое расширение, вы укажете Yii использовать ваш самодельный парсер на них, что предположительно превратит шаблон в, скажем, HTML, который вы пошлёте клиентскому приложению.

Давайте реализуем здесь простое решение – воспользуемся синтаксисом **Markdown** (см. http://daringfireball.net/projects/markdown/syntax) для составления статичных страниц. Скажем, мы хотим набор страниц пользовательской документации, которые будут вручную написаны способным техническим писателем. Скорее всего, мы не хотим заставлять его/её писать прямо на HTML или же в MS Word, впоследствии самостоятельно и в муках преобразуя DOC в HTML. Markdown будет простейшим нейтральным решением для данной задачи.

Создать новый отрисовщик для обработки статических страниц в формате Markdown довольно просто. Условимся, что мы продолжаем работать с тем же самым примером CRM-приложения, что мы делали последние две главы.

Вначале объявим то, что мы хотим:

```
./cept generate:cept acceptance Documentation
```

Затем в только что созданном файле `tests/acceptance/DocumentationCept.php`:

```
$I = new AcceptanceTester\CRMUserSteps($scenario);
$I->wantTo('see whether user documentation is accessible');

$I->amOnPage('/site/docs');
$I->see('Документация', 'h1');
$I->seeLargeBodyOfText();
```

Метод `seeLargeBodyOfText()` будет объявлен в классе `CRMUserSteps` следующим образом:

```
public function seeLargeBodyOfText()
{
    $I = $this;
    $text = $I->grabTextFrom('p'); // naive selector
    $I->seeContentIsLong($text);
}
```

Мы считаем, что мы смотрим на страницу документации, если там есть заголовок «Документация» и длинный фрагмент текста под ним. Это, конечно же, довольно наивно, но мы не можем себе позволить сейчас написать ИИ, способный автоматически распознать, является данный текст страницей документации или нет.

Метод `seeContentIsLong()` мы разместим в классе `AcceptanceHelper` внутри файла `tests/_support/AcceptanceHelper.php`:

```
public function seeContentIsLong($content, $trigger_length = 100)
{
   $this->assertGreaterThen($trigger_length, strlen($content));
}
```

Мы вынуждены так сделать, потому что в самом `AcceptanceTester` у нас нет доступа к методам проверки. Не забудьте выполнить `./cept build` впоследствии, так как мы внесли изменения в один из модулей, которые использует `AcceptanceTester` (да, `AcceptanceHelper` технически является «модулем» Codeception).

Теперь выполним тесты и посмотрим, что они не проходят:

```
$ ./cept run tests/acceptance/DocumentationCept.php

1) Failed to see whether user documentation is accessible in
DocumentationCept.php
Sorry, I couldn't see "Documentation","h1":
Failed asserting that any element by 'h1' on page /site/docs
Elements:
+ <h1> Not Found (#404)
contains text 'Documentation'

Scenario Steps:
2. I see "Documentation","h1"
1. I am on page "/site/docs"
```

Конечно же, ведь у нас ещё нет обработчика маршрута `/site/docs`. Создайте метод `SiteController.actionDocs()`:

```
public function actionDocs()
{
   return $this->render('docindex.md');
}
```

Обратите внимание на отсутствие какого-либо кода, обрабатывающего Markdown. В этом заключается весь смысл преднастроенных отрисовщиков.

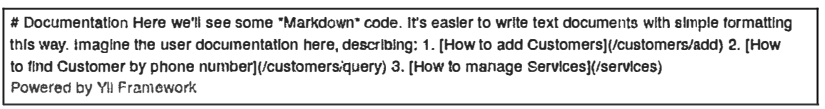
Теперь файл представления `views/site/docindex.md` должен выглядеть следующим образом:

```
# Documentation

Here we'll see some *Markdown* code.
It's easier to write text documents with simple formatting this way.

Imagine the user documentation here, describing:
```

```
1. [How to add Customers](/customers/add)
2. [How to find Customer by phone number](/customers/query)
3. [How to manage Services](/services)
```

Это является совершенно легальной страницей в формате Markdown.

Если посетить прямо сейчас страницу `/site/docs`, то мы увидим, что текст рендерится как есть, потому что `docindex.md` обрабатывается как сценарий PHP, и поскольку он не содержит инструкцию обработки `<?php`, он отрисовывается в виде HTML. Вот снимок экрана:



Наконец, давайте напишем сам отрисовщик. Так как этот класс является частью инфраструктуры приложения и не имеет ничего общего с моделью предметной области или маршрутизацией, давайте создадим отдельный подкаталог `utilities` и пространство имён для него. Таким образом, в файле `utilities/MarkdownRenderer.php` должно быть написано следующее:

```
<?php
namespace app\utilities;

use yii\helpers\Markdown;
use yii\base\ViewRenderer;

class MarkdownRenderer extends ViewRenderer
{
   public function render($view, $file, $params)
   {
      // TODO
   }
}
```

Пространство имён `app\utilities` автоматически отображается на каталог `utilities` в корневом каталоге проекта благодаря автозагрузчику, который мы упоминаем в нашей точке входа `index.php`.

«*Но как же мы будем отрисовывать Markdown?*» – можете спросить вы. Вот как:

```
   public function render($view, $file, $params)
   {
      return Markdown::process(file_get_contents($file));
   }
```

Yii 2 включает в себя полноценный процессор документов Markdown в качестве одной из своих зависимостей.

Впрочем, будьте осторожны, так как прямой вызов `file_get_contents()` довольно небезопасен. Мы полагаемся на то, что аргумент `$file` создаётся внутренностями Yii.

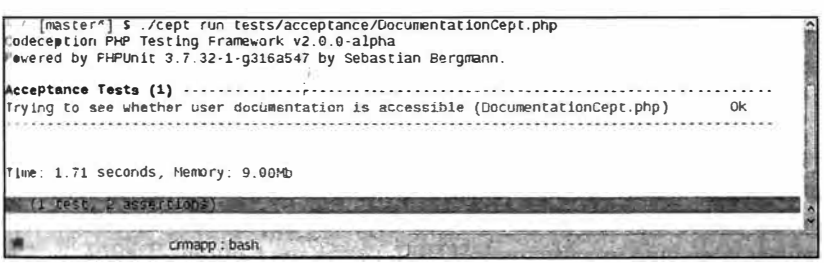
Класс `MarkdownRenderer` не имеет модульных тестов, так как всё, что мы сделали, – это обернули две встроенные и уже и без того оттестированные функции в один вызов. Чтобы упростить обсуждение, мы пропустили разработку `MarkdownRenderer` через тестирование. Заметьте, однако, что в более сложных случаях вам ни в коем случае нельзя так делать. Не всегда у вас будет такой простой парсер.

Имея написанный `MarkdownRenderer`, мы должны присоединить его к приложению. Добавьте описание нашего отрисовщика в раздел `components.view.renderers.md` конфигурации приложения в файле `config/web.php`:

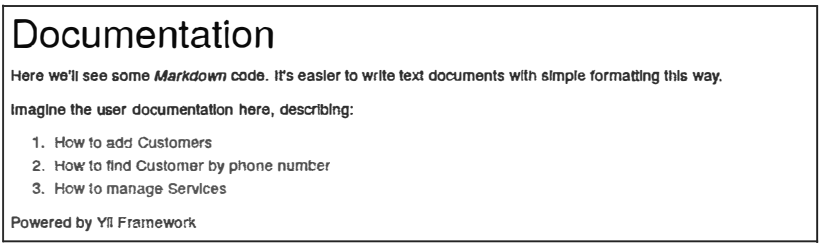
```
'components' => [
   'view' => [
      'renderers' => [
         'md' => [
            'class' => 'app\utilities\MarkdownRenderer'
         ]
      ]
   ]
]
```

Индекс в массиве `renderers` – это расширение файла. В нашем случае это будет `md`. Наш отрисовщик не имеет никаких свойств, поэтому, для того чтобы корректно сослаться на него в конфигурации приложения, нам достаточно предоставить только полностью определённое имя его класса.

Теперь запустите тесты. На этот раз они проходят успешно.



По маршруту /site/docs теперь можно увидеть корректно скомпонованную HTML-страницу:



Ручная настройка компоновщика отклика

Как уже было упомянуто в начале этой главы, последний этап обработки данных, перед тем как послать их клиенту, – это передача их в «компоновщик отклика», Response Formatter. Всё, что возвращает действие контроллера, оборачивается в объект Response, который решает, как именно в конечном счете отправлять данные по проводам. Давайте посмотрим, как мы можем использовать эту функциональность себе на пользу.

Возможно, самый очевидный пример использования преднастроенных компоновщиков отклика – это возврат данных в формате JSON с определённого маршрута. Следующий фрагмент – пример кода, возвращающего атрибуты зарегистрированных в нашем приложении услуг в формате JSON:

```
public function actionJson()
{
    $models = ServiceRecord::find()->all();
    $data = array_map(function ($model) {return $model->attributes;},
$models);

    $response = Yii::$app->response;
    $response->format = Response::FORMAT_JSON;
    $response->data = $data;

    return $response;
}
```

На следующем снимке экрана показан результат разговора с маршрутом /services/json из командной строки. Обратите внимание, что сервер выставил правильный HTTP-заголовок Content-Type.

```
Файл  Правка  Вид  Закладки  Настройка  Справка
$ curl -v http://localhost:8080/services/json
* Hostname was NOT found in DNS cache
*   Trying ::1...
* connect to ::1 port 8080 failed: В соединении отказано
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /services/json HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 03 Mar 2014 10:37:08 GMT
* Server Apache/2.4.7 (Ubuntu) is not blacklisted
< Server: Apache/2.4.7 (Ubuntu)
< X-Powered-By: PHP/5.5.9-1+sury.org~precise+1
< Content-Length: 1072
< Content-Type: application/json; charset=UTF-8
<
[{"id":86,"name":"Fugiat amet quia.","hourly_rate":92},{"id":87,"name":"Ut eum et distinctio
sit.","hourly_rate":55},{"id":88,"name":"Quis delectus at.","hourly_rate":81},{"id":90,"name"
:"Dolor possimus porro veniam.","hourly_rate":99},{"id":91,"name":"Dolorem qui et laudantium.
","hourly_rate":64},{"id":92,"name":"Maiores eum qui.","hourly_rate":75},{"id":94,"name":"Aut
 cumque fugiat.","hourly_rate":17},{"id":95,"name":"Soluta voluptas sed quod.","hourly_rate":
60},{"id":96,"name":"Necessitatibus est natus rerum.","hourly_rate":5},{"id":98,"name":"Qui e
um molestiae.","hourly_rate":37},{"id":99,"name":"Labore earum quam id.","hourly_rate":31},{"
id":100,"name":"Molestias in.","hourly_rate":45},{"id":102,"name":"Qui neque non consequatur.
","hourly_rate":7},{"id":103,"name":"Impedit quibusdam et tempora.","hourly_rate":23},{"id":1
04,"name":"Magni numquam odio.","hourly_rate":85},{"id":106,"name":"Aliquid ipsum omnis.","ho
urly_rate":94},{"id":107,"name":"Assumenda autem corporis.","hourly_rate":24},{"id":108,"name
"* Connection #0 to host localhost left intact
:"Fugiat beatae consequatur.","hourly_rate":43}]
                    hijarian : bash
```

В первую очередь обратите внимание, что мы вообще не вызываем здесь метода `render()`. Данные, которые мы получаем в виде ассоциативного массива, заворачиваются в полученный от приложения экземпляр `Response` в виде поля `Response.data` и возвращаются из действия контроллера в таком виде. Когда мы возвращаем строку, созданную методом `Controller.render()`, Yii самостоятельно заворачивает её в экземпляр `Response` вместо нас. На самом деле всё, что не является наследником класса `Response` и возвращается из действия контроллера оператором `return`, будет автоматически вложено в поле `Response.data`.

Во-вторых, мы не создавали объекта «отклика» (прямой перевод названия класса `Response`) сами, а вместо этого получили ссылку на него как компонент приложения Yii. Так значения его свойств уже будут установлены Yii на этапе инициализации приложения. Объект `Response` используется в процессе жизни приложения ровно один раз, поэтому на самом деле нет никакой разницы, кроме удобства установки настроек по умолчанию, получим ли мы его как компонент Yii или вызвав метод `__construct()`.

Поле `format` указывает объекту `Response`, как оформлять выходные данные. На момент написания этой книги разработчики Yii предоставили нам следующие типы, и нам нет необходимости заново их изобретать:

Литерал	Эффект
`Response::FORMAT_HTML`	Это литерал по умолчанию. Заголовок HTTP `Content-Type` будет установлен в значение `text/html`. Никакой обработки не будет совершено с данными, за исключением того, что объекты будут сериализованы вызовом метода `__toString()`
`Response::FORMAT_RAW`	Никакой обработки не будет совершено с данными, за исключением того, что объекты будут сериализованы вызовом метода `__toString()`. Yii даже не выставит заголовок `Content-Type`
`Response::FORMAT_JSON`	Данные будут обработаны методом `Json::encode()`, поставляемым с Yii 2. `Content-Type` будет установлен в значение `application/json`
`Response::FORMAT_JSONP`	Данные должны быть предоставлены в виде ассоциативного массива с двумя элементами: строкой по ключу `callback` и произвольными данными по ключу `data`. Откликом будет строка `callback(data)`, то есть строка по ключу `callback` – это какое-то название функции, а данные по ключу `data` будут переданы в неё. Данные по ключу `data` будут обработаны методом `Json::encode()`. Этот процесс фактически реализует рекомендацию **JSONP** (см. http://json-p.org/). `Content-Type` будет установлен в значение `text/javascript`
`Response::FORMAT_XML`	Данные будут обработаны классом `XmlResponseFormatter`. Вкратце это означает грамматически правильную строку XML в качестве отклика и заголовок `Content-Type`, установленный в значение `application/xml`. Ожидается, что данные будут либо ассоциативным массивом, либо объектом с публичными полями

Во всех случаях, когда устанавливается значение заголовка `Content-Type`, подзаголовку `charset` будет присвоено значение свойства `charset` экземпляра `Response` или, в случае если это свойство не установлено, значение свойства `charset` самого приложения. Кроме JSON, где `charset` обязан быть UTF-8, согласно спецификации.

Давайте сделаем что-нибудь ненормальное и сериализуем данные об услугах не в JSON, а в **YAML** (см. http://www.yaml.org/spec/1.2/spec.html). Одной из зависимостей Codeception является библиотека работы с форматом YAML из проекта **Symfony2**, поэтому мы просто используем её, вместо того чтобы писать свой сериализатор.

Создайте класс `app\utilities\YamlResponseFormatter` в подкаталоге `utilities` со следующим содержанием:

```
<?php
namespace app\utilities;
```

```
use Symfony\Component\Yaml\Yaml;
use yii\web\ResponseFormatterInterface;

class YamlResponseFormatter implements ResponseFormatterInterface
{
    const FORMAT = 'yaml';

    public function format($response)
    {
        $response->headers->set('Content-Type: application/yaml');
        $response->headers->set('Content-Disposition: inline');
        $response->content = Yaml::dump($response->data);
    }
}
```

Обратите внимание на выделенные части. Мы используем класс **Symfony\Component\Yaml\Yaml**, и нам нужно реализовать интерфейс **yii\web\ResponseFormatterInterface**.

Константа класса под названием `FORMAT` определена просто для большей наглядности, когда мы будем устанавливать формат отклика.

Реализация метода `format()` достаточно прямолинейна, спасибо интуитивно понятному методу `Yaml::dump()`. Особенность в том, что мы должны установить поля `headers` («Заголовки») и `content` («Содержимое») экземпляра Response, а не возвращать что-либо из этого метода.

Формат YAML не имеет зарегистрированного типа **Multipurpose Internet Mail Extensions (MIME)**, так что мы произвольно решили использовать строку `application/yaml`, чтобы подчеркнуть то, что это формат сериализации, предназначенный для прочтения некоторой программой, а не человеком. Хотя вы можете перепроверить наличие этого типа в реестре MIME по адресу http://www.iana.org/assignments/media-types/media-types.xhtml.

Чтобы подсоединить этот компоновщик к компоненту Response, нам нужно добавить его объявление к элементу `components.response.formatters` в конфигурации приложения следующим образом:

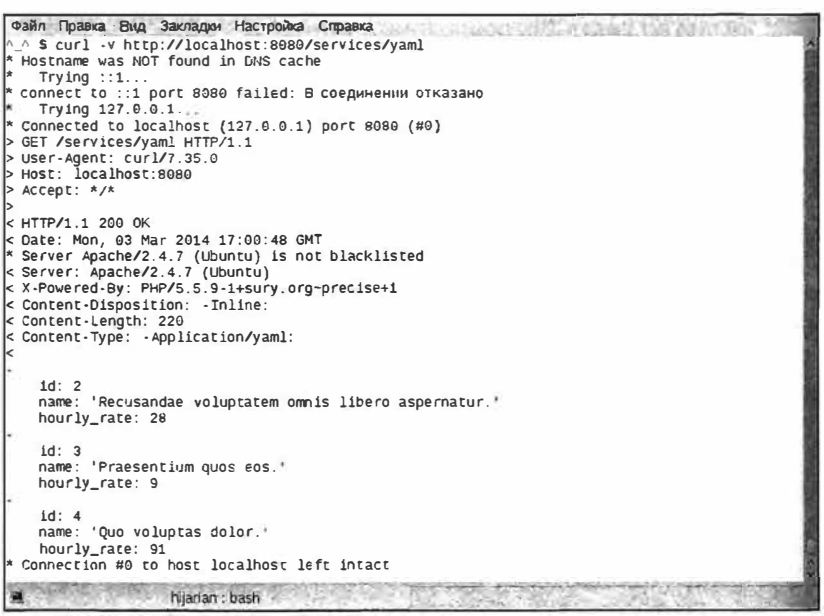
```
'components' => [
    'response' => [
        'formatters' => [
            'yaml' => [
                'class' => 'app\utilities\YamlResponseFormatter'
            ]
        ]
    ]
]
```

Выделенная часть – это объявление того, что класс `YamlResponseFormatter` будет обрабатывать формат `yaml`.

И наконец, добавим действие в `ServiceController`:

```
public function actionYaml()
{
    $models = ServiceRecord::find()->all();
    $data = array_map(function ($model) {return $model->attributes;},
$models);

    $response = Yii::$app->response;
    $response->format = YamlResponseFormatter::FORMAT;
    $response->data = $data;

    return $response;
}
```

Заметьте, что код практически идентичен коду, который компоновал данные в JSON. В этом заключается весь смысл преднастроенных компоновщиков отклика. Пример результата посещения маршрута `/services/yaml` при помощи CURL показан на следующем снимке экрана.



Эти данные в формате YAML могут быть прочитаны обратно в виде структур данных PHP методом `Yaml::parse()`, который является зеркальным отражением `Yaml::dump()`. Если вы откроете этот маршрут в браузере, браузер покажет вам диалоговое окно «Сохранить как...», поскольку тип MIME `application/yaml` не будет открываться в виде простого текста. Впрочем, мы включили HTTP-заголовок `Content-Disposition: inline` для того, чтобы заставить браузер показать эти данные, если он всё-таки это умеет.

ВОЗМОЖНОСТЬ: пакеты материалов

Мы уже упоминали и использовали пакеты материалов ранее, в *главе 3 «Автоматическая генерация кода CRUD»*. Давайте более подробно их обсудим.

Назначение пакета материалов (Asset Bundle) – собрать воедино связанные между собой файлы CSS и/или Javascript, лежащие в каком-то месте проекта, и зарегистрировать их на странице HTML одним вызовом PHP. Более того, пакеты материалов могут зависеть друг от друга, и в этом случае единственный вызов регистрации «главного» пакета материалов может вылиться в регистрацию всего пользовательского интерфейса приложения. Файлы CSS и Javascript здесь названы «материалами» (грубый перевод термина assets), отсюда и название концепции «пакета материалов».

Пакет материалов с файлами из произвольного каталога

Следующий фрагмент кода описывает типичный пакет материалов, который ссылается на произвольный каталог проекта:

```
namespace yii\web;
class YiiAsset extends AssetBundle
{
    public $sourcePath = '@yii/assets';
    public $js = [
        'yii.js',
    ];
    public $depends = [
        'yii\web\JqueryAsset',
    ];
}
```

Это встроенный пакет материалов, который требуется для работы любых встроенных виджетов Yii 2. Он расширяет класс `yii\web\AssetBundle` и имеет следующие свойства:

Название свойства	Смысл
`sourcePath`	Псевдоним пути к каталогу, который будет приписан к началу всех путей в свойствах `js` и `css` этого пакета. То есть это произвольный каталог в проекте, содержащий все материалы, которые мы собираемся зарегистрировать
`js`	Массив относительных путей до файлов, которые будут зарегистрированы как элементы `<script src=""></script>` в HTML-документе. Подразумевается, что эти файлы – сценарии Javascript
`css`	Массив относительных путей до файлов, которые будут зарегистрированы как элементы `<link rel="stylesheet" href="" />` в HTML-документе. Подразумевается, что эти файлы – описания CSS. `YiiAsset` ничего не определяет в этом свойстве, что можно условно понимать как то, что он «не содержит» никаких файлов CSS
`depends`	Массив полностью определённых имён классов, которые должны считаться пакетами материалов и быть зарегистрированными ранее этого пакета

Конечно же, это не полный список свойств. Полное описание класса `AssetBundle` можно найти в исходниках Yii 2, в файле `yii2/web/AssetBundle.php`.

Что значит «зарегистрировать» файл CSS или Javascript? Это означает сохранить путь до него некоторым способом, а потом, когда будет формироваться конечный HTML-документ, вставить соответствующие этим файлам HTML-элементы (link или script), заполнив, соответственно, атрибуты `href` или `src`. Но если мы сохраняем файлы материалов в какой-то произвольной папке, возможно, недоступной для веб-сервера, какие URL использовать?

Публикация материалов

Чтобы ответить на только что заданный вопрос, в Yii присутствуют класс `yii\web\AssetManager` и концепция «публикации материалов» (**assets publishing**). Когда вы используете `AssetBundle`, у которого определено свойство `sourcePath`, экземпляр `Application`, представляющий приложение Yii, использует компонент `AssetManager`, для того чтобы скопировать каталог из `sourcePath` в некий каталог, определённый в параметре `AssetManager.basePath`. По умолчанию `AssetManager.basePath` имеет значение `@webroot/assets`, и поэтому в результате ката-

лог, на который ссылался AssetBundle, будет скопирован в некое место, доступное из Сети. Он будет переименован в некоторый особый, уникальный хэш, зависящий от метки времени, чтобы AssetManager не переопубликовывал каталоги, в которых ничего не менялось.

Публикация материалов – это очень сложная часть механики Yii, и, безусловно, очень важная и полезная функциональность. Вы можете углубиться в детали, прочитав документацию для Yii 2, но сейчас мы сконцентрируемся на практических следствиях публикации материалов, которыми являются следующие.

- По умолчанию все материалы публикуются в подкаталог assets вашего корневого каталога, доступного из Сети. Обычно нет никакой необходимости менять данную настройку.
- Вы можете полностью удалить содержимое этого каталога, который содержит материалы, в любое время безо всяких последствий, кроме времени, потраченного на повторную публикацию материалов.
- Вам нельзя удалять сам этот каталог, потому что Yii не проверяет его существование и не создаёт его автоматически.
- В assets копируется целиком всё содержимое папки AssetBundle.sourcePath. Это означает, что у вас может быть, скажем, комплект файлов CSS в подпапке css, которые ссылаются на изображения в формате PNG, сохранённые в подпапке img, используя относительные пути вида ../img/<imagename>. То же самое применимо к файлам шрифтов.
- При разработке приложения не имеет смысла вносить изменения в файлы из каталога @webroot/assets. Вам необходимо изменить исходные файлы в AssetBundle.sourcePath и затем переопубликовать их заново. Yii 2 пытается самостоятельно определить, нужно ли ему автоматически переопубликовать материалы, сравнивая метки времени, но он может ошибаться. Поэтому беспроигрышный вариант – это просто стереть содержимое каталога assets, чтобы Yii был вынужден заполнить его заново.
- Существует свойство AssetManager.linkAssets, которое можно установить настройкой components.assetManager.linkAssets в конфигурации приложения. Когда эта настройка имеет значение true, AssetManager не копирует требуемый каталог с материалами, а делает для него символическую ссылку. Это не работает в Windows и, возможно, имеет некоторые проблемы с безопасностью. Например, вы должны обязательно включить

FollowSymlinks внутри вашей папки @webroot/assets, если вы используете Apache. Однако в системе, где это не является проблемой, создание символической ссылки вместо копирования убирает проблему, описанную в предыдущем пункте, так как @webroot/assets всегда будет содержать новейшую «копию» файлов CSS и Javascript.

Вы можете опубликовать любой каталог или файл следующим вызовом:

```
list($dir, $url) = Yii:$app->assetManager->publish($path).
```

- Переменная $path – это путь до файла или каталога в проекте. Он обрабатывается методом Yii::getAlias(), так что может быть любым псевдонимом пути, понятным Yii 2.
- Переменная $dir будет содержать полный абсолютный путь до только что опубликованного $path.
- Переменная $url будет содержать полный абсолютный URL до только что опубликованного $path. Можно будет использовать этот URL в вашем HTML-документе и быть уверенным, что запрос браузера к нему на самом деле вернёт опубликованный файл или каталог.

Пакет материалов с файлами из доступного из Сети каталога

Вот пример определения AssetBundle, который ссылается на файлы, уже расположенные в каталоге, доступном из Сети:

```
class MyUiAsset extends yii\web\AssetBundle
{
   public $basePath = '@webroot/ui';
   public $baseUrl = '@web/ui';
   public $css = ['main.css'];
   public $js = ['main.js'];
}
```

Здесь мы видим свойства basePath и baseUrl. «Базовый путь» (прямой перевод фразы base path) – это псевдоним пути до каталога, содержащего файлы, относящиеся к этому пакету. «Базовый URL» (прямой перевод фразы base URL) – это абсолютный URL, который будет приписан в качестве префикса ко всем ссылкам на файлы, так что браузер будет в состоянии правильно их запросить. Обратите

внимание на то, что система псевдонимов путей может быть применена и к URL.

Обычно базовый путь и базовый URL похожи, так как мы чаще всего используем возможность веб-сервера выдавать файл напрямую, используя путь в файловой системе. Однако в более сложных случаях у вас может быть настроена непростая система переписывания URL или, возможно, маршруты, генерирующие код CSS или Javascript «на лету». Всё это, конечно же, достаточно редко встречается, однако Yii всё равно предоставляет возможность подстраиваться под такие случаи.

Смысл в том, что когда вы используете свойство `sourcePath` в пакете материалов, Yii вначале публикует эту исходную папку и модифицирует свойства `basePath` и `baseUrl` пакета материалов за вас. В качестве побочного эффекта мы имеем то, что нет смысла использовать все три эти свойства одновременно, так как `sourcePath` всё равно будет иметь преимущество.

Использование `basePath` имеет смысл, когда вы в состоянии терпеть папки, уже существующие в корневом каталоге, доступном из Сети. Однако вся суть использования пакета материалов – хранить материалы, относящиеся к одной библиотеке, в одной папке. Если вы используете много пакетов материалов, основанных на базовом пути, это неизбежно заполнит каталог, доступный из Сети, множеством различных файлов из различных библиотек. И это может достаточно быстро стать очень уродливым.

Однако ничто не мешает вам использовать один и тот же `basePath` для всех пакетов материалов (например, `@webroot`) и просто ссылаться на различные файлы в каждом пакете. Это приведёт к запутанному месиву взаимосвязанных файлов и является кошмаром сопровождающего, ждущим своего момента.

Таким образом, пакеты материалов, основанные на `sourcePath`, несмотря на неуклюжий цикл изменения и переопубликования, выглядят более сопровождаемым и идиоматичным решением.

Ручная регистрация файлов CSS и Javascript

При наличии системы пакетов материалов вам, возможно, никогда не понадобится регистрировать материалы вручную, но в случае, когда вам это все же понадобится, вот список методов, которые вы можете для этого использовать:

Вызов метода	Эффект
`registerCss($css, $options)`	Размещает HTML-элемент <style> внутри элемента <head> со значением `$css` в качестве содержимого и атрибутами, перечисленными в необязательном аргументе `$options`. В конечном счёте делает вызов `Html::style($css, $options)`
`registerCssFile($url, $depends, $options)`	Размещает HTML-элемент <link> внутри элемента <head>, указывающего на `$url`. Если объявлены необязательные `$depends`, тогда вместо этого будет зарегистрирован дополнительный `AssetBundle` с `$url` с единственным элементом CSS и зависимостями, объявленными в `$depends`. Необязательные `$options` – это массив атрибутов для элемента <link>. В конечном счёте делает вызов `Html::cssFile($url, $options)`
`registerJs($js, $position)`	Размещает элемент <script> согласно расположению, указанному в аргументе `$position` (см. ниже), со значением аргумента `$js` в качестве содержимого. В конечном счете делает вызов `Html::script($js, ['type' => 'text/javascript'])`
`registerJsFile($url, $depends, $options)`	Размещает элемент <script> согласно указанному расположению, с атрибутом `src`, равным значению аргумента `$url`. На этот раз расположение определяется полем `$options['position']`. Объяснение понятия «расположение» – ниже. Если определён аргумент `$depends`, тогда вместо этого регистрируется дополнительный `AssetBundle` с `$url` в качестве единственного элемента Javascript и с зависимостями, объявленными в `$depends`. В конечном счете делает вызов `Html::jsFile($url, $options)` (ключ `position` будет удалён из `$options`)

Обратите внимание, что эти методы всё ещё лучше, чем писать элементы `script` и `link` в HTML-шаблонах или, ещё хуже, напрямую в файлах представлений. Одна из наиболее очевидных причин, почему это лучше, – это возможность включить только те материалы, которые нужны для маршрута, обрабатываемого в данный момент (то есть для страницы, в данный момент генерируемой).

Все эти методы объявлены в классе `View`, так что вы можете использовать их как `$this->registerCss($css)` внутри файлов представлений и как `Yii::$app->view->registerCss($css)` в любом другом месте.

Заметьте, что мы не перечислили точных определений функций, только те части, которые чаще всего используются на практике.

Для ссылок на файлы Javascript (в виде HTML-элементов <script>) существует концепция «расположения». Вот перечень всех возможных расположений:

- `View::POS_HEAD` – размещает их внутри элемента `<head>`;
- `View::POS_BEGIN` – размещает их в начале элемента `<body>`;
- `View::POS_END` – размещает их в конце элемента `<body>`;
- `View::POS_READY` – означает, что блок Javascript будет размещён внутри вызова `jQuery(document).ready()`, который, в свою очередь, будет находиться в конце элемента `<body>`. Это расположение не имеет смысла для метода `registerJsFile()`, только для отдельных фрагментов кода;
- `View:: POS_LOAD` – означает, что блок Javascript будет размещён внутри вызова `jQuery(document).load()`, который, в свою очередь, будет находиться в конце элемента `<body>`. Это расположение не имеет смысла для метода `registerJsFile()`, только для отдельных фрагментов кода.

Мы отбрасываем возможность того, что вам может понадобиться поместить элемент `<script>` где-то внутри HTML-страницы. Yii не поможет вам в таком хулиганстве.

Размещение файлов Javascript в пакетах материалов

Существует очень полезный трюк с пакетами материалов, относящийся к ручной регистрации файлов Javascript и CSS, которая была только что объяснена.

По умолчанию при регистрации пакета материалов все файлы Javascript, упомянутые в нём, будут размещены в конце элемента `body`, что соответствует расположению `View::POS_END`. Однако есть способ изменить это расположение. К сожалению, вы не можете указать расположение для каждого файла Javascript по отдельности, только для всего пакета материалов.

В классе `AssetBundle` есть свойство под названием `jsOptions`. Оно хранит параметры, которые будут переданы в вызов `registerJsFile` как аргумент `$options`, когда файлы Javascript из этого пакета будут регистрироваться. Поэтому вы можете добавить следующую строчку в ваше определение `AssetBundle`:

```
public $jsOptions = ['position' => View::POS_HEAD];
```

Все файлы Javascript, упомянутые в свойстве `js`, будут вместо конца `<body>` размещены в элементе `<head>` конечного документа HTML.

Также есть свойство `cssOptions`, которое может быть использовано для установки аргумента `$options` при вызове `registerCssFile` для

файлов CSS. Используя его, вы можете установить свойство media для элемента <link>, например:

```
public $cssOptions = ['media' => 'print,aural,tty'];
```

Создаём свой пакет материалов

Давайте создадим свой собственный пакет материалов, так что мы сможем инициализировать наши стили и поведение клиентской части одним вызовом.

Создайте каталог под названием assets в корневом каталоге проекта. Внутри него создаёте файл под названием ApplicationUiAssetBundle.php со следующим определением:

```
namespace app\assets;

use yii\web\AssetBundle;

class ApplicationUiAssetBundle extends AssetBundle
{
   public $sourcePath = '@app/assets/ui';
   public $css = [
      'css/main.css'
   ];
   public $js = [
      'js/main.js'
   ];
   public $depends = [
      'yii\bootstrap\BootstrapAsset',
      'yii\web\YiiAsset'
   ];
}
```

Надеемся, это определение теперь полностью понятно. Для того чтобы оно не было ложью, нам нужно создать два файла материалов, пока пустых: assets/ui/css/main.css и assets/ui/js/main.js.

Так как мы объявили, что наши материалы зависят от YiiAsset и BootstrapAsset, мы можем заменить регистрацию этих пакетов одной только регистрацией нашего пакета AssetBundle. Внутри главного файла шаблона мы можем заменить следующие строки:

```
\yii\bootstrap\BootstrapAsset::register($this);
\yii\web\YiiAsset::register($this);
```

такой строкой:

```
app\assets\ApplicationUiAssetBundle::register($this);
```

Теперь у нас есть основа для того, чтобы начать писать свои стили оформления для нашего примера приложения.

ВОЗМОЖНОСТЬ: темы

Одна из интересных вещей, встроенных в систему отрисовки Yii 2, – это поддержка **тем (themes)**. Вместе с поразительно гибкой системой пакетов материалов это даёт нам ещё один уровень контроля над внешним видом веб-приложения, которое мы обслуживаем.

Определение «темы» из официальной пользовательской документации и блока документации в комментариях – вероятно, самое лучшее объяснение данной концепции из всех возможных. Вот его перевод:

> *«Тема» – это каталог с файлами шаблонов и представлений. Каждый файл темы перекрывает соответствующий файл приложения, когда он отрисовывается. Одно приложение может иметь множество тем, и каждая может предоставлять совершенно иную функциональность. В любое время активна только одна тема.*

Важнейшей частью является то, что тема – это отдельный набор файлов представлений, который может перекрывать существующие файлы представлений. Данные, которые отправляются в файлы представлений из контроллера, остаются теми же самыми, но файл представления из темы может делать с ними что-то полностью другое. С поддержкой концепции пакетов материалов вы можете не только реорганизовывать вещи на конечной странице, но и менять его оформление.

Вы можете прочитать точные правила применения темы в документации; они не заслуживают долгих описаний в нашем случае. Давайте вместо этого посмотрим на достаточно небольшой пример.

Создание своей «снежной» темы

Давайте поменяем оформление главной страницы нашего приложения таким образом, что на фоне будет снег, а заголовок «Our CRM» будет увеличен. Не ищите в снеге никакого смысла, его нет.

Сейчас наш `SiteController.actionIndex()` выглядит следующим образом:

```
public function actionIndex()
{
    return 'Our CRM';
}
```

То есть на самом деле он не рендерит никакой файл представления, и в результате никакой шаблон не применяется. Замените этот метод на следующий:

```
public function actionIndex()
{
    return $this->render('homepage');
}
```

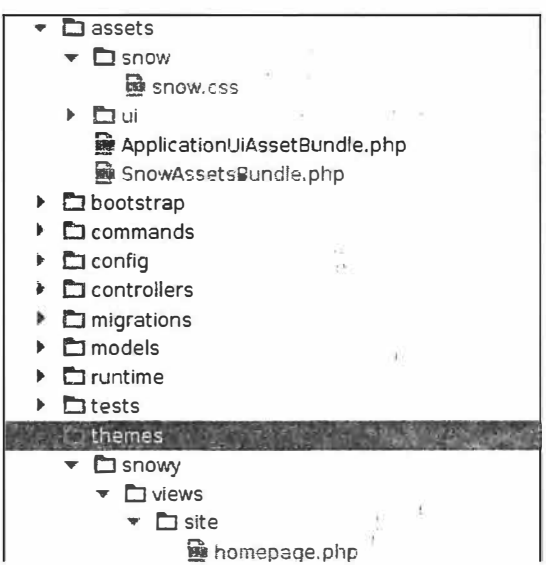
И затем создайте файл `views/site/homepage.php` с буквами `Our CRM` в качестве содержимого внутри. Больше ничего там не нужно.

Все приёмочные тесты всё ещё должны успешно завершаться после этого изменения.

Так как тема – это просто каталог с файлами представлений, и эти файлы представлений перекрывают файлы представлений, уже существующие в проекте, создадим файл `themes/snowy/views/site/homepage.php`.

Мы также будем использовать для этой темы отдельный пакет материалов, так что создайте для него каталог под названием `assets/snow`. Создайте также файл класса `SnowAssetsBundle.php` в каталоге `assets` и файл `snow.css` в каталоге `assets/snow`.

Итоговое дерево каталогов должно выглядеть следующим образом (зелёным выделены файлы, которые должны быть созданы):



Пакет «заснеженных» материалов будет иметь очевидное определение:

```
class SnowAssetsBundle extends AssetBundle
{
   public $sourcePath = '@app/assets/snow';
   public $css = ['snow.css'];
   public $depends = ['app\\assets\\ApplicationUiAssetBundle'];
}
```

Он зависит от главного пакета материалов, потому что мы хотим, чтобы он переопределил стили по умолчанию, а значит, загружался последним.

CSS в нашем случае очень простой, мы просто применяем фоновое изображение для тела HTML-документа и добавляем один класс, для того чтобы делать некоторый текст выделенным:

```
body {
   background: #a6e1ec url(snow.jpg) repeat;
}
.inside-snowflakes {
   margin: 10% 15%;
   font-size: 2em;
}
```

Изображение снега находится внутри пакета кода, который может быть скачан с сайта Packt Publishing, и является произведением Джордана Ллойда (Jordan Lloyd). Вот исходный URL: http://www.flickr.com/photos/jordanlloyd/5342749399/in/photostream/.

После этого мы применяем новый пакет материалов и оборачиваем текст в контейнер с новым стилем в `homepage.php` следующим образом:

```
<?php app\assets\SnowAssetsBundle::register($this); ?>
<p class="inside-snowflakes">Our CRM</p>
```

Итак, подготовка завершена; у нас есть тема в отдельной папке и новый пакет материалов, на который эта тема ссылается. Теперь, чтобы применить эту новую тему к сайту, мы должны добавить следующее объявление в настройку `components.view.theme` конфигурации приложения:

```
   'components' => [
   ...
      'view' => [
         ...
         'theme' => [
```

```
            'class' => yii\base\Theme::className(),
            'basePath' => '@app/themes/snowy',
        ]
    ...
    ],
```

Вот конечный результат, который вы должны получить:

Он довольно глупый, но достаточно хорошо иллюстрирует саму концепцию. Настоящие возможности системы тем заключаются в том, что мы можем включать их на основе некоторых условий, что в идеале должно быть совершено на этапе загрузки приложения. Первый вариант – это создать подкласс `yii\web\Application` с нужным переопределением метода `init()` и создавать уже его в сценарии точки входа `index.php`. Второй вариант – использовать систему событий и добавить обработчик на событие `yii\base\Application::EVENT_BEFORE_REQUEST`. Больше про события мы узнаем в *главе 10 «События»*.

Виджеты

В пользовательском интерфейсе вашего приложения у вас часто будут фрагменты, которые должны быть размещены на нескольких страницах, но которые не могут быть просто так вставлены в файл шаблона. Например, потому что они должны получить данные, которые генерирует действие контроллера (а мы не можем передавать в шаблон ничего, кроме `$content`). Yii 2 инкапсулирует это в концепцию «виджета», который является, по своей сути, способом отрисовать отдельный файл представления с некоторой дополнительной произвольной логикой.

Виджеты могут быть очень конкретными, например специальный виджет-сокращение для какой-то особенной кнопки в пользовательском интерфейсе. Они также могут быть очень абстрактными, как виджет `\yii\widgets\ListView`, инкапсулирующий процесс отрисовки файла представления несколько раз на основе массива данных, в виде списка.

Хотя и кажется, что эта глава – подходящее место для детального обсуждения виджетов, лучше будет подождать до *главы 11 «Таблица»*, где у нас будет возможность на самом деле *использовать* виджет и посмотреть, как он может быть сконфигурирован. Поэтому, если вам интересно, пролистайте до раздела «Виджеты» в той главе.

Итоги

Конечная HTML-страница генерируется фреймворком в три шага.

1. Yii подготавливает весь HTML-код, который вы предоставляете ему в файлах представлений.
2. Он «регистрирует» файлы CSS и Javascript, которые вы сказали регистрировать, или при помощи пакетов материалов, или вызовами методов register*() вручную.
3. Все зарегистрированные материалы вставляются в HTML-страницу для передачи клиенту.

Второй шаг, который разделяет добавление материалов и отрисовку страницы, позволяет нам делать интересные вещи, такие как условная загрузка файлов CSS и Javascript.

Мы узнали про компоненты приложения, центральную возможность Yii 2 и про множество возможностей, позволяющих нам контролировать и настраивать процесс отрисовки результирующей страницы, – пакеты материалов, отрисовщики представлений, компоновщики отклика, темы, – а также про внутреннюю работу компонента View в целом.

Однако мы пропустили несколько тем, относящихся к процессу отрисовки. В первую очередь это концепция «виджетов» и, в частности, список встроенных виджетов. Также мы не рассмотрели различных вспомогательных методов компонента View, имеющих дело с кэшированием содержимого. Об этом будет сказано в *главе 8 «Общее поведение приложения»*.

Теперь мы немного отойдём от фундаментальных вещей и следующие две главы посвятим практике, где добавим механизмы аутентификации и авторизации к нашему приложению. Больше никаких глупых простых примеров.

Глава 5

Аутентификация

Давайте теперь поговорим про аутентификацию пользователей. Наш пример – приложение CRM в том виде, как оно есть, – остаётся довольно бесполезным, так как предоставляет к себе доступ любому желающему, а обычно мы не хотим, чтобы кто попало мог копаться в личных данных наших клиентов.

В этой главе мы посмотрим на то, что Yii нам предоставляет для того, чтобы помочь *идентифицировать* пользователя. А в следующей главе мы ответим на вопрос *авторизации* пользователя, то есть решения о том, позволять ли пользователю выполнять определённые действия в приложении.

Мы добавим следующие возможности в наш пример CRM-приложения:

- таблицу для записи пользователей, известных системе, и соответствующий интерфейс для управления ею;
- индикатор, видимый на всех страницах, показывающий, является ли пользователь, в данный момент применяющий приложение, зарегистрированным в этой таблице. Однако он должен будет явным образом объявить это, то есть «войти» в приложение.

На этом этапе вы уже должны понимать, что такое компоненты приложения Yii и, в особенности, как их настраивать и как получать к ним доступ. Если это не так, то лучше вам перечитать предыдущую главу и официальную документацию, посвящённую этой концепции, так как с этой главы мы будем использовать всё больше и больше встроенных компонентов приложения Yii.

Анатомия входа пользователя в систему в Yii

Вам нужно следующее, чтобы успешно аутентифицировать пользователя, применяя Yii:

1. Создать экземпляр объекта класса, реализующего интерфейс `yii\web\IdentityInterface`.
2. Вызвать метод `Yii::$app->user->login()` и передать в него этот объект.

Основная хитрость заключена в концепции **«личности» (Identity)**. Для Yii 2 этим является любой класс, реализующий `IdentityInterface`. Настоятельно рекомендуем прочитать определение этого интерфейса в файле `web/IdentityInterface.php` из корневого каталога Yii 2, так как там содержится его подробнейшее объяснение.

Пока пользователь не вошёл в систему (не было осуществлено успешного вызова `yii\web\User.login()`), свойство `Yii::$app->user->isGuest` будет возвращать `true`. После успешного входа и всё время, пока пользователь остаётся аутентифицированным, `Yii::$app->user->isGuest` будет всегда возвращать `false` и `Yii::$app->user->identity` будет возвращать объект, который был передан в вызов метода `login()` (то есть личность пользователя).

В этом, по сути, заключается весь принцип использования системы аутентификации, встроенной в Yii 2.

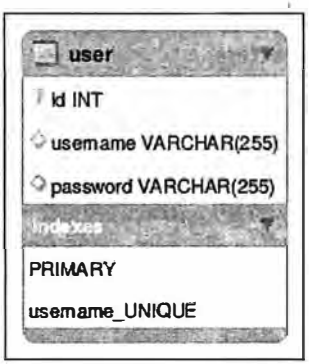
Yii только лишь поддерживает состояние аутентификации: хранит данные, идентифицирующие пользователя. Любые проверки того, что пользователь действительно тот, за кого себя выдаёт, должны производиться приложением.

Хотя сейчас аутентификация пользователя по паре логин/пароль постепенно становится профессиональной некомпетентностью, скорее всего, вам всё равно придётся реализовывать её в своём следующем приложении. Давайте рассмотрим именно этот метод входа в систему.

Механика входа в систему по логину и паролю в целом

Все знают эту древнюю схему входа в систему по логину и паролю. Записи о пользователях, известных приложению, хранятся в таблице базы данных, называемой `users`. Каждая запись содержит поля `username` и `password` (собственно, единицы информации, необходимые для аутентификации) вместе с остальной информацией о пользователе.

Минимальная таблица, представленная программой MySQL Workbench (см. http://www.mysql.com/products/workbench/), выглядит следующим образом:



Поле **username** (Имя пользователя) используется для того, чтобы идентифицировать пользователя среди прочих. Таким образом, на нём должно быть ограничение UNIQUE.

Поле **password** (Пароль) содержит строку, известную, по определению, только пользователю и приложению. Для большей защиты пароля приложение хранит его не в открытом виде, а в виде *криптографически безопасного* хэша, который генерируется алгоритмом наподобие `bcrypt`. В таком случае приложение не имеет возможности узнать изначальный текст пароля.

Важнейшая часть вышеописанного описания хранения паролей – это фраза «криптографически безопасного». Это значит, что алгоритм хэширования должен проявлять следующие характерные черты:

- он должен использовать *криптографическую функцию вычисления хэша*, главная особенность которой – быть *односторонней*, что значит, что вы легко можете вычислить хэш из текста, но получить текст из хэша очень сложно;
- в идеале он должен использовать отдельную функцию хэширования для каждого заданного текста. Это делается приёмом под названием «соление» (**salting**);
- он должен работать *медленно*.

При аутентификации пользователь предоставляет имя пользователя, чтобы заявить о том, кто он есть, и пароль, чтобы доказать это. Так как приложение знает, каким алгоритмом оно хэшировало пароль, когда пользователь регистрировался, оно хэширует предоставленный пароль тем же способом и сравнивает результат с тем, что записано в базе данных для соответствующего имени пользователя. Если хэши идентичны, значит, пользователь – на самом деле тот, за кого себя выдаёт, и мы считаем его аутентифицированным.

Ну, возможно, вы это уже знаете. Давайте реализуем этот метод аутентификации в нашем примере CRM-приложения.

Заметьте, что имя и пароль – не единственный способ аутентификации. Есть и другие методы, различающиеся как по сложности проверок безопасности, так и по пользовательскому интерфейсу. Например, есть инициатива **OpenID** (http://openid.net/), протокол **OAuth** (http://oauth.net/2/) для аутентификации в приложении с использованием личных данных из другого приложения, а также подписанные подтверждённые **сертификаты SSL** (https://developer.mozilla.org/ru/docs/Introduction_to_SSL#_Client_Authentication_).

В конечном счёте мы совершим следующие шаги:

1. Построим интерфейс манипулирования записями о пользователях.
2. Построим форму ввода данных для входа в систему.
3. Приделаем индикатор состояния пользователя (вошёл/не вошёл в систему).

Создание интерфейса управления пользователями

Нам нет никакой необходимости делать какой-то особенный пользовательский интерфейс. Также для простоты мы не будем возиться с отделением от ORM, как мы сделали в главе 2. Автоматически сгенерированный интерфейс нас полностью устроит. Мы не будем перечислять здесь в точности те же шаги, какие были описаны в главе 3 «Автоматическая генерация кода CRUD». Приведём лишь краткую сводку, перед тем как перейти к реализации возможностей, специфичных для пользователей.

Приёмочные тесты для интерфейса манипулирования пользователями

Подобно услугам в главе 3, нам нужно иметь возможность создавать, редактировать, просматривать и удалять пользователей из системы.

Так как приёмочные тесты для манипулирования пользователями будут почти в точности такими же, как для манипулирования услугами, если вы просто будете следовать инструкциям в этой книге, вам не обязательно их писать. Однако, строго говоря, в реальном приложении вам всё равно

придётся делать приёмочные тесты. Они вложены в пакет кода, соответствующий этой главе.

Прямо сейчас простейший путь – это скопировать `RegisterNewServiceCept.php`, `EditServiceCept.php` и `DeleteServiceCept.php` из каталога `tests/acceptance` в файлы `RegisterNewUserCept.php`, `EditUserCept.php` и `DeleteUserCept.php` в том же каталоге и затем переименовать все упоминания следующих двух вещей в строках, названиях методов и названиях переменных:

- «services» → «users»;
- «Service» → «User».

Сделав это, мы получим ссылку на несуществующий класс `AcceptanceTester\CRMUsersManagementSteps`, который мы создадим, скопировав файл `tests/acceptance/_steps/CRMServicesManagementSteps.php` в `tests/acceptance/_steps/CRMUsersManagementSteps.php` и заменив все упоминания `Service` на `User`.

Конечно же, это бесстыжее копирование кода, и его нужно отрефакторить, но вы можете считать это домашним заданием, потому что на самом деле мы не можем полностью избежать повторений в спецификациях возможностей манипулирования сущностями в приложении. А обучение правильным техникам рефакторинга выходит за рамки этой книги. Просто помните, что в данном случае от повторения *обязательно* нужно избавиться.

Единственное функциональное отличие в спецификации управления пользователями будет внутри класса `CRMUsersManagementSteps`. После предложенных изменений он теперь несколько раз упоминает строку `UserRecord[name]`. Наш класс `UserRecord` будет содержать не поле `name`, а поле `username`. Поэтому вам необходимо изменить `UserRecord[name]` на `UserRecord[username]`. Мы будем придерживаться слова `username` в качестве названия этого поля, потому что это не настоящее *имя* человека из реального мира, стоящего позади этой `UserRecord`, это просто идентификатор, который мы показываем приложению.

Также метод `imagineUser()` в том же классе должен выглядеть следующим образом:

```
function imagineUser()
{
   $faker = \Faker\Factory::create();
   return [
      'UserRecord[username]' => $faker->userName,
```

```
            'UserRecord[password]' => md5(time())
        ];
    }
```

Мы беспокоимся здесь о хэшах паролей, так как все эти данные – это пользовательский ввод, а не то, что будет сохранено в БД. Обратите внимание на способ, которым мы генерируем «случайный» пароль, хоть таким образом мы и можем получить только буквы и цифры. Мы делаем так только для простоты, поскольку пароль в идеале может содержать *любые* возможные символы, включая непечатные.

Таблица в БД для хранения записей о пользователях

Нам нужно подготовить таблицу, в которой мы будем хранить наши будущие записи о пользователях. Вот миграция базы данных, которая вам понадобится для реализации схемы, показанной ранее:

```
public function up()
{
    $this->createTable(
        'user',
        [
            'id' => 'pk',
            'username' => 'string UNIQUE',
            'password' => 'string'
        ]
    );
}

public function down()
{
    $this->dropTable('user');
}
```

Создание кода модели и CRUD при помощи Gii

Сделайте те же действия, что и в *главе 3 «Автоматическая генерация кода CRUD»*. Мы сохраним модель `UserRecord` в отдельное пространство имён `app\models\user`.

Название поля	Значение поля
Model class	`app\models\user\UserRecord`
Search Model class	`app\models\user\UserSearchModel`
Controller class	`app\controllers\UsersController`

Удаляем поле пароля из автоматически сгенерированного кода

Мы будем работать с полем ввода пароля автоматически за кулисами, но Gii о наших намерениях не знает. Нам нужно вручную удалить упоминания об этом поле из следующих мест:

1. `models/user/UserSearchModel.php`: мы не собираемся искать пользователей по паролю. Сначала удалите поле `password` из правила `safe` в методе `rules()`. Затем удалите следующую строчку из метода `search()`:

   ```
   $query->andFilterWhere(['like', 'password', $this->password])
   ```

2. `views/user/_search.php`: то же самое, мы не хотим искать пользователей по паролю. Удалите следующую строчку из конфигурации виджета `ActiveForm`:

   ```
   <?= $form->field($model, 'password') ?>
   ```

3. `views/user/index.php`: мы не хотим видеть пароли в записях пользователей. Удалите поле `password` из настройки `columns` в конфигурации виджета `GridView`.
4. `views/user/view.php`: то же самое, мы не хотим видеть пароли в записях пользователей. Удалите поле `password` из настройки `attributes` в конфигурации виджета `DetailView`.

Мы хотим, однако, иметь возможность вводить и менять пароль в записях пользователей, поэтому поле ввода пароля в файле `views/user/_form.php` должно остаться на месте.

Теперь начинается интересная часть: пароль должен быть захэширован при сохранении записи.

Хэширование пароля при сохранении записи пользователя

Чего нам не хватает в нашей текущей схеме? Очевидно, перед сохранением записи пользователя в БД нам нужно вычислить безопасный хэш предоставленного пароля и сохранить хэш вместо пароля в открытом виде. Более того, при обновлении записи пользователя мы не хотим, чтобы хэш был ещё раз захэширован, если мы не вводили пароль заново.

Yii 2 (так же, как и его предшественник, Yii 1.1.x) определяет несколько методов у экземпляров класса `ActiveRecord`, которые мы мо-

жем переопределить, для того чтобы делать что-либо на некоторых предопределённых этапах жизни активной записи. Хоть вы и можете прочитать о них более подробно в разделе «События класса `\yii\db\BaseActiveRecord`» в *главе 10 «События»*, мы один из них используем прямо сейчас, а именно метод `beforeSave()`, запускающийся прямо перед тем, как активная запись сохраняется в базе данных.

У него есть вот такое определение по умолчанию:

```
public function beforeSave($insert) { return true; }
```

Аргумент `$insert`, переданный в этот метод, показывает, является ли рассматриваемая активная запись новой, которую нужно добавить в БД, или уже существующей, которую нужно изменить.

Этот метод должен вернуть булево значение, показывающее, позволено ли данной записи сохраниться. Если нет, она не будет сохранена, причём безо всякой дополнительной информации пользователю или программе. Эту деталь очень важно помнить. Если вы ничего не вернёте из этого метода, будет считаться, что вы вернули значение `null`, что в PHP имеет то же булево значение, что и `false`, и в итоге вы полностью бесшумно запретите соответствующую операцию. Мы будем переопределять этот метод для наших целей в классе `UserRecord`.

В дополнение к этому в Yii 2 включён специальный компонент приложения `\yii\base\Security`, который содержит, среди прочего, вспомогательные методы для безопасного вычисления хэшей паролей. Этот компонент доступен через выражение `Yii::$app->security`. Это очень поможет нам, так как мы будем избавлены от необходимости вручную разбираться с деталями метода `crypt()` в PHP. В особенности нам интересны методы `Security::generatePasswordHash()` и `Security::validatePassword()`.

Функциональные тесты для хэширования паролей

Как мы проверим, что пароль сохранён в БД в зашифрованном виде? После сохранения записи пользователя мы просто проверим, что метод `Security::validatePassword()` вернёт `true` для переданного хэша и текста пароля.

Эта возможность не требует приёмочных тестов сквозь всё приложение. Мы будем использовать функциональные тесты, которые были включены для нас фреймворком Codeception ещё с главы 2, но до сих пор не были использованы.

Для функционального (или, другими словами, интеграционного) тестирования в Codeception существует специальный модуль `Db`, до-

кументация к которому может быть найдена здесь: http://codeception.com/docs/modules/Db. Самая важная его возможность для нас – это тот факт, что перед каждым прогоном тестов он возвращает БД в состояние, описанное в файле `tests/_data/dump.sql`, который ещё не существует. Этот файл SQL должен содержать инструкции о том, как создать схему БД. С базовым набором инструментов MySQL 5 он может быть сделан следующим образом, при условии что ваша БД не требует ввода данных учётной записи и называется `crmapp`:

```
$ mysqldump -d crmapp > tests/_data/dump.sql
```

Флаг `-d` означает «только схема, без данных».

Нам нужно настроить функциональные тесты в файле `tests/functional.suite.yml` следующим образом (строчки, которые нужно вставить, выделены):

```
class_name: FunctionalTester
modules:
   enabled: [Db, Filesystem, FunctionalHelper]
   config:
     Db:
      dsn: 'mysql:host=localhost;dbname=crmapp'
      user: 'root'
      password: 'mysqlroot'
      dump: tests/_data/dump.sql
```

Конечно же, вам нужно вставить свои собственные имя хоста, название БД, имя пользователя и пароль, соответствующие цели развёртывания. Вам также понадобится пересобрать тесты Codeception, используя консольную команду `./cept build`.

Итак, создайте новый функциональный тест следующим образом:

```
$ ./cept generate:test functional PasswordHashing
```

Вот так мы напишем этот тест в только что сгенерированном файле `tests/functional/PasswordHashingTest.php`:

```
   /** @test */
   public function PasswordIsHashedWhenSavingUser()
   {
      $user = $this->imagineUserRecord();

      $plaintext_password = $user->password; //1

      $user->save();
```

```
    $saved_user = UserRecord::findOne($user->id); //2

    $security = new \yii\base\Security();
    $this->assertInstanceOf(get_class($user), $saved_user);
    $this->assertTrue(
       $security->validatePassword( // 3
          $plaintext_password,
          $saved_user->password
       )
    );   }
```

Вся идея выражена в выделенных строчках:

- на строчке 1 модель `$user` ещё не сохранена в БД, и поле `password` всё ещё хранит текст в открытом виде, который мы сохраняем на будущее;
- на строчке 2, после того как `$user` был сохранён в БД, ему присваивается ID. Чтобы имитировать промежуток времени между созданием новой записи о пользователе и входом этого пользователя в систему, мы заново извлекаем запись об этом пользователе из БД;
- на строчке 3 мы проверяем, будет ли совпадать заново созданный хэш пароля, сохранённого в п. 1, с тем, что мы сохранили в базе данных в качестве значения поля `password` для записи о пользователе.

Нас не волнует то, как именно будет создаваться хэш пароля. Всё, что мы хотим знать, – это вернёт ли метод `validatePassword()` значение `true` для заданного хэша и текста пароля.

Мы «воображаем» экземпляр `UserRecord` тем же способом, что и в наших приёмочных тестах управления пользователями:

```
private function imagineUserRecord()
{
   $faker = Faker\Factory::create();

   $user = new UserRecord();
   $user->username = $faker->word;
   $user->password = md5(time());
   return $user;
}
```

Это, очевидно, тоже дублирование кода, которое нужно отрефакторить.

Этот тест, впрочем, ещё не будет выполняться. Если вы запустите его прямо сейчас, он скажет, что не может найти модель `UserRecord`. Это потому, что Codeception ещё не знает о нашем автозагрузчике от Yii 2. Однако структура Codeception включает в себя специальный сценарий, который выполняется до запуска тестов. Он находится в файле `tests/functional/_bootstrap.php`. Давайте внесём все необходимые вызовы `require()` в него:

```
require_once(__DIR__ . '/../../vendor/autoload.php');
require_once(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');

new yii\web\Application(
   require(__DIR__ . '/../../config/web.php')
);
```

К сожалению, мы вынуждены создавать полноценный экземпляр приложения, для того чтобы инициализировать автозагрузчик Yii 2.

После того как сценарий предзагрузки будет помещён на своё место, конфигурация должным образом изменена, и тестовая оболочка Codeception пересобрана, тест хэширования паролей должен, наконец, провалиться с сообщением «Hash is invalid». Настало время на самом деле реализовать хэширование паролей.

Реализация хэширования паролей в Active Record

Чтобы на самом деле реализовать эту возможность, учитывая всё вышесказанное, нам нужен следующий, уже очевидный, метод в классе `UserRecord`:

```
   public function beforeSave($insert)
   {
      $return = parent::beforeSave($insert);

      $this->password = Yii::$app->security->generatePasswordHash($this->password);

      return $return;
   }
```

Похоже, что он не такой уж и очевидный. Выделенная часть – это строчка, которая нам нужна. Строчки до и после неё – код, относящийся к самому фреймворку, предназначенный для сохранения исходного поведения метода `beforeSave()`, определённого для всех экземпляров `ActiveRecord`. Если коротко, то родительская реализация

`beforeSave()` использует концепцию Yii 2 под названием «события» (**Event**). Она инициирует определённое событие, говоря всем, кто слушает его на классе `ActiveRecord`, что эта запись сейчас сохраняется. Смысл в том, что слушатели могут отказать этому событию (конкретно этому, не всякому), и родительский `beforeSave()` вернёт `false` в этом случае, отменяя сохранение модели. Чтобы сохранить это поведение в целости, нам понадобились эти две дополнительные строчки.

Вы можете прочитать больше про события в главе 10 «События», но сейчас мы никак не будем использовать эту систему.

Обратите внимание, что в функциональном тесте мы вручную создали экземпляр класса `Security`. Однако в реальном коде мы используем компонент приложения. В тесте так было сделано, потому что в целом неприемлемо, чтобы тесты зависели от какого-то глобального состояния, которым синглтон `Yii` определённо является. В реальном коде так было сделано, потому что просто неразумно не использовать то, что уже у нас есть. Однако если у вас будет какой-то вручную настроенный компонент приложения под названием `security`, вам придётся из-за этого менять свои тесты.

Теперь у нас есть проблема, уже упомянутая ранее: когда мы пересохраняем экземпляр `UserRecord`, пароль будет заново перехэширован. После этого совершенно неизвестно, как пользователь будет входить в систему, так как ему фактически придётся вводить в форме входа хэш своего изначального пароля.

Следующий тест описывает эту проблему:

```
    /** @test */
    public function PasswordIsNotRehashedAfterUpdatingWithoutChangingPas
sword()
    {
        $user = $this->imagineUserRecord();
        $user->save();

        /** @var UserRecord $saved_user */
        $saved_user = UserRecord::findOne($user->id);
        $expected_hash = $saved_user->password;

        $saved_user->username = md5(time());
        $saved_user->save();

        /** @var UserRecord $updated_user */
        $updated_user = UserRecord::findOne($saved_user->id);
```

```
        $this->assertEquals($expected_hash, $saved_user->password);
        $this->assertEquals($expected_hash, $updated_user->password);
    }
```

Разница между `$saved_user->password` и `$updated_user->password` в проверках в конце этого теста заключается в том, что `$saved_user` на тот момент – это экземпляр `UserRecord`, модифицированный в памяти, а `$updated_user` на тот момент – это экземпляр `UserRecord`, заново извлечённый из обновлённой записи в базе данных. Нам нужно быть уверенными в том, что мы не испортили данных в обоих случаях.

Программировать с изменяемым состоянием *сложно*.

К счастью для нас, в Yii 2 есть средства, интегрированные в активные записи, которые позволяют нам проверить, было ли изменено то или иное поле. У нас есть выбор из следующего:

Метод класса ActiveRecord	Использование
`getOldAttributes()`	Этот метод возвращает массив исходных значений атрибутов текущей активной записи на момент последнего вызова `save()` или `find()`
`getDirtyAttributes($names = null)`	Этот метод возвращает массив всех значений атрибутов, которые были изменены с момента последнего вызова `save()` или `find()`. Фактически он возвращает те результаты `getOldAttributes()`, которые отличаются от результатов вызова `getAttributes()`. Вы можете указать в параметре `$names` конкретные имена атрибутов, которые вам интересны
`isAttributeChanged($name)`	Этот метод проверяет, изменился ли заданный атрибут с момента последнего вызова `save()` или `find()`
`markAttributeDirty($name)`	Этот метод говорит Yii, что данный конкретный атрибут должен считаться изменённым вне зависимости от того, на самом ли деле это так. Таким образом вы можете принудительно пересохранить этот атрибут в БД

Идея в том, что только значения «грязных» (**dirty**) атрибутов сохраняются в БД, когда вы вызываете метод `save()`. Конечно же, когда вы создаёте новую запись, все атрибуты являются «грязными», так что все они будут сохранены.

Метод под названием `isAttributeChanged()` – это в точности то, что нам нужно. Всего лишь добавление одной строчки в метод `beforeSave()` позволяет нашему коду пройти функциональные тесты:

```
public function beforeSave($insert)
{
    ...
    if ($this->isAttributeChanged('password'))
        $this->password = Security::generatePasswordHash($this-
>password);
    ...
}
```

Итак, теперь мы не перехэшируем пароль без необходимости. Административная часть управления пользователями, наконец, завершена.

Превращение UserRecord в Identity

Чтобы иметь применение в функциональности входа в систему, которую мы делаем, наш класс `UserRecord` должен реализовывать интерфейс `yii\web\IdentityInterface`, и он этого ещё не делает. Давайте начнём с объявления `implements yii\web\IdentityInterface` и продолжим с этого места.

Нам нужно реализовать пять методов.

Два метода напрямую относятся к концепции «личности пользователя». Первый – это метод `getId()`, который должен вернуть уникальный идентификатор пользователя. В нашем случае это будет идентификатор записи пользователя в базе данных, так что реализация тривиальна:

```
public function getId()
{
    return $this->id;
}
```

Второй метод – это статический метод `findById()`, который по идентификатору пользователя должен вернуть соответствующую модель. В нашем случае идентификатор пользователя, согласно определению метода `getId()`, – это идентификатор записи пользователя в базе данных. Мы должны вернуть экземпляр класса `UserRecord`, поэтому реализация выглядит следующим образом:

```
public static function findIdentity($id)
{
    return static::findOne($id);
}
```

Два других метода присутствуют в этом интерфейсе для поддержки широко известной функции «запомнить меня». Если мы предоставим эти методы, Yii 2 автоматически реализует эту возможность за нас. Мы не будем тратить времени на тестирование этой возможности, так как она не входит в наши первоначальные цели, но её совсем не сложно реализовать, поэтому давайте просто сделаем это.

Первый метод – это `getAuthKey()`, который должен вернуть некий постоянно хранимый идентификатор (отличающийся от идентификатора пользователя, который мы возвращаем из метода `getId()`), уникально ассоциированный с рассматриваемой личностью пользователя. Единственная сложная деталь в том, что этот «ключ аутентификации» (прямой перевод фразы **Authentication Key**) должен быть постоянно хранимым, для того чтобы Yii смог проверять его корректность между запросами.

Давайте изобретём дополнительный особенный атрибут класса `UserRecord` и будем возвращать его значение в качестве ключа аутентификации.

```
public function getAuthKey()
{
    return $this->auth_key;
}
```

Для того чтобы на самом деле «изобрести» этот атрибут, нам нужно выполнить три шага. Вначале нам нужна миграция, которая добавит соответствующую колонку в таблицу базы данных. Вызов, который нам необходим в самой миграции, должен выглядеть следующим образом:

```
$this->addColumn('user', 'auth_key', 'string UNIQUE');
```

Ограничение UNIQUE необходимо, так как записи пользователей должны быть идентифицируемы по этому полю.

Не забудьте перезаполнить файл `tests/_data/dump.sql` после выполнения этой миграции.

Мы будем заполнять это поле при создании записи и больше никогда не будем его менять. Таким образом, нам нужно добавить следующие строчки в метод beforeSave():

```
    if ($this->isNewRecord)
        $this->auth_key = Yii::$app->security-
>generateRandomKey($length = 255);
```

Значение этого кода должно быть очевидным, но желание прочитать документацию для \yii\db\BaseActiveRecord.$isNewRecord и \yii\base\Security.generateRandomKey() всё равно приветствуется.

Наконец, поле auth_key должно быть добавлено в список правил в методе UserRecord.rules(), вот так:

```
        [['username', 'password', 'auth_key'], 'string', 'max' => 255],
```

На этом наше добавление поля в модель UserRecord завершается.

Второй метод, необходимый для функциональности «запомнить меня», – это метод validateAuthKey(). Он на самом деле проверяет, соответствует ли некий заданный ключ аутентификации текущей личности пользователя. В нашем случае мы можем просто сравнить заданный ключ со значением поля auth_key, сохранённым в БД, следующим образом:

```
public function validateAuthKey($authKey)
{
    return $this->getAuthKey() === $authKey;
}
```

Последний, пятый метод интерфейса IdentityInterface, который необходимо реализовать, – это метод findIdentityByAccessToken(). По заданному токену доступа (прямой перевод фразы **access token**) он должен вернуть (в нашем случае) экземпляр UserRecord напрямую, возможно, совершив некоторые внутренние проверки. Природа этого токена не определена, и этот метод, очевидно, полезен в случае аутентификации средствами наподобие OAuth2 или OpenID. Мы не собираемся использовать эти возможности, так что давайте просто сделаем заглушку:

```
public static function findIdentityByAccessToken(
    $token,
    $type = null
) {
    throw new NotSupportedException(
        'You can only login by username/password pair for now.');
}
```

Эти пять методов: `getId()`, `findIdentity($id)`, `getAuthKey()`, `validateAuthKey($authKey)` и `findIdentityByAccessToken($token)` – используются внутри механики входа в систему, реализованной в Yii 2. Скорее всего, вам никогда не придётся вызывать их в своём клиентском коде.

После того как класс `UserRecord` стал представлять личность пользователя, вам нужно объявить его в настройке `components.user.identityClass` в конфигурации приложения. Для нашего примера приложения эта настройка будет выглядеть следующим образом:

```
'user' => [
   'identityClass' => 'app\models\user\UserRecord'
]
```

Создание интерфейса входа в систему

Наконец-то, нам нужно реализовать возможность, которая интересовала нас с самого начала: аутентификация пользователя. В первую очередь нам нужно решить, что это значит для пользователя – быть аутентифицированным.

В следующей главе, посвящённой авторизации пользователя, мы будем проверять, дозволено ли пользователю то или иное действие в системе. Но пока что мы не интересуемся авторизацией – только лишь аутентификацией, то есть *некоторым оповещением о том, что система опознала пользователя.*

Спецификация аутентификации пользователя

Давайте опишем действия по аутентификации пользователя:

- На каждой странице приложения в верхнем правом углу расположен индикатор входа пользователя в систему.
- Пока пользователь не аутентифицирован, индикатор содержит строчку **«guest»** («гость») и ссылку под названием **«login»** («вход»).
- Когда пользователь аутентифицирован, индикатор содержит имя пользователя (значение поля `username` его записи) и ссылку под названием **«logout»** («выход»).
- Щелчок по **«login»** ведёт на страницу формы входа в систему.
- Щелчок по **«logout»** деаутентифицирует пользователя и перенаправляет его на стартовую страницу приложения.

Всё это очень сложно полностью покрыть примерами использования и приёмочными тестами. Так как нашей целью является изучение

возможностей Yii 2, которые помогают нам реализовать нужные нам вещи, давайте остановимся на одном конкретном приёмочном тесте:

```
./cept generate:cept acceptance LoginAndLogout
```

Вот его полное содержимое, переведённое из вышеописанного высокоуровневого описания:

```
$I = new AcceptanceTester\CRMUsersManagementSteps($scenario);
$I->wantTo('check that login and logout work');

$I->amGoingTo('Register new User');

$I->amInListUsersUi();
$I->clickOnRegisterNewUserButton();
$I->seeIAmInAddUserUi();
$user = $I->imagineUser();
$I->fillUserDataForm($user);
$I->submitUserDataForm();

$I = new AcceptanceTester\CRMUserSteps($scenario);
$I->amGoingTo('login');

$I->seeLink('login');
$I->click('login');
$I->seeIAmInLoginFormUi();
$I->fillLoginForm($user);
$I->submitLoginForm();

$I->seeIAmAtHomepage();
$I->dontSeeLink('login');
$I->seeUsername($user);
$I->seeLink('logout');

$I->amGoingTo('logout from arbitrary page');
$I->amInQueryCustomerUi();
$I->click('logout');

$I->seeIAmAtHomepage();
$I->dontSeeUsername($user);
$I->dontSeeLink('logout');
$I->seeLink('login');
```

Вначале мы создаём пользователя, затем мы пробуем войти в систему и смотрим, отражает ли индикатор состояние аутентификации.

Потом мы пробуем выйти из системы и проверяем, вернулся ли индикатор в состояние, предназначенное для гостей.

Выделенные строчки – это шаги, которые ещё не были определены. Их реализация достаточно прямолинейна.

Проверка на то, что мы находимся в пользовательском интерфейсе формы входа:

```
public function seeIAmInLoginFormUi()
{
    $I = $this;
    $I->seeCurrentUrlEquals('/site/login');
}
```

Заполнение формы входа в систему сгенерированными ранее данными:

```
public function fillLoginForm($user)
{
    $I = $this;
    $I->fillField('LoginForm[username]', $user['UserRecord[username]']);
    $I->fillField('LoginForm[password]', $user['UserRecord[password]']);
}
```

Обратите внимание, что только что показанная функция – это точная копия подобного метода «заполнить форму» в `AcceptanceTester\CRMUsersManagementSteps`. Мы снова пропускаем шаг рефакторинга.

Отправка формы входа:

```
public function submitLoginForm()
{
    $I = $this;
    $I->click('button[type=submit]');
    $I->wait(1);
}
```

Как мы видели в *главе 3 «Автоматическая генерация кода»*, форма входа содержит валидацию данных на стороне клиента, поэтому мы вынуждены ждать.

Проверка на то, что мы на стартовой странице, то есть на маршруте `/`:

```
public function seeIAmAtHomepage()
{
    $I = $this;
    $I->seeCurrentUrlEquals('/');
}
```

Проверка того, что где-то на странице есть имя пользователя, упомянутое среди переданных данных о пользователе:

```
public function seeUsername($user)
{
   $I = $this;
   $I->see($user['UserRecord[username]']);
}
```

Проверка того, что мы *не* видим вышеупомянутое:

```
public function dontSeeUsername($user)
{
   $I = $this;
   $I->dontSee($user['UserRecord[username]']);
}
```

Этот тест будет провален на шаге, где мы пробуем щёлкнуть по ссылке под названием **login**. Давайте добавим его в шаблон.

Создание индикатора аутентификации

Как было указано, индикатор должен показывать текст **guest** и ссылку **login** для неопознанных пользователей, и имя пользователя и ссылку **logout** – для опознанных.

Мы можем проверить, является ли пользователь гостем, применяя следующий вызов:

```
Yii::$app->user->isGuest
```

Таким образом, следующий HTML-код в файле шаблона `views/layouts/main.php`, сразу после открывающего тэга `<div class="container">`, нам подойдёт:

```
<div class="authorization-indicator">
   <?php if (Yii::$app->user->isGuest):?>
      <?= Html::tag('span', 'guest');?>
      <?= Html::a('login', '/site/login');?>
   <?php else:?>
      <?= Html::tag('span', Yii::$app->user->identity->username);?>
      <?= Html::a('logout', '/site/logout');?>
   <?php endif;?>
</div>
```

Теперь, при условии что у нас есть `ApplicationUiAssetBundle`, описанный в *главе 4 «Рендерер»*, мы можем добавить следующий фрагмент

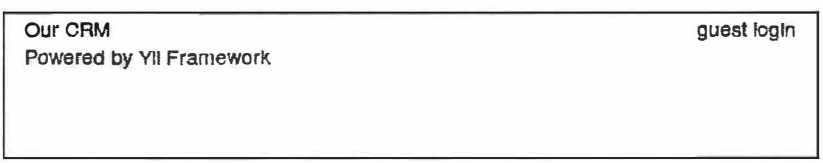
CSS-кода в файл assets/ui/css/main.css, чтобы выровнять индикатор по правому краю:

```
.authorization-indicator {
   float: right;
   width: 25%;
   text-align: right;
}
```

Это довольно грубое решение, но пока наш пользовательский интерфейс и так рудиментарен, оно сойдёт.

Функциональность формы входа

Вот как должна выглядеть стартовая страница, когда мы откроем её в браузере (без темы, введённой в предыдущей главе):



Теперь нам нужно реализовать функциональность формы входа. Как мы уже упомянули в двух местах в приёмочных тестах, маршрут для входа – /site/login, так что нам нужно предоставить метод SiteController.actionLogin(). Начиная с этого места, мы сделаем каноническую реализацию формы входа, которую можно увидеть в продвинутом шаблоне приложения от Yii 2. Каким-то другим способом сделать аутентификацию по паролю в Yii достаточно сложно, да и бессмысленно.

Вот логика для традиционного обработчика маршрута /site/login:

```
public function actionLogin()
{
   if (!\Yii::$app->user->isGuest)
      return $this->goHome();

   $model = new LoginForm();
   if ($model->load(Yii::$app->request->post()) and $model->login())
      return $this->goBack();

   return $this->render('login', compact('model'));
}
```

Если пользователь уже аутентифицирован, мы перенаправляем его обратно на стартовую страницу, используя вспомогательный метод `\yii\web\Controller.goHome()`, который делает HTTP-перенаправление со статусом 302 на маршрут `/`.

Если нам передали некоторые данные через POST-запрос, мы пробуем методом `LoginForm.login()` проверить, достаточно ли этих данных, для того чтобы аутентифицировать пользователя. В случае успеха мы перенаправляем пользователя на последний посещённый им URL при помощи вспомогательного метода `goBack()`.

Иначе рендерим HTML-код формы входа.

Самая интересная часть – в подсвеченных строчках. Мы используем здесь возможности, предоставляемые нам моделями. Используя одну и ту же модель, мы можем как аутентифицировать пользователя, так и собрать HTML-форму для ввода логина и пароля.

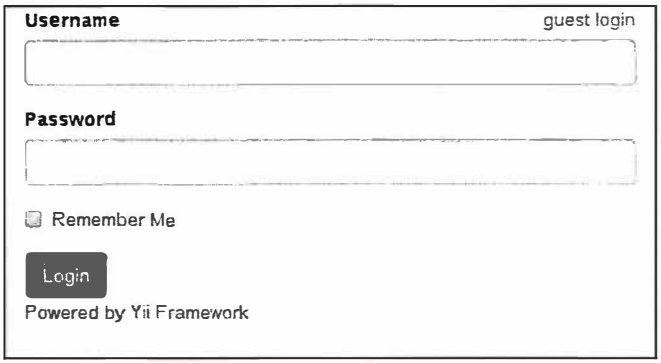
Начнём с внешнего вида формы входа в файле представления `views/site/login.php`. Вот как выглядит типичная минимальная форма входа:

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin(['id' => 'login-form']);
echo $form->field($model, 'username');
echo $form->field($model, 'password')->passwordInput();
echo $form->field($model, 'rememberMe')->checkbox();
echo Html::submitButton(
   'Login',
   ['class' => 'btn btn-primary', 'name' => 'login-button']
);
ActiveForm::end();
```

Она содержит привычный вызов виджета `ActiveForm` и три вызова для отрисовки различных видов полей ввода. Обратите внимание на стиль, в котором написаны вызовы методов отрисовки полей ввода. Вы можете узнать подробности из документации и/или исходного кода метода `ActiveForm.field()`.

У нас также есть дополнительная «галочка» для функциональности «запомнить меня», которую мы уже приготовили ранее в этой главе.

Вышеописанная форма выглядит следующим образом (с нашим обновлённым шаблоном):



Впрочем, вы её не увидите до тех пор, пока у нас не появится класс `LoginForm`. Он должен быть классом модели, но только не `ActiveRecord`, а просто `yii\base\Model`.

У этой модели – три поля, как и ожидалось, судя по файлу представления. Для наших нужд нам достаточно предоставить правила валидации данных и метод `login()`, который мы используем в действии контроллера `SiteController.actionLogin()`.

Вот фреймворк модели `LoginForm`, размещённый в файле `models/user/LoginForm.php`:

```
<?php
namespace app\models\user;

use yii\base\Model;

class LoginForm extends Model
{
   public $username;
   public $password;
   public $rememberMe;
}
```

Нам нужны три правила валидации данных:

1. И имя пользователя, и пароль обязательны.
2. Опция «запомнить меня» – булево значение (установлено/не установлено).
3. Пароль должен быть верным, то есть если есть запись о пользователе с данным именем, то данный пароль должен иметь тот же хэш, что и тот, который сохранён для этого пользователя (как было описано ранее).

Это может быть выражено в классе LoginForm следующим образом:

```
public function rules()
{
    return [
        [['username', 'password'], 'required'],
        ['rememberMe', 'boolean'],
        ['password', 'validatePassword']
    ];
}
```

Если валидатор, который нужно применить, не является одним из встроенных, мы должны хотя бы предоставить метод в этом классе модели с тем же именем, что и валидатор. Второй вариант – создать полноценный преднастроенный класс-наследник Validator и зарегистрировать его в конфигурации приложения. Сейчас нас вполне устроит валидатор в виде метода класса. Таким образом, мы создаём метод validatePassword(), и выглядит он следующим образом:

```
public function validatePassword($attributeName)
{
    if ($this->hasErrors())
        return;

    $user = $this->getUser($this->username);
    if (!($user and $this->isCorrectHash($this->$attributeName,
$user->password)))
        $this->addError('password', 'Incorrect username or password.');
}
```

Защитное выражение в начале метода стандартно: если уже есть зафиксированные ошибки, ничего не делать. Затем мы пытаемся найти запись о пользователе по предоставленному имени пользователя. Если такая запись не существует или предоставленный пароль не соответствует хэшу, сохранённому для этой записи, мы добавляем ошибку к этой модели, что и должен делать правильный валидатор.

Получение модели UserRecord по значению поля username можно сделать любым способом, но ради чистого красивого решения мы решили использовать ленивую загрузку:

```
/** @var UserRecord */
public $user;

private function getUser($username)
{
```

```
    if (!$this->user)
        $this->user = $this->fetchUser($username);

    return $this->user;
}

private function fetchUser($username)
{
    return UserRecord::findOne(compact('username'));
}
```

Для проверки корректности предоставленного пароля достаточно следующей реализации:

```
private function isCorrectHash($plaintext, $hash)
{
    return Yii::$app->security->validatePassword($plaintext, $hash);
}
```

Это – самая суть нашей аутентификации пользователя. На самом деле всё до этого места было просто поддерживающим фреймворком кода. Всё, что мы действительно хотели знать, – вернёт ли вызов метода `Security::validatePassword()` значение `true` для заданного текста пароля и хэша, сохранённого в БД.

Наконец, метод `login()` из `LoginForm`, который используется в действии контроллера `SiteController.actionLogin()`, завершает логику формы входа:

```
public function login()
{
    if (!$this->validate())
        return false;

    $user = $this->getUser($this->username);
    if (!$user)
        return false;

    return Yii::$app->user->login(
        $user,
        $this->rememberMe ? 3600 * 24 * 30 : 0
    );
}
```

Если значения полей не валидны, очевидно, вход запрещаем. Если мы не можем найти пользователя по данному имени, очевидно, вход

запрещаем. Под конец вызываем встроенный в Yii метод для входа пользователя и передаём найденную модель `UserRecord` в неё в качестве «личности». Второй параметр – это время, в течение которого следует держать сеанс пользователя активным (в секундах). Если значение равно нулю, сеанс будет активным до закрытия окна браузера. Если была поставлена галочка «запомнить меня», мы в данном случае будем держать пользователя аутентифицированным в течение месяца.

Функциональность выхода из системы и подведение итогов

Теперь нам нужен только обработчик выхода из системы, который может быть сделан практически в одну строчку в классе `SiteController`:

```
public function actionLogout()
{
    Yii::$app->user->logout();
    return $this->goHome();
}
```

Этот метод является завершением функциональности, которую мы всё это время собирали. Теперь выполняем приёмочный тест:

```
./cept run tests/acceptance/LoginAndLogoutCept.php
```

И он должен завершиться успешно, как показано на следующем снимке экрана:

```
O_O [master*] $ ./cept run tests/acceptance/LoginAndLogoutCept.php
Codeception PHP Testing Framework v2.0.0-alpha
Powered by PHPUnit 3.7.32-1-g316a547 by Sebastian Bergmann.

Acceptance Tests (1) ------------------------------------------------------------
Trying to check that login and logout work (LoginAndLogoutCept.php)        Ok
---------------------------------------------------------------------------------

Time: 4.78 seconds, Memory: 12.75Mb

OK (1 test, 11 assertions)
```

Итоги

Вход в систему по имени пользователя и паролю неудобно использовать и скучно писать. В этой форме аутентификации задействовано множество концепций. На самом деле мы повторили то, что уже по-

ставляется с расширенным шаблоном приложения Yii 2, возможно, с иным стилем написания кода. Однако если бы вы просто посмотрели на исходный код расширенного шаблона, скорее всего, вы бы потратили много времени на попытки понять, какие части, в виде методов классов, переменных, классов и файлов представлений, используются в функциональности входа в систему и какую роль каждая из них играет.

Выполнив это упражнение, мы мельком взглянули на такие приёмы разработки с Yii 2 и PHP в целом, как:

- написание чего-либо в файле представления в зависимости от того, аутентифицирован пользователь или нет;
- создание моделей, не являющихся активными записями, и использование их для генерации форм ввода данных;
- метод `ActiveRecord.beforeSave()`;
- вспомогательный класс `yii\base\Security`, для того чтобы не возиться с деталями безопасной генерации хэшей строк;
- валидаторы данных в виде методов классов моделей;
- ленивая загрузка объектов;
- понятие «грязных», то есть изменённых, атрибутов активных записей;
- возможно, что-то ещё, что сложно выделить.

Снова напомним: *конечно же*, есть и иные методы аутентификации. Протокол OAuth2, OpenID, физические токены безопасности, коды в SMS, что угодно. Мы не могли покрыть все эти темы, так что здесь был объяснён самый распространённый метод, который вас, скорее всего, когда-нибудь всё равно попросят реализовать.

Однако не стоит забывать, что мы рассмотрели только половину механики управлением доступа. Нам всё ещё нужно реализовать авторизацию пользователя после того, как он был опознан. Это станет темой следующей главы.

Глава 6

Авторизация пользователей и контроль доступа

Аутентификация пользователей – это только половина дела. Даже если вы разрабатываете веб-сайт какого-то сетевого сообщества или, прости, Господи, социальную сеть и идентификация пользователей – это важнейшая часть ваших бизнес-правил, вам практически всегда нужно контролировать, какие пользователи какую часть функциональности приложения имеют право использовать. Можно сказать, что нет смысла в аутентификации без идущего с ней в комплекте контроля доступа, то есть **авторизации пользователей**.

В этой главе мы посмотрим, как Yii 2 может помочь нам запретить или разрешить доступ пользователей к функциональности веб-приложения. Мы сосредоточимся на следующих четырёх возможностях Yii:

- использование методов-перехватчиков в контроллере;
- обработка исключений;
- фильтры действий контроллеров;
- контроль доступа, основанный на ролях.

В качестве примера кода мы реализуем в нашем примере CRM-приложения простую схему контроля доступа, основанную на ролях.

Контроль доступа с использованием состояния аутентификации пользователя

В предыдущей главе мы уже видели самый рудиментарный контроль доступа, мы писали код отрисовки индикатора аутентификации следующим образом:

```
if (Yii::$app->user->isGuest)
   // индикатор для гостей
else
   // индикатор для аутентифицированных пользователей
```

Таким образом, мы способны менять содержимое в зависимости от того, опознан пользователь или нет.

Мы можем не только менять содержимое, но и полностью запрещать посетителям открывать какие-либо маршруты без аутентификации. Для этого нам нужно узнать про две возможности Yii: обработку исключений и методы-перехватчики у класса контроллера.

Возможность: методы-перехватчики у класса контроллера

Подобно методу `beforeSave()` класса `ActiveRecord`, который мы использовали в предыдущей главе, классы-наследники `Controller` тоже имеют методы-перехватчики. Вот они:

Название метода	Что он делает
`beforeAction($action)`	Выполняется перед тем, как происходит действие контроллера. Так же, как и `ActiveRecord.beforeSave()`, он должен вернуть булево значение, показывающее, позволено ли действию выполниться
`afterAction($action, $result)`	Выполняется после того, как действие контроллера закончит выполняться, но до того, как результат будет послан в `ViewRenderer`. На самом деле аргумент `$result` – это и есть результат выполнения действия, и мы можем с ним что-нибудь сделать. Конечно же, мы должны вернуть этот результат, иначе клиенту по данному маршруту ничего не будет отображено

Оба этих перехватчика принимают аргумент `$action`, который представляет собой действие контроллера, выполняющееся в данный момент. Этот объект на самом деле не очень-то полезен, так как он всего лишь контейнер для следующих вещей:

- идентификатор действия, который может быть полезен для ультраточного контроля доступа на основе идентификаторов действий;
- ссылка на контроллер, которому принадлежит это действие;
- метод `run()`, ради которого этот объект действия и существует, так как именно этот метод и является обработчиком маршрута, который представляет это действие.

Заметьте, что в случае, если `beforeAction()` вернёт значение `false`, браузер клиента получит *пустую страницу безо всяких объяснений*. Такое на самом деле может довольно часто случаться в процессе написания этих перехватчиков, если вы регулярно забываете вернуть из них булево значение. То же самое произойдёт, если вы забудете вернуть `$result` из `afterAction()`. В целом гораздо безопаснее использовать систему событий, описанную в главе 10, но в ней применяется больше элементов и прямая связь между вызовами функций исчезает, что, возможно, затруднит чтение кода.

Хотя это и определённо интересно – чем же на самом деле является действие контроллера, – пока что вышеописанное краткое объяснение полностью нам подходит. Детальный обзор того, какую роль играет класс `Action` в обработке маршрутов, будет сделан в главе 12. Большую часть времени нет причин беспокоиться об этом.

Все возможные перехватчики, как они работают и как мы можем их использовать себе на пользу, перечислены в *главе 10 «События»*.

Исходя из определения метода `beforeAction()`, становится очевидным, каким наипростейшим образом мы можем ограничить доступ к некоторым частям приложения:

```
public function beforeAction($action)
{
   $parentAllowed = parent::beforeAction($action);
   $meAllowed = !Yii::$app->user->isGuest;
   return $parentAllowed and $meAllowed;
}
```

Вначале мы проверяем, допускает ли родительская реализация выполнение данного действия. Затем проверяем, опознан ли текущий пользователь, запрашивая значение свойства `isGuest` компонента `user`. Если оба этих условия истинны, тогда мы разрешаем выполнение действия.

Какие проверки совершает базовая реализация метода `yii\web\Controller.beforeAction()`? Она защищает нас от атак межсайтовой подделки запроса (**Cross-Site Request Forgery**, CSRF). В Yii 2 каждая форма ввода, созданная виджетом `ActiveForm`, по умолчанию включает в себя специальный токен в качестве дополнительного параметра запроса. Запрос будет обработан только в том случае, когда токен, ожидаемый от формы на сервере, равен тому, что передано в параметрах запроса. Конечно же, эта проверка не нужна (да и не делается) для запросов GET, HEAD и OPTIONS (последний встречается крайне редко). Вы также можете отключить эту проверку (не делайте этого) через настройку `enableCsrfValidation` в конфигурации самого приложения.

В главном файле шаблона, который мы сделали в *главе 3* и подробно обсудили в *главе 4*, находится вызов метода `Html::csrfMetaTags()`, о котором мы тогда даже не упомянули. Этот вызов находится там именно для поддержки защиты от CSRF. Более того, если настройка `enableCsrfValidation` имеет значение `true`, вы обязаны вызвать этот метод или потеряете возможность отправлять формы, сгенерированные виджетом `ActiveForm`.
Обычно вы не сломаете ничего особенного, если не вызовете `parent::beforeAction()`, но лучше вам так не делать, потому что если вы забудете вызвать родительскую реализацию `beforeAction()` в действии, которое принимает данные от веб-формы, то потеряете важную проверку безопасности.

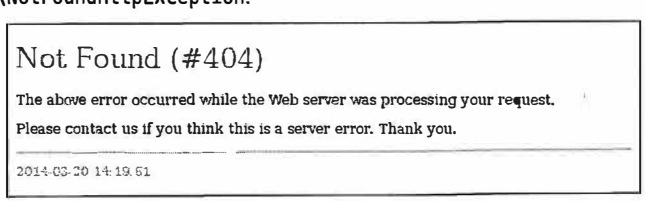
Однако, как уже было упомянуто выше, этот способ блокировки не очень-то интересен. Во-первых, он не показывает клиенту никакой обратной связи о том, что на самом деле произошло. Во-вторых, такое использование метода `beforeAction()` уже предусмотрено в Yii 2 в форме специальной концепции под названием «фильтры» **(Filters)**.

Давайте пока отложим объяснение фильтров и взглянем на то, как мы можем показать в браузере клиента, что на самом деле случилось с его запросом.

Обработка исключений в Yii 2

Yii использует свой собственный обработчик исключений, который на самом деле годится к употреблению без каких-либо дальнейших улучшений. Если коротко, то если вы бросите исключение любого класса, наследующего класс `Exception`, в каком-нибудь действии контроллера и не перехватите его, Yii покажет клиенту следующее:

Однако вот что он покажет, если вы бросите исключение класса `yii\web\NotFoundHttpException`:

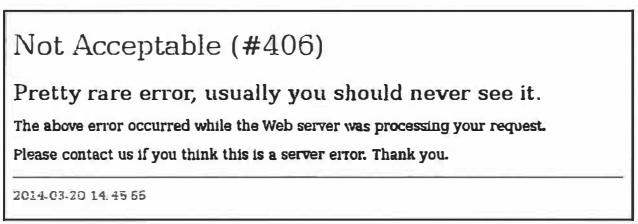


Заметьте, что он корректно показывает код состояния и сообщение об ошибке, согласно спецификации кодов состояния HTTP. Отклик сервера тоже будет на самом деле начинаться с кода 404.

Действительно, `yii\web\NotFoundHttpException` – это обёртка вокруг базового класса исключений `yii\web\HttpException`, которые вы можете бросать для того, чтобы выдавать клиенту произвольные коды состояния и сообщения, например так:

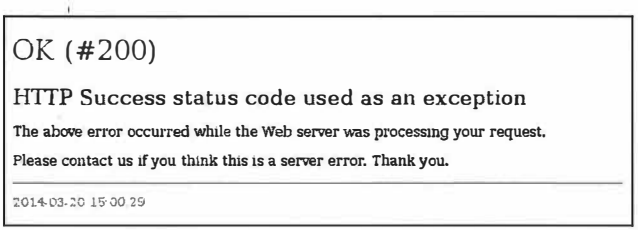
```
throw new HttpException(406, 'Pretty rare error, usually you should
never see it.');
```

Не забудьте написать `use yii\web\HttpException` вверху вашего скрипта. Вот что будет показано в результате:



Заметьте, что сообщение, которое мы указали в конструкторе исключения, показано под основным сообщением о коде состояния 406, которое автоматически создано, согласно спецификации HTTP, и неизменно. Сервер при этом на самом деле вернёт код состояния 406.

Итак, вы можете блокировать запрос пользователя с пояснительным сообщением, просто бросая различные классы-наследники `\yii\web\HttpException`. Однако обратите внимание на то, что если вы будете использовать *сам* этот базовый класс вместо его специализированных подклассов, то ничто не мешает кому-то сделать следующее:



Определённо, любой «подумает, что это ошибка сервера». Несмотря на то что Yii на самом деле вернёт код состояния 200, ответом будет

страница с ошибкой. В случае если кто-то, например, решит бросить `HttpException` с кодом состояния 302, сервер на самом деле вернёт код состояния 302, но никакого перенаправления не произойдёт. Это всегда просто статическая страница ошибки.

Вот список всех наследников `HttpException`, которые Yii 2 уже содержит для нашего удобства:

Исключение	Передаваемый код состояния HTTP
`BadRequestHttpException`	400 Bad Request
`ConflictHttpException`	409 Conflict
`ForbiddenHttpException`	403 Forbidden
`GoneHttpException`	410 Gone
`MethodNotAllowedHttpException`	405 Method Not Allowed
`NotAcceptableHttpException`	406 Not Acceptable
`NotFoundHttpException`	404 Not Found
`TooManyRequestsHttpException`	429 Too Many Requests (from RFC 6585 Additional HTTP Status Codes, see http://tools.ietf.org/html/rfc6585)
`UnauthorizedHttpException`	401 Unauthorized
`UnsupportedMediaTypeHttpException`	415 Unsupported Media Type

Ещё одна хорошая особенность обработки исключений в Yii – это сообщения об ошибках, когда включен режим отладки. Чтобы его включить, нужно в файле точки входа `index.php`, до вызова `require()` самой библиотеки Yii, определить константу `YII_DEBUG` со значением `true`:

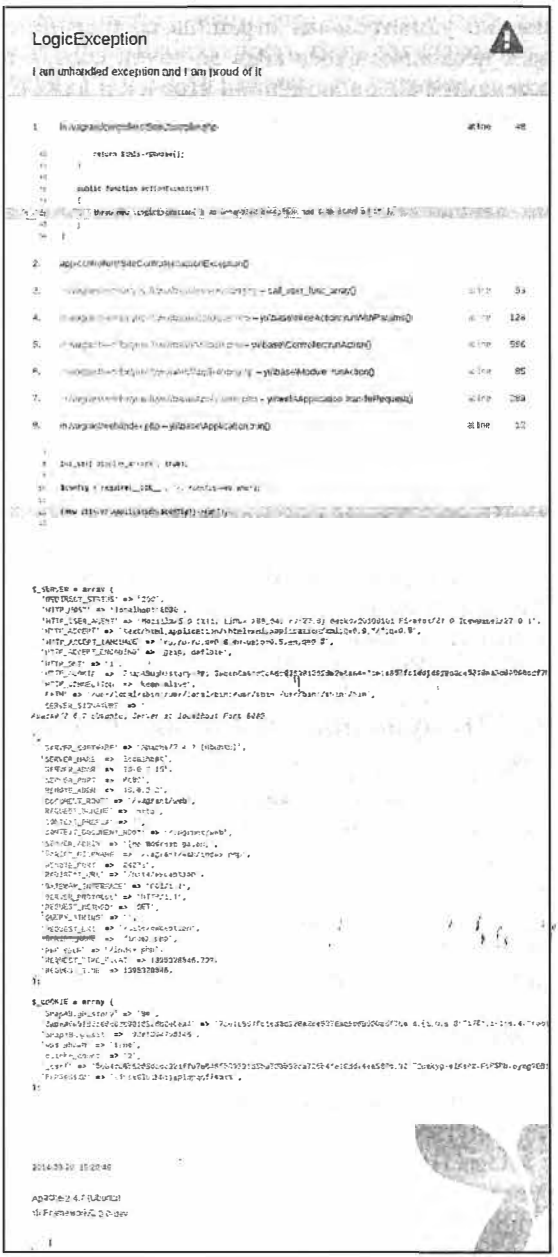
```
define('YII_DEBUG', true);
```

Совершенно необходимо определить эту константу до загрузки библиотеки Yii, так как в случае отсутствия определения этой константы Yii самостоятельно определит её со значением `false`. Это можно интерпретировать как то, что по умолчанию все приложения Yii работают в режиме реального использования. Кроме включения подробного отчёта об ошибках, `YII_DEBUG` делает ещё некоторые специфичные вещи, нам сейчас неинтересные.

Если теперь в программе появится необработанное исключение, отличающееся от `HttpException`, например это:

```
throw new \LogicException('I am unhandled exception and I am proud of it');
```

тогда вот что будет показано вместо краткой страницы ошибки, показанной ранее:


LogicException
I am unhandled exception and I am proud of it

Это совершенно удивительная страница сообщения об ошибке. Она показывает несколько слоёв кода до точки сбоя, с подсветкой синтаксиса, именами файлов, номерами строчек и даже *ссылками на страницы документации для упомянутых классов и методов Yii* (!). Она показывает действующую конфигурацию сервера. Она показывает «печеньки» (**cookies**), отправленные клиентом.

> Ни в коем случае эта страница не должна быть показана посетителям в реальном рабочем окружении. Всегда устанавливайте значение константы `YII_DEBUG` в `false` в этом случае.

Итак, вот защитное выражение, которое запрещает неопознанным пользователям использовать действие контроллера, в котором оно находится:

```
if (Yii::$app->user->isGuest)
   throw new ForbiddenHttpException;
```

Это простейший уровень контроля доступа, доступный для нас в Yii 2. Он настолько фундаментален, что в Yii есть специальный встроенный фильтр `AccessControl` для этого. Мы познакомимся с ним позже, а пока давайте узнаем, что это вообще такое – фильтры (**Filter**) – в терминологии Yii.

ВОЗМОЖНОСТЬ: фильтры действий контроллеров

Фильтр действий, если вкратце, – это методы `beforeAction()` и `afterAction()`, запакованные в один класс и подключаемые к экземплярам классов `Controller` через метод `Controller.behaviors()`.

Вот преимущества такого подхода:

- вы можете определять защитные выражения, пост- и предобработку произвольной длины и сложности, так как теперь у вас есть целый класс для этого;
- вы можете комбинировать различные фильтры в любых комбинациях и порядке.

Для того чтобы класс стал фильтром действий для некоего контроллера, ему необходимо:

- быть наследником класса `ActionFilter`;
- быть упомянутым в методе `behaviors()` этого контроллера.

Сам класс `ActionFilter` является особым случаем класса `Behavior`, который мы рассмотрим в главе 10. Пока что достаточно знать, что «поведение» (прямой перевод термина **Behavior**) – это некоторый класс, содержащий некоторые методы, которые могут быть объявле-

ны неотъемлемой частью некоторого другого класса, расширяя, таким образом, его функциональность. Если вы знакомы с концепцией «черт» (**traits**) из PHP 5.4, то вот это – то же самое.

Мы не будем подробно останавливаться на том, как создать свой собственный `ActionFilter`, так как это довольно тривиально, учитывая то, что мы уже знаем, как работают методы `beforeAction()` и `afterAction()`. Давайте вместо этого бегло пройдёмся по фильтрам, встроенным в Yii 2. Подробная информация об их использовании может быть прочитана из соответствующих страниц документации.

Класс фильтра	Что он делает
`\yii\filters\VerbFilter`	Запрещает или разрешает доступ к действиям контроллеров в зависимости от метода HTTP, использованного для запроса. Например, используя этот фильтр, вы можете разрешать только POST-запросы для ваших действий входа и выхода из системы. Это единственный фильтр, который наследует не от класса `ActionFilter`, а напрямую от класса `Behavior`. Отвечает сообщением 405 Method Not Allowed, если действие не было разрешено. Подробности на странице http://www.yiiframework.com/doc-2.0/yii-filters-verbfilter.html
`\yii\filters\PageCache`	Кэширует результат отрисовки действий данного контроллера. Имеет достаточно сложную и гибкую систему настроек для управления тем, какие именно действия, где и насколько должны быть сохранены. Подробности на странице http://www.yiiframework.com/doc-2.0/yii-filters-pagecache.html
`\yii\filters\HttpCache`	Этот фильтр, по сути, эквивалентен фильтру `PageCache`, но использует заголовки HTTP **Last-Modified** и **Etag** для «кэширования». В результате за хранение и показ кэшированной версии страницы будет отвечать браузер клиента, а не сервер. Подробности на странице http://www.yiiframework.com/doc-2.0/yii-filters-httpcache.html
`\yii\filters\AccessControl`	Предотвращает или разрешает доступ к действиям на основе набора правил, используя очень гибкий и выразительный синтаксис. Этот фильтр настолько мощный, что часто не нужно больше реализовывать вообще никакой контроль доступа в контроллере. Этот фильтр может управлять доступом на основе метода HTTP, состояния аутентификации пользователя, роли пользователя, идентификатора действия и/или контроллера, IP-адреса посетителя или даже по предоставленному произвольному анонимному методу. Отвечает кодом состояния 403 Forbidden, если блокирует действие. Подробности на странице http://www.yiiframework.com/doc-2.0/yii-filters-accesscontrol.html, но мы всё равно будем обсуждать этот фильтр в данной главе

Класс фильтра	Что он делает
\yii\filters\ ContentNegotiator	Очень удобный фильтр для больших приложений. Он автоматически меняет язык приложения и формат отклика на основе заголовков HTTP и параметров запроса, полученных от клиентского приложения. Подробности на странице http://www.yiiframework.com/doc-2.0/yii-filters-contentnegotiator.html
\yii\filters\RateLimiter	Запрещает доступ пользователям, превысившим свою частоту запросов страниц за определённый промежуток времени. Для поддержки данной функциональности ваш класс личности пользователя (про который рассказывала предыдущая глава) должен реализовать \yii\filters\RateLimitInterface. Подробности на странице http://www.yiiframework.com/doc-2.0/yii-filters-ratelimiter.html

Так как это глава про контроль доступа, давайте посмотрим на некоторые примеры использования `VerbFilter` и `AccessControl`. `AccessControl` настолько важен, что позже мы ещё раз на него посмотрим, после того как исследуем систему ролей пользователя **(user roles)**.

Вот как Gii защищает контроллеры, которые генерирует:

```
public function behaviors()
{
    return [
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'delete' => ['post'],
            ],
        ],
    ];
}
```

Обратите внимание на вызов `VerbFilter::className()`. Это идиома Yii 2, которая позволяет нам легко получить полностью определённое имя любого класса, наследующего базовому классу `yii\base\Object`, и код самого Yii 2 широко использует эту идиому. В данном показанном случае этот вызов всегда будет возвращать строку `\yii\filters\VerbFilter`.

Метод `behaviors()` должен возвращать массив конфигураций фильтров, которые нужно присоединить к контроллеру. В данном случае мы присоединяем фильтр `VerbFilter` по произвольно выбранному ключу `verbs`. Этот `VerbFilter` настроен таким образом, что действие `delete` может быть доступно только POST-запросом. Все остальные

действия остаются без дополнительной защиты, так как даже форма редактирования модели как принимает данные, отправленные через POST-запрос, так и отрисовывает HTML-страницу с этой формой.

Вот как мы можем (довольно наивным образом) защитить наши действия `login` и `logout`, используя фильтр `AccessControl`:

```
public function behaviors()
{
    return [
        'access' => [ // 1
            'class' => AccessControl::className(), // 2
            'only' => ['login', 'logout'], // 3
            'rules' => [
                [
                    'actions' => ['login'], // 4
                    'roles' => ['?'], // 5
                    'allow' => true, // 6
                ],
                [
                    'actions' => ['logout'], // 7
                    'roles' => ['@'], // 8
                    'allow' => true, // 9
                ]
            ]
        ]
    ];
}
```

Здесь мы также используем метод `behaviors()` нашего контроллера, но присоединяем класс `\yii\filters\AccessControl` с дополнительно определённой конфигурацией.

Давайте прочитаем вышеуказанные настройки строчка за строчкой:

1. Под произвольно выбранным названием `access`.
2. Мы регистрируем фильтр `AccessControl`.
3. Только для действий `login` и `logout`, которые соответствуют методам `actionLogin()` и `actionLogout()` контроллера. По умолчанию фильтр контроля доступа запрещает всё, что явно не было разрешено, поэтому мы вынуждены здесь его ограничить.
4. Для действия `login`.
5. Для неопознанных пользователей. Символ ? означает «гости».
6. Позволить запрос.
7. Для действия `logout`.

8. Для опознанных пользователей. Символ `@` означает «аутентифицированные пользователи».
9. Позволить запрос.

Конкретные параметры настройки для каждого правила можно прочитать в документации и/или исходном коде класса `yii\filters\AccessRule` (см. http://www.yiiframework.com/doc-2.0/yii-filters-accessrule.html).

Контроль доступа на основе ролей

Yii 2 использует предельно простую форму контроля доступа пользователей на основе **разрешений (permissions)**. Когда она действует, каждому пользователю может быть выдано разрешение (точный механизм сейчас не важен). Когда нужно, приложение может проверить, имеет ли пользователь это разрешение, вызовом:

```
Yii::$app->user->can($permission);
```

Здесь аргументом `$permission` является строка, которая представляет собой название этого разрешения. И всё.

Есть две важные дополнительные возможности, предоставляемые концепцией разрешений:

- разрешение может быть объявлено «дочерним» **(child)** другому разрешению. Когда пользователь имеет «родительское» **(parent)** разрешение, он автоматически имеет все его «дочерние» разрешения;
- мы можем применить некоторое дополнительное параметризованное ограничение под названием «правило» (rule) к разрешению. Технически правило – это функция, принимающая, как обычно, аргументы. Если эта функция вернёт значение `false`, даже если пользователю было присвоено рассматриваемое разрешение, он всё равно будет считаться заблокированным.

Для того чтобы называться «контролем доступа на основе ролей» **(RBAC, Role-Based Access Control)**, в этой схеме определённо не хватает понятия «роли». В Yii 2 роли работают таким же образом, что и разрешения. На самом деле они даже реализованы одним и тем же базовым классом `\yii\rbac\Item`, отличаясь только значением одной из констант класса. Также не существует вызова `Yii::$app->user->is($role)`, так что нам придётся проверять, присвоена ли пользователю та или иная роль, тем же вызовом `\yii\web\User.can()`. Система

RBAC фреймворка Yii 2 подразумевает, что роли – это просто группы разрешений, а мы всегда проверяем только сами разрешения.

Компонент приложения, который реализует RBAC, называется «менеджер авторизации» (Authorization Manager) и доступен через свойство `Yii::$app->authManager`. Метод `\yii\web\User.can()`, описанный выше, – это обёртка вокруг вызова `Yii::$app->authManager->checkAccess($user_id, $permission)`.

Для удобства существует понятие роли по умолчанию **(default role)**. Список ролей по умолчанию может быть установлен в настройке `components.authManager.defaultRoles` в конфигурации приложения, что соответствует свойству `\yii\rbac\BaseManager::$defaultRoles`. Подразумевается, что пользователь всегда имеет роль по умолчанию, то есть если настройка `defaultRoles` имеет значение `["guest"]`, вызов `Yii::$app->authManager->checkAccess($user_id, "guest")` всегда вернёт значение `true`.

В ранее рассмотренном классе `AccessFilter` каждое правило может быть настроено для проверки роли пользователя. Мы тогда использовали символы `?` и `@` для обозначения гостей и опознанных пользователей соответственно, но на самом деле там могут быть использованы названия ролей.

Самая сложная часть, необходимая для работы вышеописанной системы, – это установка необходимых связей между пользователями и ролями. Давайте рассмотрим, как это делается, разработав ещё одну функциональную единицу в нашем примере CRM-приложения.

Защита администрирования CRM от пользователей CRM

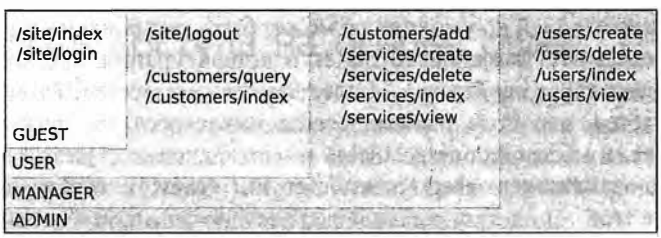
С самого начала в нашей спецификации приложения мы различали роли пользователей CRM и администраторов CRM. Каждый приёмочный тест до сих пор начинался с какой-либо работы в базе данных и заканчивался или использованием публичного интерфейса, или проверкой некоторых предположений сразу в административном интерфейсе.

Теперь настало время на самом деле запретить пользователям CRM доступ к страницам интерфейса управления базой данных. Мы собираемся реализовать следующий набор бизнес-правил:

- неопознанные пользователи (гости) не должны иметь доступа ни к чему, кроме стартовой страницы и формы входа;

- обычные пользователи должны иметь доступ к интерфейсу поиска клиентов по номеру телефона;
- пользователи уровня менеджера должны иметь доступ ко всему, кроме интерфейса управления пользователями;
- пользователи уровня администратора должны иметь доступ ко всему.

Вот картинка для наглядности:



У нас уже есть тесты для функциональности входа/выхода из системы, без последующей проверки прав доступа. Однако теперь мы заставляем всех пользователей нашего приложения в первую очередь аутентифицироваться. Это означает, что во всех наших имеющихся приёмочных тестах первым шагом должен быть вход в систему в качестве пользователя, имеющего достаточные права для того, чтобы сделать то, что должен сделать тест.

Это невероятно сложно сделать в правильных тестах через всё приложение, так как для этого необходимо не только использовать соответствующий административный интерфейс для создания сущностей, манипуляции с которыми мы собираемся проверить, но и также использовать административный интерфейс высшего уровня доступа, для того чтобы создать пользователя и дать ему достаточные права для совершения этих манипуляций. Не говоря о том, что нам нужен работающий пользовательский интерфейс для присвоения прав доступа (то есть ролей) пользователям. Это слишком большой объём подготовительной работы. Однако такой курс действий предпочтителен, если вам нужно убедиться, что все части приложения работают и ничто важное не было забыто.

Установка предопределённых пользователей

В данном примере мы обменяем время, необходимое для подготовки записей пользователей, на место для хранения их в базе данных с самого начала. Вместо создания пользователя нужного калибра для

каждого индивидуального теста мы заранее создадим в базе данных по одному пользователю на каждую роль, и наши приёмочные тесты, прежде чем сделать что-либо, будут использовать учётные данные этих пользователей для входа в систему. Конечно же, это не избавит нас от реализации самой логики входа в систему перед каждым действием.

Вот пользователи, которых нужно создать (паролями являются случайные фразы с высокой энтропией, лёгкие для запоминания):

Имя пользователя	Пароль	Название роли
Нет имени пользователя, роль по умолчанию	–	guest
JoeUser	7 wonder @ American soil	user
AnnieManager	Shiny 3 things hmm, vulnerable	manager
RobAdmin	Imitate #14th syndrome of apathy	admin

Вы можете получить хороший разбор сложности пароля на сайте http://www.passwordmeter.com/. В наше время самая важная черта хорошего пароля – это его длина и типы символов, использованных в нём (см., например, эту заметку от Microsoft: http://www.microsoft.com/en-gb/security/online-privacy/passwords-create.aspx). Мы намеренно выбрали произносимые фразы с прописными и строчными буквами, цифрами и специальными символами. Обратите внимание на то, что у нас нет никакой причины убирать пробелы из наших паролей.

Создайте миграцию для добавления их в базу данных:

```
./yii migrate/create add_predefined_users
```

Запись о пользователе должна регистрироваться следующим образом:

```
$user = new \app\models\user\UserRecord();
$user->attributes = compact('username', 'password');
$user->save();
```

Потому что наш перехватчик `beforeSave()` генерирует за нас хэши паролей и токены авторизации.

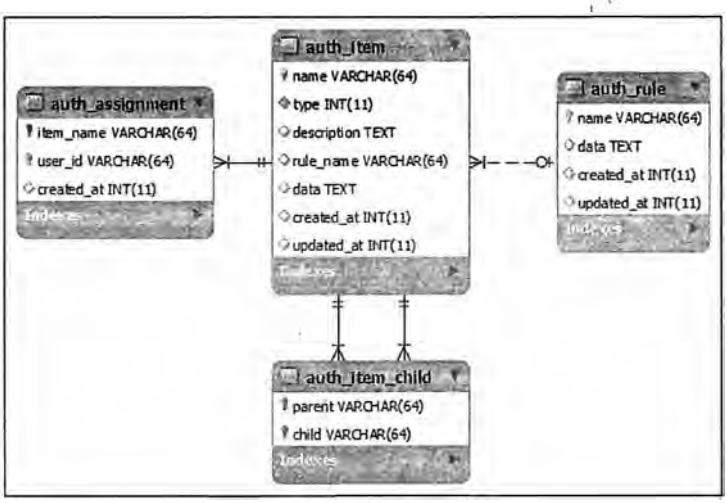
Вы можете посмотреть на то, как мы сами создали пользователей в файле `migrations/m140718_063423_add_predefined_users.php` в пакете кода, идущем в комплекте с этой книгой.

Затем, чтобы мы могли присвоить пользователям роли, нам нужно включить менеджер RBAC в нашем приложении.

Менеджеры RBAC в Yii

Вместе с Yii поставляются два менеджера RBAC. Один из них – это `\yii\rbac\PhpManager`, который считывает привязки ролей из сценария PHP при каждой загрузке приложения, а другой – это `\yii\rbac\DbManager`, который сохраняет привязки в базе данных. Мы будем использовать `DbManager`, так как он даёт больше свободы в манипулировании привязками.

Менеджер RBAC, основанный на базе данных, использует следующую схему данных для сохранения информации о ролях (и разрешениях) пользователей:



Менеджер RBAC на самом деле вообще не действует в терминах «ролей». Он действует в терминах «элементов авторизации» (**Authorization Items**), которые могут быть двух типов. Вот выдержка из определения класса:

```
namespace yii\rbac;
class Item extends Object
{
   const TYPE_ROLE = 1;
   const TYPE_PERMISSION = 2;
```

Класс `DbManager` по умолчанию сохраняет элементы авторизации в таблице под названием `auth_item`. Как обычно в Yii 2, это настраивается.

Проблема в том, что эти типы нигде ни проверяются. `Yii::$app->user->can($itemName)` подходит всем. Поэтому для простоты проще говорить в терминах «ролей» вместо «элементов авторизации», однако в случае по-настоящему сложного RBAC это различие с точки зрения проектирования может быть полезным.

Элементы авторизации, будь то роли или разрешения, хранимые менеджером авторизации, образуют направленный ацикличный граф, так что они могут иметь «дочерние» элементы, как было упомянуто выше. Давайте остановимся на этом подробнее, для простоты предполагая, что мы имеем дело с ролями.

Если пользователь `$user` имеет роль `$role` и эта роль имеет дочернюю роль `$child`, тогда вызов `$user->can($child)` будет возвращать `true`, то есть *родительская роль имеет доступ ко всему, к чему имеет доступ дочерняя роль*. Но если пользователь имеет роль `$child` и эта роль является дочерней по отношению к роли `$role`, тогда вызов `$user->can($role)` будет возвращать `false`, то есть *дочерняя роль не имеет доступа к тому, к чему имеет доступ родительская роль*.

Отношения между ролями по умолчанию сохраняются классом `DbManager` в таблице `auth_item_child`.

Собственно, связи пользователей с ролями сохраняются в таблице под названием `auth_assignment`. Обратите внимание в вышеприведённой схеме данных на то, что колонка `user_id` – это не внешний ключ, она даже не имеет тип `INT`. Это, конечно же, вызвано тем, что схема по умолчанию не имеет никакого представления о том, как вы храните свои записи о пользователях (они, в конце концов, могут быть вообще в другой базе данных). Также колонка `user_id` должна быть заполнена идентификаторами, которые возвращают метод `IdentityInterface.getId()`, который наш класс `UserRecord` счастливо реализует. Эти значения не обязательно должны быть первичными ключами таблицы `user`, пусть даже в нашем случае это и на самом деле так.

Разработчики Yii 2 снабдили нас миграцией, уже подготовленной для того, чтобы инициализировать схему таблиц базы данных для работы класса `\yii\rbac\DbManager`. Вы можете настроить все необходимые таблицы, выполнив следующую консольную команду:

```
./yii migrate --migrationPath='@yii/rbac/migrations'
```

Вы также можете использовать следующий трюк для заполнения базы данных таблицами, ожидаемыми `DbManager`. Схема хранится в виде файлов с именами `schema-*.sql` в подкаталоге `rbac` корневого каталога установки Yii 2. Каждый файл соответствует некоторой конкретной СУБД. При условии что вы используете MySQL и Yii 2 установлен при помощи Composer, нужным файлом будет `vendor/yiisoft/yii2/rbac/schema-mysql.sql`. Создайте миграцию:

```
./yii migrate/create create_rbac_tables
```

Затем в методе `up()` в созданном сценарии миграции напишите следующий код, который загрузит SQL-команды напрямую из только что упомянутого файла:

```
$this->execute(
    file_get_contents(
        Yii::getAlias('@yii/rbac/schema-mysql.sql')));
```

Этот трюк действительно полезен в случаях, когда у вас уже есть проверенная временем, очень старая схема базы данных, достаточно большая, что переписывание её в виде идиоматичного кода миграций Yii будет слишком трудозатратным.

Тесты для нашей иерархии ролей

Прежде чем мы начнём на самом деле заполнять базу данных ролями, как мы убедимся, что наша система контроля доступа на самом деле на месте? Давайте соберём функциональный тест для нашей иерархии ролей. Создайте тест:

```
./cept generate:test functional RoleHierarchy
```

С добавлением заранее созданных ролей дела становятся довольно неприглядными. Чтобы начинать с чистого состояния, в наших функциональных тестах мы используем снимок чистой базы данных, и его теперь не так-то легко получить, учитывая миграции, добавляющие записи в таблицы. И даже без этого всё равно остаётся необходимость пересоздавать этот снимок каждый раз, когда мы добавляем новую миграцию. Также с таким подходом мы полностью вычистим реальную базу данных, если запустим функциональные тесты на «боевом» сервере. Все эти сложности мы отложим до последней главы. Давайте до тех пор будем делать вид, что каким-то образом мы получаем чистое состояние до того, как выполнять какие-либо функциональные и приёмочные тесты.
Посмотрите на файлы `tests/functional.suite.yml`, `tests/functional/_bootstrap.php` и `config/test.php` в пакете кода, приложенном к этой книге, чтобы увидеть, как мы решили эту проблему.

Внутри этого теста мы напишем реализацию проверки роли по умолчанию (которой является роль `guest`) методом грубой силы, следующим образом:

```
/** @test */
public function DefaultRoleIsGuest()
{
   // no login at all

   $this->assertFalse($this->user->can('admin'));
   $this->assertFalse($this->user->can('manager'));
   $this->assertFalse($this->user->can('user'));
   $this->assertTrue($this->user->can('guest'));
}
```

Как уже было сказано, мы вынуждены использовать метод `can()` для проверки присвоенных ролей, что читается довольно странно.

Значением переменной `$this->user` на этапе подготовки теста становится компонент `\yii\web\User`, для того чтобы использовать её как сокращение и как единственное место для внесения изменений:

```
/** @var \yii\web\User */
private $user;

protected function _before()
{
   $this->user = Yii::$app->user;
}
```

Роли заранее созданных пользователей мы будем тестировать, используя возможность провайдеров данных (Data Provider) из фреймворка тестирования PHPUnit (см. http://phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.data-providers, не перепутайте это с понятием `DataProvider` в Yii):

```
public function PredefinedUserRoles()
{
   return [
      ['RobAdmin',    ['admin' => true,  'manager' => true,  'user'
=> true, 'guest' => true]],
      ['AnnieManager', ['admin' => false, 'manager' => true,  'user'
=> true, 'guest' => true]],
      ['JoeUser',     ['admin' => false, 'manager' => false, 'user'
=> true, 'guest' => true]],
   ];
}

/**
 * @test
 * @dataProvider PredefinedUserRoles
 * @param string $username
```

```
     * @param array $rbac
     */
    public function PredefinedUsersHasProperRoles($username, $rbac)
    {
        $identity = \app\models\user\UserRecord::findOne(compact('userna
me'));

        $this->user->login($identity);

        foreach ($rbac as $role => $allowed)
            $this->assertEquals($allowed, $this->user->can($role));
    }
```

Для каждого набора данных из предоставленных нашим провайдером мы ищем в базе данных запись о пользователе с указанным именем пользователя, а затем проверяем, ведёт ли себя вызов `Yii::$app->user->can()` соответственно нашей иерархии ролей для каждого из этих пользователей.

Так как мы входим в систему при каждом тесте, давайте выходить из системы на этапе завершения:

```
    protected function _after()
    {
        $this->user->logout();
    }
```

Этот тест, очевидно, будет провален, так как мы даже не присоединили компонент менеджера RBAC к приложению.

Установка иерархии ролей

Мы заполним базу данных нашими ролями и привязками ролей, используя миграцию. Это значит, что нам нужно присоединить менеджер RBAC как к консольному приложению, которое будет использовано, когда мы запустим команду `./yii migrate` в корневом каталоге проекта, так и к веб-приложению, которое на самом деле будет использовать RBAC. В дополнение к возможности написать эту конкретную миграцию мы сможем писать в дальнейшем свои консольные команды, которые будут использовать менеджер RBAC, если это нам когда-нибудь понадобится.

Чтобы присоединить менеджер RBAC, основанный на базе данных, к нашему приложению, нам нужно всего лишь сослаться на него в разделе `components` конфигурации приложения, вот так:

```
'authManager' => [
   'class' => 'yii\rbac\DbManager',
   'defaultRoles' => ['guest'],
],
```

Как только что было сказано, этот фрагмент должен быть добавлен и в `@app/config/web.php`, и в `@app/config/console.php` или выделен в дополнительный файл и всё равно включён в оба этих файла конфигурации. Указанные пути соответствуют нашему примеру CRM-приложения, как оно есть на данный момент.

Итак, с этого момента у нас доступен `Yii::$app->authManager`, указывающий на экземпляр `yii\rbac\DbManager`. Ролью по умолчанию, как указано в подсвеченной части кода, будет «гость» (`guest`), то есть все, не важно, аутентифицированные или нет, имеющие другие роли или нет, будут считаться имеющими роль `guest`.

Теперь мы можем сделать миграции, которые установят необходимые нам роли. Мы можем использовать следующие методы класса `yii\rbac\DbManager`:

Название метода	Причина использовать
`createRole($name)`	Основной метод для создания ролей. Возвращает настроенный экземпляр `\yii\rbac\Role`
`createPermission($name)`	То же самое, что `createRole()`, но создаёт экземпляры `\yii\rbac\Permission`. Мы не будем использовать этот метод, так как для простоты мы используем в нашей схеме безопасности только роли
`assign($role, $userId)`	Связать экземпляр `\yii\rbac\Role` с пользователем, обладающим `$userId`. `$userId` должен быть идентификатором, который возвращает метод `IdentityInterface.getId()`
`add($item)`	Регистрирует данный элемент авторизации, будь то экземпляр `\yii\rbac\Permission` или `\yii\rbac\Role`
`addChild($parent, $child);`	Регистрирует факт того, что `$child` является «дочерним» к `$parent`

В наличии имеется намного больше методов, но мы перечислили только самые фундаментальные. Остальные лучше узнать из справочной информации для класса `\yii\rbac\ManagerInterface` здесь: http://www.yiiframework.com/doc-2.0/yii-rbac-managerinterface.html.

Наконец, вот наша миграция для установки иерархии ролей:

```
public function up()
{
   $rbac = \Yii::$app->authManager;
```

```
            $guest = $rbac->createRole('guest');
        $guest->description = 'Nobody';
        $rbac->add($guest);

        $user = $rbac->createRole('user');
        $user->description = 'Can use the query UI and nothing else';
        $rbac->add($user);

        $manager = $rbac->createRole('manager');
        $manager->description = 'Can manage entities in database, but not
users';
        $rbac->add($manager);

        $admin = $rbac->createRole('admin');
        $admin->description = 'Can do anything including managing users';
        $rbac->add($admin);

        $rbac->addChild($admin, $manager);
        $rbac->addChild($manager, $user);
        $rbac->addChild($user, $guest);

        $rbac->assign(
           $user,
           \app\models\user\UserRecord::findOne(['username' => 'JoeUser'])-
>id
        );
        $rbac->assign(
           $manager,
           \app\models\user\UserRecord::findOne(['username' =>
'AnnieManager'])->id
        );
        $rbac->assign(
           $admin,
           \app\models\user\UserRecord::findOne(['username' =>
'RobAdmin'])->id
        );   }

    public function down()
    {
        $manager = \Yii::$app->authManager;
        $manager->removeAll();
    }
```

Мы и правда можем восстановиться после этой миграции, причём используя обычно ненужный метод `yii\rbac\DbManager.removeAll()`.

Теперь наш функциональный тест для иерархии ролей проходит успешно. Мы готовы реализовать настоящую защиту для наших контроллеров.

Тест контроля доступа в контроллерах

Вернёмся от деталей реализации RBAC к функциональности, которую мы на самом деле хотели внедрить.

Как мы собираемся проверять ограничения доступа, описанные в начале этого раздела? Сложный вопрос! Это поразительно сложно сделать «правильно». Для упрощения повествования давайте сделаем очень, очень простую реализацию, которая будет «просто работать».

Каждый из подклассов `AcceptanceTester`, которые мы до сих пор создавали для наших приёмочных тестов, неявно имеет одну из ролей в нашей иерархии. Мы будем считать каждый из них одним из предопределённых пользователей, следующим образом:

- `AcceptanceTester\CRMOperatorSteps` соответствует пользователю `AnnieManager`, который имеет роль `manager`;
- `AcceptanceTester\CRMServiceManagementSteps` тоже соответствует пользователю `AnnieManager`, который имеет роль `manager`;
- `AcceptanceTester\CRMUserSteps` соответствует пользователю `JoeUser`, который имет роль `user`;
- `AcceptanceTester\CRMUsersManagementSteps` соответствует пользователю `RobAdmin`, который имеет роль `admin`, единственную, имеющую право управлять учётными записями пользователей.

И мы напишем набор совершенно прямолинейных приёмочных тестов для проверки наших допущений в правах доступа пользователей:

```
./cept generate:cept acceptance AdminAccessRights
./cept generate:cept acceptance ManagerAccessRights
./cept generate:cept acceptance UserAccessRights
./cept generate:cept acceptance GuestAccessRights
```

Вот содержимое файла `tests/acceptance/ManagerAccessRightsCept.php` (все остальные тесты выглядят так же):

```
$I = new AcceptanceTester\CRMOperatorSteps($scenario);
$I->wantTo('Check Manager-level access rights');

$I->amOnPage('/customers/query');
$I->dontSee('Forbidden');

$I->amOnPage('/customers/index');
```

```
$I->dontSee('Forbidden');

// ... и так далее...

$I->amOnPage('/users/create');
$I->see('Forbidden');

$I->amOnPage('/users/index');
$I->see('Forbidden');
```

> Вышеуказанный код не является ни сопровождаемым, ни исчерпывающим. Не пишите ваши собственные тесты таким образом. Этот стиль был выбран, потому что он хорошо выражает идею и достаточно краток, чтобы показать его в книге.

Мы просто переходим на каждый из маршрутов в системе и проверяем, не получаем ли мы страницу ошибки с сообщением 403 Forbidden, на которой огромными буквами написано «Forbidden». Маршруты `/services/delete` и `/users/delete` были пропущены, потому что, как ранее было сказано, они защищены фильтром `VerbFilter` и доступны только по POST-запросу. Маршруты `/site/index`, `/site/login` и `/site/logout` были пропущены, потому что у них есть свои специальные тесты прав доступа.

Этот список из вызовов `see()` и `dontSee()` должен быть повторён для каждого из классов `CRMOperatorSteps`, `CRMServiceManagementSteps`, `CRMUserSteps` и `CRMUsersManagementSteps`. Тестовый сценарий для прав доступа уровня менеджера, таким образом, должен проверять сразу два класса.

Для посетителей уровня «гость» ожидаемое поведение немного отличается. Хотя и есть возможность это изменить, но по умолчанию Yii 2, вместо того чтобы показывать им «403 Forbidden», перенаправляет гостей на форму входа в систему. Подобное поведение выглядит неплохо, поэтому давайте его и использовать. Сценарий `GuestAccessRightsCept`, таким образом, должен использовать не такие проверки:

```
$I->see('Forbidden');
```

а такие:

```
$I->seeElement('#login-form');
```

Метод Codeception под названием `seeElement` из модуля WebDriver (см. http://codeception.com/docs/modules/WebDriver#seeElement) мо-

жет искать элементы дерева DOM по селекторам CSS, и `login-form` – это именно то, что мы указали в качестве HTML-идентификатора этой формы, когда настраивали её.

Нам нужен дополнительный подкласс класса `AcceptanceTester`, представляющий «гостей». Такой класс – это именно то, что нужно сценарию `LoginAndLogoutCept` из предыдущей главы. Давайте создадим его:

```
./cept generate:stepobject acceptance CRMGuest
```

Этот подкласс пока что будет просто пустым, потому что он используется только в `GuestAccessRightsCept`, который выполняет лишь базовые встроенные шаги.

Под конец, используя такую стратегию тестирования, у нас должен получиться следующий набор приёмочных тестов:

- `tests/acceptance/GuestAccessRightsCept.php` – проверяет права доступа для роли `guest`, которая представлена свежесозданным классом `\AcceptanceTester\CRMGuestSteps`;
- `tests/acceptance/UserAccessRightsCept.php` – проверяет права доступа роли `user`, представленной классом `\AcceptanceTester\CRMUserSteps`;
- `tests/acceptance/ManagerAccessRightsCept.php` – проверяет права доступа роли `manager`, представленной двумя классами: `\AcceptanceTester\CRMOperatorSteps` и `\AcceptanceTester\CRMServicesManagementSteps`;
- `tests/acceptance/AdminAccessRightsCept.php` – проверяет права доступа роли `admin`, представленной классом `\AcceptanceTester\CRMUsersManagementSteps`.

Права доступа проверяются сочетанием вызовов `see('Forbidden')` и `dontSee('Forbidden')`, согласно схеме прав доступа, которую мы решили использовать в начале этого раздела.

После этого наши тесты будут выполняться без ошибок, но и без успеха. Как мы сделаем так, чтобы наше приложение успешно проходило эти тесты, то есть как мы, собственно, защитим контроллеры от доступа неавторизованных пользователей?

Фильтр контроля доступа

Фильтр контроля доступа – это особый фильтр действий, который позволяет нам точно определять права доступа к действиям внутри некоторого контроллера.

Он должен быть присоединён к экземпляру класса `Controller` следующим образом:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'rules' => [
                // правила в формате, описанном ниже
            ]
        ]
    ];
}
```

То есть метод behaviors() должен возвращать массив, внутри которого должна быть конфигурация для создания экземпляра класса AccessControl.

Каждый элемент в настройке rules – это массив со следующими элементами:

```
[
    'allow' => true, // или false
    'actions' => ['идентификаторы', 'действий', 'к', 'которым',
'применять', 'правило'],
    'controllers' => ['идентификаторы', 'контроллеров', 'к', 'которым',
'применять', 'правило'],
    'roles' => ['роли', 'включая', 'символы', '?', 'и', '@', 'описанные',
'ранее'],
    'ips' => ['IP', 'адреса', 'включая', 'возможно', 'символ', '*'],
    'verbs' => ['HTTP', 'методы', 'запроса', 'к', 'которым', 'приме-
нять', 'правило'],
    'matchCallback' => $callable1, // произвольная проверка применимости
правила
    'denyCallback' => $callable2 // что делать в случае запрещения
запроса
]
```

Выделенные поля соответствуют публичным свойствам класса \yii\web\AccessRule. Для экономии места мы не будем подробно описывать значение каждого поля, в особенности учитывая то, что определение этого класа уже содержит полное их описание.

Правила проверяются в порядке их появления, поэтому они могут иметь пересекающиеся требования. Если никакое правило не сработало, запрос блокируется, так что если раздел rules настроек фильтра AccessControl оставить пустым, всё в этом контроллере будет запрещено для всех.

Анонимная функция по ключу `denyCallback`, упомянутая выше, как и остальные настройки, необязательна. Если её опустить, будет использовано похожее свойство `denyCallback` у класса `AccessControl`. По умолчанию оно показывает страницу «403 Forbidden» для авторизованных пользователей и страницу с формой входа для гостей, что является *в точности* тем поведением, которое мы ожидаем в наших приёмочных тестах. Да, мы снова смухлевали.

Применение контроля доступа к сайту

Для начала нам нужно защитить контроллер `UsersController` таким образом, что он будет доступен только пользователям с ролью `admin`:

```
'rules' => [
    [
        'roles' => ['admin'],
        'allow' => true
    ]
```

Мы показываем лишь правила, так как остальная конфигурация класса `AccessControl` будет оставаться одной и той же. Мы должны присоединить фильтр в методе `behaviors()`. Обратите внимание на то, что Gii уже создал для нас этот метод и даже добавил в него другой фильтр – `VerbFilter`.

Для класса `CustomersController` нам нужно следующее:

```
'rules' => [
    [
        'actions' => ['add'],
        'roles' => ['manager'],
        'allow' => true
    ],
    [
        'actions' => ['index', 'query'],
        'roles' => ['user'],
        'allow' => true
    ],
]
```

Согласно нашей спецификации прав доступа, любой пользователь может просматривать интерфейс управления клиентами, но для создания нового клиента нужны права доступа уровня менеджера.

Для класса `ServicesController` нам нужно следующее, наподобие вышеописанных настроек для класса `UsersController`, но с другой ролью:

```
        'rules' => [
            [
                'roles' => ['manager'],
                'allow' => true
            ]
        ]
```

Самая интересная часть – это изменение наших вариантов класса AcceptanceTester, для того чтобы они могли существовать в этом новом, наполненном ограничениями мире. Фактически все наши тесты теперь требуют входа в систему. Мы можем сделать так: научить CRMGuest входить и выходить из системы, а затем все остальные подклассы AcceptanceTester сделать подклассами CRMGuest, так что они тоже смогут это делать:

```
function login($username, $password) // 1
{
    $I = $this;
    $I->amOnPage('/site/login');
    $I->fillField('LoginForm[username]', $username);
    $I->fillField('LoginForm[password]', $password);
    $I->click('Login');
    $I->wait(1); // 2
    $I->seeCurrentUrlEquals('/'); // 3
}
```

Выделенные части – это нетривиальные фрагменты кода. В первую очередь этот метод универсален и принимает имя пользователя и пароль в качестве входных аргументов. Во-вторых, он использует нашу форму входа, которая всё ещё имеет валидацию при помощи Javascript, поэтому мы вынуждены ждать, прежде чем отправлять её. В-третьих, для того чтобы ускорить выполнение наших и так очень долгих приёмочных тестов в случае ошибки, мы немедленно проверяем, были ли мы перенаправлены на стартовую страницу после входа в систему. Это будет означать, что вход был выполнен успешно.

Вот код выхода из системы:

```
function logout()
{
    $I = $this;
    $I->amOnPage('/');
    // Expecting that this button is presented on the homepage.
    $I->click('logout');
}
```

Выход из системы намного проще: нам нужно всего лишь перейти на главную страницу, так как мы можем быть на странице наподобие «403 Forbidden» (и на самом деле мы будем *часто* на ней оказываться), где вообще нет ссылки «logout». Однако нам нужно быть осторожными в том, где мы используем этот тестовый шаг, так как ссылка «logout» отображается, только если пользователь вошёл в систему.

Как было сказано ранее, нам нужно также сделать все наши разновидности `AcceptanceTester` наследниками `CRMGuestSteps`: `CRMOperatorSteps`, `CRMUsersManagementSteps`, `CRMUserSteps` и `CRMServicesManagementSteps`.

Имея всё это, мы теперь можем сделать так, чтобы все наши экземпляры `AcceptanceTester` входили в систему сразу после создания. Мы просто добавим следующие два свойства и конструктор в класс `CRMGuestSteps`:

```
public $username;
public $password;

public function __construct($scenario)
{
   parent::__construct($scenario);

   if ($this->username and $this->password)
      $this->login($this->username, $this->password);
}
```

Таким образом, если у класса-наследника будут определены имя пользователя и пароль, он совершит вход в систему в качестве первого шага в любом тестовом сценарии. Этот приём позволяет нам не менять никаких из наших существующих тестов.

То есть в классе `CRMOperatorSteps` первыми строчками должны быть следующие:

```
class CRMOperatorSteps extends CRMGuestSteps
{
   public $username = 'AnnieManager';
   public $password = 'managerpass';
```

И точно так же в остальных классах `CRM...Steps`.

Несмотря на это, нам всё равно нужно сделать пять изменений в наших существующих тестах.

Первое изменение в сценарии `LoginAndLogoutCept`: после того как `AcceptanceTester\CRMUsersManagementSteps` закончил создавать нового пользователя, этот пользователь – очевидно, гость, так как у нас

нет никакого пользовательского интерфейса управления ролями. Поэтому пользователь, который будет совершать реальное тестирование функциональности входа и выхода из системы, – это не экземпляр `AcceptanceTester\CRMUserSteps`, а экземпляр `AcceptanceTester\CRMGuestSteps`. Также `AcceptanceTester\CRMUsersManagementSteps` должен выполнить `logout()` после того, как закончит.

Теперь становится ясно, что связать понятие «объектов шагов» («step objects«), навязываемое Codeception, с интуитивным представлением того, чем мы должны считать подклассы `AcceptanceTester`, очень сложно. Почему они не назвали наследников классов Tester («тестировщик») как «TesterKind» («разновидность тестировщика») или как-то наподобие этого?

Мы не будем показывать полный конечный код, так как это довольно просто реализовать в файле `tests/acceptance/LoginAndLogoutCept.php`:

```
$I = new AcceptanceTester\CRMUsersManagementSteps($scenario);
// ... шаги создания пользователя...
$I->logout();
$I = new AcceptanceTester\CRMGuestSteps($scenario);
// ... шаги проверки возможностей входа/выхода...
```

Второе изменение вызвано тем, что у нас есть следующие строчки в файле `LoginAndLogoutCept.php`:

```
$I->amGoingTo('logout from arbitrary page');
$I->amInQueryCustomerUi();
$I->click('logout');
```

При наших правилах доступа гости не могут отправиться на «произвольную страницу», так что этот тест потерял смысл. Просто удалите эти строчки.

Третье изменение нужно сделать в том же файле `LoginAndLogoutCept.php`. Там есть следующая строчка:

```
$I->seeLink('logout');
```

Эта строчка находится там для того, чтобы проверить, успешно ли мы вошли в систему. После этого идут ещё четыре шага, проверяющих, на самом ли деле отработал выход из системы. Но для этого нам нужно на самом деле выйти из системы! Вставьте следующую строчку после этого вызова `seeLink()`:

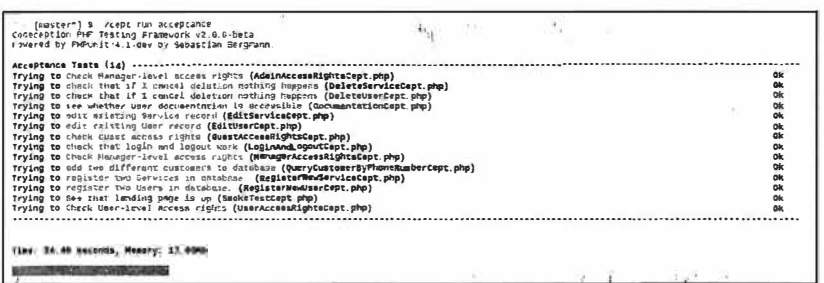
```
$I->logout();
```

В качестве четвёртого изменения мы должны переместить определения методов `seeIAmInLoginFormUi()`, `fillLoginForm()`, `submitLoginForm()`, `seeIAmAtHomepage()`, `seeUsername($user)` и `dontSeeUsername($user)` из класса `AcceptanceTester\CRMUserSteps` в класс `AcceptanceTester\CRMGuestSteps`.

Последнее изменение, которое мы должны сделать, – это выйти из системы в сценарии `QueryCustomerByPhoneNumberCept` после того, как мы закончили создавать нового клиента.

```
$I = new \AcceptanceTester\CRMOperatorSteps($scenario);
// ... шаги создания клиента ...
$I->logout();
$I = new \AcceptanceTester\CRMUserSteps($scenario);
// ... шаги испытания возможности поиска по номеру телефона ...
```

На этом реализация закончена. Выполните тесты:



Успех. Более того, теперь у нас есть достаточно надёжный комплект автоматических тестов, который избавит нас от тягот ручного прокликивания всех этих форм входа в систему.

Итоги

В этой главе мы изучили несколько возможностей Yii, которые помогают нам контролировать доступ к различным частям приложения. Также мы узнали, что Yii втайне защищает нас от TFCSRF-атак, что, вообще-то, просто чудесно.

Вдобавок к этому мы увидели, как в Yii 2 реализована обработка исключений, и посмотрели на специальные исключения, основанные на классе `HttpException`, которые показывают пользователю нужный отклик с нужным кодом состояния HTTP.

Мы узнали про концепцию фильтров действий, которая помогает нам собирать сочетания вещей, которые мы можем делать до того, как запустится действие контроллера.

Наиважнейшей частью этой главы было построение политики контроля доступа на основе ролей пользователей. Мы увидели, как компонент приложения под названием «менеджер RBAC» совместно с классом фильтра `AccessControl` предоставляют эту функциональность.

Наконец, разрабатывая тесты контроля доступа, мы рассмотрели сложности, связанные с автоматическим входом в систему в тестах через всё приложение, а также простые методы их преодоления. Про полноценные методы решения могут быть написаны целые книги.

В следующей главе мы раскроем фундаментальную основу структуры Yii 2: систему модулей, которая является воплощением техники «Модель–Вид–Контроллер».

Глава 7

Модули

Эта глава раскроет, наконец, концепцию модулей, которую мы упоминали почти во всех предыдущих главах. Существование модулей влияет на работу каждой возможности в Yii от уровня контроллеров до уровня представлений. Однако вполне возможно, что в маленьких и среднего размера приложениях вы никогда не будете их использовать (за одним исключением, которое мы скоро упомянем).

В этой главе мы изучим понятие модуля и то, как оно реализовано в Yii 2. После этого мы, используя модули, немного перераспределим вещи в нашем примере CRM-приложения, добавив в него специальный раздел исключительно для API-отчётов.

Давайте начнём.

Модули Yii

Модуль (Module) – это сущность, которая имеет свои собственные представления, контроллеры, компоненты и, возможно, другие модули. Фактически это олицетворение композитного паттерна MVC.

Как вы, возможно, уже предположили, экземпляр приложения Yii является примером модуля.

Официальная документация несколько сбивает с толку тем, что утверждает, что модуль – это приложение, которое не может быть использовано самостоятельно. Это очень неуклюжее объяснение. Наоборот, приложение – это особый вид модуля, расширенный возможностями, позволяющими ему быть исполняемым. Таким же образом, как точка входа в приложение Java – это не просто любой класс, а класс, содержащий метод `main()`. Такое намерение разработчиков ясно выражено в том, что именно класс `\yii\base\Application` расширяет базовый класс `\yii\base\Module`, добавляя в него несколько методов, а не наоборот.

Что значит «имеет свои собственные представления и т. д.»? Это означает, что экземпляр модуля имеет свойства для доступа и установки всех этих сущностей:

Свойство	Смысл
`controllerPath`	Путь к каталогу, содержащему все классы контроллеров, которые должны быть достижимы из этого модуля. Это свойство только для чтения и автоматически принимает своё значение на основе свойства `controllerNamespace`
`controllerNamespace`	Полностью определённое имя пространства имён, в котором должны находиться все классы контроллеров, чьи файлы лежат в каталоге `controllerPath`. Это свойство будет использовано для вычисления значения настройки `controllerPath`, согласно спецификации **PSR-4** (см. https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md)
`controllerMap`	Массив, точно определяющий контроллеры, которые должны быть достижимы из этого модуля. Каждая пара ключ–значение – это идентификатор контроллера и либо массив со свойствами контроллера, либо сам экземпляр контроллера
`basePath`	Псевдоним пути к папке, которая будет основой по умолчанию для всех относительных путей в этом модуле
`viewPath`	Псевдоним пути к папке, которая будет хранить файлы представлений для контроллеров в этом модуле
`layoutPath`	Псевдоним пути к папке, которая будет хранить файлы шаблонов представлений для этого модуля
`layout`	Псевдоним пути к файлу представления, который будет считаться шаблоном для всех представлений в этом модуле. Если псевдоним пути относительный, он разрешается относительно значения настройки `layoutPath`. Если он не установлен, будет использована настройка `layout` родительского модуля. Если, в конце концов, он разрешается в значение `false`, шаблон не будет применён (файл представления будет отрисован как есть)
`modules`	Список других модулей, которые должны быть достижимы из этого
`components`	Список компонентов, которые доступны в этом модуле (имитируя переменные-члены класса)
`module`	Родительский модуль, у которого этот данный модуль хранится в настройке `modules`.\

Эта таблица использует специфический термин **«достижимо» (reachable)**. Это придуманный нами для удобства термин, связанный с системой маршрутизации, реализованной в Yii 2, и его изобретение поможет нам понять, как составной паттерн MVC в Yii влияет на систему.

Неформальное понятие «достижимости»

Давайте посмотрим на действие контроллера, которое мы сделали ранее, в главе 4, и которое показывает список услуг в формате YAML. Оно реализовано в виде метода под названием `actionYaml` в контрол-

лере ServicesController, который находится в пространстве имён app\controllers, поэтому его полностью определённым именем является app\controllers\ServicesController.actionYaml(). Когда кто-то запрашивает URI http://yourdomainname.dom/services/yaml, Yii 2 вызывает в точности этот метод actionYaml(). Такое возможно по двум причинам:

1) настройка приложения controllerNamespace установлена в значение app\controllers (мы не устанавливали его специально, это значение по умолчанию);
2) по соглашению любой публичный метод контроллера, чьё имя начинается с action, считается действием контроллера (это называется «встроенное действие», **Inline Action**).

На основании этого мы неформально говорим, что метод actionYaml() «достижим из» класса ServicesController и, следовательно, из экземпляра приложения, так как мы фактически можем вызывать этот метод, переходя по некоторому известному URI.

Обратите внимание на то, что у приложения тоже есть настройка controllerNamespace, в точности потому, что приложение – это просто ещё один модуль в системе.

Модули добавляют ещё один шаг в вышеприведённое описание «достижимости». Если контроллер находится в пространстве имён, упомянутом в настройке controllerNamespace некоторого модуля, его действия будут достижимы по URI, только если этот модуль зарегистрирован в настройке modules приложения. Или в настройке modules другого модуля.

Таким образом, фактически механика маршрутизации, которую мы детально рассмотрим в главе 12, сводится к следующему шаблону URI:

`/id-модуля/id-модуля/.../id-модуля/id-контроллера/id-действия`

Конечно же, там есть много деталей: правила преобразования идентификаторов, действия, не являющиеся вписанными (inline), и т. п., но в своей основе маршрутизация работает именно так.

Исследование сложностей конфигурации модулей на глупых примерах

В основе всего модуль является просто классом, наследующим классу yii\base\Module. Этот класс должен быть зарегистрирован в настройке modules самого приложения или какого-нибудь другого модуля. Так как мы ссылаемся на модули по полностью определённому имени, точное расположение файла с определением этого класса может быть

любым, лишь бы автозагрузчик мог туда добраться. Более того, существование всех вышеописанных переменных – basePath, viewPath и др. – даёт нам большую свободу в выборе физической структуры нашего приложения.

Раз мы заговорили про физическую структуру приложения, давайте сделаем очень, очень глупую иерархию модулей. Она будет построена таким образом, что SiteController и другие контроллеры в пространстве имён app\controllers будут достижимы не только по обычным ссылкам, но и по ссылкам, содержащим цепочку вызовов модулей. Скоро станет понятнее.

Вспомним, что Yii 2 использует автозагрузчик, поддерживающий стандарт PSR-4, и пространство имён app\ соответствует корневому каталогу приложения. Таким образом, пространство имён app\controllers отображается на подкаталог controllers корневого каталога приложения, который, в свою очередь, соответствует псевдониму пути @app/controllers.

Мы используем пространство имён app\utilities для нашего проекта. Мы создали это пространство имён в главе **4**, для того чтобы было, где хранить свой отрисовщик представлений и компоновщик отклика. Давайте там создадим минимально необходимое определение модуля под названием FirstModule. Вот полностью содержимое файла utilities/FirstModule.php, который нам нужен:

```
namespace app\utilities;
use yii\base\Module;
class FirstModule extends Module { }
```

Будет лучше, если вы будете делать изменения, описанные в этом разделе, в отдельной ветке вашей системы контроля версий. Всё это – код «на выброс».

Поздравляем, вы только что создали свой первый модуль. Он, правда, абсолютно бесполезен для нашего примера, так как делает следующие допущения:

- он считает, что его basePath – это каталог, в котором он находится;
- он считает, что его представления и шаблоны находятся в подкаталогах views и views/layouts внутри basePath;
- он считает, что у него есть контроллеры, достижимые из пространства имён \app\utilities\controllers;

- он не присоединён ни к приложению, ни к какому-либо другому модулю.

Этот модуль мы присоединим к приложению, используя настройку `modules`:

```
'modules' => [
   ...
   'firstlevel' => [
      'class' => 'app\utilities\FirstModule',
   ]
]
```

Итак, мы присоединили к нашему приложению модуль, имеющий идентификатор `firstlevel`. Этот идентификатор был выбран произвольно, и он будет использован в дальнейшем при построении адресов ссылок. Класс, представляющий этот модуль, – `app\utilities\FirstModule`. Теперь создаём таким же образом второй модуль, на этот раз под названием `SecondModule`.

Этот второй модуль мы присоединим к модулю `firstlevel` там же, в конфигурации приложения. Это возможно, потому что настройки будут обработаны рекурсивно. Скорректируйте настройку `modules.firstlevel` следующим образом:

```
'modules' => [
   ...
   'firstlevel' => [
      'class' => 'app\utilities\FirstModule',
      'modules' => [
         'secondlevel' => [
            'class' => 'app\utilities\SecondModule',
         ]
      ]
   ]
],
```

Все модули имеют идентично настраиваемые свойства **`modules`**.

Будет слишком скучно добавить третий модуль тем же образом. Давайте сделаем это динамически, используя метод `\yii\base\Module.init()`, который вызывается при создании модуля после того, как отработает конструктор. Добавьте следующий фрагмент кода в класс `SecondModule`:

```
public function init()
{
   parent::init();
```

```
        $this->modules = [
            'thirdlevel' => [
                'class' => 'app\utilities\ThirdModule',
                'basePath' => '@app'
            ]
        ];
    }
```

Перекрывая методы `init()` базовых классов из Yii 2, не забывайте всегда вызывать `parent::init()`. Обычно за кулисами происходит очень много разных событий, и всё ради вас.

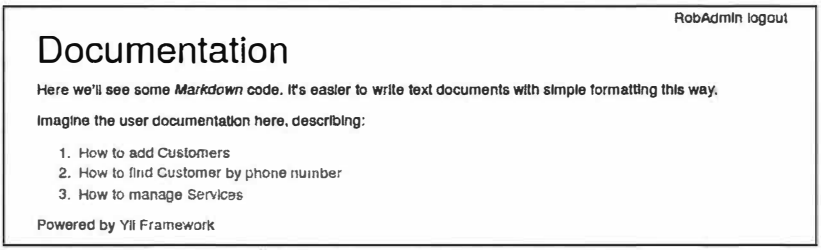
Вышеописанный метод `init()` делает то же самое, что бы сделало размещение массива настроек для `ThirdModule` в конфигурацию приложения, но привязка этого модуля на этот раз произойдёт немного позже. Конечно же, вам нужно на самом деле создать файл `@app/utilities/ThirdModule.php` с определением класса `ThirdModule`.

Мы указали, что `ThirdModule.basePath` будет указывать на корневой каталог нашего приложения. Сделав это, мы фактически превратили этот модуль в само приложение, с точки зрения механизма поиска представлений и контроллеров, за одним исключением. Для того чтобы завершить унификацию, нам нужно сбросить значение `ThirdModule.controllerNamespace` следующим образом:

```
public $controllerNamespace = 'app\controllers';
```

Обратите внимание, что в конфигурации приложения Yii мы очень активно используем строки, содержащие названия пространств имён. Так как в PHP пространства имён должны содержать обратную косую черту, нам нужно всегда помнить, что это символ экранирования. Если название какого-либо пространства имён начинается с букв `n`, `v`, `t` и т. п., в середине объявления пространства имён у нас получится управляющая последовательность (`\n`, `\v` и `\t` соответственно), что может привести к сложным неотслеживаемым ошибкам. Простейший способ предотвратить это – выработать привычку всегда использовать одинарные кавычки для строковых литералов в PHP, которые не раскрывают управляющих последовательностей. Вы также можете экранировать и саму обратную косую черту, написав "\\", если вам абсолютно необходимо использовать двойные кавычки в строках. Исходный код Yii 2 применяет именно этот метод; однако гораздо легче просто использовать одинарные кавычки.

Итак, вспомним, что ранее в главе 4 мы собрали просмотрщик документации, доступный по URL `/site/docs`. Давайте на этот раз откроем маршрут `/firstlevel/secondlevel/thirdlevel/site/docs`:



За исключением индикатора аутентификации, добавленного позже, эта страница идентична выводу маршрута `/site/docs`, который мы готовили в главе 4. На самом деле это в точности тот же метод `app\controllers\SiteController.actionDocs()`, достижимый теперь из двух URL. Все остальные контроллеры из пространства имён `app\controllers` также получили эту особенность.

Так как мы используем значение по умолчанию для настройки `viewPath` нашего модуля `thirdlevel`, то подразумевается, что этот путь отсчитывается относительно значения свойства `basePath` этого же модуля. Если бы мы не переопределили значение этого свойства, то, попытавшись перейти по маршруту `/firstlevel/secondlevel/thirdlevel/site/docs`, мы бы получили следующую ошибку:

Потому что, как было сказано ранее, по умолчанию `basePath` модуля – это каталог, в котором находится определение его класса. В вышеприведённом сообщении об ошибке можно видеть, что происходит, когда мы делаем следующее в `SiteController`:

```
return $this->render('docindex.md');
```

В зависимости от того, в каком модуле мы в данный момент находимся и каково значение его свойства `basePath`, мы ссылаемся на разные файлы. Каталог `/vagrant` – это корневой каталог приложения, если вы настроили вашу целевую машину для развёртывания, используя набор инструментов **Vagrant** (см. http://www.vagrantup.com/), как описано в *приложении 1*.

Разделение приложения на модули, используя для свойств `basePath`, `viewPath`, `layoutPath` и `controllerNamespace` только значения по умолча-

нию, подразумеваемые Yii, может быть либо слишком изощренным и несопровождаемым, либо очень простым и понятным, в зависимости от физической структуры ваших модулей и логической архитектуры вашего приложения. Вы можете полностью избавиться от переопределения этих настроек, если выполнены следующие условия:

- всё, относящееся к одному модулю, находится в одном подкаталоге, и ничего другого там нет;
- класс этого модуля находится в пространстве имён, которое отображается на этот подкаталог, то есть в простейшем случае файл, содержащий его определение, тоже находится в этом подкаталоге.

Мы будем полагаться на этот трюк далее в этой главе, когда мы на самом деле воспользуемся модулями, чтобы более красиво перераспределить кое-что в нашем примере CRM-приложения.

Модуль отладки

Кроме Gii, автоматического генератора кода, в Yii 2 встроен ещё один механизм: модуль отладки (Debug). Так же, как и Gii, он выделен в виде расширения (более подробно про расширения будет рассказано в главе 9) и присоединяется к приложению в виде модуля.

Мы уже видели модуль отладки в самом начале, в главе 1, когда устанавливали базовый шаблон приложения. Этот модуль предоставлял удобную панельку отладки в нижней части каждой страницы на сайте.

Так как это требует некоторых дополнительных действий, мы ещё не добавили такую же панель инструментов в наш пример CRM-приложения. Сейчас самое время сделать это.

На самом деле здесь мы повторим инструкции по установке из репозитория исходного кода `yii2-debug` (см. https://github.com/yiisoft/yii2/blob/master/extensions/debug/README.md). Наша цель – не только воспользоваться этим расширением, но также и понять, как оно работает.

Добавьте зависимость от пакета `yiisoft\yii2-debug` в наше приложение:

```
$ php composer.phar require --prefer-dist yiisoft/yii2-debug "*"
```

Подождите немного, пока оно закончит пересборку. Этот пакет предоставляет специальный класс модуля `yii\debug\Module`, который нам нужно присоединить к нашему приложению, как обычно:

```
'modules' => [
    ...
    'debug' => [
        'class' => 'yii\debug\Module',
    ]
],
```

Это, однако, не всё. Нам также нужно заставить Yii 2 инициализировать этот модуль в то же самое время, что и само приложение. Для этого мы воспользуемся свойством приложения bootstrap, о котором более подробно поговорим в главе 9:

```
'bootstrap' => ['debug'],
```

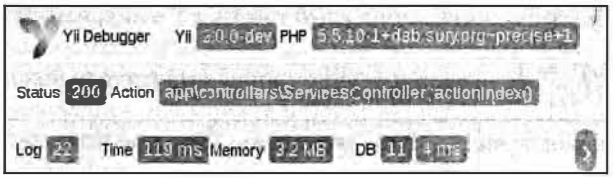
Но вы можете посмотреть документацию свойства `yii\base\Application.bootstrap`, чтобы узнать подробности того, как оно работает, прямо сейчас.

Если вы разворачиваете приложение на отдельной машине, модуль отладки не будет видим для вас, так как он защищён свойством `\yii\debug\Module::$allowedIPs`, которое по умолчанию так настроено, что модуль отображается только на соединении от локальной машины. Возможно, вам понадобится добавить эту дополнительную настройку в описание присоединяемого модуля следующим образом:

```
'debug' => [
    'class' => 'yii\debug\Module',
    'allowedIPs' => ['IP-адрес машины разработчика']
]
```

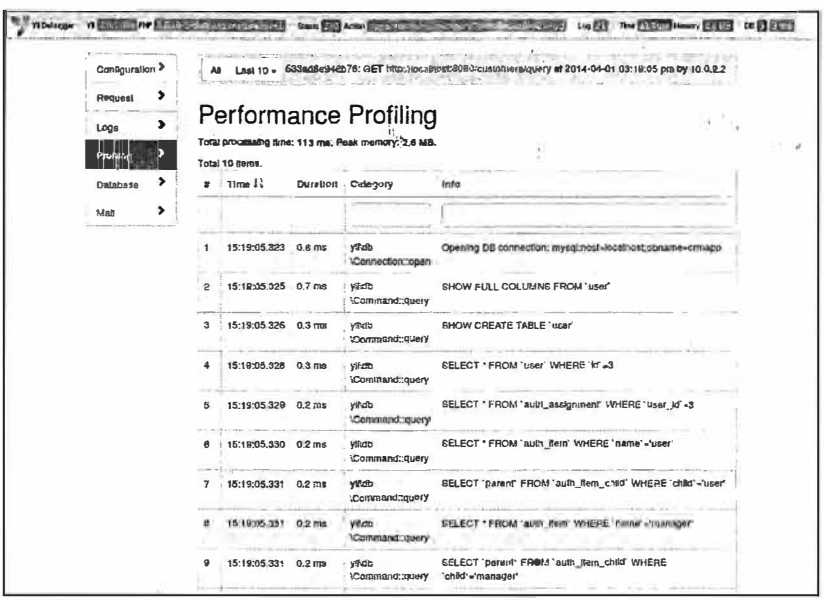
Почему нам важно инициализировать модуль отладки в начале работы приложения? Разве модуль не является всего лишь посредником для доступа к контроллерам?

На самом деле, так как модуль имеет метод `init()`, он также может быть использован ради побочных эффектов его инициализации. Модуль отладки должен быть подготовлен заранее, для того чтобы показать вам панель инструментов внизу страницы. Он делает это, отслеживая возникновение события `View::EVENT_END_BODY` и регистрируя свои материалы, когда оно наступает, что происходит (как вы, возможно, уже догадались), когда представление заканчивает отрисовку «тела» страницы и собирается регистрировать материалы. Вот как выглядит панель инструментов отладки:



В данном случае окно браузера было намеренно уменьшено по горизонтали, так что панель оказалась разбита на три строчки. Обычно все индикаторы показаны в один ряд. Все эти индикаторы – это ссылки на соответствующие разделы полного интерфейса модуля отладки.

Если вы не добавите этот модуль в бутстреппинг приложения, вы всё ещё будете иметь доступ к его полной версии по маршруту `/debug`, но панель в нижней части страниц сайта не будет отображена. Вот как выглядит раздел «Производительность» в интерфейсе модуля отладки:



Помните, что когда мы присоединяли к нашему приложению модуль Gii, мы не возились ни с чем подобным? Это именно из-за того, что модуль Gii действительно является всего лишь посредником для доступа к набору контроллеров, запакованных в отдельную папку. Он не делает ничего инвазивного с вашим приложением (по крайней

мере, пока вы не нажмёте на кнопку). Вам нужно понимать этот трюк с методом `init()`, потому что, когда вы будете разрабатывать свои собственные компоненты и модули, это поможет вам делать более сложные дополнения к приложениям. Также в главе 9 будет приведён пример применения этой возможности для построения расширений.

Теперь давайте сделаем какой-нибудь осмысленный модуль для нашего приложения самостоятельно.

Построение модуля API

У нас на данный момент есть два действия контроллера, которые:

- не нужны для пользовательского интерфейса;
- предоставляют информацию, хранящуюся в системе, как это делает традиционный публичный API приложений.

Всё это делает их идеальными кандидатами для практики по сборке модулей. Давайте сделаем модуль API, который будет содержать в себе действия, доступные на данный момент по маршрутам `/services/json` и `/services/yaml`.

Должны поддерживаться следующие два варианта использования:

- GET-запрос на `/api/services/json` должен возвращать список значений атрибутов всех зарегистрированных услуг в формате JSON;
- GET-запрос на `/api/services/yaml` должен возвращать список значений атрибутов всех зарегистрированных услуг в формате YAML.

Мы определим понятие «результат в формате JSON» как строку, которая может быть без ошибок преобразована в структуры данных PHP, с использованием метода `yii\helpers\Json::decode()`.

Мы определим «результат в формате YAML» как строку, которая может быть без ошибок преобразована в структуры данных PHP, с использованием метода `Symfony\Component\Yaml\Yaml::parse()`.

Построение набора тестов для проверки API

Мы не будем здесь делать API, совместимый с соглашением REST. Мы получим только лишь минимальный API, который будет предоставлять нам список записей, зарегистрированных в базе данных. Однако использовать тесты через всё приложение, как мы делали до сих пор, в данном случае является излишеством. Нам нужен простой способ для выполнения следующих действий:

- установить некоторые известные данные в БД;
- запросить данные у конечной точки API;
- убедиться, что возвращённые нам данные на самом деле являются сериализованным представлением данных из БД.

Codeception, который мы используем в качестве нашего исполнителя тестов, имеет в этом отношении некоторое ограничение. С одной стороны, в него включён модуль REST, который предоставляет в точности то, что нам нужно: методы `sendGET()` («послать GET-запрос»), `canSeeResponseCodeIs()` («вижу, что код отклика такой-то») и `grabResponse()` («получить отклик»). С другой стороны, модуль REST нельзя использовать одновременно с модулем WebDriver, который используется нашим набором приёмочных тестов.

Давайте просто создадим отдельный набор тестов только для проверки конечных точек API. Запустите следующий автогенератор:

```
$ ./cept generate:suite api ApiTester
```

Теперь у нас есть подкаталог `tests/api`, а также файл конфигурации `tests/api.suite.yml`.

Для того чтобы корректно настроить этот пакет тестов, нам нужно понимать следующее: эти тесты будут делать запросы к веб-серверу, предоставляющему приложение, но нам нужно подключение к базе данных, чтобы иметь возможность записывать туда данные, которые нам нужны. Таким образом, нам нужно запускать эти тесты с той же машины, на которую было развёрнуто приложение, как и в случае с функциональными тестами. Но, в отличие от функциональных тестов, данный набор тестов требует, чтобы окружение нашей программы вместе с веб-сервером было запущено и доступно, так что в этом отношении он больше похож на тесты через всё приложение. Это будет достаточно сложно реализовать на нашем тестовом сервере.

Теперь уже очевидно, что мы запутались в непроходимой массе высокоуровневых тестов, которые требуют, чтобы была подготовлена полноценная среда исполнения, а также само приложение было полностью настроено и запущено. Они медленно выполняются, и их сложно реализовать надёжным образом. Всё это произошло, потому что мы слишком полагаемся на машинерию Yii в нашем коде, ради простоты прекратив отделяться от фреймворка ещё с главы 3. В качестве грубого решения мы можем полностью отказаться от наличия работающего приложения Yii, тестируя напрямую действия контроллеров, так как они возвращают результаты своей работы, а не печатают их, скажем, напрямую на `STDOUT`. Однако это не позволит нам увидеть проблемы интеграции в Yii на реальном сервере. На самом деле вам нужно иметь несколько слоёв тестов, от модульных тестов

бизнес-правил до приёмочных тестов через всё приложение, используя пользовательский интерфейс некоторым абстрактным способом. Для того чтобы ограничить объём книги, мы показываем только самые высокоуровневые тесты. Тонкий рефакторинг был опущен по тем же причинам. Хотя её название и состоит из двух частей, эта книга всё же больше о Yii 2, нежели о веб-разработке в целом.

Зная, что мы будем выполнять тесты API на целевой машине и таким образом будем вызывать веб-сервер локально, мы теперь можем должным образом настроить набор тестов API при помощи файла конфигурации `tests/api.suite.yml`:

```
class_name: ApiTester
modules:
    enabled:
        - ApiHelper
        - PhpBrowser
        - REST
        - Db
    config:
        PhpBrowser:
            url: 'http://localhost'
        REST:
            url: 'http://localhost'
```

Посмотрим на выделенные строчки по порядку:

- Модуль `REST` посылает запросы приложению, используя модуль `PhpBrowser`.
- Нам нужен модуль `Db` для сброса базы данных после каждого запуска тестов, так как мы автоматически генерируем данные, которые будут представлены в БД.
- Модуль `PhpBrowser` вынуждает нас предоставить для приложения базовый URL. Это будет локальный сетевой интерфейс на той же машине. Веб-сервер должен работать и предоставлять доступ к нашему приложению.
- Модуль `REST` должен каким-то образом добраться до приложения, поэтому мы указываем базовый URL приложения для него. В нашем случае это очевидное дублирование кода, к сожалению, неизбежное. В общем случае это не так, поскольку модуль REST может быть настроен так, чтобы обращаться к некоторому специфическому маршруту, определённому в качестве конечной точки API, и, таким образом, все маршруты в наших тестах могли бы быть сокращены на этот префикс.

Так как мы используем модуль Db, нам нужно настроить его, чтобы он мог связаться с нашей базой данных. Мы уже делали это для наших функциональных тестов, в файле tests/functional.suite.yml. Поэтому, чтобы избежать дублирования кода, мы можем просто переместить весь раздел modules.config.Db из файла tests/functional.suite.yml в общий файл настройки codeception.yml в корневом каталоге нашего проекта.

Это не всё. Нам нужно инициализировать приложение Yii для наших тестов, чтобы иметь возможность пользоваться соединением с базой данных. Мы сделаем это в сгенерированном файле tests/api/_bootstrap.php таким же образом, как это было сделано для функциональных тестов:

```
require_once(__DIR__ . '/../../vendor/autoload.php');
require_once(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
new yii\web\Application(
   require(__DIR__ . '/../../config/web.php')
);
```

Здесь мы также создаём экземпляр именно yii\web\Application.

Не забывайте, однако, что как в случае этих тестов API, так и в случае приёмочных тестов экземпляр yii\web\Application, созданный тестовой средой, и экземпляр yii\web\Application, к которому мы будем делать запросы, – полностью отдельные сущности. Вы никоим образом не можете обмениваться данными между ними, кроме как через файловую систему или базу данных. Или, возможно, каким-либо ещё трюком из области ненормальных профессиональных обходных путей. Очень внимательно настройте тесты API таким образом, чтобы модуль Db из Codeception и само приложение разговаривали с одной и той же базой данных!

Не забудьте создать класс Tester, выполнив команду build:

```
$ ./cept build
```

Определение требований к модулю API в виде автоматических тестов

Теперь подготовка завершена, и мы можем создать сам сценарий тестирования:

```
$ ./cept generate:test api ServicesListApi
```

Мы же делаем API для списка услуг, в конце концов.

Этот тест не будет написан прозой, в стиле «Cept». Это будет обычный тестовый сценарий в виде класса PHP. Вот как наши требования могут быть выражены в виде теста Codeception:

```
/** @test */
public function ReturnsValidJson()
{
    $expectedData = [];
    $expectedData[0] = $this->registerService();
    $expectedData[1] = $this->registerService();

    $this->tester->sendGET('/api/services/json');

    $response = $this->tester->grabResponse();
    $responseData = \yii\helpers\Json::decode($response);

    $this->assertInternalType('array', $responseData);
    $this->assertEquals($expectedData[0], $responseData[0]);
    $this->assertEquals($expectedData[1], $responseData[1]);
}
```

Для простоты мы делаем здесь относительно большие шаги. Мы регистрируем в БД две услуги, чтобы сразу проверить способность нашего кода обращаться с нетривиальными наборами данных.

Затем мы отправляем GET-запрос на нашу конечную точку API и проверяем, получили ли мы отклик в формате JSON с данными, которые мы только что сохраняли в БД.

Тесты для конечной точки YAML будут в точности такими же, за исключением того, что маршрут до конечной точки будет заканчиваться на «yaml» вместо «json» и декодировать мы будем другим классом. Так что вместо строчки

```
$this->tester->sendGET('/api/services/json');
```

мы будем использовать следующую строчку:

```
$this->tester->sendGET('/api/services/yaml');
```

И ещё вместо строчки

```
$responseData = \yii\helpers\Json::decode($response);
```

мы будем использовать следующую строчку:

```
$responseData = \Symfony\Component\Yaml\Yaml::parse($response);
```

Однако до этого давайте определим «дымовой» тест, чтобы проверить, что у нас на самом деле есть конечные точки, которые нам нужны:

```
/** @test */
public function HasJsonEndpoint()
{
    $this->tester->sendGET('/api/services/json');
    $response = $this->tester->grabResponse();

    $this->tester->canSeeResponseCodeIs(200); //1
    $this->assertNotEquals('', $response); //2
}
```

Мы проверяем следующие моменты:

1. Код отклика – HTTP 200 OK.
2. Тело отклика не пусто (впрочем, оно никогда не будет пусто, так как обработчик ошибок в Yii посылает достаточно объёмное тело отклика в формате HTML в случае ошибки).

То же самое применимо для теста `HasYamlEndpoint()`.

До того, как мы сделаем сам модуль, разберёмся со вспомогательным методом `ServicesApiTest.registerService()`, который мы обошли вниманием. Идея, стоящая в основе API для чтения, заключается в том, что данные возвращаются в виде обычного «объекта» Javascript, то есть в виде набора пар ключ–значение. Поэтому мы совершенно точно не хотим полностью сериализованный объект класса `yii\db\ActiveRecord`, а только лишь атрибуты модели предметной области, которую он представляет. Поэтому мы определяем этот вспомогательный метод следующим образом:

```
private function registerService()
{
    $service = $this->imagineService();

    $service->save();

    return $service->attributes;
}
```

Таким образом, мы одним выстрелом убиваем двух зайцев: запись сохранена в БД, и мы знаем её атрибуты, которые должна вернуть наша конечная точка API.

Мы «воображаем» объект `Service`, как обычно, с помощью библиотеки `Faker`:

```
private function imagineService()
{
    $faker = \Faker\Factory::create();

    $service = new \app\models\service\ServiceRecord();
    $service->name = $faker->sentence($words = 3);
    $service->hourly_rate = $faker->randomNumber($digits = 2);

    return $service;
}
```

Это тот же самый метод, что и в приёмочных тестах для управления услугами. Запускаем тесты и видим следующее:

```
vagrant@precise64:/vagrant$ ./cept run api
Codeception PHP Testing Framework v2.0.0-beta
Powered by PHPUnit 4.1-dev by Sebastian Bergmann.

Api Tests (4) ------------------------------------------------------------------
Trying to has json endpoint (ServicesListApiTest::HasJsonEndpoint)              [F]Ok
Trying to returns valid json (ServicesListApiTest::ReturnsValidJson)          Error
Trying to has yaml endpoint (ServicesListApiTest::HasYamlEndpoint)              [F]Ok
Trying to returns valid yaml (ServicesListApiTest::ReturnsValidYaml)          Error
--------------------------------------------------------------------------------


Time: 1.2 seconds, Memory: 16.75Mb

There were 2 errors:

---------
1) ServicesListApiTest::ReturnsValidJson
yii\base\InvalidParamException: Syntax error.

#1  /vagrant/tests/api/ServicesListApiTest.php:38

---------
2) ServicesListApiTest::ReturnsValidYaml
Symfony\Component\Yaml\Exception\ParseException: Unable to parse at line 1 (near "<!DOCTYPE html>").

#1  /vagrant/vendor/symfony/yaml/Symfony/Component/Yaml/Yaml.php:67
#2  /vagrant/tests/api/ServicesListApiTest.php:57

--

There were 2 failures:

---------
1) ServicesListApiTest::HasJsonEndpoint
Failed asserting that 404 matches expected 200.

#1  /vagrant/tests/api/TestGuy.php:1463
#2  /vagrant/tests/api/ServicesListApiTest.php:16

---------
2) ServicesListApiTest::HasYamlEndpoint
Failed asserting that 404 matches expected 200.

#1  /vagrant/tests/api/TestGuy.php:1463
#2  /vagrant/tests/api/ServicesListApiTest.php:43
```

Конечно же, четыре проваленных теста за раз означают, что мы делаем слишком большие, просто огромные шаги, непозволительные для процесса TDD. Однако, во-первых, вы не пишете их сами,

а следуете руководству, а во-вторых, все эти провалы возникли из-за одной и той же простой проблемы: у нас ничего нет по маршрутам `/api/services/json` и `/api/services/yaml`.

Перемещение действий контроллера в отдельный модуль

На самом деле всё, что нам нужно, уже реализовано; просто нужно переместить это в правильное место. Мы воспользуемся силой соглашений Yii 2, начиная от маршрутов, которые нам нужно сделать доступными. Давайте восстановим структуру проекта, которая нам требуется, если мы не хотим делать какие-либо специальные настройки, помимо настроек по умолчанию.

Нам нужно сделать доступным маршрут `/api/services/json`. Это означает, что нам нужны следующие три вещи:

- модуль с идентификатором `api`, то есть некоторый наследник класса `yii\base\Module`, зарегистрированный в настройках приложения под ключом `modules.api`;
- контроллер с идентификатором `services`, то есть наследник класса `yii\web\Controller` под названием `ServicesController` (ID извлекается автоматически из имени класса), достижимый из вышеупомянутого модуля с идентификатором `api`;
- действие контроллера, имеющее идентификатор `json`, достижимое из контроллера `service`. Мы исследуем несколько других вариантов в главе 12, но в простейшем случае это означает, что нам нужен метод `actionJson()`, определённый в вышеупомянутом классе `ServicesController`.

Соглашения Yii 2 по умолчанию подразумевают, что каждый модуль находится в отдельном каталоге. Давайте создадим подкаталог `api` и поместим туда минимальное определение класса `ApiModule`:

```
namespace app\api;
use \yii\base\Module;
class ApiModule extends Module {  }
```

Так как мы следуем PSR-4 в названиях каталогов и пространств имён, нам ничего особенного не нужно делать с этим классом, чтобы он был доступен в остальном проекте. Определение класса полностью пустое, так как для модулей нет обязательных настроек.

Так как наш `ApiModule` находится в пространстве имён `app\api`, он автоматически подразумевает, что его `controllerNamespace` имеет значение `app\api\controllers`. Давайте не будем его разочаровывать и создадим

такой каталог вместе с файлом `ServicesController.php` в нём. Этот файл, очевидно, будет содержать определение класса `ServicesController` следующим образом:

```
namespace app\api\controllers;
use yii\web\Controller;
class ServicesController extends Controller
{
   // пока пусто
}
```

Мы сделали два из трёх шагов. Всё, что осталось сделать, – это перенести методы `actionJson()` и `actionYaml()` из файла `controllers/SiteController.php` в это определение нового класса.

> Заметьте, что у нас появились два класса с одинаковыми названиями. Раньше, в Yii 1.1.x, это было серьёзной проблемой для автозагрузчика, но теперь у нас везде пространства имён, и до тех пор, пока различаются полностью определённые имена классов, у нас всё в порядке.

Теперь у нас есть законченный модуль API (поверьте нам в этом на слово). Всё, что нам нужно сделать, – это сказать приложению Yii, что он существует, разместив его объявление в конфигурации в разделе `modules`:

```
'modules' => [
   ...
   'api' => [
      'class' => 'app\api\ApiModule'
   ]
],
```

Теперь мы запускаем тесты, и все четыре проходят на этот раз (при условии что мы на самом деле реализовали действия для сериализации в JSON и YAML в 4-й главе и перенесли их сюда):

```
vagrant@precise64:/vagrant$ ./cept run api
Codeception PHP Testing Framework v2.0.0-beta
Powered by PHPUnit 4.1-dev by Sebastian Bergmann.

Api Tests (4) ------------------------------------------------------------------------
Trying to has json endpoint (ServicesListApiTest::HasJsonEndpoint)                  Ok
Trying to returns valid json (ServicesListApiTest::ReturnsValidJson)                Ok
Trying to has yaml endpoint (ServicesListApiTest::HasYamlEndpoint)                  Ok
Trying to returns valid yaml (ServicesListApiTest::ReturnsValidYaml)                Ok
--------------------------------------------------------------------------------------


Time: 1.13 seconds, Memory: 16.00Mb
```

Как уже было сказано в этой главе, даный пример модуля API не соответствует REST. Если вам нужен REST API, скорее всего, вам нужно проверить встроенный класс `yii\rest\Controller`, так как это контроллер, специально настроенный для запросов, соответствующих REST API. Имеет смысл также взглянуть и на всё семейство классов в пространстве имён `yii\rest`.

Если вы откроете в браузере конечную точку JSON (уже после того, как тесты API услужливо добавили несколько записей в БД), то увидите следующее:

Простого способа показать снимок экрана для конечной точки YAML нет, так как все браузеры на момент написания этого текста будут пытаться скачать отклик сервера в файл, вместо того чтобы показать его в текстовом виде на экране.

Ретроспектива о модулях, упомянутых в предыдущих главах

Давайте вспомним, где мы ранее упоминали модули и ради простоты отмахнулись от них.

Мы упомянули модули в главе 2, когда создавали файл конфигурации для нашего приложения и узнали, что настройка `id` является обязательной. Теперь мы в подробностях узнали, что приложение является просто специальным видом модуля и все модули имеют идентификатор. Для обычного модуля этот идентификатор определён в настройке `modules` того модуля, которому он принадлежит. В примере модуля API, который мы только что реализовали, мы использовали идентификатор `api`. Само приложение, конечно же, нигде ни в каком разделе `modules` не упомянуто, поэтому оно должно получить идентификатор каким-то другим образом. Если честно, приложение нуждается в идентификаторе по иным причинам, нежели обычный модуль (оно же всё-таки *особенный* вид модуля), но он всё равно является обязательным.

Позже в той же главе мы обратили внимание, что мы остановились на структуре проекта, удивительно похожей на базовый шаблон приложения от Yii 2. Теперь вы понимаете, почему это произошло, так как мы подробно обсудили в этой главе различные настройки и соглашения, приводящие именно к такой структуре.

В главе 3 мы на самом деле *воспользовались* модулем, не зная, что это такое. Мы тогда узнали, что модуль – это нечто, чьё полностью определённое имя класса мы объявляем в разделе конфигурации под названием `modules`. После объявления вы получаете в вашем приложении набор дополнительных маршрутов, доступных для просмотра. Вы также можете заметить на этот раз, что мы можем устанавливать значения свойств модулей прямо во время объявления их в конфигурации. Это не поможет, когда вам нужно настраивать модуль во время выполнения в зависимости от данных, недоступных на этапе создания приложения (в этом случае вашим единственным выходом будет переопределение метода `init()` или перехват событий). Однако это может серьёзно помочь, если вы будете разрабатывать какой-то готовый к распространению и повторному использованию модуль, как сам Gii.

Пожалуйста, обратите внимание, что везде, где мы говорим о модулях, мы говорим про конкретное понятие фреймворка Yii 2, реализованное в виде класса `yii\base\Module`. Например, в Codeception существует совершенно иное представление о «модуле», что никак не помогает понять этот важный термин.

В *главе 4 «Рендерер»*, в процессе обсуждения того, как Yii 2 находит файлы представлений для вызовов метода `render()`, для упрощения объяснений мы достигли момента, когда нам пришлось сделать вид, будто модули вообще не существуют. Теперь мы знаем, что `viewPath`, под которым компонент `View` ищет файлы представлений, может быть определён для каждого модуля по отдельности. На самом деле эта настройка *именно так* и определена по умолчанию. Поскольку само приложение является модулем, у него есть свой `viewPath`; но когда мы выполняем действие контроллера, принадлежащее контроллеру, достигнутому через какой-то модуль, вместо `viewPath` приложения используется `viewPath` этого модуля. Те же правила применимы к настройке `layoutPath`, а настройка `layout` уже обсуждалась в той же таблице в том же разделе.

В результате, если вы будете перераспределять контроллеры и в особенности перераспределять *действия* контроллеров, вам следует не забывать перемещать между модулями и соответствующие файлы представлений!

Помимо важности для отрисовки представлений, система модулей серьёзно влияет на маршрутизацию. Если вы внимательно следили, то, возможно, уже заметили, что мы безо всяких особых действий можем иметь маршруты произвольной длины, которые выглядят как пути в файловой системе, просто определив иерархию модулей в конфигурации приложения. Но это – тема *главы 12 «Маршрутизация»*.

Итоги

Yii 2 реализует составной паттерн MVC, используя систему модулей. Каждый модуль может ссылаться на контроллеры и представления. Модули присоединяются к самому приложению и к другим модулям, создавая иерархию.

Используя систему модулей, у вас в приложении может быть произвольное количество маршрутов произвольной длины, и все они будут иметь следующий общий вид:

```
/id-модуля/id-модуля/.../id-модуля/id-контроллера/id-действия
```

Эта базовая система маршрутизации будет собрана при помощи одних лишь классов PHP, корректно связанных друг с другом соглашениями Yii 2. Никаких дополнительных настроек маршрутизации в Yii 2 не понадобится.

В следующей главе мы озаботимся вопросами безопасности и производительности. Yii 2 предоставляет нам некоторые служебные действия, для того чтобы без какой-либо серьёзной работы с нашей стороны сделать интерфейс нашего приложения немного более ухоженным и соответствующим лучшим методам разработки.

Глава 8

Поведение в целом

В этой главе мы посмотрим на возможности Yii 2, которые в некотором смысле не являются локальными. Они сами по себе не помогут вам реализовать новые возможности, но без них вам будет гораздо сложнее создать надёжные приложения как с точки зрения разработчика, так и с точки зрения пользователя.

Вот список возможностей, которые мы рассмотрим в этой главе:

1) журналирование;
2) обработка ошибок;
3) кэширование;
4) компиляция материалов.

Мы, как обычно, сосредоточимся на практических аспектах использования этих возможностей, поскольку точные технические детали их реализации вы можете посмотреть как в превосходной документации Yii, так и непосредственно в исходном коде.

Давайте начнём с того, как в Yii 2 реализовано журналирование событий.

ВОЗМОЖНОСТЬ: журнал событий

Чаще всего вы хотите знать, как меняется состояние приложения во время разработки и его отладки. В покрытом Сетью мире, основанном на PHP, обычно нет привычки использовать для просмотра индивидуальных переменных инструменты настоящего мужика вроде Xdebug, как это делают бородатые программисты на C. Большая часть проблем может быть решена продуманным размещением операторов печати, которые показывают искомые значения там, где используются.

Yii предоставляет нам возможности организовать это подглядывание во внутреннее состояние и избавить нас от необходимости писать

везде `var_dump($variable);die()`. Вы можете из любого места приложения сохранять сообщения в определённые контролируемые и централизованные места, из которых они могут быть прочитаны без необходимости вмешиваться в процесс отрисовки.

Понятие сообщения журнала **(log message)** в Yii 2 – это сущность, состоящая из следующих элементов:

- строка текста самого сообщения;
- степень важности;
- категория;
- метка времени;
- трассировка стека на момент записи сообщения.

Мы можем менять первые три элемента.

Очевидно, можно написать всё, что угодно, в качестве самого сообщения, лишь бы оно было строкой.

Степени важности предопределены заранее; технически вы можете заставить Yii записать сообщение с нестандартной степенью важности, но гарантии, что оно будет обработано так же, как и остальные, у вас уже не будет. Вот список констант, которые вы можете использовать в качестве степеней важности сообщений журнала:

- `yii\log\Logger::LEVEL_TRACE` – для сообщений, предназначенных только для отладки;
- `yii\log\Logger::LEVEL_INFO` – для сообщений общего характера;
- `yii\log\Logger::LEVEL_WARNING` – согласно названию, сообщения-предупреждения;
- `yii\log\Logger::LEVEL_ERROR` – согласно названию, сообщения об ошибках.

За исключением `LEVEL_TRACE`, о котором вы вскоре прочитаете, никакого различия между степенями принудительно не установлено. Хотя, конечно, ради собственного рассудка вам лучше присваивать степени важности соответственно журналируемым сообщениям.

Есть ещё псевдостепени `LEVEL_PROFILE`, `LEVEL_PROFILE_BEGIN` и `LEVEL_PROFILE_END`, но вам никогда не следует пользоваться ими напрямую, вместо этого применяя для профилирования производительности приложения специальные вспомогательные методы. Об этом будет также рассказано чуть позже.

«Категория» – это произвольная строка, которая может быть использована для группировки связанных между собой сообщений. В своих собственных методах Yii почти всегда использует в качестве категорий сообщений сверхглобальную константу PHP `__METHOD__`, так что он фактически предлагает вам группировать сообщения по

полностью определённому имени исполняющегося в данный момент метода. Как вы увидите позже в этом разделе, такой метод группировки имеет один положительный аспект.

Метка времени устанавливается вызовом `microtime(true)`, так что это автоматически вычисленное время на момент записи сообщения, с точностью до микросекунд.

Трассировка стека записывается из того, что возвращает вызов встроенной в PHP функции `debug_backtrace()`. Записывать его или нет, указывает значение свойства `Yii::$app->log->traceLevel` или, что то же самое, настройки приложения `components.log.traceLevel`. По умолчанию она выставлена в 0 (нуль).

> Должно быть очевидным то, что так как трассировка стека привязывается к *каждой* записи журнала, установка `traceLevel` на что угодно выше `0` серьёзно повредит производительности приложения.

Сам по себе Yii журналирует достаточно много сообщений на разных этапах своей жизни. Вам, возможно, даже не понадобится записывать дополнительные сообщения. Это, конечно, ложь, но мы же всё равно можем надеяться на лучшее, правда ведь? Вот как делается запись сообщений в журнал в Yii 2:

Название метода	Смысл
`Yii::trace($message, $category)`	Записывает сообщение степени `LEVEL_TRACE` в заданной категории. Если константа `YII_DEBUG` установлена в значение `false`, вместо этого ничего не делает
`Yii::warning($message, $category)`	Записывает сообщение степени `LEVEL_WARNING` в заданной категории
`Yii::error($message, $category)`	Записывает сообщение степени `LEVEL_ERROR` в заданной категории
`Yii::info($message, $category)`	Записывает сообщение степени `LEVEL_INFO` в заданной категории

Так как вызов `Yii::trace()` молча ничего не делает, если константа `YII_DEBUG` выставлена в `false` (что фактически значит «в реально работающем приложении»), он прекрасно подходит для записи состояния каких-либо переменных в конкретный момент времени, с целью отладки.

Сохранение сообщений журнала

Вышеперечисленные четыре метода записи сообщений журнала не сохраняют их никоим образом. Всё, что они делают, – это сообща-

ют механизму диспетчеризации сообщений о том, что эти сообщения нужно отправить в *цели журналирования*, которые мы как разработчики определили для данного приложения.

Давайте посмотрим на пример конфигурации приложения, который указывает Yii сохранять все сообщения в базе данных, но только не сообщения степени **LEVEL_ERROR**, которые нужно дополнительно отправлять прямо на адрес электронной почты менеджера проекта:

```
'components' => [
   'log' => [
      'traceLevel' => 3,
      'targets' => [
         'all_messages' => [
            'class' => 'yii\log\DbTarget',
            'levels' => ['info', 'trace', 'warning', 'error']
         ],
         'problems' => [
            'class' => \yii\log\EmailTarget::className(),
            'levels' => \yii\log\Logger::LEVEL_ERROR,
            'message' => [
               'to' => 'pm@crmapp.us'
            ]
         ]
      ],
   ],
...
],
```

Обратите внимание на выделенные части. Видно, что компонент `log` приложения содержит в себе массив «целей» (*targets* – англ.), и класс цели определяет метод, которым это сообщение будет сохранено. Названия целей журналирования произвольные и используются только в качестве подсказки для сопровождающего.

В вышеприведённом фрагменте конфигурации можно видеть два интересных приёма.

Во-первых, можно использовать для указания полностью определённого имени класса обычный строковый литерал, а можно вызвать `className()` на нужном классе. Это специальный статический метод, определённый в Yii практически на всех входящих в него классах. Этот метод вернёт нам строку с корректно сформированным полностью определённым именем класса, и вручную не нужно будет ничего писать. Это может очень пригодиться, когда вы начнёте перемещать классы между пространствами имён.

Во-вторых, в настройке `components.log.targets.[].levels` вы можете указать степени в виде списка текстовых названий степеней важности, а можете использовать побитовую комбинацию констант из набора `Logger::LEVEL_*`,

например так: `Logger::LEVEL_ERROR | Logger::LEVEL_WARNING` (в данном случае использовано побитовое ИЛИ).

У нас есть следующие доступные для использования цели журналирования:

- `yii\log\EmailTarget` отправляет сообщения по электронной почте;
- `yii\log\FileTarget` записывает сообщения в файл, причём **ротация логов** предоставляется по умолчанию;
- `yii\log\DbTarget` сохраняет сообщения в виде записей в базе данных. У вас должна быть заранее подготовлена таблица в БД, составленная согласно документации к свойству `yii\log\DbTarget.logTable`;
- `\yii\log\SyslogTarget` отправляет сообщения в системное приложение **syslog**, используя встроенные в PHP вызовы `openlog()`, `syslog()` и `closelog()`.

`EmailTarget` – особенный зверь. В первую очередь он не «сохраняет» сообщения в обычном смысле, вместо этого отправляя их на удалённый адрес электронной почты. Во-вторых, для его работы требуется компонент отправки электронной почты, настроенный и присоединённый к приложению.

Установка компонента отправки электронной почты для отправки сообщений журнала

Для нужд отправки электронной почты вместе с Yii 2 поставляется ещё одно расширение, которое привносит в приложение возможности библиотеки **SwiftMailer** (см. http://swiftmailer.org/). В качестве альтернативы можно было бы написать реализацию наследника класса `\yii\mail\BaseMailer` самостоятельно, но мы лучше оставим эту идею и просто объявим только что упомянутое расширение в качестве зависимости нашего приложения:

```
$ php composer.phar require --prefer-dist yiisoft/yii2-swiftmailer "*"
```

Это расширение предоставит нам класс `\yii\swiftmailer\Mailer`, который мы можем использовать как компонент отправки писем. Вот пример минимальной конфигурации, который будет использовать встроенную в PHP функцию `mail()` в качестве механизма доставки:

```
'components' => [
    ...
    'mail' => [
        'class' => yii\swiftmailer\Mailer::className(),
```

```
        'messageConfig' => [
            'charset' => 'UTF-8',
            'from' => 'noreply@crmapp.me'
        ],
        'transport' => [
            'class' => 'Swift_MailTransport',
        ],
    ],
    ...
]
```

Самые хитрые детали конфигурации были выделены.

Настройка `components.mail.messageConfig` определяет атрибуты всех сообщений электронной почты, которые будут отправлены этим компонентом. Так как в XXI веке в переписке по электронной почте поле «`From:`» является обязательным, нам нужно явно его определить, Yii здесь нам никак не поможет. Опять же, кодировка UTF-8 упомянута для тех систем, которые ещё не знают, что на дворе год 20xx, и всё ещё используют что-то, отличающееся от Юникода.

Настройка `components.mail.transport` определяет, какой метод отправки из пакета SwiftMailer должен и будет использован. Самые обычные, вероятно, – это `Swift_MailTransport`, который использует встроенную в PHP функцию `mail()`, и `Swift_SmtpTransport`, который может связываться с удалёнными серверами SMTP, включая корректную авторизацию и прочее. Настоятельно рекомендуем просмотреть документацию SwiftMailer (см. http://swiftmailer.org/docs/sending.html#transport-types), чтобы ознакомиться с другими интересными методами отправки, например отказоустойчивым методом и методом, балансирующим нагрузку: https://github.com/swiftmailer/swiftmailer/tree/master/lib/classes/Swift/Transport.

Чтение сохранённых записей журнала

Когда сообщение будет сохранено по цели журналирования, вы, возможно, когда-нибудь его прочтёте. Вот как выглядит сообщение, сохранённое при помощи `FileTarget`, то есть в файл:

```
2014/04/12 01:44:27 [10.0.2.2][3][pccg3bq84sm6b4mnutsa07lk31][trace]
   [yii\base\Controller::runAction] Route to run: site/index
```

Оно было разделено на несколько строк только для того, чтобы уместиться на странице книги. В файле оно занимает одну строку. Можно видеть, что у нас есть метка времени, набор меток, заключён-

ных в квадратные скобки, и сам текст сообщения. Вот метки, которые были записаны, в порядке очерёдности:

1) IP-адрес пользователя;
2) ID пользователя или символ «-» в случае, если он не аутентифицирован;
3) ID сеанса пользователя или символ «-» в случае, если он не аутентифицирован;
4) степень важности в текстовом виде;
5) категория сообщения.

Видно, что категория системных сообщений по умолчанию – это полностью определённое имя метода, из которого это сообщение было отправлено. Вот как выглядит сообщение от неаутентифицированного запроса, при условии что настройке `components.log.traceLevel` было присвоено значение 1:

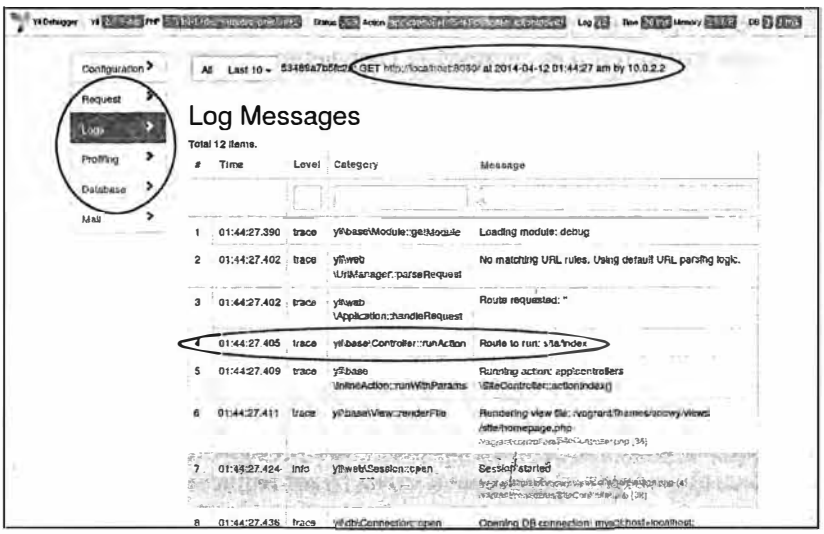
```
2014/04/12 01:44:27 [10.0.2.2][-][-][error][yii\web\HttpException:404]
   exception 'yii\base\InvalidRouteException'
   with message 'Unable to resolve the request "favicon.ico".'
   in /vagrant/vendor/yiisoft/yii2/base/Module.php:440
```

Предупреждаем: *это* сообщение *не было* на одной строчке! Последняя строчка, начинающаяся с «`in /vagrant/...`», – это специальным образом скомпонованный результат вызова `debug_backtrace()`, он выводится в файле на отдельной строке с отступом в четыре пробела. Каждый элемент трассировки стека будет записан на отдельной строке.

В общем случае формат сообщения определяется методом `\yii\log\Target.formatMessage()`. Они будут выглядеть одинаково, записанные в файл или отправленные по почте. Этот метод позволяет тексту сообщения содержать произвольные переносы строк и, как уже было сказано, дописывает по одной строчке на каждый элемент трассировки журналируемого вызова. Поэтому вам следует понимать, что сообщение журнала в Yii 2 часто не занимает ровно одну строчку, и ваш парсер логов должен быть готов к этому. Например, вы можете считать «сообщением журнала» весь текст между двумя метками времени, расположенными в начале строк.

Вы можете очень сильно упростить разбор журнала, если будете использовать `DatabaseTarget`, которая сохраняет каждый элемент сообщения в отдельном поле таблицы в БД.

Расширение с модулем отладки (которое мы внедрили в наш проект ранее в *главе 7*) использует свой собственный секретный класс `\yii\debug\LogTarget` для хранения сообщений полностью отдельно от ваших настроек журналирования. В случае установки модуля отладки через Composer этот класс находится в файле `vendor/yiisoft/yii2-debug/LogTarget.php`. Вот как первое из вышеупомянутых сообщений выглядит в модуле отладки:



Журнал, предоставляемый модулем отладки, очень сильно структурирован, потому что на самом деле Yii не сохраняет сообщений одно за другим. Он буферизует сообщения, пока текущий запрос не будет выполнен (в обычном случае) или пока не будет достигнуто максимальное количество сообщений, настраиваемое в параметре конфигурации `components.log.flushInterval`. Его значение по умолчанию – 1000. Обычно у вас будет (намного) меньше, чем 1000 сообщений журнала, так что можно относительно безопасно допустить, что на один запрос к приложению будет сохранён один пакет сообщений.

В конце пакета сообщения, в момент, когда сообщения действительно будут записываться в предопределённые цели, компонент журналирования запрашивает некоторые глобальные переменные, то есть «контекст» текущего запроса, и выполняет на них `var_export()`, записывая результат в качестве завершающего сообщения журнала.

Переменные, которые будут записаны, перечислены в настройке `logVars` каждой отдельной цели журналирования. По умолчанию сохраняются переменные `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION` и `$_SERVER`, что в целом полностью описывает запрос, но вы можете указать название любой другой переменной, лишь бы она была доступна через суперглобальную переменную `$GLOBALS`.

Итак, зная об этом поведении, модуль отладки отражает его, группируя сообщения по пакетам и проделывая несколько других вещей, чтобы предоставить вам полную информацию о приложении, подверженном отладке.

Заметьте, что у вас нет никакого контроля над этой особенной целью журналирования, которую использует модуль отладки. Он, впрочем, всё равно настроен так, чтобы записывать так много информации, насколько это возможно.

Настоятельно рекомендуем попробовать модуль отладки самостоятельно, потому что он достаточно прост, чтобы объяснение его работы в виде текста было более громоздким, нежели простое прокликивание интерфейса мышкой.

Хоть модуль отладки и не даёт вам контроля над своими сообщениями журнала, у вас всё ещё есть полный контроль над сообщениями, которые записываются вашим собственным приложением. Есть два способа изменить их формат: один предоставлен разработчиками Yii, а другой, как обычно, инвазивный.

Во-первых, у вас есть контроль над токенами с IP, ID пользователя и ID сеанса. Их можно заменить на что угодно, меняя настройку `prefix` каждой конкретной цели журналирования. Значение этой настройки должно быть анонимной функцией, которая ожидает единственного аргумента `$message`. Структуру `$message` можно подсмотреть в блоке самодокументации свойства `\yii\log\Logger::$messages`:

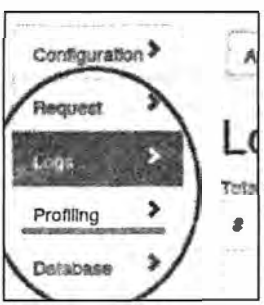
```
[
  [0] => сообщение (тип mixed, может быть строкой или какими-нибудь
сложными данными, например объектом исключения)
  [1] => степень важности (integer)
  [2] => категория (string)
  [3] => метка времени (float, получен вызовом microtime(true))
  [4] => трассировка стека (array, содержит отдельные её элементы)
]
```

По умолчанию IP, ID пользователя и ID сеанса в квадратных скобках форматирует метод `\yii\log\Target::getMessagePrefix()`, и он даже не использует предоставленного аргумента `$message`. Вы же можете написать всё, что хотите.

Второй, инвазивный способ заключается в том, чтобы написать свою собственную цель журналирования, расширяя класс `\yii\log\Target`, и переопределить там метод `formatMessage()`. Этим способом вы получите полный контроль над тем, как должно быть сериализовано `$message`.

ВОЗМОЖНОСТЬ: профилирование

Возможно, на предыдущем снимке экрана вы заметили пункт меню **Profiling** («Профилирование»):



На самом деле профилирование, то есть замеры производительности, не является отдельной функциональностью в архитектуре Yii, но является частью механизма журналирования. Мы пропустили его, чтобы не переусложнять объяснения.

Есть ещё три степени важности сообщений журнала:

- `yii\log\Logger::LEVEL_PROFILE` – показывает, что сообщение предназначено для целей профилирования;
- `yii\log\Logger::LEVEL_PROFILE_BEGIN` – помечает начало блока замера производительности;
- `yii\log\Logger::LEVEL_PROFILE_END` – помечает конец блока замера производительности.

Вы не можете указать степени `LEVEL_PROFILE_BEGIN` и `LEVEL_PROFILE_END` в настройке `levels` цели журналирования, и они, вообще говоря, не предназначены для того, чтобы их использовали вручную. Механизм профилирования в Yii 2 имеет следующий простой API:

Метод	Как используется
`\yii\BaseYii::beginProfile( $token, $category = 'application' )`	Начинает блок замера производительности, вставляя специальное сообщение со степенью важности `LEVEL_PROFILE_BEGIN`. `$token` – это уникальное название блока. Указание конкретной `$category` может быть использовано для того, чтобы фильтровать различные блоки профилирования позже, в вызове `getProfiling()`
`\yii\BaseYii::endProfile( $token, $category = 'application')`	Заканчивает блок замера производительности, вставляя специальное сообщение со степенью важности `LEVEL_PROFILE_END`. `$token` должен быть точно такой же, какой использовался в соответствующем вызове `beginProfile()`, или блок будет пропущен. Значение `$category` здесь несущественно (при фильтрации в вызове `getProfiling()` рассматриваются только категории, записанные вызовом `beginProfile()`), так что оно только для категоризации самого сообщения
`\yii\log\Logger::getProfiling( $categories, $excludeCategories )`	Проходит по всем собранным сообщениям на момент вызова и вычисляет промежутки времени между соответствующими парами сообщений степеней `LEVEL_PROFILE_BEGIN` и `LEVEL_PROFILE_END`. Вы можете указать конкретные категории сообщений, которые следует учесть в отчёте. Описание возвращаемого результата приведено ниже
`\yii\log\Logger::getDbProfiling()`	Вспомогательный метод для получения статистики использования базы данных. Возвращает массив с количеством выполненных команд SQL и суммарным временем, проведённым за общением с базой данных
`\yii\log\Logger::getElapsedTime()`	Микроскопический метод, который возвращает время, включая микросекунды, с самого начала запуска фреймворка Yii. Однако бойтесь, ибо внутри он вычитает друг из друга числа с плавающей точкой; не ждите от него какой-либо сверхъестественной точности

Вышеприведённые сигнатуры методов не очень полезны в понимании того, как добраться до самих методов в уже запущенном приложении Yii. Вот пример рабочего кода:

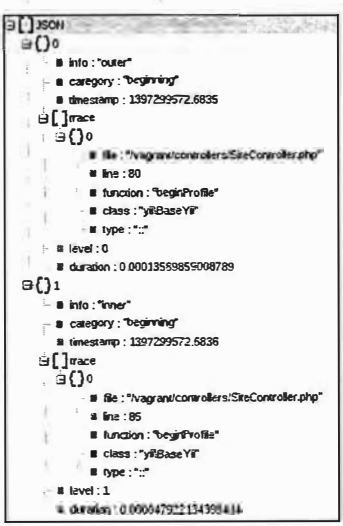
```
public function actionProfile()
{
    Yii::beginProfile('outer', 'beginning');
    Yii::getLogger()->log('first', Logger::LEVEL_PROFILE);
    Yii::trace('second');
```

```
        Yii::info('third');

        Yii::beginProfile('inner', 'beginning');
        Yii::warning('fourth', 'nonapplication'); // note the category
        Yii::error('fifth');
        Yii::endProfile('inner', 'ending');

        Yii::endProfile('outer', 'ending');

        $result = Yii::$app->response;
        $result->data = Yii::getLogger()->getProfiling(
           ['beginning', 'application', 'ending']);
        $result->format = Response::FORMAT_JSON;

        return $result;
    }
```

Как уже было сказано, для сообщений журнала, записанных вручную, категорией по умолчанию является «application», так что мы указываем в вызове getProfiling(), что мы заинтересованы только в ней.

Заметьте, что мы можем вкладывать блоки профилирования друг в друга.

Приведённое здесь действие контроллера составляет в конечном счёте текст, кодирующий объект в формате JSON. Вот графическое представление результата:

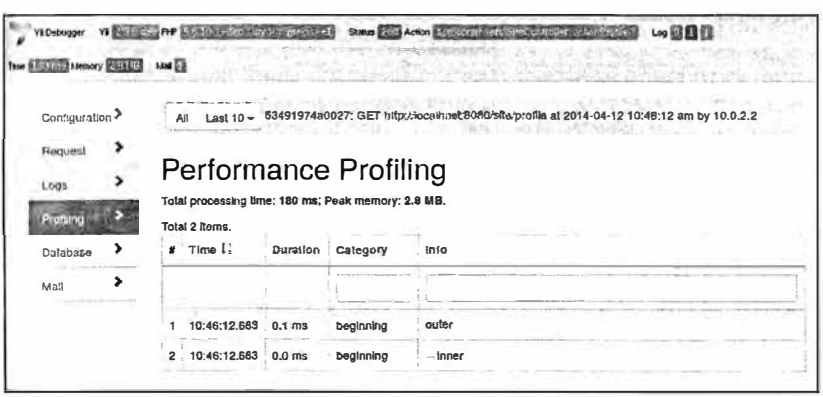


Обратите внимание, что промежуточные сообщения были опущены, Yii показал только сообщения, записанные вызовами `beginProfile()`.

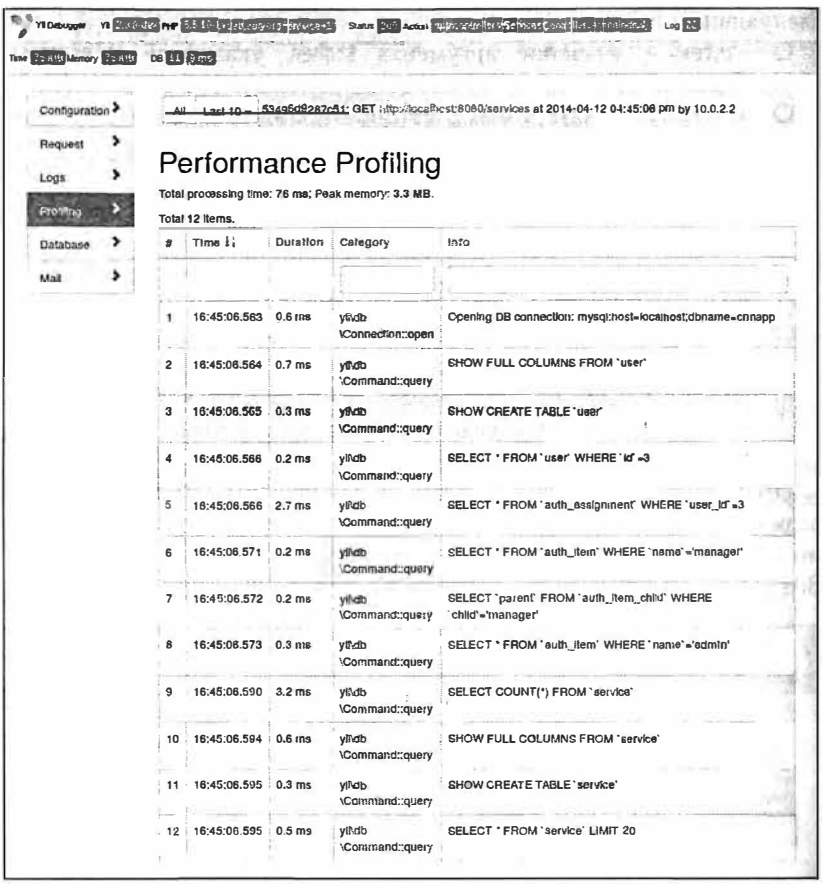
Профилировщик вычисляет время между соответствующими парами сообщений степеней `LEVEL_PROFILE_BEGIN` и `LEVEL_PROFILE_END`, которые находятся на одном и том же уровне вложенности. Результат профилирования каждого уровня возвращается в виде массива со следующими полями:

- `«info»` – значение аргумента `$token`, указанное в вызове `beginProfile()`;
- `«category»` – категория, к которой принадлежит данный процесс профилирования. Он тоже берётся из вызова `beginProfile()`;
- `«timestamp»` – метка времени вызова `beginProfile()`;
- `«trace»` – трассировка стека для сообщения профилирования (чтобы знать, в каком методе оно было записано);
- `«level»` – индикатор того, насколько глубоко в стеке блоков профилирования мы находимся, начиная с `0`;
- `«duration»` – время в секундах (!) от вызова `beginProfile()` до вызова `endProfile()`. Это значение вычисляется вычитанием результатов вызовов `mictotime($as_float = true)`, так что это дробное число, представляющее продолжительность в секундах, с некоторой точностью после запятой.

Примера использования этого низкоуровневого API для просмотра результатов профилирования не будет по той простой причине, что модуль отладки уже содержит в себе реализацию их отображения. Вот тот же самый запрос, открытый во вкладке **Profiling**:



Конечно же, здесь особо нечего смотреть, потому что всё, что мы сделали, – это пометили два участка кода. По умолчанию Yii 2 помечает для профилирования каждый вызов к СУБД, так что любой запрос, который делает обращение к базе данных, является гораздо более интересной целью для изучения:



Мы ничего специально не сделали, чтобы получить эту крайне полезную информацию, Yii предоставляет её по умолчанию. Это наиболее ценный источник сведений об узких местах в производительности вашего приложения. Однако профилированием покрыты только вызовы к БД, всё остальное вы должны разметить вручную.

Подробности обработки ошибок

Мы на самом деле детально рассмотрели обработку ошибок и исключений ранее, в *главе 6*. Однако мы обошли стороной исключения, не связанные с кодами состояния HTTP, и едва упомянули способ, которым вы можете по-настоящему управлять тем, как Yii работает с ошибками.

Важный момент – в том, что вам, возможно, никогда не понадобится менять то, как Yii обрабатывает ошибки, потому что это действительно хорошо продуманный алгоритм, покрывающий все виды исключительных ситуаций, возможных в приложении на основе PHP, так что вряд ли вам понадобится что-то к нему добавлять.

История о том, как Yii 2 работает с ошибками, довольно проста и коротка. К приложению Yii по умолчанию присоединён специальный компонент под названием `\yii\base\ErrorHandler`. Если быть точным, то к веб-приложению Yii присоединено расширение этого базового обработчика ошибок, представленное объектом класса `\yii\web\ErrorHandler`. Этот компонент содержит три метода, которые регистрируются в среде выполнения PHP как обработчик исключений, обработчик ошибок и обработчик фатальных ошибок соответственно. Все эти методы преобразуют все три типа отслеживаемых проблем в исключение того или иного типа и передают это исключение в метод под названием `renderException`, таким образом единообразно обрабатывая все возможные ошибки, которые могут случиться в коде PHP, и показывая информацию о них пользователю.

Вот точная копия кода, который привязывает всю обработку ошибок к Yii, на момент написания этой главы:

```
public function register()
{
    ini_set('display_errors', false);
    set_exception_handler([$this, 'handleException']);
    set_error_handler([$this, 'handleError']);
    if ($this->memoryReserveSize > 0) {
        $this->_memoryReserve = str_repeat('x', $this-
>memoryReserveSize);
    }
    register_shutdown_function([$this, 'handleFatalError']);
}
```

Переменная `$this` является экземпляром `\yii\base\ErrorHandler`. Как и было сказано, все три метода: `handleException()`, `handleError()` и `handleFatalError()` – в конечном счёте разрешаются в вызов `\yii\base\ErrorHandler::renderException(\Exception $exception)`. Что просто великолепно, так то, как подобным образом мы имеем всего одно место для внесения изменений.

Давайте посмотрим детально, что именно будет показано клиенту в зависимости от различных условий. Если начать с того, что ошибки и фатальные ошибки PHP уже превращены в исключения, у нас есть следующие варианты:

Условия	Что показываем
Глобальный компонент `Response` имеет значение поля `format`, отличающееся от `FORMAT_HTML` (которое является значением по умолчанию)	Исключение будет преобразовано в массив, согласно методу `\yii\web\ErrorHandler::convertExceptionToArray`, и этот массив будет отрисован, согласно тому, как выставлено значение `Yii::$app->response->format`
Константа `YII_ENV_TEST` имеет значение `true`	Исключение будет преобразовано в строку, согласно методу `\yii\base\ErrorHandler::convertExceptionToString`, и эта строка будет отрисована закодированной для HTML внутри пары тегов `<pre>` как единственный отклик от сервера (то есть сервер вернёт только фрагмент HTML, а не полную страницу)
Запрос является AJAX-запросом, что проверяется проверкой значения `$_SERVER['HTTP_X_REQUESTED_WITH']`	
`YII_DEBUG` установлено в `true`, и исключение не является наследником `\yii\base\UserException`	Исключение будет отрисовано, используя файл представления, на который ссылается свойство `\yii\web\ErrorHandler::$exceptionView`, которое по умолчанию настроено показывать полный отчёт об исключении, показанный ранее в *главе 6*
У компонента `ErrorHandler` настроено свойство `errorAction` (значением должна быть строка, являющаяся некоторым маршрутом в приложении, например `site/error`)	Будет выполнено действие контроллера, достижимое по этому маршруту. Ожидается, что оно проверит значение свойства `Yii::$app->errorHandler->exception` и каким-либо образом его отрисует. Традиционно этим действием является `\yii\web\ErrorAction`
Все прочие случаи	Исключение будет отрисовано, используя файл представления, на который ссылается свойство `\yii\web\ErrorHandler::$errorView`, которым по умолчанию является короткий отчёт об исключении для конечных пользователей, показанный ранее в *главе 6*, непосредственно перед полным отчётом об исключении

Эти варианты исключают друг друга сверху вниз, то есть используется первый по порядку подходящий вариант.

Обратите внимание на то, что исключения, наследующие классу `UserException`, всегда отрисовываются «представлением ошибки», которое короткое и неинформативное. Исключения этого типа предназначены для конечного пользователя, то есть они являются «исключительной ситуацией, случившейся из-за действий пользователя». Поэтому даже на реальном сервере мы будет показывать не просто «internal server error occured», но точное сообщение об ошибке, хотя и без подробностей.

Во всех случаях, если было брошено исключение, наследующее классу `HttpException`, будет установлен код состояния HTTP, соответствующий этому исключению. В противном случае будет использован код состояния 500.

Вышеописанная таблица описывает алгоритм отображения исключения, закодированный в методе `\yii\web\ErrorHandler::renderException`. Если вам когда-нибудь понадобится использовать другую логику для отображения исключений, вы свободно можете переопределить этот метод в наследнике класса `\yii\web\ErrorHandler` и присоединить этот новый класс к настройке `components.errorHandler` приложения. Если же вам когда-нибудь понадобится использовать другую логику *обработки* исключений, ошибок или фатальных ошибок PHP, вы даже можете что-нибудь переопределить в базовом классе обработки исключений `\yii\base\ErrorHandler`. Хотя сложно представить пример ситуации, когда это вам может понадобиться.

ВОЗМОЖНОСТЬ: действие контроллера, обрабатывающее ошибки

В Yii 2 есть одно встроенное место, где вы можете использовать композицию вместо наследования для контроля того, как исключения будут отрисованы вашим приложением. Это место упомянуто в виде предпоследней строчки в таблице из предыдущего раздела.

Мы можем присвоить настройке `components.errorHandler.errorAction` какой-нибудь маршрут в нашем приложении, который будет использован для отрисовки ошибок. Эта настройка, очевидно, соответствует свойству `\yii\web\ErrorHandler::$errorAction`.

Это будет работать, только если все предыдущие варианты были неприменимы!

Хотя мы можем указать любой маршрут в этом свойстве, в Yii 2 уже содержится заранее подготовленное действие контроллера под

названием \yii\web\ErrorAction, которое может делать рендеринг за нас. Нужно внести два изменения, и в результате у вас появится преднастроенная страница об ошибке, которую приложение будет показывать посетителям.

Во-первых, вы говорите обработчику ошибок в конфигурации приложения, что вы хотите использовать ваш собственный маршрут для обработки ошибок, например site/error:

```
'errorHandler' => [
   'errorAction' => 'site/error',
]
```

Во-вторых, вы идёте в SiteController и объявляете действие error, используя метод actions():

```
public function actions()
{
   return [
      'error' => ['class' => 'yii\web\ErrorAction'],
   ];
}
```

На этом всё. Теперь любое исключение, которое встретится посетителям вашего приложения, будет показано, используя файл представления views/site/error.php. Вы можете указать другой путь до этого файла представления, используя настройку view класса ErrorAction.

Этот файл представления получит следующие три переменные (помимо $this, привязанной к экземпляру класса View):

- $name, которая содержит имя исключения, составленное для вас классом ErrorAction так, чтобы его можно было безопасно (и без стыда) показывать посетителям;
- $message, которая содержит сообщение этого исключения, опять же подготовленное классом ErrorAction специально для показа посетителям;
- $exception, которая содержит исходный объект исключения для того, чтобы вы могли покопаться в нём самостоятельно.

Обратите внимание на то, что класс ErrorAction очень старается быть обобщённым и обрабатывает AJAX-запросы, обращённые к нему, возвращая строку "$name: $message", дословно игнорируя файл представления.

И да, снова никаких примеров из реальной жизни.

Список встроенных исключений

В *главе 6* мы перечислили все встроенные в Yii 2 исключения, основанные на классе `HttpException`. Но наследники `HttpException` – не все исключения, доступные для вас в Yii 2. Вот список всех остальных исключений, которые вы можете использовать. В колонке «Назначение» мы объясним, как и где Yii 2 сам использует соответствующий класс исключения. В кавычках содержится перевод прямых цитат из блоков самодокументации соответствующих классов. От вас, по сути, ожидается такое же поведение, иначе какой смысл в таких конкретных названиях классов?

Класс исключения	Назначение
`\yii\base\ErrorException`	Ошибки и фатальные ошибки PHP превращаются в исключения этого класса. Вам лучше не бросать их вручную, так как они содержат внутри себя много логики
`\yii\base\Exception`	«Общее исключение для любых целей»
`\yii\base\ExitException`	Изображает выход из приложения, как, например, нажатие *Ctrl+C* в POSIX-совместимом терминале. Первый аргумент конструктора – код возврата, полезен для консольных команд
`\yii\base\InvalidCallException`	«Представляет исключение из-за вызова метода неправильным образом»
`\yii\base\InvalidConfigException`	«Представляет исключение, вызванное неправильной конфигурацией объекта». Вы увидите *много* таких исключений, когда будете экспериментировать с настройками приложения или виджетов
`\yii\base\InvalidParamException`	«Представляет исключение, вызванное неверными параметрами, переданными в метод»
`\yii\base\InvalidRouteException`	«Представляет исключение, вызванное неверным маршрутом». Если это исключение обрабатывается экземпляром `\yii\web\Application`, он пробросит это исключение дальше в виде исключения класса `NotFoundHttpException`
`\yii\base\NotSupportedException`	«Представляет исключение, вызванное доступом к возможностям, которые не поддерживаются»
`\yii\base\UnknownClassException`	«Представляет исключение, вызванное использованием неизвестного класса»
`\yii\base\UnknownMethodException`	«Представляет исключение, вызванное доступом к неизвестному методу объекта»

Класс исключения	Назначение
\yii\base\UnknownPropertyException	«Представляет исключение, вызванное доступом к неизвестному свойству объекта»
\yii\base\UserException	Это исключение, о котором мы говорили ранее. Ожидается, что вы будете бросать его тогда, когда хотите указать посетителю на его собственную (фатальную) ошибку
\yii\console\Exception	Это исключение по смыслу полностью идентично классу \yii\base\UserException, но предназначено для использования в консольном приложении
\yii\db\Exception	Представляет проблемы, о которых рапортует механизм взаимодействия с базой данных. Когда Yii бросает такое исключение, он заполняет дополнительное свойство errorInfo результатом вызова PDO::errorInfo()
\yii\db\StaleObjectException	Объект, на который мы ссылаемся в базе данных (операцией UPDATE или DELETE), более не существует. Это возможно при параллельном доступе к БД

Кэширование

До тех пор, пока программы на языке PHP остаются сценариями, которые запускаются, выдают результаты и умирают, идея **кэширования** результатов вычислений будет оставаться актуальной. Математическое понятие мемоизации, реализованное на уровне среды выполнения языка, повсеместно используется в экосистеме PHP, и фреймворк Yii поддерживает широкий спектр техник кэширования для ускорения вашей программы.

В Yii 2 есть следующие четыре уровня кэширования:

1) кэширование запросов к базе данных;
2) кэширование фрагментов HTML-страниц;
3) кэширование запроса к серверу целиком;
4) кэширование HTML-отклика при помощи некоторых заголовков HTTP.

Перед тем как мы посмотрим на все эти возможности по очереди, нам нужно узнать, что в Yii 2 есть специальный компонент кэширования, который лежит в основе этого механизма.

ВОЗМОЖНОСТЬ: компонент кэша

Три из четырёх перечисленных в предыдущем разделе уровней кэширования используют централизованный компонент кэширования,

который настраивается через параметр конфигурации приложения components.cache. Этот компонент представляет некое абстрактное хранилище пар «ключ–значение», способное сохранять данные, помеченные некоторым ключом, и затем возвращать эти данные, если был передан их ключ.

Конечно же, хотя кэширование в Yii 2 использует этот компонент неявно, вы как разработчик можете также использовать его и вручную, обращаясь к объекту Yii::$app->cache. Если не трогать многие другие детали API-кэширования, вы можете надёжно полагаться на следующие вызовы:

Метод	Смысл
add($key, $value, $duration = 0, $dependency = null)	Добавить значение $value, помеченное ключом $key, в кэш на количество секунд, равное $duration. Если что-то уже сохранено по ключу $key, ничего не делать. Если зависимость $dependency изменится, значение $value будет считаться неверным вне зависимости от значения $duration
set($key, $value, $duration = 0, $dependency = null)	Установить значение $value по предоставленному ключу $key и установить новое количество секунд хранения $duration. Если по этому ключу $key ничего не сохранено, данный метод идентичен методу add(). Если зависимость $dependency изменится, значение $value будет считаться неверным вне зависимости от значения $duration
get($key)	Изъять объект, сохранённый по ключу $key
delete($key)	Удалить объект, сохранённый по ключу $key
exists($key)	Проверить, сохранено ли что-нибудь по ключу $key
flush()	Полностью очистить кэш, удалив все сохранённые значения

При сохранении значение $value будет сериализовано либо встроенным в PHP методом serialize(), либо анонимной функцией, указанной в настройке components.cache.serializer. То же самое относится к процедуре извлечения $value, которая будет использовать встроенную в PHP функцию unserialize() в случае отсутствия преднастроенного сериализатора. Сверьтесь с документацией по этой настройке, чтобы узнать подробности.

Когда вы регистрируете какой-либо объект в кэше, вы можете указать два способа, которым он может быть аннулирован (**invalidated**), то есть стать недоступным по тому ключу, под которым был сохранён.

Первый способ – при помощи параметра `$duration`, по сути своей ограничивая время «жизни» для этого кэшированного объекта.

Второй способ намного более интересный. Вы можете также установить произвольную «зависимость» (**dependency**) для кэшируемого объекта. Аргумент `$dependency` у методов `add()` и `set()` – это объект некоторого подкласса класса `\yii\caching\Dependency`. Чтобы лучше понять, о чём всё это, вот список всех встроенных видов зависимостей кэширования:

- `\yii\caching\DbDependency` – кэшированный объект будет зависеть от результата некоторого запроса SQL к некоторой БД (возможно, отличающейся от основной);
- `\yii\caching\ExpressionDependency` – кэшированный объект будет зависеть от результата некоторого произвольного выражения PHP, которое будет вычислено методом `eval()`;
- `\yii\caching\FileDependency` – кэшированный объект будет зависеть от времени последнего изменения некоторого файла в файловой системе;
- `\yii\caching\TagDependency` – этой зависимостью кэшированный объект будет помечен некоторым «тегом». Позже в некоторый момент времени приложение может аннулировать все кэшированные объекты, имеющие один и тот же «тег», вызовом `TagDependency::invalidate($cache, $group)`, передав в него компонент кэша и желаемый «тег»;
- `\yii\caching\ChainedDependency` – специальный вид зависимости, который группирует другие зависимости вместе, позволяя создавать сложные условия аннулирования кэшированных элементов.

Главная проблема с кэшированием, конечно же, заключается в том, что иногда реальное содержимое обновляется быстрее, чем закэшированные объекты становятся аннулированными. Это выливается в то, что посетителям показываются устаревшие сведения. «Зависимости» являются решением этой проблемы, потому что с их помощью вы фактически можете заставить содержимое быть заново закэшированным на основании некоторого условия, такого как изменённый файл, результат запроса к БД или какое-либо произвольное выражение PHP.

Мы не будем углубляться в детали тонкой настройки вашего приложения этими сложными методами. Для этого лучше посмотреть официальную документацию.

Компонент кэша в Yii 2 – это класс, расширяющий класс \yii\caching\Cache, и фреймворк предоставляет «из коробки» восемь решений по кэшированию:

- \yii\caching\ApcCache – использует расширение PHP под названием **APC** (см. http:// php.net/manual/book.apc.php);
- \yii\caching\DbCache – использует таблицу в базе данных (вы можете даже указать соединение с базой данных, отличающееся от вашего основного соединения);
- \yii\caching\DummyCache – не предоставляет никакого кэширования, но реализует обязательный API, так что вы можете использовать этот компонент для удовлетворения требований некоторой другой подсистемы;
- \yii\caching\FileCache – сохраняет объекты в файлах в некотором каталоге, таким образом полагаясь на возможности файловой системы. Обратите внимание на то, что в зависимости от используемых типов хранения данных и файловой системы вы можете получить *уменьшение* производительности с этим видом кэширования;
- \yii\caching\MemCache – использует решение **memcache** (см. http:// php.net/manual/intro.memcache.php). Вы можете переключиться на memcached, используя одну настройку, имеющую булево значение (см. документацию на этот класс);
- \yii\caching\WinCache – использует **Windows Cache** посредством расширения PHP для WinCache (см. http://www.iis.net/expand/wincacheforphp);\yii\caching\XCache – использует **XCache** (см. http://xcache.lighttpd.net/);
- \yii\caching\ZendDataCache – использует расширение PHP **Zend Data Cache** (см. http://www.zend.com/en/products/server/).

Как обычно, вам настоятельно рекомендуется прочитать соответствующую документацию, чтобы узнать подробности корректной настройки каждого из перечисленных компонентов.

В дополнение ко встроенным видам кэширования есть ещё расширение Yii 2 для сервиса Redis, которое можно установить следующей командой:

```
$ php composer.phar require --prefer-dist yiisoft/yii2-redis "*"
```

После установки у вас появится класс \yii\redis\Cache, который предоставляет кэширование, используя серверы **Redis** (см. подробности на http://redis.io/).

Вот пример конфигурации, необходимой для включения компонента кэширования с Redis в качестве провайдера кэширования:

```
'components' => [
   'cache' => [
      'class' => 'yii\redis\Cache',
      'redis' => [
         'hostname' => 'localhost',
         'port' => 6379,
         'database' => 0,
      ]
   ],
],
```

ВОЗМОЖНОСТЬ: кэширование запросов к базе данных

В компоненте соединения с базой данных в Yii 2 есть несколько свойств, которые контролируют кэширование запросов к БД. Вот они:

Настройка	Смысл
`enableSchemaCache`	Включать ли вообще кэширование схемы данных
`schemaCacheDuration`	Продолжительность в секундах, на которую нужно кэшировать схему данных. По умолчанию значение этого свойства равно 3600 (1 час)
`schemaCacheExclude`	Схему каких таблиц *не* кэшировать, в виде массива строк
`schemaCache`	Какой компонент кэша использовать для кэширования. По умолчанию используется системный компонент, который устанавливается по ключу `components.cache` в настройках приложения
`enableQueryCache`	Включать ли вообще кэширование результатов запросов
`queryCacheDuration`	Продолжительность в секундах, на которую кэшировать результаты запросов. По умолчанию значение этого свойства равно 3600 (1 час)
`queryCacheDependency`	Зависимость, которая дополнительно будет контролировать аннулирование кэшированных данных
`queryCache`	Какой компонент кэша использовать для кэширования. По умолчанию используется системный компонент, который устанавливается по ключу `components.cache` в настройках приложения

Обычно всё, что вам нужно, чтобы *чрезвычайно* ускорить взаимодействие с базой данных, при условии что мы рассматриваем «боевой» сервер и у вас тысячи записей в таблицах, – это следующие настройки компонента подключения к базе данных:

```
'components' => [
   'db' => [
      // … все обычные вещи вроде 'class', 'dsn', 'username' и 'password'…
      'enableSchemaCache' => true,
      'enableQueryCache' => true,
   ]
]
```

При условии что вы настроили компонент кэширования, вы на этом сделали всё, что нужно. Результаты запросов и схема данных по умолчанию будут кэшироваться на 1 час.

ВОЗМОЖНОСТЬ: кэширование фрагментов страницы

На уровне выше, чем кэширование результатов запроса к БД, вы можете закэшировать части файлов представлений. Вы можете это сделать в файле представления следующим образом:

```
$this->beginCache($уникальный_ключ);
```

... это содержимое будет взято из кэша, если оно там есть, и перевычислено и положено в кэш в противном случае...

```
$this->endCache();
```

Виджет под названием `\yii\widgets\FragmentCache` инкапсулирует реализацию этой возможности. Показанные выше методы, `\yii\base\View::beginCache()` и `\yii\base\View::endCache()`, являются вспомогательными методами для облегчения использования этого виджета.

Метод `beginCache()` принимает два аргумента: `$id` и `$properties`. `$id` должен быть некоторым уникальным идентификатором, а `$properties` является свойством виджета `FragmentCache`. Посмотрите документацию на этот виджет, чтобы узнать подробности.

Этот уровень кэширования идеально подходит для таких вещей, как динамические рекламные баннеры, чей контент должен быть отрисован на основе данных из БД.

ВОЗМОЖНОСТЬ: кэширование страницы целиком

Затем, на следующем уровне кэширования, вы можете положить в кэш всю страницу, отрисованную в ответ на запрос к серверу. Следует отметить, что предыдущие два уровня кэширования всё ещё будут продолжать ускорять работу, уменьшая время, необходимое на отрисовку всего файла представления, начиная с шаблона.

Для этой задачи вы используете фильтр действий \yii\web\PageCache, и если вы помните, мы упоминали его ранее, в *главе 6*. Когда вы хотите, чтобы контроллер кэшировал содержимое отклика целиком, вы добавляете следующие установки в метод behaviors() этого контроллера:

```
public function behaviors()
{
    return [
        'pageCache' => [
            'class' => \yii\web\PageCache::className(),
            'only' => ['список', 'идентификаторов', 'кэшируемых', 'действий'],
            'duration' => 60, // секунд
            'dependency' => [конфигурация для зависимости],
        ]
    ]
}
```

Абсолютно необходимые свойства – имя поведения (может быть любым) и название класса фильтра PageCache. Если вы опустите настройку only, все действия этого контроллера будут кэшироваться, что, скорее всего, не то, что вам на самом деле нужно.

Пожалуйста, обратите внимание на то, что если вы не глядя сунете PageCache в пример CRM-приложения, с которым мы экспериментируем в этой книге, вы, скорее всего, сделаете так, что *приёмочные тесты перестанут успешно завершаться*. Например, если вы просто зарегистрируете PageCache на все действия в контроллере ServicesController, система из-за кэширования не сможет даже сохранить новый экземпляр ServiceRecord. Даже если вы ограничите PageCache до одного только действия actionIndex(), проблемы с приёмочными тестами всё равно сохранятся, потому что вам фактически нужно аннулировать кэш после каждого добавления, изменения или удаления услуги. Это требует довольно изощрённой зависимости кэширования, объявлённой в настройке dependency экземпляра PageCache. Вы можете сколько угодно сами экспериментировать с реализацией кэширования страницы целиком в ServicesController, но в целом это просто не очень хорошая идея: реализовывать такой вид кэширования в крайне нестабильных частях веб-приложения вроде CRUD-интерфейса.

Кэширование на уровне страницы фильтром PageCache – на самом деле это то же самое, что вызвать метод beginCache() до того, как начать отрисовку файла представления, а потом после отрисовки вызвать метод endCache(). Более того, фильтр PageCache совершает в точности такие действия, как можно увидеть из исходного кода этого класса. Использование этого фильтра избавит вас от повторения одного и того же кода во всех местах, требующих кэширования.

ВОЗМОЖНОСТЬ: кэширование запроса заголовками HTTP

Это наименее надёжная, последняя линия кэширования, которая совершается (почти) полностью на стороне браузера. Используя заголовки **Last-modified** и **Etag** протокола HTTP, сервер может сказать браузеру, была ли страница модифицирована с момента последнего к ней запроса и будет ли иметь смысл скачивать её заново. Корректно используя эти заголовки, мы можем сэкономить как время на скачивание HTML и всех связанных с ним материалов, так и время на отрисовку этого HTML на сервере.

> За официальным объяснением того, как работают эти заголовки, обращайтесь в RFC2616, раздел 13.3, по адресу http://tools.ietf.org/html/rfc2616#section-13.3.

Этот вид кэширования осуществляется фильтром действий `\yii\web\HttpCache`. Его настройка немного более сложная, чем всех предыдущих методов, так как мы должны предоставить две анонимные функции, одна из которых будет генерировать значение для заголовка `Last-modified`, а вторая – значение для заголовка `Etag`.

Скажем, мы добавим поле `last_updated` («последний раз модифицировано...») в таблицу клиентов `customers` в базе данных, которое будет автообновляться по триггеру, когда любое другое поле в соответствующей записи изменит своё значение. Мы можем добавить дополнительный уровень кэширования к действию `CustomersController.actionIndex()`, если установим в качестве значения заголовка `Last-Modified` метку времени той записи о клиенте, у которой эта метка времени – самая поздняя. В итоге, если с момента последнего обращения к списку клиентов более новых клиентов не появилось, мы даём понять браузеру, что страница никак не должна была поменяться. Для того чтобы корректно различать версии конечной страницы по заголовку `Etag`, мы будем использовать ту же функцию.

Вот код, который реализует такой план:

```
public function behaviors()
{
   return [
      'httpCache' => [
         'class' => \yii\web\HttpCache::className(),
         'only' => ['index'],
         'lastModified' => [$this, 'getMaxCustomerTimestamp'],
```

```
            'etagSeed' => [$this, 'getMaxCustomerTimestamp'],
        ],
    ];
}
public function getMaxCustomerTimestamp($action, $params)
{
    return strtotime((new Query())->from('customers')->max('last_updated'));
}
```

Минимизация материалов

Конечно же, в сложных случаях у вас будет множество файлов CSS и Javascript, используемых на страницах приложения, и это серьёзная проблема. Современный передовой опыт заключается в объединении и сжатии всех файлов CSS в один, и то же самое относится к файлам Javascript. В результате мы будем передавать всего два файла вместо целой кучи. Вы, возможно, уже всё это знаете. Было бы просто замечательно, если бы система материалов в Yii делала это за вас неявно, но такой функциональности нет и не будет по соображениям производительности, так как минимизация материалов занимает время, и её определённо лучше не выполнять во время запроса посетителя. Однако в Yii присутствует вспомогательная консольная команда, которую вы можете использовать для упрощения процесса сжатия ваших материалов и привязывания их к приложению. Самая красивая часть в том, что вам не придётся менять ващи существующие файлы представлений; (почти) все существующие вызовы `*Asset::register()` могут продолжать оставаться на своих местах.

Хотя документация полностью описывает процесс внедрения объединённых материалов в приложение на Yii 2, мы повторим её здесь, чтобы рассмотреть намного больше подробностей в этом по-настоящему сложном процессе.

Общая идея, стоящая за компиляцией материалов в Yii, заключается в том, что вы подготавливаете специальный пакет материалов, состоящий из всего двух файлов: один, содержащий все скомпилированные файлы CSS, и другой, содержащий все скомпилированные файлы Javascript. Затем вы переопределяете настройки пакетов материалов, используя настройку `components.assetManager.bundles` (прочитайте о ней в документации), чтобы сказать Yii, что нужно использовать этот свежесозданный пакет материалов вместо всех обычных пакетов.

Класс `\yii\console\controllers\AssetController`, который предоставляет нам команду `./yii asset`, автоматизирует эту процедуру. Он даже

решает за вас проблему файлов, на которые ссылаются комбинируемые файлы CSS, и это на самом деле поразительно.

Давайте посмотрим, что нам нужно, чтобы этого добиться. Во-первых, мы запомним структуру HTML, которую мы получаем, переходя по некоторому маршруту в нашем приложении, например `/site/login`. Следующее изображение – это снимок экрана из Firebug, на котором видно количество частей, из которых состоит наш интерфейс:

Внутри элемента `head` можно увидеть как таблицы стилей Bootstrap, так и наши собственные. В нижней части элемента `body` находится целая пачка элементов `script`. Наша цель – заменить все эти ссылки на одну ссылку на CSS и одну ссылку на Javascript.

Нам понадобятся исполняемые файлы, которые будут заниматься сжатием материалов. Yii 2 использует утилиту **YUI Compressor** для сжатия CSS и утилиту **Google Closure Compiler** для сжатия Javascript, доступные, соответственно, по адресам http://yui.github.io/yuicompressor/ и https://developers.google.com/closure/compiler/.

Давайте создадим каталог под названием `compression` внутри каталога `assets` и будем складывать всё, относящееся к нашей задаче, в него. Вам нужно скачать с вышеприведённых ссылок в эту папку два файла JAR. Файл YUI Compressor должен быть назван `yuicompressor.jar`, файл Google Closure Compiler – `compiler.jar`.

После этого нам нужно создать специальный файл настроек для компрессора. В Yii есть консольная команда, которая поможет вам создать шаблон такого файла. Выполните следующую команду в корневом каталоге проекта:

```
$ ./yii asset/template assets/compression/config.php
```

Если вы откроете созданный шаблон, то должны увидеть следующий код:

```
<?php
return [
   // Adjust command/callback for JavaScript files compressing:
   'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_
file {to}',
   // Adjust command/callback for CSS files compressing:
   'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o
{to}',
   // The list of asset bundles to compress:
   'bundles' => [ // 1
      // 'yii\web\YiiAsset',
      // 'yii\web\JqueryAsset',
   ],
   // Asset bundle for compression output:
   'targets' => [ // 2
      'app\assets\AllAsset' => [
         'basePath' => 'path/to/web',
         'baseUrl' => '',
         'js' => 'js/all-{hash}.js',
         'css' => 'css/all-{hash}.css',
      ],
   ],
   // Asset manager configuration:
   'assetManager' => [ // 3
      'basePath' => __DIR__,
      'baseUrl' => '',
   ],
];
```

Этот фрагмент довольно длинный для того, чтобы вставлять его в книгу «как есть», но в нём находится множество важных и неочевидных частей.

Начало кода достаточно простое: вы должны определить список пакетов материалов, которые хотите объединить и сжать. Заметьте, что они упоминаются по полностью определённым именам классов.

Пожалуйста, указывайте полностью определённые имена классов пакетов материалов *без* косой черты в качестве первого символа, иначе вы сломаете механику сжатия.

По определению, содержимое пакетов материалов, перечисленных в настройке `bundles`, будет зарегистрировано на каждой странице вашего веб-приложения. Если вы хотите регистрировать некоторые пакеты на основе каких-либо условий, как наш пакет `SnowAssetsBundle` из *главы 4*, вам лучше оставить их отдельно, так как они могут содержать переопределения стандартных стилей.

Было бы неплохо соединить все файлы CSS и JavaScript из нашего основного пакета материалов и всех его зависимостей. Для этого нам нужны только следующие значения для настройки `bundles`:

```
'bundles' => [
    'app\assets\ApplicationUiAssetBundle',
]
```

Не беспокойтесь о файлах, которые упоминаются в файлах CSS, таких как изображения и шрифты. В любом случае все пакеты материалов будут опубликованы сервером как обычно, поэтому все дополнительные файлы всё так же будут находиться в папках, доступных из Сети. Команда `./yii asset` делает это для того, чтобы иметь возможность переписать все аннотации вида `url('...')` в сжатом файле CSS так, чтобы они указывали на свои настоящие URL. В итоге всё будет выглядеть так, как если бы соответствующие пакеты материалов были зарегистрированы как обычно, безо всякого сжатия.

Вторая часть кода намного хитрее. В настройке `targets` вы должны перечислить пакеты материалов, которые получат сжатые файлы. Это на самом деле обобщённая настройка, но для нашего случая, когда нам нужен единственный, всеобъемлющий пакет материалов, содержащий всего два сжатых файла, нам полностью достаточно кода, предоставленного нам шаблоном. Этот код гласит, что там будет пакет материалов `app\assets\AllAsset`, который будет содержать один файл Javascript и один файл CSS. Нам нужно только исправить пути в этом определении.

Однако настройка `targets` нам лжёт. Никакого определения класса `app\assets\AllAsset` нигде не существует, и Yii его за нас не создаст. Вы должны самостоятельно положить тривиальное определение этого класса в соответствующий файл `assets/AllAsset.php`. Вот это определение:

```
namespace app\assets;
use yii\web\AssetBundle;
class AllAsset extends AssetBundle { }
```

Что касается путей, для них нам нужен план. Он, впрочем, будет прост: два сжатых файла, в которых мы заинтересованы, будут размещены в подкаталоге `compiled-assets` прямо в `@webroot`, чтобы до них можно было добраться без отдельного опубликования менеджером материалов.

Итак, `AllAsset` должен иметь значение настройки `basePath`, равное `@webroot`, и значение настройки `baseUrl`, равное `/`. Проблема в том, что когда файл `assets/compressed/config.php` будет обрабатываться, псевдонима пути `@webroot` ещё не будет определено, так что нам остаётся только использовать абсолютные пути:

```
'targets' => [
   'app\\assets\\AllAsset' => [
      'basePath' => realpath(__DIR__ . '/../../web'),
      'baseUrl' => '/',
      'js' => 'compiled-assets/all-{hash}.js',
      'css' => 'compiled-assets/all-{hash}.css',
   ],
],
```

Этот забавный токен `{hash}` будет заменён на уникальный хэш для соответствующего файла.

Третья часть кода показывает настройки менеджера материалов, в которых нам придётся повторить настройки менеджера материалов, присоединённого к главному приложению. Нам нужно указать здесь рабочие пути, потому что, как было сказано ранее, на момент выполнения команды `./yii asset` псевдонима пути `@webroot` не будет.

Логика настроек `assetManager.basePath` и `assetManager.baseUrl` в том, чтобы менеджер материалов знал, куда класть опубликованные материалы. Мы уже решили на этапе составления приложения, что материалы будут опубликованы в подкаталог `web/assets` и поэтому будут доступны по маршрутам `/assets/*` (так как `@webroot` – это корневой каталог сайта, публикуемый веб-сервером). Учитывая всё это, нам следует написать следующее в настройку `assetManager` в конфигурации для команды `./yii asset`:

```
'assetManager' => [
   'basePath' => realpath(__DIR__ . '/../../web/assets'),
   'baseUrl'  => '/assets',
],
```

Последняя часть – это настройки путей к компрессорам в самом начале шаблона конфигурации. По умолчанию команда `./yii asset` ожидает, что JAR-файлы будут лежать в корневом каталоге проекта. Но в нашем случае они находятся в подкаталоге `assets/compression`, чтобы вещи где попало не валялись. Контроллер `yii\console\controllers\AssetController`, чьи свойства мы устанавливаем в этом дополнительном файле конфигурации, содержит для нас две специальные настройки, довольно, впрочем, неудобные для переопределения:

```
    public $jsCompressor = 'java -jar compiler.jar --js {from} --js_
output_file {to}';
    public $cssCompressor = 'java -jar yuicompressor.jar --type css
{from} -o {to}';
```

Как видите, это полные консольные команды, требуемые для запуска компрессоров Javascript и CSS, и они на самом деле предполагают, что JAR files будут в корневом каталоге проекта (или, по крайней мере, в каталоге, откуда мы вызовем команду `yii asset`). Так как мы всегда будем использовать команду сжатия материалов из корневого каталога проекта, мы переопределим эти настройки в файле `assets/compression/config.php` следующим образом:

```
    'cssCompressor' => 'java -jar assets/compression/yuicompressor.jar
--type css {from} -o {to}',
    'jsCompressor' => 'java -jar assets/compression/compiler.jar --js
{from} --js_output_file {to}',
```

С этим у вас будет законченная, правильная конфигурация для компрессора. Однако до того, как его запускать, вам нужно ещё подготовить подкаталог `web/compiled-assets/`, который вы обещали контроллеру `AssetController` в настройках `targets.app\\assets\\allAsset.js` и `targets.app\\assets\\allAsset.css`. И не забудьте положить этот каталог в систему контроля версий, которую вы используете, он вам теперь всегда будет нужен.

Теперь, наконец, выполните следующую команду:

```
$ ./yii asset assets/compression/config.php config/assets_compressed.php
```

Первый аргумент команды `./yii asset` – это путь до файла конфигурации, над которым мы так тяжело трудились до этого момента. Второй аргумент – это путь до файла, который будет хранить итоговый фрагмент конфигурации, который нам нужно будет присоединить к приложению. Мы решили положить его в файл `assets_compressed.php` рядом с другими нашими основными файлами настроек.

Сгенерированный фрагмент настроек будет содержать определения всех пакетов материалов, перечисленных в настройке `bundles`, с пустыми значениями настроек `js` и `css`. Все эти пакеты материалов будут объявлены зависимыми от нашего новосозданного пакета материалов, содержащего сжатые CSS- и Javascript-файлы. Нам нужно использовать содержимое этого фрагмента конфигурации в качестве значения настройки `components.assetManager.bundles`, так что в нашем случае просто вставьте следующее в раздел `components`:

```
'assetManager' => [
    'bundles' => (require __DIR__ . '/assets_compressed.php')
],
```

Наконец, всё это будет бессмысленно, если мы не изменим вставку пакетов материалов в нашем шаблоне. Сейчас она выглядит так:

```
app\assets\ApplicationUiAssetBundle::register($this);
```

Эта строчка нам теперь фактически не нужна, потому что наш всеобъемлющий пакет материалов включает в себя содержимое пакета материалов интерфейса приложения. Её можно спокойно удалить. Однако нам нужно зарегистрировать пакет материалов класса `AllAsset`, который мы столько времени создавали:

```
app\assets\AllAsset::register($this);
```

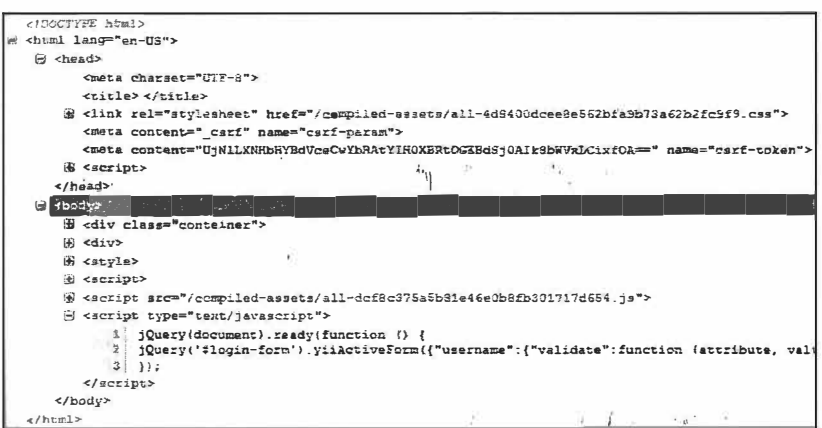
Теперь вы можете открыть тот же самый маршрут, который мы использовали для предыдущего снимка экрана (который, кстати говоря, был маршрутом `/site/login`), и увидеть следующее:

Как вы можете видеть, мы избавились от запросов к материалам `main.css`, `main.js`, `bootstrap.css`, `yii.js` и `jquery.js`. Однако всё ещё остаются сценарии для валидации и виджета `ActiveForm`. Их мы тоже можем сжать (всё равно мы используем эти возможности по всему приложению), включив их в настройку `bundles`:

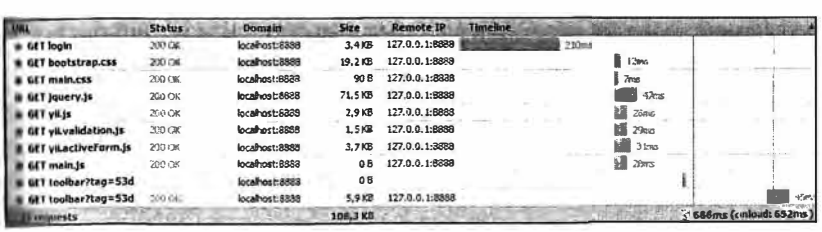
```
'bundles' => [
    'app\assets\ApplicationUiAssetBundle',
    'yii\widgets\ActiveFormAsset',
    'yii\grid\GridViewAsset',
    'yii\validators\ValidationAsset',
],
```

`GridViewAsset` был добавлен просто так. У нас всё равно уже есть два виджета `GridView`, и в последующих главах добавятся ещё.

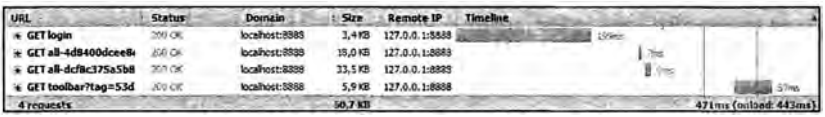
После того как вы заново пережмёте материалы, используя только что приведённые настройки, вы больше не должны видеть материалы для валидатора полей ввода и виджета `ActiveForm`:



Итак, хронометраж. Вот результаты без сжатия:



А вот результаты после сжатия:



Мы убрали 6 запросов из 10, и наше время запроса на 31% меньше. Конечно же, любая оценка производительности по части времени является ложью, но для нас наиболее важно уменьшение количества запрашиваемых материалов. Эти снимки экрана получены после очистки кэша и пересборки материалов в Yii, так что в показанных замерах времени никакого кэширования на стороне браузера не должно было участвовать.

Итоги

Мы взглянули на два аспекта общего поведения приложений, основанных на Yii 2, а именно на интроспекцию, предоставляемую механизмами журналирования и отчётов об ошибках, и скорость отклика, предоставляемую механизмами четырёхуровневого кэширования и сжатия материалов.

На самом деле журналирование тоже влияет на производительность. Включение записи запросов к базе данных обычно серьёзно ухудшает скорость обработки данных, так что, скорее всего, вы никогда не будете использовать её на «боевом» сервере.

Ничего из того, что мы здесь обсуждали, не добавляет каких-либо возможностей, видимых для конечного пользователя, за исключением, возможно, преднастроенной страницы об ошибке. И даже она имеет сомнительную ценность, так как страница об ошибке для конечных пользователей, которую Yii 2 предоставляет по умолчанию, соответствует практически всем лучшим рекомендациям для страниц об ошибках.

Однако в серьёзном приложении любого разумного размера вам, безусловно, понадобятся свои собственные страницы об ошибках.

В следующей короткой главе мы посмотрим на то, как мы можем создать своё собственное расширение Yii 2. Мы уже использовали два расширения и одно вкратце упомянули. Давайте обсудим концепции, которые фреймворк подготовил, чтобы мы могли сами паковать и распространять свои расширения к нему.

Глава 9

Создание расширения

Мы уже использовали довольно много встроенных в Yii 2 расширений, распространяемых в виде библиотек, которые можно установить посредством Composer отдельно от основного фреймворка. В этой главе мы рассмотрим, как сделать своё собственное расширение, которое можно будет установить таким же простым способом.

Для этого нам нужно будет следовать определённой последовательности действий, хотя понадобятся некоторые отдельные приготовления, для того чтобы связать наши новые классы с приложением. Вся эта глава будет описанием данного процесса.

Идея расширения

Итак, как же мы будем расширять Yii 2 в качестве примера для этой главы? Давайте на этот раз поступим злодейски и создадим *вредоносное* расширение, которое будет предоставлять что-то, что можно назвать бэкдором для фишинга **(phishing backdoor)**.

> Ни за что на свете на самом деле не делайте того, что мы будем описывать в этой главе! Это всё равно не даст вам мгновенного доступа к атакованному веб-сайту. Однако для опытного взломщика это вредоносное расширение даёт достаточно информации, чтобы крайне облегчить получение полного контроля над приложением, поэтому, воспользовавшись подобным кодом в реальной жизни, вы теоретически можете даже нарушить какой-нибудь закон!

Идея в следующем: наше расширение будет предоставлять специальный маршрут (контроллер с единственным действием внутри), который будет сбрасывать полный список настроек приложения на страницу в браузере. Скажем, этим маршрутом будет маршрут `/app-info/configuration`.

Однако мы не можем получить содержимое файла конфигурации никаким надёжным образом. На тот момент, когда мы будем присоединять себя к экземпляру приложения, исходный массив настроек уже будет недоступен, и даже если бы он был доступен, мы всё равно не можем быть уверены, откуда он был получен. Поэтому мы будем исследовать состояние приложения во время его выполнения и возвращать наиболее важные части информации, которую мы можем получить на этапе разрешения действия контроллера. Вот в точности та «боевая часть», которую мы хотим внедрить:

```
public function actionConfiguration()
{
 $app = \Yii::$app;
 $config = [
   'components' => $app->components,
   'basePath' => $app->basePath,
   'params' => $app->params,
   'aliases' => \Yii::$aliases
 ];
 return \yii\helpers\Json::encode($config);
}
```

Этот фрагмент кода – ядро расширения, оно подразумевается во всех последующих разделах этой главы.

На самом деле, если вы знаете значение настройки `basePath` приложения, список псевдонимов путей, настройки компонентов (среди которых может быть соединение с базой данных) и все дополнительные параметры, которые разработчики установили вручную, вы можете достаточно надёжно составить карту атакуемого приложения. Так как в данном случае вы получаете все учётные данные для входа в различные подсистемы приложения, у вас итоге появляется огромное количество крайне ценной информации. Всё, что нам остаётся, – чтобы пользователь установил это расширение.

Создание содержимого для расширения

Наш план выглядит следующим образом:

Мы будем разрабатывать наше расширение в папке, отдельной от нашего примера CRM-приложения.

Это расширение будет называться `yii2-malicious`, чтобы сохранить единообразие с именами других расширений Yii 2.

Учитывая, какая у нашего расширения полезная нагрузка, приведённая выше, оно будет состоять из одного контроллера и неболь-

шого количества кода (о котором мы ещё не узнали), для того чтобы автоматически присоединить этот контроллер к приложению.

Наконец, чтобы считать этот дополнительный проект «настоящим» расширением Yii 2, а не просто какой-то произвольной библиотекой исходного кода, мы хотим сделать его устанавливаемым тем же образом, что и остальные расширения Yii 2.

Подготовка шаблонного кода для расширения

Давайте сделаем отдельный каталог, инициализируем там репозиторий git и добавим туда файл `AppInfoController.php`. В командной строке bash это можно сделать следующими командами:

```
$ mkdir yii2-malicious && cd $_
$ git init
$ > AppInfoController.php
```

Внутри файла `AppInfoController.php` мы напишем обычный код-заглушку для контроллера Yii 2:

```
namespace malicious;
use yii\web\Controller;
class AppInfoController extends Controller
{
    // Здесь будет действие
}
```

Вставьте фрагмент кода с определением действия, приведённый в начале этой главы, в этот контроллер, и мы с ним закончили. Обратите внимание на пространство имён: оно называется не так, как каталог, в котором находится этот контроллер, и не соответствует нашим обычным правилам автозагрузки. Позднее в этой главе мы рассмотрим, что это не проблема, благодаря тому как Yii 2 осуществляет автозагрузку классов из расширений.

Теперь этот контроллер нужно как-то подсоединить к приложению. Мы уже знаем, что у экземпляра приложения есть специальное свойство под названием `controllerMap`, используя которое, мы можем вручную присоединять классы контроллеров к приложению. Однако как мы можем это сделать автоматически, а ещё лучше – прямо в момент запуска приложения? В Yii 2 присутствует специальная возможность под названием **бутстреппинг** (**bootstrapping**, в теории компиляторов для этого понятия есть более красивый, но нам не подходящий перевод «самораскрутка»), предназначенная для поддержки именно этой функциональности: выполнить некоторые действия

в начале жизненного цикла приложения, хоть и не в самое начало, но совершенно точно до обработки запроса. Эта возможность тесно связана с концепцией расширений в Yii 2, так что сейчас самое время объяснить, как она реализована.

ВОЗМОЖНОСТЬ: бутстреппинг

Если объяснить концепцию бутстреппинга вкратце, то она заключается в том, что вы можете объявить некоторые компоненты приложения в его свойстве `\yii\base\Application::$bootstrap`. Они будут корректно инициализированы в начале работы приложения. У тех компонентов, которые реализуют интерфейс `BootstrapInterface`, будет вызван метод `bootstrap()`, так что вы заодно получите инициализацию расширения приложения. Давайте теперь углубимся в подробности.

Свойство `\yii\base\Application::$bootstrap` содержит массив обобщённых «значений», которые вами указаны фреймворку как заранее нуждающиеся в инициализации. Фактически это развитие понятия «предзагрузки» из Yii 1.x. Вы можете указать четыре типа «значений» для инициализации:

- идентификатор компонента приложения;
- идентификатор некоторого модуля;
- название класса;
- массив настроек.

Если это идентификатор компонента, этот компонент полностью инициализируется. Если это идентификатор модуля, этот модуль полностью инициализируется. Это имеет большое значение, потому что в Yii 2 реализована «ленивая» загрузка модулей и компонентов, и они обычно инициализируются только при первом использовании. То, что они зарегистрированы для бутстреппинга, означает, что их инициализация, независимо от того, насколько она долгая или ресурсозатратная, всегда будет выполнена, и всегда именно в начале работы приложения.

Если у вас есть компонент и модуль с идентичными идентификаторами и они оба зарегистрированы для бутстреппинга, то компонент будет инициализирован, а модуль инициализирован *не будет*!

Если упомянутое «значение» – это название класса или массив настроек, то экземпляр данного класса будет создан при помощи механики `\yii\BaseYii::createObject()`. Созданный экземпляр будет немедленно выброшен, если он не реализует интерфейс `\yii\base\BootstrapInterface`. Если же он реализует этот интерфейс, то дополни-

тельно будет выполнен его метод `bootstrap()`. Затем этот объект всё равно будет выброшен.

Итак, в чём эффект возможности бутстреппинга? Мы уже использовали эту возможность в главе 7, когда устанавливали расширение отладки. Нам необходимо было зарегистрировать модуль отладки для бутстреппинга по его идентификатору, для того чтобы он мог добавить свой обработчик событий, чтобы мы получили панель отладки внизу каждой страницы нашего веб-приложения. Эта возможность совершенно незаменима, если вам нужно быть уверенным в том, что некоторая деятельность будет совершена в начале жизненного цикла приложения.

Интерфейс `BootstrapInterface` фактически представляет собой воплощение паттерна **Команда**. Реализовав этот интерфейс, мы получаем возможность добавить к инициализации приложения любую деятельность, не обязательно связанную с данным компонентом или модулем.

ВОЗМОЖНОСТЬ: регистрация расширений

Возможность бутстреппинга повторена в том, как обрабатывается свойство `\yii\base\Application::$extensions`. Это свойство – единственное место в Yii 2, где вообще можно увидеть упоминание концепции «расширений». Расширения в этом свойстве описываются в виде массива массивов, и каждый из вложенных массивов должен иметь следующие поля:

- `name`: это поле содержит название расширения;
- `version`: это поле содержит версию расширения (ничто в Yii 2 не будет на самом деле проверять его, так что оно здесь просто для справки);
- `bootstrap`: это поле содержит данные для бутстреппинга этого расширения. Оно заполняется теми же элементами, что и ранее описанное свойство `Yii::$app->bootstrap`, и имеет тот же смысл;
- `alias`: это поле описывает отображение псевдонимов путей Yii 2 на реальные пути в файловой системе.

Когда приложение регистрирует расширение, оно делает две вещи в следующем порядке:

1. Регистрирует псевдонимы, упомянутые в расширении, используя метод `Yii::setAlias()`.
2. Инициализирует *вещи*, упомянутые в бутстреппинге расширения, точно таким же образом, который был описан в предыдущем разделе.

Обратите внимание, что бутстреппинг расширения происходит до бутстреппинга самого приложения!

Регистрация псевдонимов критически важна для всей концепции расширений. Всё из-за автозагрузчика Yii 2, совместимого со стандартом PSR-4, о котором мы немного поговорили в *главе 2*.

Вот цитата из блока самодокументации метода `\yii\BaseYii::autoload()` (перевод автора):

> *Если класс находится в пространстве имён (например, `yii\base\Component`), он [автозагрузчик – прим. пер.] попытается подключить файл, связанный с соответствующим псевдонимом пути (например, `@yii/base/Component.php`).*
> *Этот автозагрузчик позволяет загружать классы, которые следуют стандарту PSR-4 и имеют корневое пространство имён или пространства имён нижележащих уровней, определённые в виде псевдонимов путей.*

Стандарт PSR-4 доступен в Сети здесь: http://www.php-fig.org/psr/psr-4/.

Согласно этому поведению, настройка `alias` расширения фактически является способом сказать автозагрузчику название корневого пространства имён для классов в вашем расширении. Допустим, у вас в расширении определено такое значение настройки `alias`:

```
"alias" => [
  "@имякомпании/имярасширения" => "/некоторый/абсолютный/путь"
]
```

Если у вас есть файл `/некоторый/абсолютный/путь/подкаталог/ИмяКласса.php` и, согласно правилам PSR-4, он содержит класс, чьим полностью определённым именем является `\имякомпании\имярасширения\подкаталог\ИмяКласса`, то Yii 2 сможет автоматически загрузить этот класс безо всяких проблем.

Создание бутстреппинга для нашего расширения – тайное присоединение контроллера

В нашем расширении уже заранее подготовлен контроллер. Теперь мы хотим, чтобы этот контроллер был автоматически присоединён к атакованному приложению, когда расширение будет обрабатываться. Это можно сделать, используя только что изученную возможность бутстреппинга. Давайте создадим для этих целей класс `\malicious\`

Bootstrap внутри каталога нашего расширения и напишем в нём следующий фреймворк кода:

```
<?php

namespace malicious;

use \yii\base\BootstrapInterface;

class Bootstrap implements BootstrapInterface
{
  /** @param \yii\web\Application $app */
  public function bootstrap($app)
  {
   // Здесь будет добавление контроллера.
  }
}
```

Благодаря этой подготовке на старте приложения будет вызван метод `bootstrap()`, при условии что мы всё корректно соединим. Но сначала нам нужно разобраться с тем, как мы можем воздействовать на приложение, чтобы оно начало использовать наш контроллер. На самом деле из-за того, что существует свойство `\yii\web\Application::$controllerMap`, сделать это будет очень просто (и не забывайте, что это свойство унаследовано от класса `\yii\base\Module`, а значит, мы можем так сделать не только с самим приложением).

Вот единственная строчка, которую нужно добавить в метод `bootstrap()`:

```
$app->controllerMap['app-info'] = '\malicious\AppInfoController';
```

Мы будем полагаться на автозагрузчики Composer и Yii 2, для того чтобы приложение на самом деле нашло `\malicious\AppInfoController`.

> Просто представьте, что внутри этой предзагрузки вы можете делать всё, что угодно. Например, вы можете открыть соединение при помощи CURL с каким-нибудь ботнетом и сразу отправить собранную информацию туда. Никогда не верьте случайным расширениям в Сети.

Всё, что теперь осталось, – это сделать наше расширение устанавливаемым тем же способом, что и другие расширения Yii 2, которые мы до сих пор использовали. Если вам нужно присоединить это вредоносное расширение к вашему приложению вручную и код расши-

рения находится в каталоге /некий/путь/в/файловой/системе, тогда вы можете присоединить расширение «напрямую», написав в конфигурации приложения следующее:

```
'extensions' => array_merge(
    (require __DIR__ . '/../vendor/yiisoft/extensions.php'),
    [
        'malicious\app-info' => [
            'name' => 'Application Information Dumper',
            'version' => '1.0.0',
            'bootstrap' => '\malicious\Bootstrap',
            'alias' => ['@malicious' => '/некий/путь/в/файловой/системе']
        ]
    ]
)
```

Этот вид присоединения расширений проиллюстрирован в ветке extension-manual-loading репозитория git в пакете кода, приложенном к этой книге. Эта ветка доступна по следующей консольной команде:

```
$ git checkout "extension-manual-loading"
```

Пожалуйста, обратите внимание на специфический способ указания настройки extensions. Мы объединяем содержимое файла extensions.php, который поставляется вместе с Yii 2, и наше вручную написанное определение расширения. Файл extensions.php – это то, что позволяет компании Yiisoft распространять расширения при помощи простых вызовов composer require. Давайте узнаем, что нам нужно, чтобы повторить эту функциональность.

Делаем расширение устанавливаемым как... хм... расширение

Для начала проясним ситуацию: мы здесь говорим только о случае, когда Yii 2 установлено при помощи Composer и мы хотим, чтобы наше расширение тоже можно было установить через Composer. Это будет нашим основным допущением.

Давайте вспомним, какие расширения мы уже установили:

- Gii, генератор кода, в *главе 3*;
- расширение Twitter Bootstrap в *главе 4*;
- расширение для отладки в *главе 7*;
- расширение Swiftmailer в *главе 8*.

Все эти расширения мы установили при помощи Composer. Можно вспомнить, что когда мы установили Gii, мы добавили в систему ссылку на файл extensions.php и затем никогда до этого момента не трогали эту часть конфигурации приложения:

```
'extensions' => (require __DIR__ .
  '/../vendor/yiisoft/extensions.php')
```

Если открыть файл vendor/yiisoft/extensions.php (при условии что все расширения из предыдущих глав на самом деле были установлены) и взглянуть на его содержимое, мы увидим следующий код (обратите внимание, что в вашем случае могут быть незначительные отличия):

```
<?php

$vendorDir = dirname(__DIR__);

return array (
  'yiisoft/yii2-bootstrap' =>
  array (
   'name' => 'yiisoft/yii2-bootstrap',
   'version' => '9999999-dev',
   'alias' =>
   array (
     '@yii/bootstrap' => $vendorDir . '/yiisoft/yii2-bootstrap',
   ),
  ),
  'yiisoft/yii2-swiftmailer' =>
  array (
   'name' => 'yiisoft/yii2-swiftmailer',
   'version' => '9999999-dev',
   'alias' =>
   array (
     '@yii/swiftmailer' => $vendorDir . '/yiisoft/yii2-swiftmailer',
   ),
  ),
  'yiisoft/yii2-debug' =>
  array (
   'name' => 'yiisoft/yii2-debug',
   'version' => '9999999-dev',
   'alias' =>
   array (
     '@yii/debug' => $vendorDir . '/yiisoft/yii2-debug',
   ),
```

```
  ),
  'yiisoft/yii2-gii' =>
  array (
   'name' => 'yiisoft/yii2-gii',
   'version' => '9999999-dev',
   'alias' =>
   array (
     '@yii/gii' => $vendorDir . '/yiisoft/yii2-gii',
   ),
  ),
);
```

Одно расширение было выделено, чтобы показать его среди других. Итак, что весь этот код значит для нас?

- Во-первых, он значит, что Yii 2 каким-то образом автоматически генерирует необходимый фрагмент настроек автоматически, когда вы устанавливаете пакет Composer расширения.
- Во-вторых, он значит, что каждое расширение, предоставленное в стандартной поставке Yii 2, в конце концов будет зарегистрировано в настройке `extensions` приложения.
- В-третьих, все классы в расширении становятся доступными в основной базе кода приложения при помощи аккуратно собранного значения настройки `alias` в конфигурации расширения.
- В-четвёртых, в конечном счёте лёгкая установка расширений Yii 2 возможна благодаря некоторой интеграции между Yii и системой распространения пакетов Composer.

Магия скрыта в манифесте `composer.json` всех встроенных в Yii 2 расширений. Детали того, как составлен этот манифест, написаны в документации по Composer, которая доступна по адресу https://getcomposer.org/doc/04-schema.md. Хотя нам на самом деле нужно только одно поле, и это поле `type`.

Yii 2 использует особый тип пакетов Composer под названием `yii2-extension`. Если вы проверите манифесты `yii2-debug`, `yii2-swiftmail` и других расширений, то увидите, что все они имеют вот такую строчку внутри:

```
"type": "yii2-extension",
```

Обычно Composer не сможет понять, как установить пакет такого типа. Однако главный пакет `yii2`, который содержит в себе сам фреймворк, зависит от специального вспомогательного пакета `yii2-composer`:

```
"require": {
    … прочие зависимости ...
    "yiisoft/yii2-composer": "*",
```

Этот пакет предоставляет преднастроенный установщик Composer (Composer Custom Installer, прочитайте об этом по адресу https://getcomposer.org/doc/articles/custom-installers.md), который включает этот тип пакета.

Весь смысл типа пакетов `yii2-extension` – в том, чтобы автоматически обновлять файл `extensions.php` информацией из манифеста расширения. Фактически всё, что нам теперь нужно, – это собрать корректный манифест `composer.json` внутри каталога расширения. Давайте шаг за шагом его напишем.

Подготовка корректного манифеста composer.json

Манифест composer.json – это текстовый файл, содержимое которого является описанием объекта в формате JSON. Поэтому первым символом, который должен быть в этом файле, является открывающая фигурная скобка: {, а последним символом, который должен быть в этом файле, является закрывающая фигурная скобка: }. Всё остальное содержимое между этими двумя скобками мы прямо сейчас постепенно и опишем.

Для начала нам нужен блок с описанием. Предположим, мы опишем себя таким образом:

```
"name": "malicious/app-info",
"version": "1.0.0",
"description": "Пример расширения, которое раскрывает важную информацию
о приложении",
"keywords": ["yii2", "application-info", "example-extension"],
"license": "CC-0",
```

Технически мы обязаны предоставить только `name`. Даже `version` можно опустить, если пакет удовлетворяет следующим двум требованиям:

- он распространяется из какой-то системы контроля версий, например репозитория Git;
- в этом репозитории есть теги (**tags**), корректно идентифицирующие версии в истории фиксации изменений.

Но пока что мы не собираемся с этим возиться.

Затем нам нужно зависеть от Yii 2, просто на всякий случай. Обычно пользователи будут устанавливать расширения только после того,

как фреймворк уже будет на месте, но если расширение уже перечислено в разделе `require` файла `composer.json` среди множества других вещей, мы не можем быть уверены в точном порядке этих выражений, поэтому лучше (и проще) просто объявить зависимость явно.

```
"require": {
   "yiisoft/yii2": "*"
},
```

Затем мы должны указать тип:

```
"type": "yii2-extension",
```

После этого для установщика Yii 2 нам нужно предоставить два дополнительных блока. Первым блоком будет `autoload`. Он нужен для того, чтобы корректно заполнить раздел `alias` конфигурации расширения:

```
"autoload": {
   "psr-4": {
      "malicious\\": ""
   }
},
```

Это объявление означает, что наши классы разложены, согласно правилам PSR-4, таким образом, что классы в пространстве имён `malicious` находятся прямо в корневом каталоге.

Вторым блоком является блок `extra`, в котором мы говорим установщику, что в конфигурации расширения мы хотим объявить раздел `bootstrap`:

```
"extra": {
   "bootstrap": "malicious\\Bootstrap"
},
```

Наш файл манифеста теперь готов. Зафиксируйте всё в системе контроля версий:

```
$ git commit -a -m "Добавлен манифест Composer"
```

Теперь, наконец, мы добавим тег, соответствующий объявленной нами `version`:

```
$ git tag 1.0.0
```

Мы уже упоминали, зачем мы это делаем.

Всё, что осталось, – это сказать Composer, откуда брать содержимое расширения.

Настройка репозиториев

Для того чтобы иметь возможность устанавливать наше расширение, нам нужно настроить какого-то рода репозиторий для него.

Наипростейший способ – это использовать сервис Packagist, доступный по адресу https://packagist.org/, у которого есть прозрачная интеграция с Composer. У этого способа есть следующие плюс и минус:

- плюс: вообще ничего не нужно объявлять в файле `composer.json` приложения, к которому мы хотим присоединить расширение;
- минус: у нас должен быть общедоступный репозиторий системы контроля версий (Git, Subversion или Mercurial), где расширение должно быть опубликовано.

Сейчас мы просто учимся тому, как устанавливать что-либо с помощью Composer; совершенно определённо мы не хотим публиковать своё расширение.

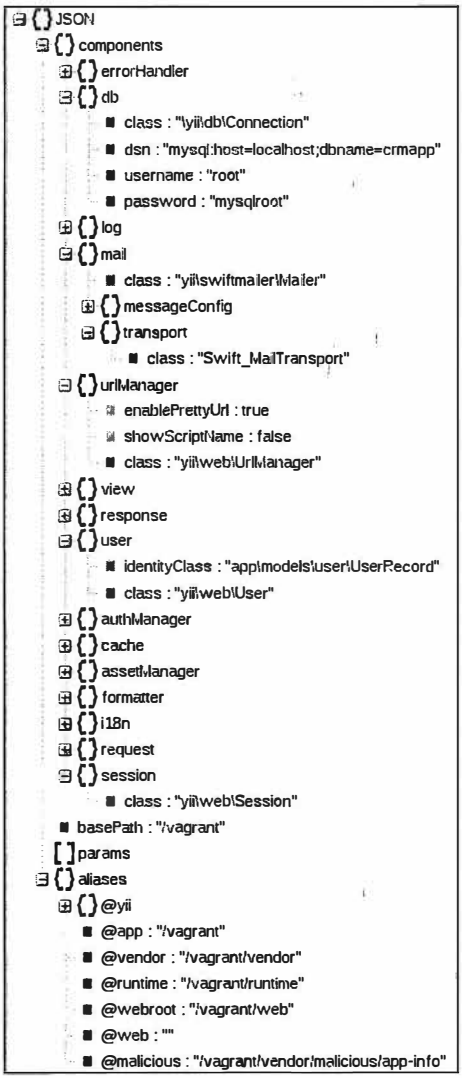
> Ни в коем случае не используйте Packagist для расширения, которое мы делаем в этой главе!

Давайте вспомним о нашей конечной цели. Наша цель – иметь возможность установить расширение, вызвав следующую команду из корневого каталога некоторого приложения Yii 2:

```
$ php composer.phar require "malicious/app-info:*"
```

После этого, перейдя по маршруту `/app-info/configuration`, мы должны увидеть что-то вроде следующего:

Это соответствует следующей структуре данных (снимок экрана сделан при помощи сервиса http://jsonviewer.stack.hu/):



Разместите это расширение в некотором общедоступном репозитории кода, например GitHub, и зарегистрируйте пакет в сервисе Packa-

gist. После этого данная команда будет работать без каких-либо подготовительных действий в манифесте `composer.json` целевого приложения.

Но в нашем случае мы не будем делать это расширение общедоступным, так что у нас остаются два варианта.

Первый вариант заключается в том, чтобы использовать архивированный пакет напрямую. Для этого нужно добавить в `composer.json` целевого приложения раздел `repositories`:

```
"repositories": [
   // определения репозиториев, используемых этим приложением
]
```

> Обратите внимание на то, что и в этом, и в следующем вариантах мы работаем не с манифестом нашего расширения (над которым у нас есть полный контроль), а с манифестом приложения, куда мы хотим поставить наше расширение (контроль над которым сомнителен)! В этом заключается главное неудобство установки необщедоступных расширений.

Чтобы указать репозиторий для пакета, который должен быть установлен из ZIP-архива, нам нужно взять полное содержимое `composer.json` этого пакета (в нашем случае нашего расширения `malicious/appinfo`) и вставить его в виде элемента раздела `repositories` без изменений. Это одновременно самый простой и самый сложный способ настроить зависимость от пакета Composer, но он позволяет зависеть от абсолютно любой папки с файлами (запакованной в архив).

Конечно же, содержимое `composer.json` расширения не указывает местонахождения файлов расширения. Вам нужно добавить это в раздел `repositories` вручную. В конечном счёте в `composer.json` целевого приложения должен появиться следующий дополнительный раздел:

```
"repositories": [
   {
      "type": "package",
      "package": {
// ... пропускаем то, что было скопировано без изменений из манифеста
расширения...
         "dist": {
            "url": "/home/vagrant/malicious.zip", // например
            "type": "zip"
         }
      }
   }
]
```

В JSON нет комментариев, поэтому удалите их, если вы собираетесь копировать и вставлять код прямо из книги.

Этим способом мы указываем местонахождение пакета в файловой системе на той же машине и говорим Composer, что этот пакет – ZIP-архив. Теперь вам просто нужно запаковать содержимое папки `yii2-malicious`, в которой находится наше расширение, положить архив куда-нибудь на целевую машину и указать в параметре `dist.url` корректный URL. URL, который был использован в предыдущем примере, предназначен для установки Vagrant, описанной в приложении 1.

Вам нужно запаковать только содержимое каталога расширения, а не сам каталог.

После этого вы запускаете Composer на машине, с которой этот URL на самом деле достижим (вы, конечно же, можете также использовать URL со схемой `http://`), и получаете следующий ответ от Composer:

```
vagrant@precise64:/vagrant$ php composer.phar require "malicious/app-info:1.0.0"
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing malicious/app-info (1.0.0)
    Downloading: 100%

Writing lock file
Generating autoload files
```

Чтобы проверить, что Yii 2 на самом деле установил расширение, можно открыть файл `vendor/yiisoft/extensions.php` и посмотреть, на самом ли деле он теперь содержит следующий блок:

```
'malicious/app-info' =>
array (
 'name' => 'malicious/app-info',
 'version' => '1.0.0.0',
 'alias' =>
 array (
   '@malicious' => $vendorDir . '/malicious/app-info',
 ),
 'bootstrap' => 'malicious\\Bootstrap',
),
```

(отступы сохранены такими же, как в оригинальном файле). Если этот блок на самом деле там, то теперь можно открыть маршрут `/app-info/configuration` и посмотреть, будет ли он показывать вам отчёт в формате JSON. Должен показывать.

Плюсы и минусы установки из файла такие:

Плюсы	Минусы
Можно указать любой файл, лишь бы он был достижим по какому-нибудь URL. Возможности работать с архивами ZIP существуют на сегодняшний день на практически любой платформе	Слишком много работы в `composer.json` целевого приложения. Требование скопировать манифест целиком в раздел `repositories` просто убийственное. Оно ведёт к очень опасному дублированию кода
Не нужно настраивать репозиторий системы контроля версий исходного кода, хотя это на самом деле сомнительный плюс	Манифест из устанавливаемого пакета расширения вообще не будет обработан. Это значит, что вы не можете убрать дублирующиеся данные из раздела `repositories`, оставив только элементы `dist` и `name`, потому что установщик Yii 2 не сможет получить разделы `autoloader` и `extra`

Последний метод, который можно использовать, – это локальный репозиторий системы контроля версий. У нас уже всё зафиксировано в репозиторий Git, и мы проставили корректный тег, соответствующий версии, объявленной в манифесте. Это всё, что нам нужно подготовить в самом приложении. Теперь нам нужно модифицировать манифест целевого приложения и добавить туда раздел `repositories`, так же, как мы сделали ранее, но на этот раз мы внедрим намного меньше кода:

```
"repositories": [
   {
      "type": "git",
      "url": "/home/vagrant/yii2-malicious/" // пример URL
   }
]
```

Нужно только указать корректный URL до репозитория Git расширения. После того как вы сделаете это, вы можете выполнить команду установки (повторим её ещё раз):

```
$ php composer.phar require "malicious/app-info:1.0.0"
```

Всё будет установлено так же, как обычно. Проверьте успешность установки, взглянув на содержимое файла `vendor/yiisoft/extensions.php` и открыв маршрут `/app-info/configuration` в приложении.

Плюсы и минусы установки на основе репозитория такие:

Плюсы	Минусы
Сравнительно немного кода, который нужно написать в манифесте целевого приложения	Всё равно нужно ковыряться в манифесте целевого приложения, который в общем случае вне вашего контроля, а это значит, что вам придётся давать указания вашим пользователям о том, как устанавливать расширение, что, прямо скажем, не очень хороший пиар
Не нужно на самом деле публиковать расширение. В некоторых случаях это действительно полезно, например для приложений с закрытым исходным кодом	

Итак, вкратце, следующие элементы внутри манифеста `composer.json` превращают произвольный пакет Composer в расширение Yii 2:

- вначале мы говорим Composer, что нужно использовать специальный инсталлятор Yii 2 для пакетов:

  ```
  "type": "yii2-extension"
  ```

- затем мы говорим инсталлятору расширений Yii 2, где находится бутстреппинг нашего расширения (если таковой имеется):

  ```
  "extra": {"bootstrap": "<Fully qualified name>"}
  ```

- далее мы говорим инсталлятору расширений Yii 2, как подготовить псевдонимы путей, чтобы классы могли быть автоматически загружены:

  ```
  "autoloader": {"psr-4": { "namespace": "<folder path>"}}
  ```

- наконец, мы добавляем явную зависимость от самого Yii 2, чтобы гарантировать то, что инсталлятор расширений Yii 2 вообще будет установлен:

  ```
  "require": {"yiisoft/yii2": "*"}
  ```

Всё остальное – это детали установки любого другого пакета Composer, о которых вы можете прочитать в официальной документации по Composer.

В коде, приложенном к этой книге, для всех трёх описанных вариантов (за исключением публикации в Packagist) в репозитории git есть отдельная ветка. Так что проверьте вывод команды `git branch` и выберите, на какой вариант взглянуть. Расширение содержится только в этих ветках; вы больше никогда не увидите его в главной ветке.

Итоги

В этой главе мы взглянули на то, как Yii 2 реализует свои расширения, так что их легко установить одной командой Composer, и после этого они автоматически присоединяются к приложению. Мы узнали, что это требует некоторой степени взаимодействия между двумя системами, Yii 2 и Composer, и это, в свою очередь, требует некоторых дополнительных приготовлений от нас как разработчиков расширения.

Мы использовали достаточно глупый, даже немного опасный пример расширения. Это было сделано по трём причинам:

1) расширение было (мы надеемся) интересно собирать;
2) мы показали, что, используя механику бутстреппинга, мы можем автоматически соединить части расширения с целевым приложением без сложных инструкций по установке вручную;
3) мы показали потенциальную опасность установки случайных расширений из Сети, так как расширение может выполнять полностью произвольный код на этапе инициализации приложения Yii 2. Что происходит при каждом запросе к вашему веб-приложению, между прочим.

Мы обсудили три способа распространения пакетов Composer, которые применимы также и к расширениям Yii 2. Общее правило такое: если вы хотите, чтобы ваше расширение было общедоступным, просто используйте сервис Packagist. Во всех остальных случаях используйте локальные репозитории систем управления версиями исходного кода. Вы всё равно можете использовать как URL на внешние ресурсы, так и локальные пути в файловой системе. Мы также рассмотрели вариант присоединения расширения полностью вручную, не используя установки через Composer.

В следующей главе, *главе 10 «События и поведение»*, мы нырнём с высоты расширений Yii 2, парящих над наивысшей точкой его архитектуры, в тёмные глубины системы событий, простирающейся до самых нижних его уровней.

Глава 10

События

В этой главе мы вернёмся к нашему разделённому дизайну приложения, который мы использовали в главе 2. Мы оставили модель клиента из предметной области, как она была, и начали изучать различные раздельные полезные возможности Yii 2. Давайте теперь вернёмся обратно к модели клиента и реализуем одно интересное и в то же время полезное поведение, которое, без сомнения, потребуется в любом деловом приложении. Во время реализации этого поведения мы узнаем про два отдельных понятия Yii 2: **события (events)** и **поведение (behaviors)**.

Автоматическая пометка записей в БД меткой времени и ID пользователя

Давайте реализуем одну полезную возможность, которую совершенно точно у вас попросят однажды в любом реальном проекте промышленного масштаба. Она описывается следующим образом, словами некоего неизвестного мистера Произвольного Совладельца:

> *«Когда мы записываем нового клиента, дата и время, а также пользователь, который это сделал, тоже должны быть записаны. При любом обновлении клиента дата и время изменения, а также пользователь, который это делает, также должны быть сохранены».*

Мы как разработчики можем перевести это в следующую спецификацию:

1. Добавить четыре поля в таблицу `customer` в базе данных:
   `created_at`: типа `integer` хранит метку времени Unix;
   `created_by`: типа `integer`, является внешним ключом к полю `user.id`;
   `updated_at`: типа `integer`, хранит метку времени Unix;
   `updated_by`: типа `integer`, является внешним ключом к полю `user.id`.

2. Когда новая запись клиента добавляется в базу данных, автоматически заполнить поле `created_at` текущей меткой времени, а поле `created_by` – идентификатором текущего авторизованного пользователя. Автоматически заполнить поле `updated_at` тем же значением, что и поле `created_at`, и заполнить поле `updated_by` тем же значением, что и поле `created_by`.
3. Когда запись о клиенте обновляется в базе данных, автоматически заполнить поле `updated_at` текущей меткой времени, а поле `updated_by` – идентификатором текущего авторизованного пользователя.

Так как мы хотим сохранять точный момент времени, в который произошло событие, нам следует использовать метки времени Unix, которые не зависят от часовых поясов.

Тест создания пользователя

Начнём со случая создания новой записи о пользователе. Закодируем его спецификацию без обиняков в виде интеграционного теста, проверяющего реальное состояние базы данных. Тест, который мы получим в итоге, будет достаточно длинным, поэтому, вместо того чтобы написать его целиком одним фрагментом текста, мы опишем его создание шаг за шагом.

Вот шаги создания этого теста.

1. Создаём файл теста следующим образом:

```
$ ./cept generate:test functional CustomerAudit
```

2. Очищаем сгенерированный файл `tests/functional/CustomerAuditTest.php` до тех пор, пока он не будет выглядеть вот так:

```
class CustomerAuditTest extends \Codeception\TestCase\Test
{
   /** @var \FunctionalTester */
   protected $tester;

   /** @test */
   public function NewCustomerHasAuditInfo()
   {
      // Здесь тестовый сценарий...
   }
}
```

3. Объявляем зависимости для сценария. Для простоты мы будем работать с базой данных, напрямую используя активные записи. Также, так как нам нужно хранить идентификатор текуще-

го авторизованного пользователя, нам нужны экземпляр класса UserRecord, заранее нам известный, и система для обработки аутентификации. Мы не отвязались от Yii в механике аутентификации, поэтому просто напрямую используем компонент yii\web\User. Это не очень хороший стиль, но в функциональных тестах всё равно синглтон Yii валяется рядом, почему бы и не использовать его.

```
// Зависимости
$identity = UserRecord::findOne(['username' =>
  'RobAdmin']);
$user = Yii::$app->user;
```

Для нашего теста будем использовать учётную запись администратора.

4. На шаге подготовки мы входим в систему, «воображаем» запись о клиенте и сохраняем её в базе данных.

```
// Given (Дано)
$user->login($identity);
$customer = $this-> imagineCustomerRecord();
$before = time();
$customer->save();
$after = time();
```

Мы «воображаем» следующим образом:

```
private function imagineCustomerRecord()
{
    $faker = \Faker\Factory::create();
    $record = new CustomerRecord();
    $record->name = $faker->name;
    return $record;
}
```

Для простоты мы устанавливаем только обязательный атрибут клиента, которым является его/её имя.

Мы ожидаем, что будет сохранена метка времени сохранения, поэтому для проверки её корректности мы запоминаем метки времени до и после сохранения записи.

5. Затем мы немедленно извлекаем сохранённую запись о клиенте из базы данных:

```
// When (Когда)
$saved = CustomerRecord::findOne($customer->id);
```

Мы используем идентификатор записи о клиенте, который Yii 2 для нас сгенерировал при сохранении объекта `$customer` в базе данных. Этот идентификатор передаётся методу запроса `findOne()`, для того чтобы создать новый экземпляр активной записи, так что мы будем уверены в том, что мы действительно используем данные из БД, а не запись, хранимую в оперативной памяти.

6. Наконец, мы проверяем наши ожидания от записи клиента, которую мы только что извлекли:

```
// Then (Тогда)
$this->assertInstanceOf
  ('app\models\customer\CustomerRecord', $saved);
$this->assertBetween
  ($before, $saved->created_at, $after);
$this->assertEquals($user->id, $saved->created_by);
$this->assertEquals
  ($saved->created_at, $saved->updated_at);
$this->assertEquals
  ($saved->created_by, $saved->updated_by);
```

Мы проверяем корректность метки времени `created_at` проверкой `assertBetween()`, которая должна проверить, идут ли три её аргумента в порядке увеличения значений. Нам её ещё нужно сделать. Мы проверяем корректность поля `created_by`, сравнив её значение со значением `Yii::$app->id`, потому что мы ещё не вышли из системы после создания записи о клиенте, поэтому эта переменная всё ещё должна хранить тот же идентификатор учётной записи администратора.

Как было сказано в спецификации, значения `updated_at` и `updated_by` должны быть равны `created_at` и `created_by` соответственно.

7. Самодельная проверка, которую мы используем для того, чтобы проверить корректность метки времени, реализована просто:

```
private function assertBetween($before, $value, $after)
{
   $this->assertLessThanOrEqual($before, $value);
   $this->assertGreaterThanOrEqual($after, $value);
}
```

В нашем случае мы сравниваем целочисленные значения, но благодаря особенностям используемых встроенных проверок мы можем

сравнивать всё, что сравнимо в PHP этими проверками. Например, строки.

Теперь, так как мы мухлюем и знаем заранее, как должен выглядеть рабочий код, удовлетворяющий этому тестовому случаю, мы, вместо того чтобы написать его, напишем тест обновления записи о клиенте.

> Нам вообще не нужно даже прикасаться к пользовательскому интерфейсу, чтобы реализовать данную функциональность и убедиться в том, что она работает как положено. Этот и следующий тестовый случай покроют всё, что нам нужно. Для того чтобы протестировать эту функциональность вручную, нам нужно будет внести изменения в существующий пользовательский интерфейс.

Тестовый случай обновления записи о клиенте

Давайте сделаем второй тестовый метод в том же самом классе `\CustomerAuditTest`:

```
    /** @test */
    public function CustomerRecordRemembersUpdateDatetimeAndUser()
    {
        // Здесь сценарий теста...
    }
}
```

Давайте следовать тем же шагам, что и в предыдущем разделе, то есть «зависимости», «дано», «когда» и, наконец, «тогда».

1. Мы всё ещё зависим от класса `yii\web\User`, и мы будем использовать две учётные записи пользователей, потому что мы хотим на самом деле проверить изменения в атрибуте `updated_by`.

```
// Dependencies
$first_identity = UserRecord::findOne
  (['username' => 'RobAdmin']);
$second_identity = UserRecord::findOne
  (['username' => 'AnnieManager']);
$user = Yii::$app->user;
```

2. В качестве подготовки мы входим в систему под первой учётной записью, сохраняем запись `CustomerRecord` и запоминаем начальные значения полей `updated_at` и `updated_by`:

```
// Given
$user->login($first_identity);
$record = new CustomerRecord;
```

```
$record-> name = 'John';
$record->save();

$initial_updated_at = $record->updated_at;
$initial_updated_by = $record->updated_by;
```

3. В наиболее интересной части теста мы выходим из первой учётной записи пользователя, входим под второй учётной записью и пересохраняем запись:

```
// When
$user->logout();
sleep(1);
$user->login($second_identity);
$record-> name = 'Bill';
$record->save();
```

Обратите внимание на вызов `sleep()`. Метки времени Unix имеют точность до секунды, а наш тест, скорее всего, будет выполняться намного быстрее. Для того чтобы поле `updated_at` хоть на сколько-то изменилось, нам придётся подождать как минимум секунду.

Мы *обязаны* что-нибудь изменить в записи, поскольку если ничего не будет изменено в объекте активной записи, вызов `save()` даже не будет возиться с тем, чтобы на самом деле обратиться к базе данных. Это, кстати, очень полезная функциональность.

---

4. Проверки в этом случае намного более простые, потому что всё, что нам нужно проверить, – это то, что текущие значения полей `updated_at` и `updated_by` отличаются от начальных значений:

```
// Then
$this->assertGreaterThan
  ($initial_updated_at, $record->updated_at);
$this->assertNotEquals
  ($initial_updated_by, $record->updated_by);
$this->assertEquals($user->id, $record->updated_by);
```

В этом случае мы не проверяем точность метки времени, но, конечно же, мы проверяем, что идентификатор пользователя верен.

Мы часто входим и выходим из системы в этих тестах. Нам нужно быть уверенными в том, что мы вышли из системы после каждого запуска тестов, чтобы сохранить чистое состояние для всех

остальных тестов. Поэтому давайте вернём метод очистки на место в следующем виде:

```
public function _after()
{
    Yii::$app->user->logout();
}
```

Теперь запускаем тесты и наблюдаем, как они проваливаются:

```
$ ./cept run functional
```

Вот ожидаемый результат запуска:

```
Time: 3.29 seconds, Memory: 17.00Mb

There were 2 errors:

---------
1) CustomerAuditTest::NewCustomerHasAuditInfo
yii\base\UnknownPropertyException: Getting unknown property: app\models\customer
\CustomerRecord::created_at

#1  /vagrant/vendor/yiisoft/yii2/db/BaseActiveRecord.php:246
#2  /vagrant/tests/functional/CustomerAuditTest.php:39

---------
2) CustomerAuditTest::CustomerRecordRemembersUpdateDatetimeAndUser
yii\base\UnknownPropertyException: Setting unknown property: app\models\customer
\CustomerRecord::first_name

#1  /vagrant/vendor/yiisoft/yii2/db/BaseActiveRecord.php:266
#2  /vagrant/tests/functional/CustomerAuditTest.php:56

FAILURES!
Tests: 8, Assertions: 21, Errors: 2.
```

Начнём реализацию функциональности с полей, необходимых в записях в базе данных.

Подготовка полей в базе данных

Как обычно, мы создаём миграцию:

```
$ ./yii migrate/create add_audit_fields_to_customer
```

Давайте напишем следующий код в качестве сценария миграции:

```
$this->addColumn('customer', 'created_at', 'integer');
$this->addColumn('customer', 'created_by', 'integer');
$this->addColumn('customer', 'updated_at', 'integer');
$this->addColumn('customer', 'updated_by', 'integer');

$this->addForeignKey('customer_created_by', 'customer',
  'created_by', 'user', 'id');
```

```
        $this->addForeignKey('customer_updated_by', 'customer',
          'updated_by', 'user', 'id');
```

Внешние ключи на самом деле не требуются для нашей функциональности, но ради полноты картины мы их включили, потому что в реальной жизни они всё равно понадобятся.

Запустим миграцию:

```
$ ./yii migrate
```

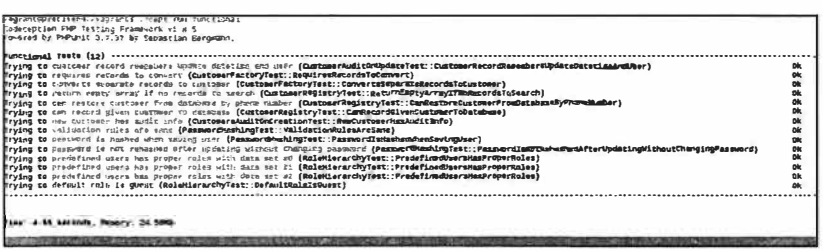
Класс `CustomerRecord` автоматически подберёт эти поля. Какой же код теперь нужно написать, чтобы пройти наконец наши тесты?

Использование поведений «timestamp» и «blameable»

Нам нужно добавить следующий код в класс `CustomerRecord`:

```
public function behaviors()
{
   return [
      'timestamp' => \yii\behaviors\TimestampBehavior::className(),
      'blame' => \yii\behaviors\BlameableBehavior::className()
   ];
}
```

В результате наши тесты волшебным образом окажутся пройденными, что показано на следующем снимке экрана:



Что такого мы здесь сделали?

Объясняя на высоком уровне, метод `behaviors()` класса `yii\base\ActiveRecord` существует для того, чтобы объявлять некоторые классы, которые будут предоставлять дополнительную функциональность для рассматриваемой активной записи. Эта функциональность может быть трёх видов:

- новые методы, доступные для использования;
- новые свойства, доступные для чтения/записи;

- присоединённые обработчики событий.

На самом деле методом behaviors() обладает всё, что является наследником класса \yii\base\Component, поэтому вы можете подсоединить что-нибудь также и к контроллерам, модулям и компонентам приложения.

В случае класса CustomerRecord мы присоединили два поведения, \yii\behaviors\TimestampBehavior и \yii\behaviors\BlameableBehavior, встроенные в Yii 2.

Мы использовали короткий синтаксис для указания присоединяемого поведения. В полном варианте мы указываем не просто полностью определённое имя класса, но ассоциативный массив со значениями свойств этого класса:

```
public function behaviors()
{
   return [
      'timestamp' => [
         'class' => \yii\behaviors\TimestampBehavior::className(),
         // здесь остальные настройки ...
      ]
      'blame' => [
         'class' => \yii\behaviors\BlameableBehavior::className()
         // здесь остальные настройки ...
      ]
   ];
}
```

То, что вы указываете в качестве поведения, будет отправлено в метод \yii\BaseYii::createObject().

И TimestampBehavior, и BlameableBehavior расширяют более обобщённый \yii\behaviors\AttributeBehavior. Следующий фрагмент кода – пример из официальной документации:

```
public function behaviors()
{
   return [
      'attributeStamp' => [
         'class' => AttributeBehavior::className(),
         'attributes' => [
            ActiveRecord::EVENT_BEFORE_INSERT => ['attribute1',
'attribute2'],
            ActiveRecord::EVENT_BEFORE_UPDATE => 'attribute2',
         ],
```

```
            'value' => function ($event) {
                return 'some value';
            },
        ],
    ];
}
```

У AttributeBehavior есть две настройки:

- настройка attributes: это массив, который отображает названия событий из класса ActiveRecord на названия атрибутов. Вы можете указать массив атрибутов для одного и того же события;
- настройка value: это анонимная функция, которая получает в качестве аргумента событие, на которое среагировало AttributeBehavior. Эта функция должна вернуть значение, которое нужно присвоить атрибуту, ассоциированному с этим событием. Вместо анонимной функции вы можете сразу указать фиксированное значение. Конечно же, вам нужно позаботиться о том, чтобы тип значения был чем-то, что ActiveRecord может обработать; он не может магическим образом сохранять объекты или массивы в базу данных.

Поведение «timestamp» – это особая разновидность AttributeBehavior. Оно устанавливает настройку attribute согласно следующему коду:

```
BaseActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
BaseActiveRecord::EVENT_BEFORE_UPDATE => 'updated_at',
```

Эти установки выливаются в точности в то поведение, которое мы определили в начале этого раздела, заполняя поля created_at и updated_at в нужные моменты жизненного цикла ActiveRecord. Это было наше первое мошенничество, когда мы назвали новые поля согласно значениям по умолчанию в классе TimestampBehavior.

По умолчанию это поведение использует результат вызова функции time() в качестве значения настройки value. Вы можете сделать то же самое, присвоив настройке value следующее значение в классе AttributeBehavior:

```
function ($event) {
    return time();
},
```

Это было наше второе мошенничество, когда мы решили использовать метки времени в качестве значений полей created_at и updated_at.

Поведение «blameable» – это тоже специальная форма поведения `AttributeBehavior`. Оно устанавливает настройку `attribute` почти так же, как это делает `TimestampBehavior`, но названия полей отличаются:

```
BaseActiveRecord::EVENT_BEFORE_INSERT => ['created_by', 'updated_by'],
BaseActiveRecord::EVENT_BEFORE_UPDATE => 'updated_by',
```

Так что мы смухлевали третий раз, снова с выбором имён полей.

Значение по умолчанию, которое поведение «blameable» устанавливает для этих атрибутов, – это идентификатор текущего пользователя, взятый прямо из `Yii::$app->user->id`. Поэтому с традиционной механикой аутентификации, которую мы реализовали в главе 5, у нас всё уже было на месте без какой-либо дополнительной подготовки.

Нужно сказать, что всё это предыдущее «мошенничество» было нужно только ради великолепия прохождения сразу двух функциональных тестов после добавления всего четырёх строчек кода, из которых две – синтаксис литералов массивов. В целом концепция остаётся той же самой: в Yii 2 (и, если честно, ранее в Yii 1.x в том числе) вы можете присоединять к объектам некоторое поведение, определённое в отдельных классах. Давайте более детально рассмотрим понятие «поведения».

ВОЗМОЖНОСТЬ: поведение

Три класса поведений, `TimestampBehavior`, `BlameableBehavior` и `AttributeBehavior`, описанных ранее, плюс ещё одно узкоспециализированное поведение `\yii\behaviors\SluggableBehavior`, о котором вы можете прочитать отдельно, если интересно, – это все виды поведений, которые поставляются с Yii 2. Само понятие «поведения» реализовано классом `\yii\base\Behavior`, и поэтому ничто вас не останавливает от реализации новых видов поведения, расширяя его.

Основная идея уже была ранее объяснена: вы можете присоединить поведение к некоторому другому объекту, и этот объект получит методы и свойства, определённые в этом поведении. Это означает, что «поведение» получается тем же самым, что и «особенности» (**traits**), введённые в PHP 5.4 (прочитайте об «особенностях» на странице http://www.php.net/manual/en/language.oop5.traits.php). На самом деле понятие поведения было введено в Yii 1.x для того, чтобы получить возможности «особенностей» до того, как они были реализованы в PHP на уровне языка.

Но поведение в Yii 2 имеет ещё одну возможность: оно также присоединяет обработчики событий к объекту, к которому привязано. И это именно то, благодаря чему работает функциональность `AttributeBehavior`.

Давайте посмотрим на четыре отдельные части, из которых состоит класс поведения:

- во-первых, у поведения есть владелец. Внутри методов поведения вы можете полагаться на то, что по ссылке `$this->owner` вы доберётесь до объекта, к которому вы в данный момент присоединены;
- во-вторых, у поведения есть специальный метод под названием `events`. Этот метод можно (и нужно) переопределить в подклассах, и он должен возвращать ассоциативный массив, отображающий названия событий на анонимные функции;
- наконец, поведение содержит методы `attach($component)` и `detach($component)`, которые вам, возможно, никогда не придётся переопределять. Используя их, вы можете присоединить поведение к объекту `$component`, и оно присоединит все объявленные в нём обработчики событий. Обратите внимание, что присоединение таким методом не предоставит объекту `$component` методов и свойств из поведения! Так что хоть эти методы и присутствуют, и могут быть использованы, в реальности они фактически бесполезны.

Названия событий, указанные в `events()`, должны быть событиями владельца по причинам, о которых мы расскажем позже. Это крайне важно, но в то же самое время довольно полезно, так как события, которые не происходят в текущем владельце поведения, будут просто молча проигнорированы.

Вы можете в качестве обработчика события присоединить любой объект, пригодный для исполнения в PHP, и ещё вы можете указать просто строку, которая будет интерпретирована как название метода в самом поведении (так что вместо того, чтобы писать `[$this, 'handlerName']`, вы можете писать просто `'handlerName'`). Обработчик события будет принимать объект класса `\yii\base\Event` в качестве единственного аргумента.

Чтобы правильно присоединять и отсоединять поведение, вам следует использовать четыре специальных метода класса `\yii\base\Component`:

Название метода	Назначение метода
`attachBehavior($name, $behavior)`	Присоединяет поведение `$behavior` к этому компоненту под именем `$name`
`attachBehaviors($behaviors)`	Присоединяет целый массив видов поведения к этому компоненту. Массив должен отображать название поведения на объекты поведения, подобно аргументам метода `attachBehavior()`
`detachBehavior($name)`	Отсоединяет именованное поведение от этого компонента
`detachBehaviors()`	Выполняет `detachBehavior()` на всех объектах поведения, присоединённых к этому компоненту

Присоединяя (дословный перевод глагола attach) поведение к компоненту, вы делаете обработчики событий, свойства и методы, объявленные в поведении, доступными в компоненте. *Отсоединяя* (дословный перевод глагола detach) поведение, вы делаете эти обработчики событий, свойства и методы недоступными.

В качестве примера мы можем взять `TimestampBehavior`, который объявляет вспомогательный метод `touch($attribute)`. Мы знаем, что у нас нет такого метода в модели `UserRecord`, но мы можем сделать следующее:

```
$user = UserRecord::findOne($id);
$behavior = new TimestampBehavior();
$behavior->value = function () { return date('Y-m-d'); };
$user->attachBehavior('ts', $behavior);
```

После этого представим, что у `UserRecord` есть атрибут `lastLoggedDatetime`. Совершив вышеописанные манипуляции, мы можем сделать следующее:

```
$user->touch('lastLoggedDatetime');
```

В этот момент атрибут `lastLoggedDatetime` примет в качестве значения результат выполнения `date('Y-m-d')`.

Чтобы избавить вас от необходимости присоединять поведение вручную, каждый компонент в Yii 2 определяет специальный метод `behaviors()`. Переопределяя этот метод, вы можете указать поведение, которое должно быть присоединено к этому компоненту в момент создания его экземпляра. Вы уже знаете, что почти всё в Yii 2 является наследником `\yii\base\Component`. Пример присоединения поведения с использованием метода `behaviors()` уже был дан ранее в этой главе.

Важно помнить, что свойства класса поведения тоже будут доступны объекту-владельцу. Нужно понимать, что в Yii 2, если у вас есть методы `getSomething()` и `setSomething()`, у вас условно уже есть свойство `something`, даже если в классе вообще не определено переменной-члена `$something`. Исходный код для волшебных методов `__call()`, `__get()` и `__set()` в определении класса `\yii\base\Component` содержит точное описание механики, лежащей в основе такого... поведения.

ВОЗМОЖНОСТЬ: события

Сама по себе концепция «событий» в Yii 2 – это почти что воплощение паттерна Наблюдатель (см. http://c2.com/cgi/wiki?ObserverPattern). В определённый момент времени объект может осознать, что с ним собирается случиться некое событие. Если в этом объекте содержатся какие-либо обработчики этого события, он их выполняет. Затем объект продолжает делать то, что делал до этого. Таким образом, здесь нет явных «наблюдателей», просто отдельные функции, присоединённые к наблюдаемому в качестве обработчиков событий.

Эта концепция реализована в Yii 2 тремя методами класса `\yii\base\Component`:

Метод	Смысл
`on( $name, $handler, $data = null, $append = true )`	Присоединяет обработчик событий `$handler` к событию под названием `$name`. В зависимости от значения `$append` этот `$handler` будет добавлен либо в конец, либо в начало списка обработчиков, возможно, уже присоединённых к этому событию. Указав `$data`, вы можете передать некоторые произвольные данные в `$handler` (об этом позже)
`off($name, $handler = null)`	Удаляет обработчик событий `$handler` из списка обработчиков для события под названием `$name`. Никакой магии здесь нет: если вы не предоставите этому методу аргумент `$handler`, будут удалены все обработчики. Если же вы предоставите, то будет удалён тот обработчик, который равен `$handler`. Сравнение выполняется простым оператором ==, поэтому вы не сможете удалить таким образом обработчик события, определённый в виде замыкания или анонимной функции
`trigger($name, $event = null)`	«Вызывает» событие под названием `$name`, что означает просто-напросто вызов всех обработчиков событий, ассоциированных с этим событием. Аргумент `$event`, если передан, обязан быть наследником `\yii\base\Event`. Если этот аргумент оставлен неуказанным, `trigger()` сам создаст экземпляр `\yii\base\Event`

Как этой системой пользоваться? Так же, как и с паттерном Наблюдатель, такая система событий полезна, когда вам нужно выполнить некоторые действия в ответ на что-то, произошедшее в другом слое приложения. Другое важное преимущество в том, что вы можете динамически изменять поведение, которое должно проявляться в ответ на события, что тоже может быть очень полезно. И, конечно же, к одному событию можно присоединить любое количество обработчиков из совершенно разных слоёв приложения.

Давайте посмотрим на проблему обработки контента, генерируемого пользователями, а именно на загрузку фотографий. Предположим, что у вас к приложению присоединён отдельный специальный компонент работы с изображениями, который доступен через вызов `Yii::$app->images`. В этом компоненте доступен один метод, который называется `handleUpload()`; таким образом, когда вы загружаете изображение на сервер, вы в конечном счёте вызываете `Yii::$app->images->handleUpload($_FILES)`.

Скажем, у вас уже есть бизнес-правило, согласно которому все загруженные изображения должны быть уменьшены до размера в 2000 пикселей по большей стороне. Таким образом, в этом методе обработки фотографий у вас есть должный вызов к фоновой задаче (не важно, как вы реализовали фоновые задачи в вашем приложении), которая вызывает какую-нибудь отдельную библиотеку для осуществления преобразования.

Теперь представьте, что в определённый момент «сверху» пришло сообщение о том, что нам нужно дополнительно скопировать это загруженное изображение (без изменений) в отдельную подсистему приложения, где менеджер вручную проверит его на предмет нарушений авторских прав. Конечно же, это добавит всего лишь один вызов к ещё одной функции в вашем методе обработки фотографий, но высока вероятность того, что подобные запросы продолжат приходить, и этот метод превратится в свалку разнородного кода (с условиями и циклами, конечно же).

Вместо этого процедура `handleUpload()` может заниматься всего одной вещью: класть изображение в некоторое заранее известное временное место, где оно не будет удалено нижележащей операционной системой (как заканчивают все файлы, загруженные в PHP обычным образом). После этого данная процедура выполнит следующий вызов:

```
$this->trigger(self::IMAGE_UPLOADED, $event);
```

объявив таким образом всем заинтересованным сторонам о том, что изображение вправду было загружено. Параметр `$event` будет новым экземпляром некоторого подкласса `\yii\base\Event`, загруженного информацией о том, как найти это загруженное изображение.

В таких условиях мы можем следующим образом объявить в каком-нибудь удобном месте приложения, что нам нужно уменьшить размер загруженного изображения и положить его в базу данных:

```
Yii::$app->images->on(Images::IMAGE_UPLOADED, $downscale_and_save);
```

Здесь параметр `$downscale_and_save` является анонимной функцией следующего вида:

```
function ($event) {
   // … получаем путь до файла из параметра $event …
   // … уменьшаем изображение ...
   // … сохраняем изображение в БД ...
}
```

Возможно, в другом месте (например, в методе `init()` контроллера, который будет отрисовывать интерфейс загрузки изображений) вы размещаете иное объявление обработчика события `IMAGE_UPLOADED`. Этот обработчик будет заниматься публикацией изображения в некоторую часть административного интерфейса для осмотра на предмет нарушений.

Сложно подтвердить ценность метода `off()` в PHP, где приложение на каждый отдельный HTTP-запрос запускается, работает и останавливается. Более всего он бы был полезен в системах, которые тихо ждут пользовательского ввода и отвечают на постоянно появляющиеся события. С методом `off()`, удаляющим обработчики событий, система могла бы фактически переконфигурировать сама себя, когда нужно, в зависимости от каких-либо факторов. Когда же ваш сценарий живёт только в рамках одного HTTP-запроса, обычно не нужна такая степень полиморфизма. В вашем случае, однако, всё может быть по-другому.

Класс `\yii\base\Event` представляет информацию о произошедшем событии. В нём содержатся четыре фрагмента информации:

- `$name`: это название данного объекта события. Когда вы вызываете `trigger()`, вам нужно передать название события в качестве первого аргумента. Вот это имя и будет значением атрибута `$name`. У вас нет контроля над значением этого атрибута, даже если вы передадите в вызов `trigger()` вручную созданный объект `$event`;
- `$sender`: это объект, который вызвал событие. Если он не установлен вручную, то `trigger()` устанавливает в качестве значения

этого атрибута ссылку на `$this`, что значит, что по умолчанию вы можете полагаться на то, что значением `$sender` действительно является объект, вызвавший `trigger()`;

- `$data`: это дополнительные данные, связанные с обработчиком события, когда вы устанавливали его методом `on()`. Таким образом, в момент присоединения обработчика события вы можете указать некоторые данные, которые будут доступны в обработчике через `$event->data`. Следует подчеркнуть, что это данные, вычисленные не в момент вызова `trigger()`, а в момент вызова `on()`!
- `$handled`: это специальный флаг (изначально установленный в `false`), который, будучи установленным в `true`, останавливает обработку этого события. Таким образом, если были ещё какие-либо обработчики этого события, они не будут выполнены. Вот зачем нужен аргумент `$append` в вызове `on()`: порядок присоединения обработчиков события имеет значение.

Методы `on()`, `off()` и `trigger()`, описанные ранее, были определены в терминах определённого экземпляра класса, то есть конкретный объект будет запускать обработчики своего события, присоединённые конкретно к нему. Но иногда полезно иметь обработчики событий, привязанные не к объекту, а к самому классу, которые будут выполняться вне зависимости от конкретного экземпляра класса, в котором произошло событие. В классе `\yii\base\Event` есть три статических метода, которые предоставляют эту функциональность:

Метод	Смысл
`on( $class, $name, $handler, $data = null, $append = true )`	Присоединяет обработчик `$handler` к событию `$name`, срабатывающему в любом экземпляре `$class`. Значением `$class` должно быть полностью определённое имя класса. Смысл параметров `$data` и `$append` тот же, что и в методе `Component.on()`
`off($class, $name, $handler = null)`	Удаляет `$handler`, связанный с событием по имени `$name`, из списка обработчиков для `$class`. Значением `$class` должно быть полностью определённое имя класса. В остальном метод работает так же, как метод `Component.off()`
`trigger($class, $name, $event = null)`	Подобно методу `\yii\base\Component::trigger($name, $event = null)`, вызывает обработчики для события под названием `$name`, определённые для некоторого `$class`, передавая им `$event`. Значением `$class` должно быть полностью определённое имя класса или объект. Если это объект, то `trigger()` извлечёт имя класса из него. Если `$class` – это имя класса, а не объект, тогда свойство `$sender` объекта, передаваемого в обработчики, будет равно `null`

Например, скажем, что ваш бизнес требует, чтобы каждый раз, когда новый пользователь регистрируется в вашем приложении, нужно было отправлять электронное письмо. Почему метод `createUser()`, чья единственная обязанность – создавать новую запись о пользователе в базе данных, должен, грубо говоря, вызывать также функцию `mail()`? Это излишне раздует его и нарушит принцип единой ответственности (**Single Responsibility Principle**, это достаточно известный принцип, посмотрите о нём, например, здесь: http://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html). Использовать передачу событий, основанных на классах, будет гораздо более чистым решением.

Основная идея будет такова: мы устанавливаем обработчик на событие вставки новой `UserRecord` (или как она названа в вашем проекте) внутри какого-нибудь кода инициализации на подходящем слое. Например, в методе `init()` для всего приложения мы можем вставить следующую строчку:

```
Event::on(
   '\app\models\user\UserRecord',
   \yii\db\ActiveRecord::EVENT_AFTER_INSERT,
   $send_email
);
```

Переменная `$send_email` – это анонимная функция следующего вида:

```
function ($event) {
   // … извлекаем данные о UserRecord из параметра $event …
   // … отправляем электронное письмо ...
}
```

Довольно важно знать, что когда вы инициируете событие на объекте, вы также инициируете событие с тем же именем на классе этого объекта. Упрощая, это значит, что `\yii\base\Component::trigger()` вызывает в качестве последнего действия `\yii\base\Event::trigger()`, передавая в него свои собственные аргументы.

Если объект события, переданный в привязанный к объекту `trigger()`, будет иметь флаг `$handled`, установленный в `true`, и таким образом его обработка должна будет остановиться, он всё равно будет передан в привязанный к классу `trigger()`, и его флаг `$handled` будет заново переустановлен в значение `false`, так что вы не можете предотвратить появление событий, привязанных к классу, манипулируя флагом `$handled`.

В предыдущем разделе мы вкратце упомянули метод `events()` класса `Behavior`, который нужно переопределять, для того чтобы указать обработчики событий будущего владельца этого поведения. Теперь должно быть довольно очевидно, что поведение делает, когда его присоединяют к владельцу методом `attach()`: оно вызывает `$owner->on()` на каждой паре событие–обработчик, определённой методом `events()`. И это то, почему поведение должно объявлять только события, ожидаемые от его будущего владельца: именно владелец будет вызывать `trigger()`, и есть лишь ограниченный выбор событий, которые он инициирует. Любой другой обработчик, объявленный в методе `Behavior.events()`, имеет шанс никогда не быть вызванным.

Теперь можно ясно увидеть разницу между тем, что делает этот код:

```
$behavior->attach($owner);
```

и тем, что делает этот код:

```
$owner->attachBehavior($behavior);
```

Форма `$behavior->attach()` только лишь вызовет `$owner->on()` для всех событий, перечисленных в его методе `events()`. Форма `$owner->attachBehavior()` сделает то же самое и заодно зарегистрирует `$behavior` внутри компонента `$owner`, так что он будет иметь возможность использовать свойства и методы, определённые в `$behavior`.

Система событий в Yii 2 позволяет вам писать своё приложение хотя бы частично в основанной на событиях парадигме, что уже само по себе неплохо. Но то, что делает её по-настоящему полезной с самого начала, – это то, что многие встроенные в Yii 2 компоненты уже содержат события, которые инициируются для вас. Это означает, что вы можете подцепиться к различным фазам жизненного цикла компонентов вроде активных записей, контроллеров или модулей. Между прочим, именно поэтому мы могли использовать наследников `AttributeBehavior` ранее в этой главе. Если бы класс `ActiveRecord` не вызывал `trigger()` на событиях `EVENT_BEFORE_INSERT` и `EVENT_BEFORE_UPDATE`, тогда все вышеописанные манипуляции были бы бессмысленными. В следующем разделе мы взглянем на события, которые уже имеются для вашего использования.

Встроенные события

Здесь мы избавим вас от полнотекстового поиска через достаточно обширную базу кода Yii 2 и перечислим из него всё, что определено под названием, начинающимся с символов «`EVENT_`".

Заметьте, что вы должны предоставить в качестве аргумента `$name` метода `on()` строковое значение. Все встроенные события объявляют эти строки в виде констант класса, и вы, присоединяя обработчики событий, всегда должны использовать именно эти константы класса, а не их настоящие значения.

События класса \yii\base\Application

Следующие события вызывают как классы веб-приложения, так и классы консольного приложения, поставляющиеся с Yii 2. В качестве `$event` в метод `trigger()` ничего не будет передано, так что ожидайте объект события по умолчанию.

Название события	Когда инициируется
`EVENT_BEFORE_REQUEST`	Прямо перед началом обработки запроса, фактически в начале жизненного цикла приложения. Это самое раннее событие, происходящее в приложении, к которому вы можете привязаться
`EVENT_AFTER_REQUEST`	Сразу после успешной обработки запроса или когда вы вызываете `Yii::$app->end()`, принудительно, таким образом, выключая приложение. Это *не* самое последнее событие, происходящее в приложении, таковым является событие `\yii\web\Response::EVENT_AFTER_SEND`

События класса \yii\base\Controller

Следующие события будут инициировать как контроллеры веб-приложения, так и контроллеры консольного приложения. Обратите внимание, что в обработчики будет передан экземпляр класса `\yii\base\ActionEvent`, так что проверьте документацию этого класса здесь: http://www.yiiframework.com/doc-2.0/yii-base-controller.html, в нём есть некоторые полезные возможности.

Название события	Когда инициируется
`EVENT_BEFORE_ACTION`	Прямо перед запуском любого действия контроллера. Если контроллер принадлежит модулю, тогда вначале инициируется `EVENT_BEFORE_ACTION` того модуля. Как было сказано ранее, в обработчики этого события будет передан экземпляр класса `ActionEvent`. Вы можете установить его атрибут `isValid` в значение `false`, и это запретит действию контроллера выполниться (приложение выключится без отрисовки чего бы то ни было, только последующие события будут инициированы)
`EVENT_AFTER_ACTION`	Прямо после выполнения действия контроллера. Обратите внимание на то, что в поле `result` экземпляра класса `ActionEvent`, переданного в обработчик, будет находиться объект отклика, сгенерированный действием, так что вы можете его как-нибудь дополнительно обработать

Заметьте, что оба этих события инициируются внутри методов `beforeAction()` и `afterAction()`. С фреймворком Yii версии Yii 1.1.x традиционно эти методы переопределялись в дочерних классах, для того чтобы добиться каких-либо эффектов пред- или постобработки. Хотя вы и можете делать то же самое в Yii 2, вы всегда в таком случае должны вызывать в переопределённых методах методы `parent::beforeAction()` и `parent::afterAction()` соответственно, потому что иначе вы предотвратите инициацию вышеописанных событий.

Лучше использовать полноценные обработчики событий, чем перекрывать методы `beforeAction()` и `afterAction()`.

События класса \yii\base\Module

Семантика событий класса `\yii\base\Module` в точности такая же, как и у соответствующих событий в классе `\yii\base\Controller`. Но они происходят *вокруг* соответствующих событий контроллера.

Название события	Когда инициируется
`EVENT_BEFORE_ACTION`	Прямо перед выполнением любого действия контроллера и до того, как будет инициировано такое же событие в контроллере. В обработчики этого события будет передан экземпляр класса `ActionEvent`. Вы можете установить его атрибут `isValid` в значение `false`, и это запретит действию контроллера выполниться (приложение выключится без отрисовки чего бы то ни было, только последующие события будут инициированы)
`EVENT_AFTER_ACTION`	Прямо после выполнения действия контроллера и после того, как будет инициировано то же самое событие в контроллере. Обратите внимание на то, что в поле `result` экземпляра класса `ActionEvent`, переданного в обработчик, будет находиться объект отклика, сгенерированный действием, так что вы можете его как-нибудь дополнительно обработать, даже если постпроцессинг уже был совершён контроллером в своём `EVENT_AFTER_ACTION`

Модули тоже содержат свои методы `beforeAction()` и `afterAction()`. Посмотрите раздел «События класса \yii\base\Controller» для объяснения тех сложностей, которые этот факт вызывает.

События класса \yii\base\View

Обратите внимание на то, что компонент представлений, использующийся в веб-страницах, – это класс `\yii\web\View`, и он определяет два дополнительных события, к которым можно подцепиться. Этот конкретный класс определяет только базовые события в процессе отрисовки.

Название события	Когда инициируется
EVENT_BEGIN_PAGE	Вызов к $this->beginPage() внутри файла представления инициирует это событие. Этот метод в деталях описан в главе 4 и используется во фреймворке шаблонов представлений
EVENT_END_PAGE	Вызов $this->endPage() внутри файла представления инициирует это событие. Этот метод в деталях описан в главе 4 и используется во фреймворке шаблонов представлений
EVENT_BEFORE_RENDER	При отрисовке файла представления это событие инициируется перед тем, как отрисовка на самом деле произойдёт. Обработчики получат экземпляр \yii\base\ViewEvent, и вы можете установить его свойство isValid в false, для того чтобы предотвратить отрисовку. Сам вызов trigger() для этого события вложен в метод под названием beforeRender(), и, таким образом, если вы переопределите его, вам придётся вызывать parent::beforeRender() или потерять обработчики этого события. Базовая реализация beforeRender() возвращает в точности значение ViewEvent.isValid
EVENT_AFTER_RENDER	При отрисовке файла представления это событие инициируется после того, как отрисовка будет закончена. Обработчики получат экземпляр \yii\base\ViewEvent, и вы можете использовать его свойство output для какой-нибудь постобработки результата отрисовки. Сам вызов trigger() для этого события вложен в метод под названием afterRender(), и, таким образом, если вы переопределите его, вам придётся вызывать parent::afterRender() или потерять обработчики этого события. Базовая реализация afterRender() возвращает результат отрисовки

События класса \yii\web\View

Давайте посмотрим на дополнительные события, объявленные специально для компонента представлений, используемого в веб-приложениях.

Название события	Когда инициируется
EVENT_BEGIN_BODY	Вызов к $this->beginBody() внутри файла представления инициирует это событие. Этот метод в деталях описан в главе 4 и используется во фреймворке шаблонов представлений
EVENT_END_BODY	Вызов к $this->endBody() внутри файла представления инициирует это событие. Этот метод в деталях описан в главе 4 и используется во фреймворке шаблонов представлений. Это хорошее место для работы с пакетами материалов, так как сразу после инициации этого события пакеты материалов регистрируются в представлении

События класса \yii\base\Model

Базовый класс модели – основа для активных записей. Здесь определены только самые базовые события, но они применимы также и к жизненному циклу ActiveRecord.

Название события	Когда инициируется
EVENT_BEFORE_VALIDATE	Прямо перед валидацией атрибутов методом validate() (вспомните, что метод save() также вызывает validate(), если вы не подавите это поведение). Экземпляр \yii\base\ModelEvent передаётся в обработчик этого события. Вы можете использовать его атрибут isValid, чтобы указать, должна ли эта модель считаться проверенной. Если isValid имеет значение false, тогда модель проваливает валидацию, даже если никакие валидаторы не сигнализировали ошибку, так что пользуйтесь этой возможностью с осторожностью, поскольку в таком случае вы не получите никакого автоматического отчёта о причинах провала
EVENT_AFTER_VALIDATE	Сразу после того, как валидаторы закончат работу. Ничего особенного не передаётся в обработчики этого события. Это событие здесь для того, чтобы вы могли провести какую-либо постобработку атрибутов рассматриваемой модели

Заметьте, что, подобно методам beforeAction() и afterAction() у контроллера, инициация этих событий вложена в методы beforeValidate() и afterValidate(), и те же самые предупреждения насчёт их переопределения также применимы.

Запомните, что вы можете добраться до модели, которая проходит валидацию, через поле $event->sender. Это единственное неудобство, которое вам придётся пережить, используя полноценные обработчики событий вместо переопределения методов beforeValidate() и afterValidate().

События класса \yii\db\BaseActiveRecord

Класс BaseActiveRecord является базовым классом для класса ActiveRecord, который вы будете использовать большую часть времени. Активные записи, будучи подклассами \yii\base\Model, вызывают также и его события.

Чтобы не повторять то же самое предупреждение снова и снова, мы дадим одно прямо сейчас. Все перечисленные ниже события инициируются внутри специальных методов класса BaseActiveRecord. В Yii 1.1.x было стандартной практикой переопределять эти методы

в классах моделей, но если вы продолжите так делать в Yii 2, то должны всегда сначала вызывать родительскую реализацию метода, или вы потеряете вызов `trigger()`.

Название события	Когда инициируется
`EVENT_INIT`	В конце метода `init()` класса `BaseActiveRecord`, который вызывается сразу после конструктора
`EVENT_AFTER_FIND`	После того, как `ActiveRecord` найден каким-либо запросом к базе данных. Вызов этого события находится в методе `afterFind()` класса `BaseActiveRecord`
`EVENT_BEFORE_INSERT`	Перед тем, как на самом деле сохранить новую активную запись в базе данных. Вызов этого события находится в методе `beforeSave()` класса `BaseActiveRecord`. Экземпляр класса `\yii\base\ModelEvent` будет передан в обработчик. Если вы присвоите его атрибуту `isValid` значение `false`, сохранение будет молча отменено
`EVENT_AFTER_INSERT`	После того, как запись была сохранена в базе данных. Вызов этого события находится в методе `afterSave()` класса `BaseActiveRecord`
`EVENT_BEFORE_UPDATE`	Перед тем, как на самом деле обновить активную запись в базе данных. Вызов этого события находится в методе `beforeSave()` класса `BaseActiveRecord`. Экземпляр класса `\yii\base\ModelEvent` будет передан в обработчик. Если вы присвоите его атрибуту `isValid` значение `false`, сохранение будет молча отменено
`EVENT_AFTER_UPDATE`	После того, как запись была обновлена в базе данных. Вызов этого события находится в методе `afterSave()` класса `BaseActiveRecord`
`EVENT_BEFORE_DELETE`	Перед удалением активной записи из базы данных. Вызов этого события находится в методе `beforeDelete()` класса `BaseActiveRecord`. Экземпляр класса `\yii\base\ModelEvent` будет передан в обработчик. Если вы присвоите его атрибуту `isValid` значение `false`, удаление будет молча отменено
`EVENT_AFTER_DELETE`	После удаления активной записи из базы данных. Вызов этого события находится в методе `afterDelete()` класса `BaseActiveRecord`

Два события инициируются в одном и том же методе `beforeSave()`: `EVENT_BEFORE_INSERT` и `EVENT_BEFORE_UPDATE`. Подобным образом два события инициируются в одном и том же методе `afterSave()`: `EVENT_AFTER_INSERT` и `EVENT_AFTER_UPDATE`. Обратитесь к реализации этих методов в исходном коде Yii 2, чтобы узнать, как они работают, потому что они собраны таким образом, чтобы выборочно инициировать события в зависимости от того, вставляется новая запись или обновляется существующая.

Для примера давайте посмотрим на последовательность событий, когда вы создаёте и сохраняете модель, основанную на классе ActiveRecord: EVENT_INIT → EVENT_BEFORE_VALIDATE → EVENT_AFTER_VALIDATE → EVENT_BEFORE_INSERT → EVENT_AFTER_INSERT.

События класса \yii\db\Connection

Экземпляр \yii\db\Connection – это компонент, который вы присоединяли к приложению и который доступен по идентификатору db. Когда вы делаете Yii::$app->db, то получаете доступ к подключению к базе данных.

Название события	Когда инициируется
EVENT_AFTER_OPEN	После того, как сделана вся работа по установлению соединения с базой данных. Это событие не передаёт ничего особенного в свои обработчики
EVENT_BEGIN_TRANSACTION	Прямо перед тем, как транзакция в базе данных начнётся, используя нижележащую библиотеку PDO. Обратите внимание, что это произойдёт достаточно глубоко внутри вызова Yii::$app->db->beginTransaction()
EVENT_COMMIT_TRANSACTION	Сразу после того, как транзакция в базе данных будет зафиксирована, используя нижележащую библиотеку PDO
EVENT_ROLLBACK_TRANSACTION	Сразу после того, как транзакция в базе данных будет отменена из-за какой-либо ошибки

События класса \yii\web\Response

Все события отклика происходят внутри метода \yii\web\Response::send(). Они отмечают три шага в жизненном цикле отклика.

Название события	Когда инициируется
EVENT_BEFORE_SEND	В самом начале отправки отклика
EVENT_AFTER_PREPARE	После того, как отклик был отформатирован подходящими компоновщиками в методе \yii\web\Response::prepare()
EVENT_AFTER_SEND	После того, как отклик будет отправлен. Это последнее событие, происходящее в приложении и к которому вы можете привязаться

События класса \yii\web\User

Компонент пользователя рассказывает миру только о событиях входа и выхода из системы. Обработчики всех этих событий получат экземпляр класса \yii\web\UserEvent. Помимо прочих свойств, о которых вы

можете прочитать в документации, у него есть свойство `isValid`. Оно имеет тот же смысл и назначение, что мы уже видели в экземплярах объектов событий для классов `Response`, `View`, `ActiveRecord` и `Controller`.

Название события	Когда инициируется
`EVENT_BEFORE_LOGIN`	В начале процедуры входа в систему, расположенной в методе `login()`. Если вы установите атрибут `isValid` объекта `UserEvent` в значение `false` внутри обработчика, `login()` молча отменится
`EVENT_AFTER_LOGIN`	После того, как пользователь успешно войдёт в систему, в конце метода `login()`
`EVENT_BEFORE_LOGOUT`	В начале процедуры выхода из системы, расположенной в методе `logout()`. Если вы установите атрибут `isValid` объекта `UserEvent` в значение `false` внутри обработчика, `logout()` молча отменится
`EVENT_AFTER_LOGOUT`	После того, как пользователь успешно выйдет из системы, в конце метода `logout()`

Обратите внимание на довольно удивительную возможность отказать пользователю в *выходе* из приложения одним только манипулированием объекта события. Серьёзно сложно представить себе ситуацию, в которой вам может понадобиться принудительно держать пользователей аутентифицированными.

События класса \yii\mail\BaseMailer

Класс `BaseMailer` – абстрактный класс, который предназначен для переопределения некоторой конкретной реализацией настоящего отправителя электронной почты. Мы видели пример в главе 10, когда настраивали отправку сообщений журнала через Swiftmailer. Он публикует только два события, и оба передают в обработчик экземпляр класса `\yii\base\MailEvent`.

Название события	Когда инициируется
`EVENT_BEFORE_SEND`	В начале метода `send()`. Инициация этого события вложена в метод `beforeSend()`, так что вам нужно вызывать `parent::beforeSend()`, если вы переопределяете его. Объект класса `MailEvent`, переданный в обработчик, содержит атрибут `isValid`. Если вы установите его в `false`, отправка будет молча отменена
`EVENT_AFTER_SEND`	В конце метода `send()`. Инициация этого события вложена в метод `afterSend()`, так что вам следует вызывать `parent::afterSend()`, если вы переопределяете его. Объект `MailEvent`, переданный в обработчик, содержит атрибут `isSuccessful`, который говорит, была ли отправка успешной

Обратите внимание, что значение `isSuccessful` является просто копией значения, возвращаемого из метода `sendMessage()`, который должны реализовать конкретные классы.

Итоги

Если честно, сложно оправдать использование должного программирования на основе событий в PHP, где программа запускается, обрабатывает запрос и умирает. Однако в сочетании с концепцией «поведения» события предоставляют места, в которых мы можем расширять существующую функциональность, что может привести к очень простым решениям, таким как представленное в начале этой главы.

Вот сила, данная вам событиями в Yii 2.

В этой главе мы сначала пощупали поверхность системы событий, использовав поведение, поставляемое вместе с Yii. Затем мы в мельчайших деталях изучили, чем в точности являются понятия «событий» и «поведения» в Yii 2 и как они реализованы. Наконец, мы полностью разобрали типы событий, которые уже происходят внутри приложения Yii 2 и к которым вы можете прицепиться без каких-либо особых приготовлений.

Следующая глава обещает нам много работы. Вначале мы усилим нашу модель клиента всеми отсутствующими до сих пор частями, которые мы оставили нереализованными в главе 2. После этого мы воспользуемся самым сложным виджетом в Yii 2: виджетом `GridView`.

Глава 11

Таблица

Ранее в главе 2 мы кое-где срезали путь и реализовали только часть функциональности нашего и без того рудиментарного примера CRM-приложения. Эта глава завершит пример.

Нашей целью здесь будет закончить CRUD для модели клиента и получить эффективный пользовательский интерфейс для фильтрации записей о клиентах.

Мы будем учиться использовать самый сложный виджет среди всех, поставляемых вместе с Yii: виджет `GridView`, отвечающий за автоматическую отрисовку табличных интерфейсов. Однако виджет `GridView` здесь только для показа записей из базы данных; нам также нужно сделать веб-форму, для того чтобы записывать клиентов в базу данных, и для этого существует виджет `ActiveForm`. Рассмотрение этого важного виджета мы отложим на потом.

В этой главе мы рассмотрим следующие темы:

- концепция виджетов в целом;
- структура виджета `GridView`, в особенности его понятие «колонок»;
- встроенные возможности виджета `GridView`;
- два способа создания преднастроенных колонок в `GridView`;
- реализация сортировки и фильтрации для преднастроенных колонок.

Избавление от слоя предметной области

В главе 2 мы очень старались отвязаться от фреймворка, для того чтобы иметь возможность тестировать и расширять наше приложение в дальнейшем, когда у нас должна появиться настоящая бизнес-логика, помимо безмозглого CRUD-интерфейса. Конечно же, это подразумевает очень много дополнительной работы, так как мы фактически добавляем ещё один уровень абстракции над активными записями.

Это не идиоматичный код Yii. Yii предоставляет всеобъемлющий и удобный API для множества областей, но в то же самое время он

очень старается привязать вас как к своему слою данных, так и к своему слою презентации, что потом всю дорогу будет причинять вам одни неудобства.

Скорее всего, вы много раз увидите такой код при обслуживании чужих приложений, основанных на Yii, и поэтому мы решили показать вам, как веб-приложения разрабатываются в стиле Yii, в пику разработке в стиле использования Yii, показанной в главе 2. Мы будем вызывать активные записи прямо из контроллеров и передавать их напрямую в представления. Большая часть нашего кода будет автоматически сгенерирована и оставлена как есть. Мы фактически выкидываем слои приложения и предметной области. Давайте посмотрим и прочувствуем результат, который у нас получится.

Так как у нас уже есть активные записи в нашем слое данных, такое изменение ничего не будет нам стоить, потому что мы будем генерировать их в любом случае. Мы также не будем писать никаких автоматических тестов, полагаясь только на визуальный осмотр страниц, которые мы делаем. В этом случае мы также получим работоспособное приложение, и мы получим его даже ещё быстрее.

Дизайн списка клиентов

Подразумевается, что вы не начинаете с нуля, но продолжаете с примером, с которым мы до сих пор работали. Однако если вы на самом деле хотите начать с нуля, тогда используйте миграцию, которая создаёт таблицу `customers`, и автоматически сгенерируйте модель клиента и связанный с ней CRUD при помощи Gii. Вам также понадобятся изменения, описанные в главе 10, так как мы будем использовать поля аудита, введённые там, и система пользователей из главы 5.

Мы будем работать с маршрутом `/customers/index` нашего примера CRM-приложения. Мы хотим, чтобы эта страница показывала нам список всех клиентов, зарегистрированных в базе данных. В дальнейшем мы сделаем так, чтобы в ней были возможности отбора клиентов по их именам, номерам телефонов, стране и дате рождения, а также сортировки списка по тем же полям.

Для того чтобы это работало, нам нужно реализовать модель `Address`, и нам нужно серьёзно улучшить форму создания клиента, для того чтобы можно было приписывать клиентам адреса. То же самое относится к модели `Email`. Эта глава посвящена лишь `GridView`, поэтому мы вынесли описание изменений в формах создания и редактирования клиента в приложение 2.

Нам нужно действовать внимательнее, так как у клиента теперь наличествует слишком много элементов информации. Если мы будем показывать каждое поле записи о клиенте в виде отдельной колонки в таблице, эта таблица станет слишком широкой, чтобы показывать её на экране. Наша главная задача в обращении с `GridView` будет в том, чтобы показывать несколько полей активной записи в одной и той же колонке таблицы.

Вот набросок желаемого `GridView` для нашей модели клиента:

Name	Birth Date	Addresses	Emails	Phones	
John Doe	1973.12.10	some first address some second address etc	a@sbdy.dom b@sbdy.dom etc	666-66-66	View Edit Delete
...	...	...	...	+79133334343 +43432224410	View Edit Delete
...	...	...	...	...	View Edit Delete

Создание активных записей телефонов, адресов и адресов электронной почты

Нам нужно подготовить таблицы и определения активных записей для всех наших подчинённых моделей: `Address`, `Email` и `Phone`.

Давайте, используя следующий сценарий миграции, создадим таблицу для `AddressRecord`:

```
$this->createTable(
    'address',
    [
        'id' => 'pk',
        'purpose' => 'string',
        'country' => 'string',
        'state' => 'string',
        'city' => 'string',
        'street' => 'string',
        'building' => 'string',
        'apartment' => 'string',
        'receiver_name' => 'string',
        'postal_code' => 'string',
        'customer_id' => 'int not null'
    ]
);
```

```
$this->addForeignKey('customer_address', 'address',
   'customer_id', 'customer', 'id');
```

Для простоты все элементы почтового адреса будут просто строками. Нам нужно явно создать внешний ключ по причинам, о которых мы скоро скажем.

Функция `\yii\db\Migration.addForeignKey()` достаточно ужасна, учитывая пять её позиционных аргументов, и, возможно, единственный вариант для того, чтобы её запомнить, – это IDE с возможность перехода к определению функции. Один из способов запомнить то, как правильно оформлять её вызов, – это использовать следующее предложение, в котором используются все аргументы в корректном порядке и с более-менее запоминающимися названиями: «добавить внешний ключ с *названием* в *таблицу* на *колонку*, ссылающийся в *таблицу* на *колонку*» («add foreign key with *name* on *table column* referencing *table column*»).

После того как эта миграция применена, мы открываем Gii и генерируем модель `AddressRecord`, используя следующие настройки в генераторе моделей:

- **Table Name**: `address`;
- **Model Class**: `AddressRecord`;
- **Namespace**: `app\models\customer`;
- **Generate Relations**: галочка должна быть установлена.

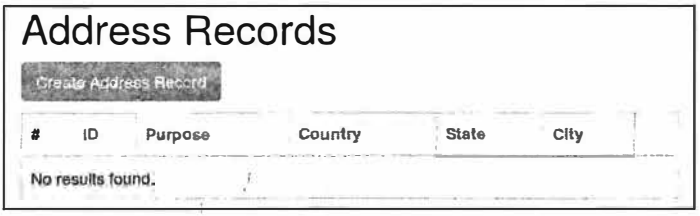
Класс `ActiveRecord` в Yii имеет некоторые особенные средства для использования связей между таблицами по внешним ключам. Галочка **Generate Relations** включает генерацию этих средств в нашем классе `AddressRecord`.

Затем откройте генератор CRUD и создайте CRUD для класса `AddressRecord`, используя следующие настройки:

- **Model Class**: `app\models\customer\AddressRecord`;
- **Search Model Class**: оставьте это поле пустым;
- **Controller Class**: `app\controllers\AddressesController`.

Мы убрали модель поиска, потому что её назначение в автогенерированном CRUD – предоставлять фильтрацию в виджете `GridView`, а нам это для модели `AddressRecord` не нужно. Нам это не нужно, потому что мы вообще не намерены смотреть на полный список моделей `AddressRecord`, без контекста определённого клиента.

Маршрут `/addresses` теперь должен нам показывать следующее:



Затем мы создаём таблицу email следующей миграцией:

```
$this->createTable(
   'email',
   [
      'id' => 'pk',
      'purpose' => 'string',
      'address' => 'string',
      'customer_id' => 'int not null'
   ]
);
$this->addForeignKey('customer_email', 'email',
   'customer_id', 'customer', 'id');
```

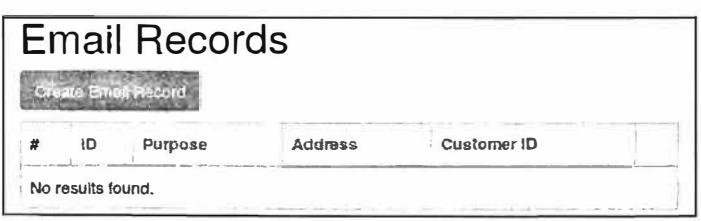
Создайте класс EmailRecord тем же самым генератором моделей и со следующими настройками:

- **Table Name**: email;
- **Model Class**: EmailRecord;
- **Namespace**: app\models\customer;
- **Generate Relations**: галочка должна быть установлена.

Затем создайте CRUD со следующими настройками:

- **Model Class**: app\models\customer\EmailRecord;
- **Search Model Class**: оставьте это поле пустым;
- **Controller Class**: app\controllers\EmailsController.

Маршрут /emails должен показать вам следующее:

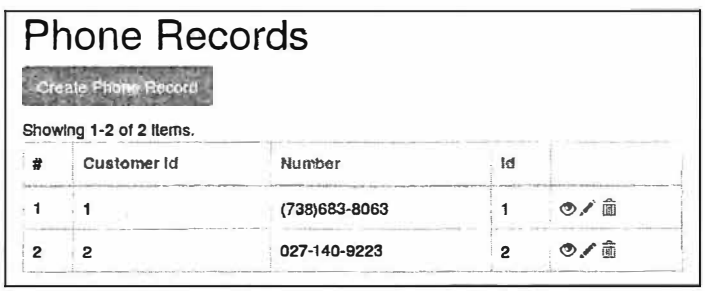


У нас уже есть модель PhoneRecord.

Создайте CRUD со следующими настройками:

- **Model Class**: app\models\customer\PhoneRecord;
- **Search Model Class**: оставьте поле пустым;
- **Controller Class**: app\controllers\PhonesController.

Наш свежесозданный CRUD по маршруту /phones должен выглядеть так, как на следующем снимке экрана (при условии, что в БД уже есть две записи о телефонах):



Во всех трёх случаях мы сталкиваемся с виджетом GridView, с которым мы вскоре будем иметь возможность поиграть. Но сначала мы займёмся небольшой уборкой.

Создание общего базового контроллера для подчинённых моделей

Сейчас у нас есть три абсолютно идентичных контроллера, различающихся только именами использующихся классов активных записей. Оставлять такое дублирование кода – безвкусица, поэтому давайте посмотрим, как мы можем их объединить.

Вначале возьмите один из файлов – AddressesController.php, EmailsController.php или PhonesController.php – и скопируйте его в папку @app/utilities, переименовав в SubmodelController.php. Замените объявление пространства имён внутри этого файла на app\utilities и имя класса на SubmodelController. Теперь мы готовы обобщить контроллеры для подчинённых моделей.

Нам нужно следующее свойство класса, для того чтобы хранить кусочек конфигурации, который будет меняться:

```
/**@var string Название класса, которым мы манипулируем */
public $recordClass;
```

Это причина, по которой мы обобщаем определение контроллера. Контроллеры, созданные генератором CRUD, почти идентичны, за исключением имени класса активной записи, которой мы управляем. Мы не добавляем никакого дополнительного поведения к этим контроллерам, так как поведение по умолчанию нас полностью устраивает, так что мы просто передадим имя класса в качестве значения настройки.

Далее идём в метод `actionIndex()` и полностью его удаляем. Мы никогда не будем смотреть на список подчинённых моделей отдельно от корневой модели.

Затем идём в метод `actionCreate()`, его первая строчка должна выглядеть следующим образом:

```
$model = new AddressRecord;
```

Замените эту строчку такими строчками:

```
/** @var ActiveRecord $model */
$model = new $this->recordClass;
```

Первая строчка – подсказка как для IDE, если таковая используется, так и для людей, которые будут читать ваш код, о том, что вы создаёте экземпляр, по крайней мере, класса `ActiveRecord`.

На второй строчке мы видим применение по-настоящему высокоуровневой возможности PHP, когда мы можем создать класс по его имени, сохранённому в строковой переменной (строковые литералы, впрочем, не сработают).

Продолжая работать с методом `actionCreate()`, взгляните на следующее условное выражение:

```
if ($model->load(Yii::$app->request->post()) &&
   $model->save()) {
   return $this->redirect(['view', 'id' => $model->id]);
} else {
   return $this->render('create', [
      'model' => $model,
   ]);
}
```

Улучшите читаемость этого условного выражения, заменив его на следующую команду с защитным предложением:

```
if ($model->load(Yii::$app->request->post())
   && $model->save())
   return $this->redirect(['view', 'id' => $model->id]);

return $this->render('create', compact('model'));
```

Проделайте то же самое упрощение в методе actionUpdate():

```
if ($model->load(Yii::$app->request->post()) &&
   $model->save())
   return $this->redirect(['view', 'id' => $model->id]);

return $this->render('update', compact('model'));
```

Обратите внимание на выделенную часть. Это единственная строчка, которая отличается от того же блока кода в методе actionCreate(). Но, для того чтобы избавиться от этого дублирования, нам нужно будет пересмотреть наш подход к концепциям «создания» и «обновления» моделей, так что мы просто оставим всё как есть, в особенности учитывая то, что это не так важно в данной конкретной главе.

Затем переходим к методу findModel(). Взгляните на следующий вызов, который находится внутри условного выражения:

```
$model = CustomerRecord::findOne($id)
```

Замените этот вызов следующим кодом (в том же самом месте):

```
$model = call_user_func([$this->recordClass, 'findOne'], $id)
```

Это очевидное обобщение, учитывая то, что функция call_user_func способна вызывать статические методы классов.

Затем давайте улучшим читаемость метода findModel() в целом. У вас должен быть такой код в начале:

```
if (($model = call_user_func([$this->recordClass,
   'findOne'], $id)) !== null) {
   return $model;
} else {
   throw new NotFoundHttpException
      ('The requested page does not exist.');
}
```

И в конце вы должны получить такой код:

```
$model = call_user_func([$this->recordClass, 'findOne'],
   $id);
if (!$model)
   throw new NotFoundHttpException
      ('The requested page does not exist.');

return $model;
```

Совершенно излишне проверять, является ли результатом запроса в точности значение null. Обычно нам не интересно, является ли он пустым массивом, пустой строкой, нулём, булевой ложью или null, до тех пор, пока это не экземпляр ActiveRecord, который нам нужен. Не говоря уж о том, что ActiveRecord::findOne() по определению не может вернуть никакое другое ложное значение.

Теперь у нас есть обобщённый базовый класс контроллеров, который делает то же самое, что и контроллер, сгененированный автоматически при помощи Gii. За исключением безжалостно удалённого метода actionIndex(). Используя этот контроллер, мы можем сократить класс AddressesController до следующего кода:

```
class AddressesController extends SubmodelController
{
    public $recordClass = 'app\models\customer\AddressRecord';
}
```

Класс EmailsController сокращается до следующего кода:

```
class EmailsController extends SubmodelController
{
    public $recordClass = 'app\models\customer\EmailRecord';
}
```

И класс PhonesController – до следующего кода:

```
class PhonesController extends SubmodelController
{
    public $recordClass = 'app\models\customer\PhoneRecord';
}
```

Обратите внимание на то, что в мире PHP, покрытого пространствами имён, нам нужно использовать полностью определённые имена для классов ActiveRecord. Не забывайте, что вам нужно использовать предложение use app\utilities\SubmodelController, если вы будете копировать и вставлять вышеуказанный код без изменений.

Создание отношений между моделью клиента и подчинёнными моделями

Достаточно очевидно, что для отображения в таблице телефонов, почтовых адресов и адресов электронной почты нам нужно иметь возможность выбирать из базы данных все подчинённые записи для данной записи о клиенте.

Для решения этой задачи в Yii 2 есть очень удобная возможность в активных записях: вы можете иметь полностью виртуальные свойства только для чтения, которые будут возвращать активные записи, связанные внешними ключами в нижележащих таблицах с рассматриваемой записью.

Следующий метод добавляет связь с моделями Phone в модель CustomerRecord:

```
public function getPhones()
{
   return $this->hasMany(PhoneRecord::className(),
      ['customer_id' => 'id']);
}
```

Это – концепция «отношения» **(relation)** в Yii 2. Она даёт нам следующие возможности:

- вызов $customer->phones с экземпляром CustomerRecord в качестве $customer вернёт нам массив экземпляров PhoneRecord, у которых значение поля customer_id равно значению $customer->id;
- вызов $customer->getPhones() вернёт экземпляр \yii\db\ActiveQuery, который мы можем скормить конструктору какого-нибудь DataProvider, чтобы использовать тот в качестве источника данных для виджетов.

Всё это становится возможным благодаря магии переопределённого метода __get().

Нам также нужны «отношения» с почтовыми адресами и адресами электронной почты, что делается в точно такой же манере:

```
public function getAddresses()
{
   return $this->hasMany(AddressRecord::className(),
      ['customer_id' => 'id']);
}

public function getEmails()
{
   return $this->hasMany(EmailRecord::className(),
      ['customer_id' => 'id']);
}
```

Кроме hasMany(), класс ActiveRecord также содержит метод hasOne(), который принимает те же самые аргументы, но значение второго аргумента другое: в методе hasMany() мы «указываем» из связанной

таблицы на «эту» таблицу, но в методе `hasOne()` мы «указываем» из «этой» таблицы на связанную.

Заметьте, что технически нам не нужны заранее определённые внешние ключи в таблицах базы данных, чтобы эти отношения работали, так как мы их определяем вручную на уровне приложения. Определения внешних ключей были даны по двум причинам:

- это документация в коде: любой, читающий схему базы данных, будет в состоянии понять, что между таблицами есть логическая связь;
- Gii может автоматически генерировать методы отношений на основании внешних ключей, определённых в таблицах. Он создаёт методы отношений в обе стороны, то есть для класса `EmailRecord` он создаст также метод `getCustomer()`.

Мы можем вообще пропустить эту ручную генерацию методов отношений. Генератор моделей в Gii может перезаписывать уже существующие классы моделей, и вы даже можете перед этим посмотреть на различия, которые он внесёт. Поэтому при наличии всех этих внешних ключей в таблицах, связанных с таблицей `customer`, если мы запустим генератор моделей, для того чтобы «создать» модель `CustomerRecord` «заново», Gii регенерирует этот класс, добавив в него все необходимые методы отношений. Вы, однако, потеряете поведение, присоединённое в главе 10, и вам понадобится заново его добавлять.

Перед тем как мы действительно начнём работать с виджетом `GridView`, нам нужно решить, как мы будем вставлять записи в БД, чтобы посмотреть, что наш пользовательский интерфейс на самом деле работает.

У нас есть два варианта:

- вставить несколько вручную собранных записей в базу данных вручную, используя консольный интерфейс к MySQL. Это довольно просто, при условии что мы аккуратно убедимся, что идентификатор клиента корректно вставлен во все поля внешних ключей. Вместо консольного интерфейса мы можем использовать созданный нами CRUD. В данном случае удаление метода `actionIndex()` даже помешает;
- создать пользовательский интерфейс, который позволяет нам создавать записи о клиентах сразу вместе со всеми связанными записями. Это требует много работы с виджетом `ActiveForm`, и, как уже было кратко упомянуто, мы решили выделить это в приложение 2. Если записи в БД, созданные вручную, – это не ваш

метод, то вам настоятельно рекомендуется прямо сейчас пролистать до приложения и реализовать описанный там интерфейс.

Всё последующее обсуждение подразумевает, что у вас есть как минимум одна запись в таблице `customer` с несколькими связанными записями в таблицах `address`, `phone` и `email`. Это прямое следствие работы, полагаясь только на тестирование вручную.

ВОЗМОЖНОСТЬ: виджеты

Пока что мы видели два стиля использования виджетов Yii 2. Первый выглядел так:

```
$form = ActiveForm::begin();
// … много другого кода ...
ActiveForm::end();
```

Когда Gii создаёт интерфейсы создания и обновления записей, он использует этот формат в файлах представлений `_form.php`.

Второй вариант выглядел так:

```
GridView::widget([
// … много настроек …
]);
```

Когда Gii создаёт пользовательский интерфейс списка записей, он использует этот формат в файлах представлений `index.php`.

Теперь мы можем поговорить о понятии «виджета» (**widget**) в идеологии Yii 2.

Очень сильно упрощая, это что-то, что позволяет нам рендерить файл представления внутри другого файла представления, который уже находится в процессе отрисовки. В отличие от вызова `$this->render($viewFile)` внутри обычных файлов представлений, виджеты позволяют нам применять любое количество логики для отрисовки этого подчинённого файла представления, что помогает упростить представления верхнего уровня и даёт возможность повторно использовать одинаковые части контента в нескольких местах веб-приложения. А это определённо полезно.

Конечно же, так как файлы представлений в Yii 2 – это просто обычные сценарии PHP, теоретически вы можете иметь в них любое количество логики, и таким образом отпадает необходимость в абстракции «виджетов». Однако если вы будете так использовать ваш слой представления, то вы совершенно точно заслуживаете последствий, которые вас неотвратимо настигнут.

Более конкретно, виджет в Yii 2 – это подкласс `\yii\base\Widget`, который имеет несколько вспомогательных методов для использования данной концепции.

Самое важное, что в нём есть, – это метод `render()`, позволяющий ему отрисовывать выходные данные, используя компонент `View` приложения.

Предполагается, что основная логика любого виджета скрыта внутри методов `init()` и `run()`, которые любой подкласс `\yii\base\Widget` должен переопределить.

Когда вы вызываете `ИмяКлассаВиджета::widget($configuration)`, он вызывает свои `init()` и `run()` по порядку. В этом случае вся отрисовка полностью скрыта в виджете, и если вы хотите, чтобы виджет что-то вывел, вы должны передать это через `$configuration`.

Когда вы вызываете `ИмяКлассаВиджета::begin($configuration)`, он вызывает свой метод `init()` и буферизует весь вывод, до тех пор, пока вы не вызовете `ИмяКлассаВиджета::end()`. В тот момент он вызывает свой метод `run()`, собирает всё, что было буферизовано, присоединяет к нему результат вызова метода `run()`, а затем возвращает получившуюся строку. Таким образом вы можете делать пригодные для повторного использования виджеты, которые будут декорировать то, что вы будете выводить между вызовами `begin()` и `end()`.

Обратите внимание на то, что и `widget()`, и `end()` так же, как это делает метод `Controller.render()`, возвращают результат отрисовки в виде строки. Ничто не будет на самом деле напечатано на странице, если вы явно это не напечатаете.

Базовый класс виджетов включает в себя специальные средства, для того чтобы мы могли вкладывать вызовы виджетов друг в друга. Таким образом мы можем использовать несколько вызовов `begin()`, чтобы начать несколько виджетов подряд, и затем позднее использовать `end()`, чтобы один за другим их закончить, и всё это будет «просто работать».

Вся настройка, которую мы делаем для виджетов, передавая массивы параметров настройки в вызовы `widget()` и `begin()`, возможна благодаря всемогущему методу `Yii::createObject()`. Все настройки в этих массивах соответствуют публичным свойствам класса виджета. Это очень помогает читать код других разработчиков, даже если вы не умеете обращаться с конфигурируемым виджетом: если вы встретите какую-нибудь настройку, которую не понимаете, вы можете просто перейти к определению класса виджета и найти публичное свойство с тем же именем. В случае виджетов, встроенных в Yii 2, обычно это

сильно помогает, потому что исходный код фреймворка имеет превосходную самодокументацию.

Создание страницы списка клиентов

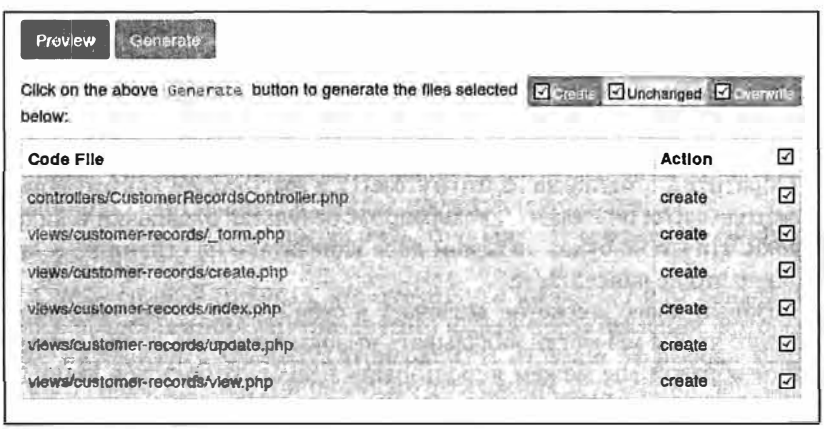
Теперь мы готовы создать интерфейс, о котором говорили в начале этой главы.

Мы оставим в стороне интерфейс, который мы создали в главе 2, и на этот раз создадим CRUD напрямую для модели `CustomerRecord`.

Используйте следующие настройки для генератора CRUD:

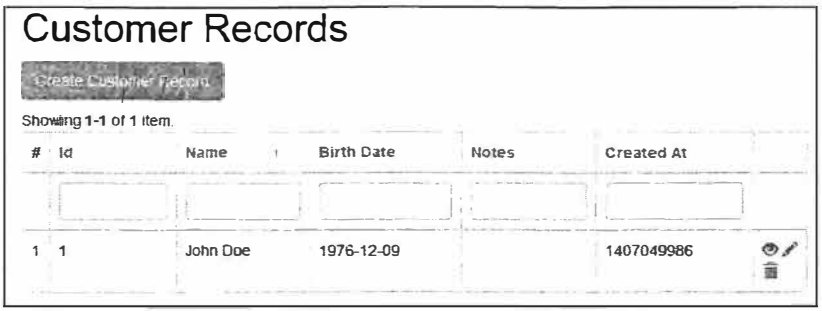
- **Model Class**: `app\models\customer\CustomerRecord`;
- **Search Model Class**: `app\models\customer\CustomerRecordSearch`;
- **Controller Class**: `app\controllers\CustomerRecordsController`.

Обратите внимание на список файлов, которые он для нас будет создавать:



У контроллера под названием `CustomerRecordsController` имеется папка `customer-records` внутри папки `views`. В ней этот контроллер будет искать свои файлы представлений, если ссылки на них будут полностью относительными.

Нажмите на кнопку **Generate** и затем откройте маршрут `/customer-records`. Вот что он покажет при наличии одной записи о пользователе в базе данных:



Эта довольно тривиальная реализация – то, с чем мы будем работать до конца главы.

Создание базового GridView для клиентов

Давайте определимся, какие колонки мы хотим видеть в таблице со списком клиентов:

- ID;
- имя;
- дата рождения;
- телефоны;
- адреса электронной почты;
- почтовые адреса.

Этот список отсортирован в порядке увеличения сложности. Давайте посмотрим на текущий автоматически сгенерированный код для GridView внутри файла views/customer-records/index.php:

```
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        'id',
        'name',
        'birth_date',
        'notes:ntext',
        'created_at',
        // 'created_by',
```

```
        // 'updated_at',
        // 'updated_by',

        ['class' => 'yii\grid\ActionColumn'],
    ],
]); ?>
```

Отметим некоторые вещи, которые Gii для нас сделал.

Во-первых, в качестве первой колонки мы имеем `\yii\grid\SerialColumn`. Это всего лишь вспомогательная колонка, которая даёт нам номера строк. В таблицах, разделённых на страницы, это может быть достаточно полезно, так как номер строки отображается не относительно текущей страницы, а относительно всего набора данных из `dataProvider`.

Во-вторых, у нас среди колонок перечислены все поля из модели `CustomerRecord` , но все, кроме первых пяти, закомментированы, чтобы не загромождать отображаемую таблицу.

Затем отметьте специальный синтаксис для поля `notes`: `notes:ntext`. Мы обсудим этот синтаксис позже.

Также Gii уже сделал для нас колонку типа `ActionColumn` с настройками по умолчанию. В этой колонке будут кнопки для управления записями таблицы.

И наконец, у нас есть настройка `filterModel`, в которую передаётся экземпляр `\app\models\customer\CustomerRecordSearch`. Эта настройка – то, ради чего Gii спрашивает у нас имя класса в поле `Search Model Class`. Она отвечает за то, чтобы у таблицы была дополнительная строчка, которая будет предоставлять нам фильтрацию по значениям полей в таблице. Позже в этой главе мы реализуем эту функциональность для нескольких случаев. Если мы ничего не передадим в `filterModel`, то фильтра у таблицы не будет.

Теперь давайте сделаем что-то простенькое, например вставим те фрагменты информации, которые нам нужны, в то небольшое пространство, которое у нас есть.

Изменение формата содержимого колонки

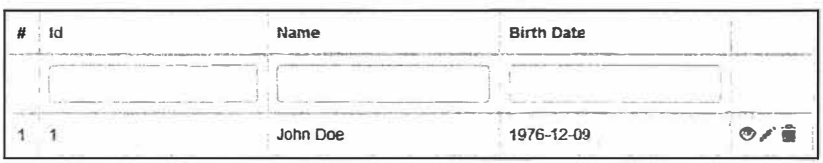
Довольно просто очистить наш список клиентов, оставив там только простые поля: `id`, `name` и `birth_date`. Вот соответствующие определения колонок для этого:

```
'columns' => [
    ['class' => 'yii\grid\SerialColumn'],
```

```
        'id',
        'name',
        'birth_date',
        ['class' => 'yii\grid\ActionColumn'],
    ],
```

Во всех последующих фрагментах кода ради краткости мы опустим строчки для `SerialColumn` и `ActionColumn`. Мы не будем их трогать вообще в этой главе. Однако это не значит, что мы удалим их из кода, так как эти колонки всё равно должны остаться в таблице.

На следующем снимке экрана представлена наша отправная точка, как она отображается на странице. Обратите внимание на то, что у колонок уже есть автоматически сгенерированные заголовки с правильным регистром букв.



Колонка `id` выглядит точно так же, как и колонка с порядковым номером; давайте перенесём её на правый край таблицы. Также даты в формате ISO выглядят очень строго. Мы можем их сделать более привлекательно выглядящими, используя следующие настройки:

```
'columns' => [
    'name',
    [
        'attribute' => 'birth_date',
        'format' => ['date', 'jS M, Y'],
    ],
    'id',
],
```

Класс `DataColumn` содержит свойство под названием `format`, используя которое, мы можем объявить, как надо отображать данные в ячейках таблицы. В случае формата `date` мы можем указать в качестве дополнительного параметра саму строку формата даты. Чтобы это сделать, мы предоставляем массив настроек для свойства `format`. Многие другие форматы не имеют дополнительных параметров, например ранее

упомянутый формат ntext для поля notes. В таком случае мы можем предоставить формат в виде строки или одноэлементного массива.

Давайте углубимся в тему форматирования данных для вывода и рассмотрим компонент приложения, специально созданный для решения этой задачи.

ВОЗМОЖНОСТЬ: компонент форматирования

К приложению Yii всегда присоединён один специальный компонент под названием format, принадлежащий по умолчанию классу \yii\i18n\Formatter. Этот компонент инкапсулирует возможность форматирования входных данных согласно указанным правилам. Основное предназначение форматировщика – предоставлять следующий метод:

```
public function format($value, $format)
```

> Несколько других компонентов зависят от этого метода форматировщика. Так что если вы будете переопределять этот компонент, убедитесь, что он всё ещё имеет метод format(), иначе вы сломаете значительную часть Yii 2.

В качестве аргумента $format вы можете передать строку или массив. В случае массива его первый элемент должен быть строкой. В обоих случаях эта строка является названием формата, согласно которому должно быть преобразовано $value. Чтобы на самом деле совершить преобразование, компонент попробует найти у себя метод, названный так же, как и указанный формат, но с префиксом as.

Таким образом, следующий код, в конце концов, совершит вызов \yii\i18n\Formatter.asRaw($string):

```
Yii::$app->formatter->format($string, 'raw');
```

Следующий код, в конце концов, совершит вызов \yii\i18n\Formatter.asDatetime($datetime, 'Y-m-d H:i:s'):

```
Yii::$app->formatter->format(
   $datetime,
   ['datetime', 'Y-m-d H:i:s']
);
```

Эта возможность широко используется в тех местах, где вы можете указать формат чего-то, предназначенного для отрисовки, например в объявлениях колонок GridView.

На момент написания этой главы (если точнее, перевода этой главы – *прим. перев.*) в стандартном форматировщике были определены следующие форматы:

Название	Смысл
raw	Никакой обработки не будет применено. Заметьте, что этот формат довольно полезен в случае DataColumn, так как форматом по умолчанию является text
text	Значение будет обработано методом Html::encode(), и в результате все HTML-теги станут обычным видимым текстом
ntext	То же самое, что и text, но результат будет дополнительно обработан методом nl2br(), встроенной в PHP-функцией, которая заменяет переносы строк на HTML-элементы br. Скажем, если какой-то текст был введён с помощью поля ввода textarea, то безопасно вывести его на HTML-страницу с сохранением переносов строк можно именно при помощи этого формата
paragraphs	То же самое, что text, но блоки текста, разделённые двумя или более пустыми строками подряд, будут обёрнуты HTML-элементом p
html	Принимает дополнительный параметр $config. Значение будет обработано вызовом HtmlPurifier::process($value, $config). Таким образом, вместо превращения разметки HTML в видимый текст мы вообще полностью её удаляем
email	Значение будет обработано вызовом Html::mailto(Html::encode ($value), $value), таким образом, вы получите HTML-код ссылки mailto: с самим адресом электронной почты в качестве видимого текста
image	Значение будет использовано в качестве атрибута src элемента img, HTML-код которого будет возвращён. Этот формат полезен для превращения ссылок на изображения в настоящие отрисованные изображения на конечной HTML-странице
url	Переданное значение будет использовано как атрибут href для объявления элемента a, HTML-код которого будет возвращён. Если нужно, будет вставлен пропущенный префикс http://. Этот формат полезен для превращения URL в ссылки, которые на самом деле можно нажать
boolean	Если переданное значение в конечном счёте равно false, этот формат вернёт строку No, локализованную согласно настройкам приложения. Иначе вернёт локализованную строку Yes. Вы можете изменить свойство booleanFormat класса Formatter на ваше собственное определение ["No", "Yes"]. Оно должно продолжать быть двухэлементным массивом, как показано
date	Конвертирует данное значение согласно формату, переданному вторым аргументом. Если формат явно не задан, будет использовано свойство dateFormat класса Formatter. Этот метод принимает строки, которые понимает strtotime(), целочисленные метки времени Unix и экземпляры DateTime
time	То же самое, что date, но по умолчанию будет использовано свойство timeFormat класса Formatter
datetime	То же самое, что date, но по умолчанию будет использовано свойство datetimeFormat класса Formatter
timestamp	То же самое, что date (как бы не казалось иначе), но всегда форматирует значение в виде метки времени Unix (целого числа, представляющего количество секунд с 1 января 1970 г.)

Название	Смысл
`relativeTime`	Переданное значение расценивается как определение даты/времени, и дополнительный аргумент также расценивается как определение даты/времени, с текущими датой/временем в качестве значения по умолчанию. Возвращает текстовое представление интервала времени между этими определениями. В качестве обоих аргументов можно передавать метки времени Unix, строки, распознаваемые функцией `strtotime()`, и экземпляры `DateTime`. Дополнительно вы можете в качестве первого и единственного аргумента передать экземпляр `DateInterval` или строку, не распознаваемую конструктором `DateTime`, но распознаваемую конструктором `DateInterval`. Результатом будет также текстовое представление переданного интервала времени. Посмотрите на определение метода `\yii\i18n\Formatter::asRelativeTime()`, чтобы увидеть, какие именно локализованные строки будет использовать этот формат
`integer`	Возвращает целочисленное представление данного значения. Обратите внимание, что этот формат работает не так, как простое преобразование типа в тип `int`. Проверьте определение метода `\yii\i18n\Formatter::asInteger`, чтобы точно понять, как он осуществляет преобразование
`decimal`	Возвращает строковое представление данного значения в виде действительного числа с плавающей точкой. В качестве дополнительного аргумента указывается количество отображаемых точек после запятой, по умолчанию их 2. Символ разделителя целой и дробной частей берётся из свойства `decimalSeparator` класса `Formatter`, а символ разделителя тысяч – из свойства `thousandSeparator`. По умолчанию они зависят от текущих настроек локали
`percent`	Переданное значение должно быть дробным числом, так как оно будет умножено на `100` и к нему будет приписан символ «`%`». Таким образом, значение `0.75` будет превращено в строку «`75%`». Дополнительный аргумент настраивает количество цифр после запятой в полученном количестве процентов, которое нужно отображать (по умолчанию равно нулю)
`scientific`	Форматирует переданное число в научном формате, например число `100000` будет отображено как `1.00000E+5`. Можете посмотреть документацию к ключу `E` для функции `sprintf` здесь: http://php.net/manual/ru/function.sprintf.php, чтобы узнать подробности представления чисел в научном формате в PHP. Так же, как и формат `decimal`, принимает в качестве дополнительного аргумента количество точек после запятой, которые нужно отображать
`currency`	Форматирует переданное число в виде суммы денег. В качестве дополнительного аргумента передаётся трёхбуквенный код валюты, согласно стандарту ISO 4217. Если его нет, то будет использовано значение свойства `currencyCode` класса `Formatter`, которое устанавливается в зависимости от текущей локали. Данный формат очень сильно зависит от расширения PHP intl. Если оно установлено, то форматирование переданного значения будет производиться методом `NumberFormatter::formatCurrency()` (см. http://php.net/manual/en/numberformatter.formatcurrency.php). Если нет, то тогда код валюты будет приписан к числу в качестве префикса, а само число будет преобразовано форматом `decimal` с двумя точками после запятой

Название	Смысл
`spellout`	Превращает переданное число в его текстовое представление, как оно произносится, используя из PHP класс `NumberFormatter` в «стиле» `NumberFormatter::SPELLOUT`. Если не установлено расширение PHP intl, бросает исключение `InvalidConfigException`
`ordinal`	Превращает переданное число в порядковое числительное, используя из PHP класс `NumberFormatter` в «стиле» `NumberFormatter::ORDINAL`. Если не установлено расширение PHP intl, бросает исключение `InvalidConfigException`
`shortsize`	Расценивает переданное значение как количество байт и очень старается описать это число в больших размерах, например кило- или мегабайтах. Дополнительный аргумент – количество цифр после запятой. Свойство `sizeFormatBase` класса `Formatter` условно определяет число байт в килобайте. Если значением этого свойства является число 1024 (чем оно является по умолчанию), то данный формат будет использовать двоичные единицы измерения (KiB, MiB и т. д.), в противном случае будут использованы обычные десятичные префиксы (KB, MB и т. д.)
`size`	То же самое, что и `shortsize`, но вместо сокращённых обозначений единиц измерения будут использованы полные локализованные названия этих единиц измерения (кибибайты, мебибайты, килобайты, мегабайты и т. д.)

Множество из этих методов (например, все методы, относящиеся к форматированию чисел) гораздо эффективнее подстраиваются под текущую локаль приложения, если установлено расширение PHP intl (см. http://php.net/manual/en/book.intl.php), а некоторые вообще не работают без этого расширения.

Обратите внимание на то, что если вы передадите в любой метод, форматирующий числа тем или иным образом, значение, на которое функция `is_numeric()` возвращает `false`, то вы получите `InvalidParamException`, и если это произошло внутри какой-нибудь ячейки `GridView`, то это выльется в полностью разрушенный пользовательский интерфейс!

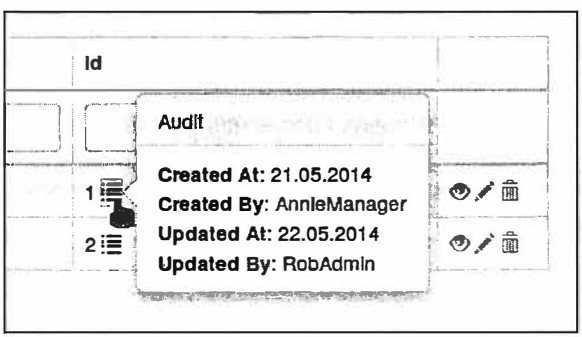
Мы упомянули не все дополнительные параметры к соответствующим методам компонента форматирования, а только самые важные для их логики. Мелкие подробности вы можете прочитать в официальной документации здесь: http://www.yiiframework.com/doc-2.0/guide-output-formatter.html.

По умолчанию список всех форматов, доступных для вас, вы можете вывести из методов класса `\yii\i18n\Formatter`, чьи имена начинаются с префикса `as`. Например, формат `ntext` соответствует методу `asNtext($value)`, а `date` соответствует методу `asDate($value, $format)`.

Создание преднастроенной колонки GridView

Можно сделать небольшой трюк, для того чтобы увеличить полезность информации, представленной в колонке ID. Давайте вложим туда информацию об аудите клиента, которую готовили в главе 10, используя возможность Popover из библиотеки Twitter Bootstrap (прочитайте о ней на странице http://getbootstrap.com/javascript/#popovers). Однако это потребует очень большого количества кода, так что мы сразу создадим новый тип колонки в виде класса `app\utilities\AuditColumn`.

Представим очень просто выглядящее содержимое ячеек этой колонки. В них будет показан идентификатор записи о клиенте, и рядом с ним – специальная иконка. По щелчку на этой иконке появляется всплывающая панелька с информацией о том, кто и когда создал и последний раз изменял эту запись, как на следующем снимке экрана:



Мы будем перекрывать класс `DataColumn`. Давайте создадим класс **`app\`**`utilities\AuditColumn` в файле `AuditColumn.php` в подкаталоге `utilities`:

```
namespace app\utilities;

use yii\grid\DataColumn;

class AuditColumn extends DataColumn
{
   // будущий код здесь
}
```

В Yii 1.x и PHP 5.3 было достаточно сложно создать преднастроенный класс колонки, потому что это обычно требовало перекрытия метода `renderDataCellContent()`, причём с осторожным сохранением поддерживающего кода, включённого в него.

В Yii 2 класс колонки таблицы содержит очень мощное свойство под названием content, которое принимает произвольную анонимную функцию и передаёт ей следующие аргументы в указанном порядке:

1. *Объект данных*, возвращённый из DataProvider для этой строчки таблицы. В случае ActiveDataProvider это будет экземпляр ActiveRecord; в случае ArrayDataProvider это будет ассоциативный массив.
2. *Ключ*, который DataProvider ассоциировал с этой строкой. В случае ActiveDataProvider это обычно значение поля, объявленного первичным ключом в соответствующей таблице БД.
3. *Номер* этой строки, начиная с нуля, среди всех строк, возвращённых DataProvider.
4. Наконец, весь экземпляр класса самой колонки.

Для зрительного запоминания, вот как выглядит установка значения свойства content у колонки:

```
$column->content = function ($model, $key, $index, $column) {
    return "содержимое ячейки таблицы в виде строки";
}
```

Это свойство предоставляет нам возможность создавать очень чистые и простые решения.

Конечно же, если вы хотите настраивать содержимое ячеек заголовков или фильтров в DataColumn, вам не повезло, и придётся перекрывать методы \yii\grid\DataColumn::renderHeaderCellContent и \yii\grid\DataColumn::renderFilterCellContent соответственно. В них очень много сложной логики, так что это не будет лёгкой прогулкой. Впрочем, такие изменения нужны довольно редко.

Мы не будем показывать шаг за шагом полный процесс рефакторинга, который приведёт нас к результирующему коду, так как он занял довольно много времени. Покажем только конечный результат.

Метод init() в классе AuditColumn наиболее подходит для инициализации свойства content. Давайте напишем следующий простой код для этого:

```
public function init()
{
    $this->content = [$this, 'makeAuditCellContent'];
}
```

Метод makeAuditCellContent() будет местом, где мы будем создавать внутренности нашей ячейки. Этот метод должен иметь видимость

как минимум protected, иначе наша колонка не сможет его вызывать (вызов метода call_user_func будет осуществлять родительский класс DataColumn).

Этот метод выглядит следующим образом:

```
protected function makeAuditCellContent($model)
{
   $id = $this->formatID($model);
   $audit = $this->makeAuditPopoverElement
      ($this->getAuditValues($model));

   return sprintf('%s %s', $id, $audit);
}
```

Нам не нужно здесь ничего особенного. Ячейка будет состоять из приукрашенного идентификатора и символа для вызова всплывающей панельки.

Под «приукрашиванием» идентификатора мы имеем в виду превращение его в семизначный номер, заполненный нулями слева:

```
protected function formatID($model)
{
   return sprintf("%07d", $model->id);
}
```

Из документации о всплывающих панелях в Twitter Bootstrap (http://getbootstrap.com/javascript/#popovers) и после краткого осмотра встроенных в этот пакет иконок (http://getbootstrap.com/components/#glyphicons-glyphs) мы делаем вывод, что нам нужен следующий HTML-код для показа дополнительной информации:

```
<span class="audit-toggler glyphicon glyphicon-list"
     data-toggle="popover"
     data-html="true"
     data-title="Audit"
     data-content="...">
</span>
```

Это можно представить в виде следующей функции, создающей HTML для данного переключателя:

```
protected function makeAuditPopoverElement($values)
{
   return Html::tag(
      'span',
      '',
```

```
            [
                'class' => 'audit-toggler glyphicon
                    glyphicon-list',
                'data-toggle' => 'popover',
                'data-html' => 'true',
                'data-title' => 'Audit',
                'data-content' =>
                    $this->makePopoverContent($values)
            ]
        );
    }
```

Не забудьте вставить предложение `use yii\helpers\Html` в начало файла.

«Содержимое», которое метод `makePopoverContent()` обещает сделать, будет следующим:

- `Created At` (жирным шрифтом): дата создания записи в формате PHP `Y-m-d`;
- `Created By` (жирным шрифтом): имя пользователя, которого следует винить в создании записи;
- `Updated At` (жирным шрифтом): дата последнего обновления записи в формате PHP's `Y-m-d`;
- `Updated By` (жирным шрифтом): имя пользователя, виновного в последнем обновлении.

Здесь очень много повторов, поэтому давайте станем здесь функциональными и напишем такой код для создания содержимого всплывающей панели:

```
protected function makePopoverContent($values)
{
    $formatter = function ($pair) {
        return sprintf(
            "<div><strong>%s:</strong> %s</div>",
            $pair[0],
            $pair[1]
        );
    };

    $appender = function ($accumulator, $value) {
        return $accumulator . $value;
    };

    return array_reduce(array_map($formatter, $values), $appender, "");
}
```

Этот код, безусловно, намного более высокоуровневый, чем что бы то ни было, написанное в этой книге, но намерение должно быть очевидно: мы делаем каждую строчку вышеупомянутой информации функцией `$formatter`, а затем соединяем все строчки в одну вызовом `array_reduce`. Определение функции `$formatter` подразумевает, что `$values`, которые превращаются в текстовое содержимое панели, – это массив массивов. Внутренние массивы состоят из двух элементов, и поэтому мы зовём их `$pair`. Каждая такая «пара» состоит из метки (которую нужно вывести жирным шрифтом) и значения (которое нужно написать после двоеточия). Вот как мы создадим этот список значений:

```
protected function getAuditValues($model)
{
    return [
        [
            $model->getAttributeLabel('created_at'),
            date('d.m.Y', $model->created_at)
        ],
        [
            $model->getAttributeLabel('created_by'),
            UserRecord::findOne($model->created_by)->username
        ],
        [
            $model->getAttributeLabel('updated_at'),
            date('d.m.Y', $model->updated_at)
        ],
        [
            $model->getAttributeLabel('updated_by'),
            UserRecord::findOne($model->updated_by)->username
        ]
    ];
}
```

Это завершает создание HTML-содержимого для нашей новой колонки.

В документации для всплывающих панелей в Bootstrap сказано, что нам нужно вручную включить соответствующий плагин Javascript. Среди зависимостей нашего `\app\assets\ApplicationUiAssetBundle` у нас уже есть пакет материалов под названием `BootstrapAsset`, но он содержит только файлы CSS от Bootstrap. Для того чтобы включить часть Twitter Boostrap, содержащую Javascript, нам нужно объявить зависимость от другого пакета материалов под названием `yii\bootstrap\BootstrapPluginAsset`.

Давайте просто создадим маленький пакет материалов для нашей колонки. Создайте файл assets/AuditColumnAssetsBundle.php с таким определением класса:

```
namespace app\assets;

use yii\web\AssetBundle;

class AuditColumnAssetsBundle extends AssetBundle
{
   public $sourcePath = '@app/assets/audit-column';
   public $css = [
      'styles.css'
   ];
   public $js = [
      'scripts.js'
   ];
   public $depends = [
      'yii\bootstrap\BootstrapPluginAsset',
   ];
}
```

Внутри файла assets/audit-column/scripts.js мы напишем следующее:

```
$(".audit-toggler").popover();
```

Это включит наши всплывающие панели, так как их нужно включать автоматически.

Внутри файла assets/audit-column/styles.css мы напишем следующее:

```
.audit-toggler {
   cursor: pointer;
}
.audit-toggler:hover {
   outline: 1px solid cyan;
}
```

Это добавит немного стиля кнопкам вызова информационных панелей.

Давайте также объявим, что наш главный пакет материалов ApplicationUiAssetsBundle зависит от этого нового пакета:

```
class ApplicationUiAssetBundle extends AssetBundle
{
// … другие объявления...
```

```
    public $depends = [
        'yii\bootstrap\BootstrapAsset',
        'yii\web\YiiAsset',
        'app\assets\AuditColumnAssetsBundle',
    ];
}
```

После того как вы подготовили пакет материалов, зарегистрируйте его в конце метода `AuditColumn.init()`:

```
        app\assets\AuditColumnAssetsBundle::register
            ($this->grid->view);
```

Не забудьте заново вызвать минификатор материалов, если вы его настроили в главе 8.

На самом деле мы могли обойтись всего двумя командами в конце метода `init()` вместо регистрации целого пакета материалов:

```
$this->grid->view->registerJs(
    '$(".audit-toggler").popover();'
);
$this->grid->view->registerCss(
    '.audit-toggler { cursor: pointer; }
     .audit-toggler:hover { outline: 1px solid cyan; }'
);
```

Подобные строки могут довести любого сопровождающего до сумасшествия. Можете представить более пяти таких преднастроенных виджетов по всему приложению, которое имеет 50 разных файлов представлений и несчитанное количество действий контроллеров? Когда они начнут конфликтовать друг с другом, или, ещё хуже, другой код начнёт зависеть от их стилей или сценариев Javascript, единственный вариант, который вам предложат коллеги в какой-то момент, – это переписать пользовательский интерфейс с нуля.

Не рассовывайте, не глядя, вызовы `registerCss()` и `registerJs()` где попало. В настоящем сложном проекте всегда нужно следить за разделением ответственности, и все ваши материалы должны лежать отдельно.

Давайте также добавим заголовок по умолчанию в шапку этой колонки, вставив следующий код в начало метода `init()`:

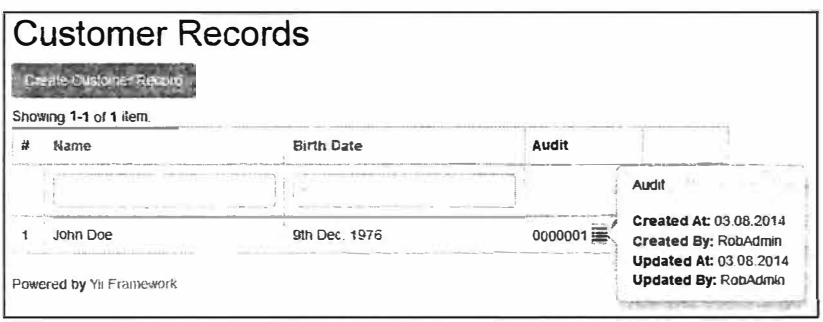
```
$this->label = $this->label ?: 'Audit';
```

Определив класс `AuditColumn`, мы можем сослаться на него среди колонок внутри файла представления `views/customer-records/index.php`:

```
<?= GridView::widget([
    'dataProvider' => $dataProvider,
```

```
        'filterModel' => $searchModel,
        'columns' => [
            ['class' => 'yii\grid\SerialColumn'],
            'first_name',
            'last_name',
            [
                'attribute' => 'birth_date',
                'format' => ['date', 'jS M, Y'],
            ],
            ['class' => 'app\utilities\AuditColumn'],
            ['class' => 'yii\grid\ActionColumn'],
        ],
    ]); ?>
```

Наконец, мы открываем маршрут `/customer-records` и видим следующее:



Обратите внимание на то, что эта колонка не предоставляет ни сортировки, ни фильтрации. Мы решим эту проблему позже.

Сжатие подчинённых моделей в одну колонку

Наиболее проблематичная часть информации о клиенте – это доступность нескольких почтовых адресов, телефонных номеров и адресов электронной почты. Один из способов упростить их отображение – это выводить, скажем, все номера телефонов в одной ячейке. То же самое можно применить к адресам электронной почты и почтовым адресам.

Мы решим эту задачу более простым способом, нежели создание преднастроенного класса колонки. Вот полный код решения, чтобы вам было, на что медитировать:

```
        [
            'label' => 'Addresses',
            'format' => 'paragraphs',
            'value' => function ($model) {
                $result = '';
                foreach ($model->addresses as $address) {
                    $result .= $address->fullAddress . "\n\n";
                }
                return $result;
            }
        ],
```

Мы используем формат paragraphs, описанный ранее. Он будет считать все блоки текста, разделённые двумя символами переноса строки, как элементы p, и в качестве значения для него мы производим именно то, что он ожидает: строку с адресами, разделёнными двумя символами переноса строки. Единственное, чего у нас ещё нет, – это свойства fullAddress модели AddressRecord. Следующий фрагмент кода – это определение метода AddressRecord.getFullAddress(), который нам нужен:

```
public function getFullAddress()
{
    return implode(', ',
        array_filter(
            $this->getAttributes(
                ['country', 'state', 'city', 'street', 'building',
'apartment']))));
}
```

Волшебный геттер в активных записях вызовет метод getFull·Address(), когда мы попробуем прочитать значение несуществующего свойства fullAddress.

Тем же способом мы можем сделать колонки **Emails** и **Phones**, за исключением того, что нам уже не понадобится виртуальное агрегирующее поле и мы можем напрямую использовать поля address и number соответственно.

ВОЗМОЖНОСТЬ: колонки GridView

Итак, GridView состоит из колонок. Эта концепция реализована в Yii 2 классом \yii\grid\Column. Каждая колонка состоит из четырёх элементов:

- строка шапки, в которой записывается заголовок колонки. Yii 2 корректно размещает шапку в HTML-элементе thead. Шапка отрисовывается методом \yii\grid\Column::renderHeaderCell();
- строка фильтра, в которой отображаются поля ввода, для того чтобы мы могли указать условие фильтрации для набора данных. Она находится во второй строке в том же самом элементе thead. Фильтр отрисовывается методом \yii\grid\Column::renderFilterCell();
- строки тела таблицы, содержащие значения, которые нужно показать в элементе tbody рассматриваемого GridView. Строки тела таблицы отрисовываются методом \yii\grid\Column::renderDataCell();
- строка подвала, которая корректно отрисовывается внутри элемента tfoot методом \yii\grid\Column::renderFooterCell().

Мы уже показали, что вы можете просто использовать свойство content класса \yii\grid\Column, для того чтобы совершать довольно тяжёлую обработку данных в ячейках таблицы. Однако если вы хотите так же сильно поменять отрисовку остальных трёх частей колонки, вам придётся переопределять эти методы (и конечно же, сперва прочитать их реализацию по умолчанию).

Когда GridView отрисовывает себя, он проходит по списку колонок в своём свойстве columns и инициализирует всё, что там упомянуто. Если колонка описана в виде массива, он отправляется напрямую в метод Yii::createObject(). Обратите внимание на то, что по умолчанию подразумевается класс \yii\grid\DataColumn, а не обычная \yii\grid\Column.

Если колонка описана строкой, эта строка расценивается одним из следующих вариантов, по порядку:

- атрибут:формат:метка;
- атрибут:формат;
- атрибут;
- InvalidConfigException.

Конечно же, InvalidConfigException – не значение, а то, что вы получите, если в отображаемой модели нет упомянутого атрибута.

Метка – это то, что будет вставлено в ячейку шапки (Yii 2 туда, возможно, вставит что-нибудь ещё, например стрелочку, показывающую направление сортировки). Формат – это идентификатор формата для компонента форматирования. Атрибут – это или название атрибута активной записи, или, в случае ассоциативных массивов, ключ внутри массива.

Класс DataColumn отличается от обычной колонки тем, что полагается на механику ActiveRecord. У него сразу определён метод renderFilterCell(), и он может полностью автоматически собрать ячейку шапки. Он даже вставляет ссылку, для того чтобы делать запросы на сортировку из ячейки шапки, если это нужно.

Вот список всех типов колонок, встроенных в Yii 2 (их не так много):

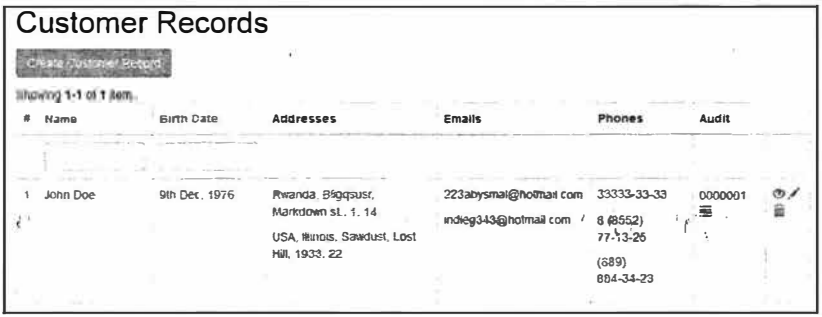
- тип \yii\grid\CheckboxColumn отрисовывает в качестве ячейки данных поле для установки галочки. Это полезно для богатого пользовательского интерфейса, усиленного JavaScript, потому что у вас есть возможность получить номера строк, в которых установлены галочки;
- тип \yii\grid\ActionColumn невероятно полезный. Он позволяет вам добавить в таблицу колонку, в которой будут находиться произвольные кнопки. По умолчанию отрисовываются три кнопки, для просмотра, редактирования и удаления соответствующей записи, которые вы уже видели. Свойство buttons этого класса определяет, какие кнопки могут быть отрисованы в этой колонке, а свойство template определяет, какие и в каком порядке кнопки на самом деле будут отрисованы;
- тип \yii\grid\DataColumn уже подробно обсуждался, и в следующих разделах мы изучим его возможности по сортировке и фильтрации. Он позволяет выводить значения атрибутов объектов из набора данных, которые возвращает DataProvider, переданный в GridView;
- тип \yii\grid\SerialColumn просто предоставляет вам колонку со счётчиком, который показывает номер строки в рамках всего набора данных, что полезно во многих областях. Как уже было сказано, этот счётчик никак не связан с первичным ключом отрисовываемой активной записи.

И вы всегда можете просто использовать базовый класс \yii\grid\Column с его универсальным свойством content.

Настоятельно рекомендуем прочитать документацию Yii 2 о колонках таблицы и узнать, как пользоваться колонкой с кнопками (это самая полезная колонка после DataColumn), или просто прочитать блоки самодокументации в исходном коде.

Реализация фильтрации в GridView

К этому моменту, после подготовки колонок **Emails** и **Phones**, у вас должна получиться такая таблица:



Можно увидеть, что у некоторых колонок есть фильтры в строке фильтра (**Name** и **Birth Date**), а у некоторых – нет. В данном разделе мы сделаем два изменения в этой таблице: вначале мы добавим фильтрацию по идентификатору записи в колонке **Audit** (что сделать просто), а затем добавим фильтрацию по названию страны в колонке **Addresses** (что намного сложнее).

Если вы попробуете возможность фильтрации таблицы, то увидите, что она работает следующим образом:

1. Вводим некоторый текст в поле ввода в строке фильтра.
2. Нажимаем *Enter* или теряем фокус ввода на этом поле ввода каким-нибудь другим образом.
3. Вся страница перезагружается.
4. На перезагруженной странице отображаются только строчки, соответствующие условию фильтрации.

Давайте посмотрим, на каком URL мы окажемся, если введём в таблице на предыдущем снимке экрана строку `Kasey` в фильтр колонки **Name**:

```
/customer-records
   ?CustomerRecordSearch[name]=Kasey
   &CustomerRecordSearch[birth_date]=
```

Вы, возможно, сразу подумали «эй, но это ведь просто GET-запрос, и вся эта фильтрация должна представлять из себя простую HTML-форму с небольшой добавкой Javascript». Так и есть. Обратите внимание на то, что мы передаём не `$_GET["CustomerRecord"]`, но `$_GET["CustomerRecordSearch"]`, не активную запись, которую мы сделали генератором моделей, но модель поиска, созданную генератором CRUD. Давайте теперь посмотрим на реализацию метода `CustomerRecordsController.actionIndex()`:

```
public function actionIndex()
{
    $searchModel = new CustomerRecordSearch;
    $dataProvider = $searchModel->search
        (Yii::$app->request->getQueryParams());

    return $this->render('index', [
        'dataProvider' => $dataProvider,
        'searchModel' => $searchModel,
    ]);
}
```

Действительно, действие контроллера index не просто показывает весь список моделей класса CustomerRecords, в него встроены возможности запросов к БД.

Идея проста. В файле представления для действия index мы отрисовываем GridView, который требует экземпляра класса DataProvider. Очевидно, что мы будем создавать этот экземпляр в действии контроллера, до того, как отрисовывать представление.

Созданием необходимого DataProvider будет заниматься специальная «модель поиска». DataProvider можно создать на основе некоторого экземпляра ActiveQuery. Таким образом, как сказано на выделенной строчке в вышеуказанном коде, мы просто берём все параметры GET, передаём их в модель поиска, и она создаст нужный экземпляр ActiveQuery, который загрузит в DataProvider и вернёт нам этот провайдер данных. Отсутствие параметров означает неограниченный DataProvider, который будет возвращать все записи из базы данных.

Мы передаём использованную модель поиска в файл представления (и далее в GridView внутри него), для того чтобы строка фильтра показала нам условия отбора, а также в целом в качестве условия показа строки фильтра. Если свойству filterModel экземпляра GridView не будет присвоена некоторая модель поиска, мы вообще не получим строку фильтра.

Давайте откроем метод CustomerRecordSearch.search():

```
public function search($params)
{
    $query = CustomerRecord::find();

    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);
    if (!($this->load($params) && $this->validate())) {
```

```
            return $dataProvider;
        }

        $query->andFilterWhere([
            'id' => $this->id,
            // … same lines for birth_date, created_at, created_by,
updated_at and updated_by ...
        ]);

        $query->andFilterWhere(['like', 'name', $this->name])
            ->andFilterWhere(['like', 'notes', $this->notes]);
        return $dataProvider;
    }
```

Это – каноническая подготовка фильтра для `GridView`.

Обратите внимание на выделенную строчку. Это важное защитное предложение, так как оно гласит, что если никаких данных не было передано для модели `CustomerRecordSearch` или они были неверно сформированы, мы возвращаем `DataProvider` с пустой `ActiveQuery`, ничего не фильтруя.

Следующие строчки несколько подлые, в том смысле, что мы в них модифицируем объект, который уже был передан в качестве входного аргумента другому объекту. В общем случае это довольно опасно.

Вначале мы добавляем условия отбора по нетекстовым полям, используя специальный метод `andFilterWhere()`. Затем мы добавляем условия отбора для нестрогого сравнения текстовых полей, используя тот же метод, но другой синтаксис входных аргументов.

В этом заключается настоящая сила метода `search()`. У нас есть вся функциональность класса `ActiveQuery` и целый класс соответствующей модели поиска для использования. Это позволяет нам создавать фильтры произвольной сложности. Более того, мы не ограничены здесь только виджетами `GridView`, так как мы можем передать полученный `DataProvider` куда угодно, где он принимается.

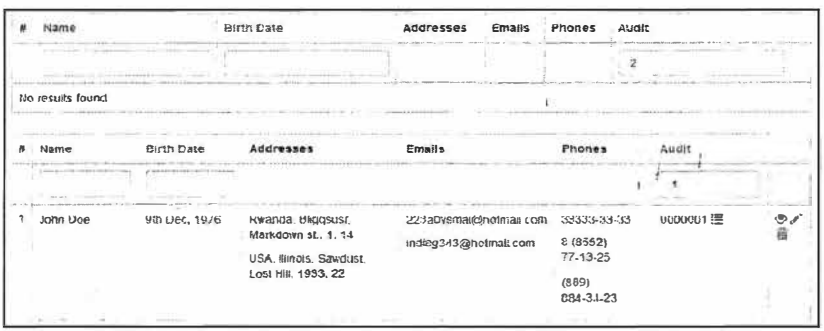
Чтобы включить фильтрацию по атрибуту `id` в нашей колонке Audit, нам нужно сделать смехотворно маленькое изменение в коде. Всё, что нужно, – это сказать классу этой колонки, к какому атрибуту она принадлежит, то есть заполнить его свойство `attribute`. Мы можем передать значение `id` при объявлении колонки в свойстве `columns` экземпляра `GridView` следующим образом:

```
        [
            'class' => 'app\utilities\AuditColumn',
            'attribute' => 'id'
        ],
```

Мы также можем быть более настойчивыми и инициализировать это свойство в начале метода `AuditColumn.init()`:

```
$this->attribute = $this->attribute ?: 'id';
```

Мы не будем такими настойчивыми, какими могли бы быть, и всё-таки позволим переопределять свойство `attribute` в объявлениях этой колонки. Теперь мы можем фильтровать таблицу по идентификатору записи о клиенте:



Сейчас давайте реализуем гораздо более интересное условие фильтрации: по названию страны в колонке **Addresses**.

Сначала нам нужно определить витруальный атрибут модели `CustomerRecordSearch`, то есть атрибут, который не существует среди полей таблицы клиентов, но который будет считаться таковым механикой фильтрации.

Чтобы определить виртуальный атрибут, нам нужно сделать две вещи. Во-первых, нам нужно объявить его как публичное свойство модели `CustomerRecordSearch`:

```
public $country;
```

Во-вторых, нам нужно упомянуть его среди каких-нибудь правил валидации, кроме правила `safe => false`. Суть в том, что вам нужно объявить этот атрибут как «безопасный» для использования в пользовательском интерфейсе. Давайте использовать правило «safe», которое уже есть в классе модели:

```
public function rules()
{
   return [
      // … неинтересные строчки пропущены ...
      [[ /* … остальные поля ... */, 'birth_date', 'notes',
```

```
            'country'], 'safe'],
        ];
    }
```

OK, так как нам нужно фильтровать по значению поля в связанной таблице, нам, очевидно, нужно присоединять некоторые таблицы в запросе SQL, лежащем в основе `ActiveQuery`. К счастью, у нас есть именно такой метод под названием `joinWith()`, который говорит `ActiveQuery` принимать во внимание указанное отношение, объявленное в базовой активной записи. Так как мы определили метод `getAddresses()` в классе `CustomerRecord`, наше отношение называется `addresses`, так что давайте вызовем `joinWith()` в конце метода `search()` (конечно же, до предложения `return`):

```
$query->joinWith('addresses');
```

И после этого, так как у нас теперь есть таблица `address`, присоединённая через `LEFT JOIN`, мы можем просто добавить такое же условие поиска, как и те, что мы уже видели для других текстовых атрибутов:

```
$query->andWhere(['like', 'address.country', $this->country]);
```

Если вас передёргивает от предложений `JOIN` в запросах SQL из-за возможного падения производительности, вы можете добавить простейшее защитное предложение вокруг всего этого условия:

```
if ($this->country)
{
    $query->joinWith('addresses');
    $query->andWhere(['like', 'address.country', $this->country]);
}
```

Но знайте, что вы всегда должны проверять такие изменения через профилировщик. Например, это конкретное изменение (присоединение таблиц, только когда название страны передаётся в запрос) на самом деле *увеличивало* количество запросов, которые ORM в Yii 2 делает к базе данных, когда авторы проверяли это в разделе «Database Queries» расширения отладки.

Нам также нужно поправить условие отбора по полю `id` в том же методе `CustomerRecordSearch.search()`, из-за того же самого `LEFT JOIN`. Нам нужно быть более специфичными, так чтобы механика `ActiveQuery` смогла понять, какой именно `id` вы хотите сравнить:

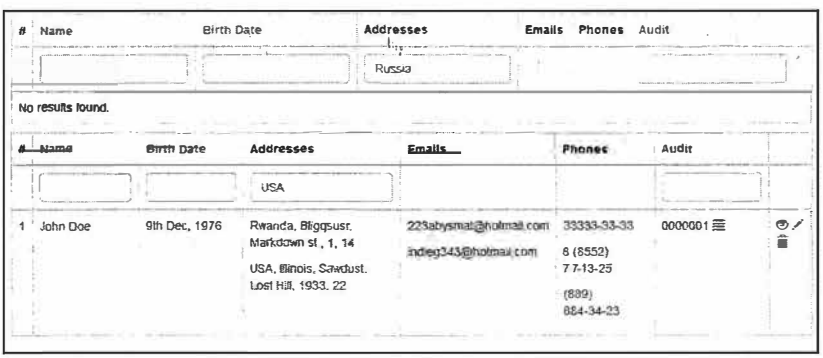
```
$query->andFilterWhere([
    'customer.id' => $this->id,
    'birth_date' => $this->birth_date,
    // … другие объявления ...
]);
```

Выделенную часть нужно вставить.

Улучшив модель `CustomerRecordSearch` таким образом, теперь мы можем объявить, что колонка **Addresses** на самом деле соответствует атрибуту `country`:

```
[
    'attribute' => 'country',
    'label' => 'Addresses',
    'format' => 'paragraphs',
    // … длинное несущественное определение значения ...
],
```

И мы получаем фильтрацию по названию страны в этой колонке:



Можно заметить, что `LEFT JOIN` при отношении один ко многим на самом деле создаёт частично дублирующиеся строки в результирующем наборе данных. Однако, как вы можете видеть в `GridView` после описанных изменений, ORM в Yii 2 автоматически заботится об этом, так что вы видите только различающиеся записи верхнего уровня в таблице и можете использовать все связанные записи как пожелаете.

Реализация сортировки в GridView

Настало время раскрыть тайну цветового кодирования заголовков в шапке `GridView`. Синие заголовки – ссылки. Щёлкая по ним, вы вызываете сортировку таблицы: страница перезагружается, и строчки меняют порядок, согласно выбранной колонке. Первый щелчок на заголовке осуществляет сортировку по этой колонке в восходящем порядке, второй щелчок сортирует в нисходящем порядке, и все последующие щелчки продолжают переключать сортировку таким же

образом. Щелчок на любом другом заголовке сортирует по соответствующей колонке, снова начиная с восходящего порядка.

Давайте снова посмотрим на URL, на котором мы оказываемся после щелчка по шапке колонки **Name**:

```
/customer-records/index?sort=name
```

Второй щелчок по тому же заголовку перемещает нас на следующий маршрут:

```
/customer-records/index?sort=-name
```

Если активен фильтр, мы получим что-то, напоминающее следующий код:

```
/customer-records/index
   ?CustomerRecordSearch[name]=
   &CustomerRecordSearch[birth_date]=
   &CustomerRecordSearch[country]=
   &CustomerRecordSearch[id]=1
   &sort=-name
```

Ясно видно, что сортировка и фильтрация являются интегральными частями пользовательского интерфейса `GridView` и могут работать совместно. Но мы уже видели полную реализацию метода `actionIndex()`, и там не было ничего, что принимало бы параметр `sort`, правильно? На самом деле `GridView`, с некоторой помощью от `DataProvider`, здесь мухлюет.

Деятельность по сортировке в Yii 2 инкапсулирована в классе `\yii\data\Sort`. Настроенный экземпляр этого класса может быть присоединён к экземпляру `DataProvider`, чтобы указать ему, как упорядочивать элементы результирующего набора данных. Если экземпляр `Sort` не был специально определён, `ActiveDataProvider` сам настроит сортировку по всем атрибутам, доступным в используемом экземпляре `ActiveRecord`. Все остальные провайдеры данных просто применяют тот порядок элементов, который использовала нижележащая база данных.

Хотя у объектов `Sort` есть достаточно много свойств, нас больше всего интересуют свойства `sortParam` и `attributes`. Если вы прочитаете исходный код класса `Sort`, то увидите, что значение по умолчанию для `sortParam` – это `sort`, в точности то, что `GridView` передаёт в `actionIndex()` в первом примере URL двумя абзацами выше. Свойство `attributes` хранит определения того, как `DataProvider` потенциально может сортировать набор данных.

Идея в следующем: мы настраиваем объект Sort, присоединённый к DataProvider, описывая ему различные варианты сортировки набора данных. После этого мы можем сказать этому объекту Sort, что мы хотим сортировать тем или иным образом; если этот образ был настроен ранее, то Sort проинструктирует своего владельца, экземпляр DataProvider, как ему следует переупорядочить набор данных. Это, конечно, вывернутый наизнанку поток управления, но именно это позволяет Yii 2 спрятать своё мошенничество внутри объекта Sort, где оно и должно находиться и не протекать в компонент DataProvider.

Согласно этим объяснениям процесса, вы должны осуществлять сортировку следующим образом:

```
$query = $this->сделатьНекоторыйЗапросЗаActiveRecords();
$sort = new Sort;
$sort->sortParam = 'sort'; // только для ясности, это всё равно значение по умолчанию
$sort->attributes = $this->сделатьОписанияВидовСортировки();
$dataProvider = new ActiveDataProvider(compact('query', 'sort'));
// … возможно, через некоторое время ...
$dataProvider->sort->params = ['sort' => 'один-из-видов-определённых-в-свойстве-attributes'];
$models = $dataProvider->getModels();
```

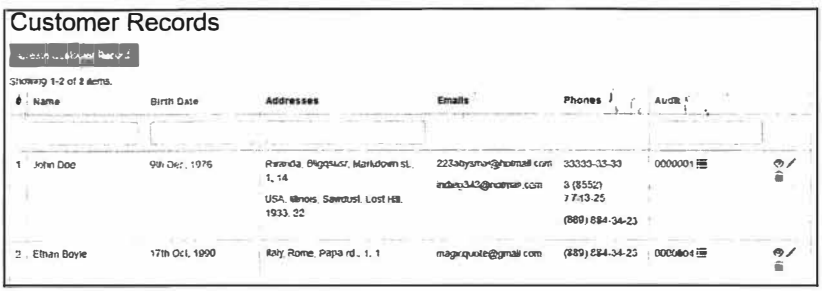
После этого $models является коллекцией объектов ActiveRecord, которую можно перебирать и которая отсортирована по атрибуту, переданному через $dataProvider->sort->params['sort'].

Мошенничество здесь в следующем: поскольку GridView не может залезть так глубоко внутрь DataProvider и для того, чтобы избавить разработчиков от необходимости возиться с этими низкоуровневыми деталями аж в действиях контроллеров, как мы делали с фильтрацией, Sort, когда приходит время выбирать порядок сортировки, просто идёт в компонент приложения Request и спрашивает там о параметрах запроса GET, которые передал клиент. Это, конечно, поразительное нарушение границ между абстракциями, но это то, что позволяет нам использовать удобный интерфейс для сортировки GridView. На момент написания этой главы вы можете посмотреть, как именно это сделано, в методе \yii\data\Sort::getAttributeOrders(), который в конце концов вызывается методом \yii\data\ActiveDataProvider::prepareModels(), который необходим для вызова метода \yii\data\BaseDataProvider::getModels().

Итак, давайте снова решим две задачи в качестве упражнения, одну лёгкую и одну не очень. В качестве лёгкой задачи мы реализуем сор-

тировку по нашей колонке **Audit**, а в качестве сложной задачи реализуем сортировку по колонкам **Addresses** и **Emails**.

Давайте посмотрим на то, что у нас уже есть...



Постойте, что? Заголовок колонки **Audit** – это синяя ссылка, и она уже сортирует таблицу!

Это действительно так. Когда мы хотели включить фильтрацию по атрибуту `id`, мы объявили настройку `attribute` для колонки **Audit**. Этого было достаточно, чтобы также включить сортировку по тому же самому атрибуту.

Как уже было сказано ранее, если особой конфигурации для настройки `sort` у `ActiveDataProvider` нет, он создаёт пустой экземпляр класса `Sort` и затем на самом деле идёт в модель, берёт оттуда все объявленные атрибуты, соответствующие колонкам таблицы, и вставляет их все одну за другой в свойство `attributes` объекта `Sort`.

У этого есть интересный побочный эффект: вы можете сортировать `GridView` по полям, которые не видны в виде колонок таблицы, но существуют в нижележащей таблице базы данных. Например, перейдите по следующему маршруту:

```
/customer-records/index?sort=-created_at
```

Это пересортирует таблицу. Строчки должны будут идти в порядке уменьшения значения ID в колонке **Audit**, так как поле `customer.id` автоматически увеличивается, и более поздние записи, очевидно, будут иметь более позднюю метку времени создания.

Это на самом деле было слишком легко. Однако при нашей подготовке мы можем также довольно легко реализовать сортировку по названию страны.

Исходя из всех объяснений до этого момента, становится довольно ясно, что нам нужно настроить провайдер данных, для того

чтобы включить новые возможности сортировки. Поэтому наилучшим местом для внесения наших изменений снова становится метод `CustomerRecordSearch.search()`.

> Достаточно важно понимать к этому моменту, что на самом деле сортировка, которую будет делать провайдер данных, будет совершена аккуратной сборкой запроса SQL на стороне базы данных. Сам провайдер данных не будет средствами PHP менять порядок записей в извлечённом наборе данных!

После быстрого взгляда на документацию по свойству `\yii\data\Sort::$attributes`, чтобы узнать точный синтаксис настройки, давайте добавим определение сортировки по названию страны в метод `search()`:

```
$dataProvider->sort->attributes['country'] = [
    'asc' => ['address.country' => SORT_ASC],
    'desc' => ['address.country' => SORT_DESC]
];
```

Мы добавляем в уже заполненное свойство `attributes`, потому что иначе нам понадобится заново писать определения сортировки по остальным атрибутам. Абсолютно необходимо, чтобы вы вставили эту команду настройки перед защитным предложением, показанным в предыдущем разделе:

```
if (!($this->load($params) && $this->validate())) {
    return $dataProvider;
}
```

Иначе вы окажетесь в глупой ситуации, когда у вас сортировка будет работать, только если таблица как-либо отфильтрована. Также вызов `$query->joinWith('addresses')` должен быть перемещён туда же по тем же причинам.

Эта строчка – всё, что нам нужно, чтобы включить сортировку по названию страны. Если вы перезагрузите страницу со списком клиентов, вы должны заметить, что заголовок колонки **Addresses** превратился в синюю ссылку и на самом деле сортирует таблицу после щелчка по себе. Впрочем, полученный порядок может выглядеть довольно странно и неочевидно, потому что база данных сортирует таблицу, созданную `LEFT JOIN`, и затем Yii 2 за сценой сжимает эту таблицу.

Мы можем добавить сортировку по атрибуту `emails`, сделав в точности то же самое, за исключением того, что нам нужно сказать `ActiveQuery` присоединить ещё одну таблицу к вечеринке:

```
$query->joinWith('emails');
$dataProvider->sort->attributes['email'] = [
    'asc' => ['email.address' => SORT_ASC],
    'desc' => ['email.address' => SORT_DESC],
];
```

Похожий код для стран работал без дополнительных изменений, потому что мы уже ранее объявили колонку **Addresses** соответствующей виртуальному атрибуту `country`, когда приделывали фильтрацию. Колонка **Emails** ещё не имеет объявления свойства `attribute`, и это единственное, что осталось сделать:

```
[
    'attribute' => 'email',
    'label' => 'Emails',
    // … здесь объявлено сжатие строк в одну ...
],
```

После вставки этой выделенной строчки кода в файл представления и перезагрузки страницы вы получите сортировку и по колонке **Emails**. Заметьте, что здесь нам не понадобилось объявлять виртуальный атрибут `email` в классе `CustomerRecordSearch`, как мы делали при реализации фильтрации. Это потому, что просто побочного эффекта объявления этой настройки хватает, чтобы превратить заголовок в ссылку для сортировки, а это всё, что нам нужно. В случае если нам понадобится фильтрация, объявление виртуального атрибута в классе модели поиска станет обязательным.

За всю эту главу мы не написали ни строчки кода тестов. Частично это было из-за того, что у нас вообще нет никакой бизнес-логики. Частично это было из-за того, что единственный способ, которым мы могли протестировать проделанную работу, – это приёмочные тесты через всё приложение. Виджет `GridView` предоставляет такой огромный набор функциональных возможностей, что набор тестов, необходимый для покрытия его целиком, будет совершенно необъятным. Поэтому весь код, который мы произвели в этой главе, – тяжёлое наследие, которое вы кладёте в своё приложение, обречённое на ручное тестирование.

Это урок, который мы хотели преподать вам в этой главе, когда отбросили подход ATDD: вы можете очень быстро собрать поразительный пользовательский интерфейс, слепо используя возможности Yii 2 повсюду. Но вы должны понимать, что вы осознанно обмениваете уверенность в том, что весь ваш код работает, как ожидается, на скорость разработки. Если вы всё ещё не уверены или не понимаете смысла этого обмена, тогда представьте, что вам понадобится сделать несколькими годами позже, когда Yii 2 превратится в Yii 3 или Twitter Bootstrap внезапно придётся заменять на что-то типа Foundation (см. http://foundation.zurb.com/).

Итоги

Итак, наше путешествие в мир настройки табличных интерфейсов подошло к концу. Сейчас вы, возможно, хотите обратиться к приложению 2, которое продолжает с этого места и показывает разработку пользовательского интерфейса для обновления нашей модели `CustomerRecord` сразу со всеми её подчинёнными моделями.

Виджет `GridView` – поразительно сложный компонент Yii 2. Это вообще больше не виджет в обычном понимании «виджета». Это полноценный пользовательский интерфейс для управления наборами данных в табличном формате. Множество движущихся частей вовлечено в его функционирование.

Мы покрыли очень много тем в этой главе, занимаясь следующим:

- агрессивно создавая CRUD-интерфейсы для таблиц базы данных, связанных между собой внешними ключами;
- бесстыже используя стили Bootstrap для наших целей;
- быстро слепив целый самосборный класс колонки `GridView`, чтобы иметь возможность щёлкнуть по иконке и гордиться появившейся всплывающей панелью;
- ещё быстрее сделав серьёзные изменения во внешнем виде данных в `GridView`, написав анонимные функции для отрисовки содержимого ячеек;
- поражаясь тому, как легко реализовать простые и не очень простые виды фильтрации и сортировки в `GridView`.

И ещё несколько не таких великих достижений тоже было сделано.

В то же самое время мы прошли мимо множества механик, которые мы частично использовали в этой главе. Среди них:

- другие виды фильтров `GridView`, например выпадающие списки выбора. Мы можем вставлять в фильтры произвольный HTML-код!
- биндинги JavaScript к событиям, происходящим в таблице. Концепция «ключей» строчек и то, как мы можем использовать её в сложном интерфейсе.

Нам нужно сохранять некоторое подобие ограничения по объёму, и эта книга всё равно не претендует на то, чтобы быть всеобъемлющим справочником.

Следующая, предпоследняя глава будет посвящена системе маршрутизации в Yii 2. Там мы рассмотрим последние две реальные функциональные возможности, которые будут реализованы в нашем примере CRM-приложения.

Глава 12

Маршрутизация

В этой главе мы изучим, как работает система маршрутизации Yii 2, то есть как фреймворк откликается на различные URL, запрошенные от него.

Мы начнём с описания процесса, который происходит в приложении Yii, для того чтобы определить действие контроллера, которое нужно выполнить в ответ на запрос клиента. Затем мы реализуем небольшую функциональную возможность, которая продемонстрирует то, как мы можем управлять нашими маршрутами, используя только конфигурацию приложения.

Наконец, мы реализуем одну особенно интересную возможность, которая потребует своего собственного класса правил обработки маршрутов, и покажем наши соображения насчёт этого.

Продвинутый курс маршрутизации

В главе 2 мы прошли вводный курс маршрутизации; пора углубить наши знания по этой теме.

Как мы уже знаем, всё в Yii 2 начинается от сценария точки входа, который в нашем примере приложения представлен файлом `web/index.php`. Этот сценарий должен быть единственным сценарием PHP в каталоге, опубликованным веб-сервером.

Так как всё приходит в этот сценарий, все маршруты, которые мы использовали в предыдущих 10 главах (исключая главу 1, которая обсуждала установку фреймворка), выглядят следующим образом:

```
протокол://имядомена/путь/до/
  index.php?r=модуль/контроллер/действие&параметр=значение&итд
```

Таким образом, любой запрос обрабатывается при помощи доступа к файлу `index.php` и передачи ему дальнейшего маршрута по приложению в виде GET-параметра под названием `r`. Это название настраивается в свойстве `\yii\web\UrlManager::$routeParam`, так что следующий фрагмент конфигурации установит строку `icecream` в качестве названия параметра маршрута:

```
[
   'components' => [
      'urlManager' => [
         'routeParam' => 'icecream'
      ]
   ]
]
```

В большинстве случаев, впрочем, такие манипуляции бессмысленны, так как у нас есть два других свойства класса: `UrlManager`: `enablePrettyUrl` и `showScriptName`, – назначение которых мы обсудили в разделе «Вводный курс маршрутизации» в главе 2.

- Свойство `enablePrettyUrl` фактически удаляет из URL'ов, которые ваше приложение принимает, часть `?r=`. То, что было доступно по `index.php?r=module/controller/action`, будет доступно просто по `index.php/module/controller/action`. Но в дополнение к этому оно включает поддержку определения преднастроенных правил разбора и генерации маршрутов. На самом деле оно полностью изменяет способ, которым приложение Yii обрабатывает маршруты.
- Свойство `showScriptName`, установленное в значение `false`, запретит менеджеру URL добавлять к URL, которые он создаёт, название сценария точки входа. Само название сценария точки входа не имеет значения; оно автоматически выводится из текущих настроек `$_SERVER`.

Говоря прагматично, вы почти всегда будете использовать следующую комбинацию параметров в разделе `components` конфигурации приложения:

```
'urlManager' => [
   'enablePrettyUrl' => true,
   'showScriptName' => false,
]
```

Единственный недостаток этого подхода – в том, что вам придётся настраивать ваш веб-сервер так, чтобы он перенаправлял все запросы на вашу точку входа, подобно следующей директиве переписывания URL:

```
RewriteRule . index.php
```

Обратите внимание на то, что режимы `enablePrettyUrl` не совместимы. Если вы используете «украшенные URL» (прямой перевод смыс-

ла словосочетания pretty url – *прим. пер.*), вы не сможете запрашивать ваше приложение, используя нотацию `?r=:route`, и наоборот.

Получение собственно маршрута из строки запроса – это только первый шаг в системе маршрутизации Yii 2.

Как мы обсудили в разделе *«Неформальное понятие "достижимости"»* в главе 7, концепция «маршрута» сводится к следующей строке:

```
/id-модуля/id-модуля/.../id-модуля/id-контроллера/id-действия
```

Эта строка позволяет фреймворку понять, какое действие контроллера оно должно выполнить. Если `id-модуля` отсутствует, тогда целевым модулем считается само приложение.

Когда же мы установим `enablePrettyUrl` в значение `true`, мы получим возможность определять специальные правила URL для разбора клиентских запросов к приложению. Фактически мы сможем использовать произвольные строки в качестве маршрутов в нашей системе.

Есть следующие три способа управлять маршрутами:
- имена модулей, контроллеров и действий;
- преднастроенные определения правил в настройке `components.urlManager.rules` конфигурации приложения;
- преднастроенные классы правил, упоминающиеся в той же настройке.

ВОЗМОЖНОСТЬ: маршрутизация с использованием имён модулей, контроллеров и действий

Если только вы не делаете по-настоящему сложные правила URL, используя две другие возможности (преднастроенные правила и преднастроенные классы правил, только что упомянутые в конце предыдущего раздела), вы можете полагаться на базовый формат пути *ID Модуля – ID Контроллера – ID Действия* с аргументами, предназначенными для самого действия, передаваемыми через параметры запроса. Это уже даёт вам много возможностей и подходит в большинстве случаев. Внимательно и аккуратно выбирая имена модулей, контроллеров и действий, вам, возможно, никогда не понадобится определять какие-либо ещё правила маршрутизации.

Внутри модуля у вас есть свойство `controllerMap`, которое позволяет вам вручную присваивать идентификаторы к конкретным классам контроллеров.

В противном случае в качестве запасного варианта работает свойство `controllerNamespace`. Идентификатор контроллера, переданный в URL, будет использован, чтобы вывести имя ожидаемого класса контроллера, а пространство имён в `controllerNamespace` будет использовано для того, чтобы физически найти файл, содержащий определение этого класса.

Идентификатор контроллера превращается в имя класса, используя в точности следующее преобразование:

```
$className = str_replace(' ', '', ucwords(str_replace
   ('-', ' ', $className))) . 'Controller';
```

Таким образом, формат записи идентификатора «через чёрточку» заменяется на формат записи имени контроллера «без пробелов с больших букв». И ещё Yii 2 ожидает, что имя класса контроллера заканчивается суффиксом `Controller`.

Внутри контроллера используется похожая механика. Используя свойство `actions` (никогда ранее не упоминавшееся в этой книге), вы можете присваивать идентификаторы действий к конкретным классам действий контроллера, которые должны быть подклассами `\yii\base\Action`.

Если никаких действий таким образом не объявлено, контроллер будет в качестве действий использовать свои публичные методы, имена которых должны начинаться со строки `action`. Такие методы называются «встроенные действия» (inline action). На самом деле, когда вы запрашиваете маршрут `user/view` и в свойстве `actions` контроллера `UserController` ничего не присвоено идентификатору `view`, он создаёт экземпляр `\yii\base\InlineAction`, указывает ему использовать метод `actionView`, а затем использует это встроенное действие так же, как любое другое действие-наследник класса `\yii\base\Action` (см. документацию и определение метода `\yii\base\Controller::runAction()` по адресу: http://www.yiiframework.com/doc-2.0/yii-base-controller.html#runAction%28%29-detail).

Возможно, это довольно иронично, но эти два *запасных* механизма – то, что вы обычно будете использовать при разработке веб-приложений с Yii 2.

Фундаментальные правила работы с URL в Yii 2

Довольно важно понимать некоторые фундаментальные понятия, лежащие в основе системы маршрутизации Yii 2. Благодаря этому всё станет намного понятнее.

1. Любой запрос, который вы передаёте в приложение Yii, будет в конечном счёте преобразован в название модуля, название класса, название действия и параметры этого действия.
2. Универсальный формат, который позволяет механике маршрутизации Yii 2 понять, какое действие контроллера вызывать, выглядит следующим образом:

```
[
   "/id-модуля/.../id-модуля/id-контроллера/id-действия",
   [
      "параметр1" => "значение1",
      "",
      "параметрN" => "значениеN"
   ]
]
```

3. Компонент работы с URL разбирает входящий запрос в вышеописанный формат, используя правила URL.
4. Компонент работы с URL превращает маршрут из этого формата в строку, которая будет помещена на HTML-страницу в качестве URL, используя те же самые правила URL.
5. Правило URL определяет разбор URL и создание URL как полностью отдельные действия. На самом деле, используя свои правила URL, вы можете создать URL из одного определения маршрута, который будет разобран в полностью другое определение маршрута.

ВОЗМОЖНОСТЬ: создание URL в Yii 2

До того, как мы углубимся в детали того, как URL разбираются Yii 2 и превращаются в вызовы действий контроллера, упомянем крайне важную функцию, которая создаёт URL'ы для отображения клиенту. Эта функция – функция `\yii\web\UrlManager.createUrl($params)`. Вы можете добраться до этого метода из любого места вашего приложения при помощи следующего вызова:

```
Yii::$app->urlManager->createUrl($params);
```

Этот метод принимает один аргумент, которым являются параметры в формате, показанном в предыдущем разделе, и возвращает текстовое представление описанного URL. Этот метод настолько широко и часто используется, что существует вспомогательный статический метод для упрощения его использования, метод `\yii\helpers\BaseUrl::toRoute()`, который нужно вызывать следующим образом:

```
URL::toRoute($params);
```

Преднастроенные маршруты с использованием конфигурации

Взгляните на следующий путь, которому мы следуем, открывая страницу просмотра записи о клиенте:

```
/customer-records/view?id=1
```

Это довольно многословно. Почему бы не использовать просто следующее:

```
/customer/1
```

Чтобы такое реализовать, нужно добавить следующее объявление в конфигурацию приложения:

```
'components' => [
   'urlManager' => [
      'rules' => [
         'customer/<id:\d+>' => 'customer-records/view',
      ]
   ]
]
```

Это правило следует читать и понимать следующим образом:

1. Если запрос начинается с `customer/`.
2. И после этого идут только цифры.
3. Сохранить эти цифры в аргументе под названием `id`.
4. Считать запрос маршрутом `customer-records/view`.
5. Передать сохранённый аргумент `id` в итоговое действие контроллера.

ВОЗМОЖНОСТЬ: правила URL

Настройка приложения `components.urlManager.rules`, которая соответствует свойству `\yii\web\UrlManager::$rules`, может быть заполнена объектами в двух формах записи.

Полная форма – это форма записи, предназначенная для передачи в метод `Yii::createObject()`. Вы в данном случае будете определять экземпляры класса `\yii\web\UrlRule`. В следующей таблице перечислены самые важные свойства правил URL:

Свойство	Смысл
`pattern`	Как должен выглядеть входящий запрос, чтобы это правило сработало

Свойство	Смысл
verb	Какой должен быть использован метод HTTP, чтобы это правило сработало
route	Маршрут до действия контроллера в формате /id-модуля/.../id-модуля/id-контроллера/id-действия, в который должен разрешиться pattern
suffix	Какую строку добавлять при создании URL и какая строка должна находиться в конце pattern при разборе запроса, чтобы это правило сработало
mode	Если установлено в 0, это правило будет использоваться как для создания, так и для разбора URL. Иначе вы можете выставить его в значения \yii\web\UrlRule::PARSING_ONLY или \yii\web\UrlRule::CREATION_ONLY

Есть ещё некоторые свойства, имеющие более специфичное использование, такие как host и defaults. Вам настоятельно советуется прочитать документацию (http://www.yiiframework.com/doc-2.0/yii-web-urlrule.html) и/или исходный код класса \yii\web\UrlRule, чтобы узнать точные детали.

Наиболее важное и полезное свойство здесь – это свойство pattern. Оно определяет, что должно быть передано в Yii в качестве строки запроса (без параметров запроса), с добавлением именованных параметров. Именованный параметр – это конструкция следующей формы:

```
<Имя:РегулярноеВыражение>
```

Угловые скобки и двоеточие – обязательный синтаксис. Именованные параметры могут быть позднее упомянуты в свойстве route, что увеличивает обобщённость правила. Если они не упоминаются в route, они сохраняются под тем же именем в качестве параметра в $_GET (переписывая ранее сохранённые под тем же именем значения, если таковые имеются).

Сокращённая запись определения правил URL выглядит следующим образом:

```
"[verb ]pattern" => "route"
```

Это инициализирует объект правила URL с указанными свойствами pattern, verb, и route, все прочие свойства оставляя пустыми. Параметр verb необязателен, что показано квадратными скобками (они не являются синтаксисом определения правила). Документация Yii 2 для \yii\web\UrlManager::$rules показывает прекрасный самоописывающий пример менеджера URL, настроенного для запросов, совместимых с REST:

```
'rules' => [
   'dashboard' => 'site/index',
```

```
    'POST <controller:\w+>s' => '<controller>/create',
    '<controller:\w+>s' => '<controller>/index',
    'PUT <controller:\w+>/<id:\d+>' => '<controller>/update',
    'DELETE <controller:\w+>/<id:\d+>' => '<controller>/delete',
    '<controller:\w+>/<id:\d+>' => '<controller>/view',
];
```

Это на самом деле довольно легко читается, при условии что вы всегда помните, что белый шум между двоеточием и закрывающей угловой скобкой является всего лишь регулярным выражением. Как вы видите, именованные параметры `id` из шаблона не упоминаются в `route`, что означает, что они будут переданы в соответствующее действие контроллера в качестве аргументов с теми же именами. Параметр под названием `controller` делает этот набор правил обобщённым, применимым ко всем контроллерам в системе.

Правила, объявленные в настройке `rules`, проверяются в порядке появления. Первое правило, соответствующее текущему запросу, применяется, и все остальные после этого игнорируются. Если ни одного правила не сработало, тогда менеджер URL начинает полагаться на разбор запроса в формате `/id-модуля/id-контроллера/id-действия`.

Помимо манипулирования свойствами базового класса `\yii\web\UrlRule`, мы можем в настройке `rules` сослаться на любой другой класс в качестве правила URL, при условии что этот класс реализует `\yii\web\UrlRuleInterface`. Этот интерфейс объявляет два фундаментальных действия правила URL:

Метод	Смысл
`parseRequest( $manager, $request )`	Аргумент `$manager` – это экземпляр класса `UrlManager`. Аргумент `$request` – это экземпляр класса `Request`, который вы можете использовать, чтобы получить строку запроса и параметры для разбора. Этот метод должен вернуть либо маршрут в канонической форме, показанной ранее в этой главе, либо в точности значение `false`, что будет означать, что это правило не может быть применено к текущему запросу
`createUrl( $manager, $route, $params )`	Аргумент `$manager` – это экземпляр класса `UrlManager`. Аргумент `$route` – это строка, представляющая маршрут в базовом формате `/id-модуля/id-контроллера/id-действия`. Могут быть варианты, например отсутствующие части, принимайте это во внимание. Аргумент `$params` – это массив пар ключ–значение, определяющий параметры запроса для этого маршрута. Ожидается, что этот метод вернёт строку, которая должна быть относительным URL, которую в идеале должно быть возможно разобрать обратно в маршрут/параметры вызовом метода `parseRequest()` того же класса `UrlRule`. Если маршрут/параметры не соответствуют этому правилу, этот метод должен вернуть в точности значение `false`

Преднастроенные маршруты с использованием классов правил URL

В качестве более сложной задачи давайте реализуем более изощрённую возможность. Учитывая наш сценарий миграции, который создаёт пользователей по умолчанию в базе данных, следующий URL приносит нас к странице просмотра того пользователя, который имеет имя `AnnieManager`:

```
/users/view?id=2
```

Совершенно реальный запрос от вышестоящего начальства может выглядеть следующим образом: сделать так, чтобы та же страница была доступна по следующему URL:

```
/AnnieManager
```

Это правило не может быть выражено в виде пары шаблон–маршрут, как ранее, потому что нам нужно соотносить переданное имя пользователя, сохранённое в базе данных, с идентификатором, который ожидает действия `actionView()` контроллера.

Чтобы понимать преднастроенные правила URL, вы должны очень хорошо понимать следующее: в конце концов, будет выполнено определённое действие контроллера, и мы должны передать ему аргументы, которые оно ожидает. То, о чём мы говорим в этой главе, – это абстракция на один уровень выше, где мы скрываем эту базовую маршрутизацию за другим словарным запасом, с различными целями, будь то SEO, большая регулярность вроде REST или просто приятность глазу. Но этот уровень всё равно будет в конечном счёте разрешён в пару маршрут/параметры.

Мы решим эту задачу, внедрив преднастроенный класс правил URL. Объявите его в наборе правил следующим образом:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'showScriptName' => false,
    'rules' => [
        'customer/<id:\d+>' => 'customer-records/view',
            // наше правило из предыдущей задачи
        [
            'class' => 'app\utilities\UsernameUrlRule'
        ]
    ]
],
```

На этот раз мы начали с объявления, вместо того чтобы подсоединять уже существующий класс. Давайте тогда создадим этот класс, как объявлено, внутри файла `@app/utilities/UsernameUrlRule.php`:

```
namespace app\utilities;

use yii\web\UrlRuleInterface;

class UsernameUrlRule implements UrlRuleInterface
{
   public function parseRequest($manager, $request)
   {
      // здесь разбор запроса...
   }
   public function createUrl($manager, $route, $params)
   {
      // здесь создание URL из пары маршрут/параметры...
   }
}
```

Вначале разбор методом `parseRequest()`. Мы проверим, применяется ли вообще это правило, следующим образом:

```
$maybeUsername = $request->pathInfo;

$user = UserRecord::findOne(['username' => $maybeUsername]);
if (!$user)
   return false;
```

Здесь очень простая логика. Если строка между именем хоста и символом `?`, отмечающим начало параметров запроса, не является именем пользователя ни для какой `UserRecord` в нашей базе данных, мы говорим `UrlManager`, что ничего здесь сделать не можем.

Это очень серьёзная брешь в безопасности, которую вы должны тем или иным образом закрыть, если реально будете реализовывать подобную функциональность в вашем веб-приложении. Если пользователи могут сами себя регистрировать и выбирать себе имя, то ничто не останавливает их от того, чтобы выбрать имя наподобие `site`, или `users`, или `customer-records`, которое соответствует идентификатору существующего контроллера. Так как правила URL проверяются в первую очередь, существование такого пользователя нарушит доступ к упомянутому контроллеру. Например, URL `/customer-records`, который обычно идентичен вызову `/customer-records/index`, будет разрешаться в страницу просмотра пользователя с именем `customer-records`.

Один из способов, который достаточно неудобно сопровождать, – это проверка того, находится ли `$maybeUsername` среди некоторого набора запре-

щённых ключевых слов, и этот набор можно автоматически создавать из идентификаторов контроллеров, использованых в вашем приложении.

С другой стороны, если `$user` на самом деле найден, мы создаём необходимую пару маршрут/параметры и возвращаем её:

```
$route = 'users/view';
$params = ['id' => $user->id];
return [$route, $params];
```

Наконец, нам нужно иметь возможность создавать URL в том же формате, используя метод `createUrl()`. Одно из мест, где мы можем найти URL'ы страниц просмотра пользователей, – это страница `/users/index`, на которой есть виджет `GridView`, перечисляющий модели `UserRecord`.

Нам нужно вначале проверить, относятся ли вообще к нам переданные маршрут и параметры:

```
if ($route !== 'users/view' || !array_key_exists ('id', $params))
   return false;
```

Нас не заботят никакие маршруты, кроме `users/view`, и мы требуем, чтобы нам передали `id`.

Также нам нужно убедиться, что в базе данных на самом деле существует нужный `UserRecord`, иначе мы не сможем получить имя пользователя:

```
$user = UserRecord::findOne($params['id']);
if (!$user)
   return false;
```

Если всё в порядке, всё, что нам нужно сделать, – это вернуть имя пользователя:

```
return "{$user->username}";
```

Обратите внимание на отсутствие косой черты в начале, что сделано явным при помощи интерполяции строк. Yii 2 автоматически вставит нам косую черту, так что метод `createUrl()` должен вернуть только оставшуюся часть строки.

Теперь откройте страницу `/users` и наведите курсор на любую кнопку с иконкой глаза. Она должна иметь URL, соответствующий имени пользователя. После щелчка по этой кнопке вы должны оказаться на странице просмотра соответствующего объекта `UserRecord`.

Итоги

Эта глава подводит итог механике маршрутизации, с которой мы имели дело в течение всей этой книги.

Мы посмотрели на два практических примера из реальной жизни, которые показали, как мы можем использовать эту механику. Всё остальное можно прочитать в документации для классов `UrlManager` и `UrlRule` и ещё для класса `\yii\helpers\Url helper`.

Следующая глава будет завершающей главой нашего путешествия. Мы разберёмся с инфраструктурными проблемами разработки приложения Yii 2, в особенности с проблемами, возникающими при обмене кодом между разработчиками и при обмене кодом между окружениями разработки и реальной работы.

Глава 13

Совместная работа

Это последняя глава книги, за исключением приложений (в которых тоже содержится полезная информация; не пропускайте их).

Мы завершим наше приключение, начатое 12 глав назад, возможностями Yii 2, которые относятся не к значению приложения для бизнеса, а к его инфраструктуре.

При работе с базой кода в составе команды других разработчиков и постоянно развёртывая код между производственными и тестовыми серверами, вы неизбежно столкнётесь с некоторыми проблемами, которые вам придётся решить, чтобы продолжить эффективно поставлять новую функциональность.

Во-первых, мы выучим некоторые трюки для управления конфигурацией приложения, чтобы приспособиться к различным целям развёртывания.

Во-вторых, мы взглянем на консольные приложения Yii: ту сторону, которую мы несколько раз неявно использовали в этой книге, но никогда не обсуждали открыто.

И под конец поговорим о миграциях базы данных, которые всё это время использовали. Мы надеемся, что у вас теперь достаточно практического опыта использования миграций. Здесь мы поговорим о причинах их использовать и о некоторых трюках, для того чтобы более эффективно управлять ими.

Конструирование конфигурации

Нам всем известна следующая проблема.

Представим, что мы разрабатываем веб-приложение и делаем это на своей локальной рабочей машине. Приложение использует экземпляр MySQL, также установленный локально, с определённым названием базы данных, именем пользователя и паролем. Когда мы

развёртываем это приложение, оно будет использовать экземпляр MySQL, установленный на цель развёртывания, над которым мы можем иметь, а можем и не иметь контроля в выборе имён и паролей. Даже если мы имеем полный контроль над целью развёртывания и можем использовать те же самые настройки подключения, очень непрактично вынуждать всех остальных людей в команде использовать наши имена пользователей и пароли на их собственных рабочих станциях.

Так как конфигурация приложения Yii – это просто сценарий PHP, возвращающий ассоциативный массив, у нас есть простой способ решить эту проблему: все части конфигурации, которые должны быть указаны для каждой цели развёртывания индивидуально, перенести в отдельные файлы. Основная конфигурация просто включит их содержимое в себя, используя стандартные вызовы `require()`.

На самом деле наша конфигурация примера CRM приложения уже так и настроена. Ниже показаны строчки кода, которые включают другие фрагменты конфигурации в основной файл настроек, который в конце концов передаётся в конструктор `\yii\web\Application`:

```
return [
   // … глобальные настройки здесь пропущены ...
   'components' => [
      'db' => require(__DIR__ . '/db.php'),
      // … множество других компонентов пропущено …
      'assetManager' => [
         'bundles' => (require __DIR__ .
            '/assets_compressed.php')
      ],
   ],
   'extensions' => (require __DIR__ .
     '/../vendor/yiisoft/extensions.php')
];
```

Мы уже использовали раздельную конфигурацию для базы данных, сжатых файлов материалов (из главы 8) и расширений.

Это, возможно, единственное место, где альтернативный синтаксис вызова `require()` выглядит полезным: нотация `(require PATH)` явным образом выражает, что мы берём что-то из `PATH` и вставляем прямо внутрь скобок, что, возможно, легче читать, чем обычную нотацию в стиле вызова функции.

Используя ту же технику, можно разделить объявления преднастроенных параметров в настройке `params` приложения. Этот трюк

использует базовый шаблон приложения Yii (см. https://github.com/yiisoft/yii2/tree/master/apps/basic/config).

Перекрытие конфигурации можно вывести на новый уровень, если принять во внимание встроенную в PHP функцию `array_merge_recursive()` и метод `\yii\helpers\ArrayHelper::merge()` из Yii. Статическая функция `ArrayHelper::merge()` особенно полезна, потому что вместо комбинирования значений, имеющих одинаковые ключи (что делает вызов `array_merge_recursive()`), она переопределяет старое значение новым.

Как это можно использовать? Очевидно, у нас могут быть конфигурация приложения по умолчанию в одном файле и переопределения, специфичные для цели развёртывания, в другом файле. Конечная конфигурация, которая должна быть передана в экземпляр приложения Yii, будет собрана вызовом `ArrayHelper::merge()` следующим образом:

```
// config/web.php, который мы всегда имели
return \yii\helpers\ArrayHelper::merge(
   (require "default.php"),
   (require "local.php")
);
```

Внутри файла `local.php` вы будете поддерживать в точности ту же самую структуру, что и в файле `default.php`, что гораздо проще делать, чем постоянно запоминать, какая часть конфигурации лежит в том или ином отдельном файле. Конечно же, некоторые файлы, такие как файл `extensions.php`, придётся оставить как есть. В случае с файлом `extensions.php` у нас всё равно нет контроля над его содержимым.

Добавление локальных переопределений в конфигурацию

В главе 2 мы внедрили отдельный фрагмент конфигурации `db.php`, для того чтобы соответствовать базовому шаблону приложения Yii, без объяснения того, зачем это нужно было вообще делать. На самом деле такое разделение – это рудиментарный шаг к тому, чтобы сделать код более переносимым между разработчиками и между целями развёртывания. На каждой из них вы можете иметь разные сценарии `db.php` с разными учётными записями и т. п. Для того чтобы этого добиться, вам нужно всего лишь не фиксировать этот фрагмент в системе контроля версий. Однако такой подход имеет серьёзный недостаток: каждый экземпляр приложения должен иметь настройки соединения с базой данных, написанные с нуля. Вам нужно сообщать любые об-

щие настройки между разработчиками каким-то образом, отличающимся от фиксации изменений в репозитории исходного кода, так что вы не сможете, например, раз и навсегда записать настройки кэширования или указать какой-то другой класс подключения к БД.

Давайте применим описанную ранее технику к нашей базе исходного кода. Для того чтобы это осуществить, нам нужно сделать несколько очень маленьких шагов.

1. Вначале создайте подкаталог под названием `overrides` внутри папки `config`. Это название подразумевает, что фрагменты исходного кода будут «перекрывать» друг друга.
2. Внутри `overrides` создайте файл под названием `base.php`, который будет хранить базовые настройки как для консольного, так и для веб-приложения на любой цели развёртывания.
3. Мы перенесём очень небольшой объём настроек в файл `config/overrides/base.php`. Вот они:

```
return [
    'basePath' => realpath(__DIR__ . '/../../'),
    'components' => [
        'db' => [
            'class' => '\yii\db\Connection'
        ],
    ]
];
```

В расширенном шаблоне приложения Yii имеются раздельные подкаталоги для консольного и двух веб-приложений, так что у них `basePath` будет различным. В нашем случае оба приложения имеют одно и то же значение настройки `basePath`.

Мы инициализировали компонент соединения с базой данных только лишь именем класса. Данные для подключения будут предоставлены в дальнейших перекрытиях. Нам нужно подключение к БД как в консольном приложении, так и в веб-приложении из-за миграций.

Вот как мы разделим конфигурацию дальше:

1. Переместите файл конфигурации `web.php` в файл `config/overrides/web_base.php`. Это имя было выбрано таким образом, потому что мы потом создадим файл `config/web.php` заново, и в общем случае не очень хорошо иметь в проекте два разных файла с одинаковыми названиями, даже если они в разных подкаталогах.
2. Внутри файла `web_base.php` нам нужно удалить те части конфигурации, которые уже определены в базовой конфигурации. Ими являются только настройки `basePath` и `components.db`.

3. Кроме этого, важно изменить пути к файлам `extensions.php` и `assets_compressed.php` в файле `web_base.php`, так как мы теперь находимся на один подкаталог глубже. Вам решать, перемещать ли куда-нибудь файл `assets_compressed.php`, но переместить его в каталог `overrides` будет нарушением концепции «перекрытий».
4. Далее переместите файл конфигурации `console.php` в файл `config/overrides/console_base.php`. Как и раньше, удалите из него настройки `basePath` и `components.db`.
5. Теперь мы подходим к самой интересной части. Создайте файл конфигурации `config/overrides/local.php`, который будет содержать только лишь настройки соединения с базой данных, в таком виде:

```
return [
   'components' => [
      'db' => [
         'dsn' => 'mysql:host=localhost;dbname=crmapp',
         'username' => 'root',
         'password' => 'mysqlroot'
      ]
   ]
];
```

   Вне зависимости от того, насколько специфичны локальные переопределения, вам нужно содержать в точности ту же структуру файла конфигурации, как и в главном файле конфигурации Yii. Вся идея «переопределений» основывается на этом поддержании общей структуры.
6. Имея это локальное переопределение, мы наконец можем создать заново файлы `config/web.php` и `config/console.php`, которые ожидают наши точки входа. Вот как будет выглядеть файл `config/web.php`:

```
return \yii\helpers\ArrayHelper::merge(
   (require __DIR__ . '/overrides/base.php'),
   (require __DIR__ . '/overrides/web_base.php'),
   (require __DIR__ . '/overrides/local.php')
);
```

Мы просто сливаем воедино три конфигурации, и их порядок имеет важнейшее значение. Теперь, если вы откроете наш пример CRM-приложения в браузере, ничего не должно измениться, что является именно тем, что нам нужно. Все тесты также должны проходить успешно.

Тем же образом создайте заново файл config/console.php, за исключением того, что вместо файла web_base.php нужно использовать файл console_base.php.

Эти изменения приводят к важным последствиям. Главное преимущество, которое у нас есть, – это то, что теперь мы можем зафиксировать в системе контроля версий только базовую конфигурацию и базовое переопределение конфигурации для веб-приложения. Локальное переопределение должно быть создано для каждого отдельного развёртывания. Для того чтобы помочь разработчикам и системным администраторам, мы можем зафиксировать в репозитории специально подготовленную копию файла local.php, где все конкретные учётные данные заменены на какие-либо имеющие смысл заменители. Такие шаблоны конфигурации обычно называются так же, как и файлы, которые они представляют, с добавлением суффикса -example, вот так: local-example.php. Вот как мы можем подготовить такой пример в нашем CRM-приложении:

```
<?php
/**
 * Это пример локального переопределения настроек.
 * Вы должны определить хотя бы параметры подключения к базе данных.
 */

return [
   'components' => [
      'db' => [
         'dsn' => 'mysql:host=localhost;dbname=DB_NAME',
         'username' => 'DB_USERNAME',
         'password' => 'DB_PASSWORD'
      ]
   ]
];
```

Для того чтобы точно не зафиксировать настоящий фрагмент конфигурации local.php в репозиторий, в системе контроля версий Git вы можете добавить для него правило в .gitignore-файл.

Расширенный пример приложения из пакета Yii уже использует подобный трюк с переопределениями, но только для преднастроенных параметров приложения (см., например, https://github.com/yiisoft/yii2/blob/master/apps/advanced/frontend/config/main.php). Чтобы увидеть реальный пример сложного многоуровневого построения конфигу-

рации, вы можете обратиться к проекту YiiBoilerplate от Clevertech по адресу https://github.com/clevertech/YiiBoilerplate. Он для Yii 1.x, но концепция остаётся той же самой, возможно, более расширенной.

Консольное приложение

В главе 2 мы настроили сущность, которую назвали «исполнителем консольных команд», в виде сценария PHP под названием `yii` в корневом каталоге приложения, и пропустили всякие объяснения того, что это такое. После этого мы широко использовали два вызова в командной строке: `./yii migrate/create` и `./yii migrate`. Мы также использовали вызов `./yii asset` в главе 8, чтобы подготовить наши скомпилированные материалы. Теперь настало время объяснить ценность и возможности этой сущности.

Кроме класса `\yii\web\Application`, который представляет веб-приложение, которое мы строили предыдущие 11 глав (исключая главу 1), фреймворк Yii 2 включает в себя класс `\yii\console\Application`, который представляет консольное приложение. Этот вид приложения концептуально – в точности то же самое, что и веб-приложение, в том смысле, что он тоже является модулем, поддерживающим MVC. Специфика консольного приложения в том, что оно должно отрисовывать результаты своей работы на консоль и принимать входные параметры из командной строки. За исключением этого, консольное приложение использует ту же самую концепцию контроллеров (которые в этом случае должны быть наследниками `\yii\console\Controller`) и конфигурируется и создаётся тем же образом, что становится очевидным, если вы сравните содержимое сценариев `yii` и `web/index.php`.

В сущности, для того чтобы достичь действия контроллера, используя консольное приложение, вы должны выполнить следующий вызов в командной строке:

```
./yii id-контроллера/id-действия значение1параметра значение2параметра
  --свойство1=значение1
```

Детали следует читать в документации по фреймворку (см. http://www.yiiframework.com/doc-2.0/guide-tutorial-console.html). Обратите внимание на то, что аргументы действия контроллера передаются как неименованные позиционные аргументы командной строки, а значения для свойств контроллера передаются как именованные аргументы.

Если только вы не делаете конкретно консольное приложение, используя фреймворк Yii, сложно оправдать разделение консольных

контроллеров на модули, но, так как консольное приложение остаётся модулем и использует ту же механику маршрутизации, что и веб-приложение, вы можете также использовать маршруты в виде `/id-модуля/.../id-модуля/id-контроллера/id-действия`.

Поскольку несколько неуклюже говорить о консольных «контроллерах», мы будем говорить о них как о консольных «командах», как в Yii 1.1.x, но то, что мы на самом деле имеем в виду, – это классы контроллеров, наследующие классу `\yii\console\Controller`. Вместе с фреймворком вы получаете набор встроенных консольных команд, как, например, `\yii\console\controllers\MigrateController`. Они перечислены в методе `\yii\console\Application::coreCommands()`, и вам рекомендуется прочитать самодокументацию их классов. Эти команды всегда доступны из консольного приложения, если только вы не переопределите этот метод (и зачем вам вообще это делать?). Мы не будем их обсуждать подробно, так как, помимо команд `migrate` и `asset`, все остальные несколько специфичны.

Преднастроенные консольные команды

Давайте сделаем какую-нибудь свою консольную команду. Мы сделаем небольшой хак, для того чтобы разработчики могли вручную вносить записи о пользователях прямо в базу данных.

Как вы помните из главы 5, мы храним пароли пользователей в хэшированном виде, используя вспомогательные классы, встроенные в Yii. Это мешает возможности разработчиков создавать записи о пользователях вручную, используя прямой доступ к базе данных. Мы легко можем придумать и использовать пароль в открытом виде, но мы должны хранить его в БД хэшированным, и этот хэш должен быть как-то вычислен, а при помощи ручки и бумажки сделать это практически невозможно. Более того, мы вообще точно не знаем, какой метод хэширования использует Yii.

Поэтому давайте сделаем консольную команду, которая, получив некоторую строку, показывает её хэш, вычисленный тем же способом, который использует метод `\app\models\user\UserRecord::beforeSave()`.

Поскольку консольные контроллеры являются классами контроллеров, Yii требует, чтобы все они были в одном и том же каталоге. Мы произвольно решили, что это будет подкаталог `commands` корневого каталога проекта. Согласно правилам PSR-4 и соглашениям Yii, этот подкаталог представляет пространство имён `app\commands`; поэтому давайте прямо сейчас присоединим это пространство имён к нашему

консольному приложению. Откройте файл config/overrides/console_base.php и вставьте туда настройку controllerNamespace:

```
return [
    'id' => 'crmapp-console',
    'controllerNamespace' => 'app\commands',
];
```

Пусть наша новая команда называется hash, так чтобы её можно было вызвать командой ./yii hash. Значит, нам нужен класс HashController, суффикс Controller которого обязателен, внутри подкаталога commands, со следующим кодом внутри:

```
namespace app\commands;
use yii\console\Controller;
class HashController extends Controller
{
    public function actionIndex()
    {
        // пока не знаем, что тут делать
    }
}
```

Вы уже должны знать, как объявлять новые контроллеры, так что пространство имён и предложения use здесь – самые важные части. Очень важно, чтобы мы помнили, что мы создаём консольный контроллер, а не веб-контроллер, так как они по-разному выполняют действия. Действием по умолчанию является index, так же, как и в веб-контроллерах.

Вместо того чтобы использовать встроенное в PHP echo для вывода на консоль, мы будем использовать вспомогательный класс \yii\helpers\Console. Он содержит метод output(), который автоматически вставляет переносы строк, так как добавлять их вручную всегда было тем ещё неудобством. Вот что нам нужно вставить, чтобы удовлетворить нашим нуждам:

```
public function actionIndex($string)
{
    \yii\helpers\Console::output(
        \Yii::$app->security->generatePasswordHash($string)
    );
}
```

Вызов Security::generatePasswordHash() – это в точности то, как мы генерируем хэш паролей для наших записей пользователей. Вот то,

что вы должны получить после вызова команды `./yii hash 1234`, чтобы посмотреть хэш очень глупого пароля пользователя:

```
vagrant@precise64:/vagrant$ ./yii hash 1234
$2y$13$GnGFWlYaecpoKoJTIUlFcuzqI7qyaODej8dQ6ZBMZHtmxgApSpiRq
```

Эту же команду можно вызвать таким образом: `./yii hash/index 1234`. Хэш, конечно же, будет другим, но это всё равно будет то же самое действие контроллера.

Обратите внимание на то, что аргументы действий контроллера передаются как неименованные аргументы командной строки. Это означает, среди прочего, что вы должны передавать строки, содержащие специальные символы, заключёнными в кавычки. Чтобы помочь представить, что мы на самом деле хэшируем, давайте также выводить строку, которую мы передали в метод хэширования. Однако довольно мило, что наша исходная команда соответствует пути Unix (то есть она выводит результаты сразу на стандартный вывод безо всякой дополнительной информации). Мы воспользуемся встроенным флагом консольных контроллеров, чтобы проверить, на самом ли деле нам нужно выводить эти дополнительные сведения. Добавьте следующие две строчки в начало метода `actionIndex()`:

```
if ($this->interactive)
    Console::output(sprintf('Input string was: %s', $string));
```

Свойство `$this->interactive`, на которое мы ссылаемся, – это свойство `\yii\console\Controller::$interactive`, доступное изо всех консольных команд. Как было сказано ранее, вы можете устанавливать значения таких свойств, используя именованные параметры. В нашем случае это означает, что нам нужно передать аргумент `--interactive=0` в вызов `./yii hash 1234`, чтобы подавить вывод отладочной информации. Иначе мы получим полезное напоминание:

```
vagrant@precise64:/vagrant$ ./yii hash some string with spaces
Input string was: some
$2y$13$.C40HrK8Muj5SN3fw2fpp.JJvdm7wPQ1YnwOkS59kWGzWTJ2IqegS
vagrant@precise64:/vagrant$ ./yii hash "some string with spaces"
Input string was: some string with spaces
$2y$13$NWOHw.n9NEkt4wJvr.EjOuc.hFtwqS/qFq0hY5R5PkHibk9CHME.y
vagrant@precise64:/vagrant$ ./yii hash "some string with spaces" --interactive=0
$2y$13$2npFbKFax3n7hOoiOn1rieNsvYp7vXU.QlJM.kzWpRFOel4wRrztu
```

Это поможет избежать глупых ошибок при ручном создании разумно сложных паролей, так как в противном случае вы бы не знали, что вы на самом деле хэшируете. Обратите внимание на то, какая строка

была передана в наше действие контроллера в первом случае, когда мы не экранировали входную строку.

Для того чтобы передать значение `false` в свойства контроллеров, используйте любую строку, которая будет оценена как значение, подобное `false`, например пустую строку или нуль. Строки `null` или `false` будут расценены как непустые строки и, соответственно, будут считаться булевым значением `true`. Так что `--interactive=false` не будет работать, как хотелось бы; а вот `--interactive=` будет.

Миграции базы данных

Зависимость от базы данных очень серьёзна. В идеале ваше приложение вообще не должно зависеть от системы управления базами данных (**СУБД**). Однако большую часть времени клиентам нужны приложения, которые концептуально являются всего лишь изощрёнными CRUD-интерфейсами над базой данных, так что эта зависимость может считаться неизбежной.

Проблемы, вызванные совместной разработкой в таких условиях, широко известны. Если приложение ожидает, что база данных имеет определённую структуру, и в процессе разработки в код вносится изменение, которое меняет некоторые из этих ожиданий, то нам нужно исправлять схему базы данных на всех машинах, на которых развёрнуто это приложение. Конечно же, мы также должны иметь какого-то рода сценарий инициализации схемы базы данных, и это изменение также должно быть внесено в этот сценарий.

Трюк с миграциями, реализованный в Yii (и описанный в документации по адресу http://www.yiiframework.com/doc-2.0/guide-db-migrations.html), имеет корни в концепции миграций из Ruby on Rails (описанной в их чужеродной документации по адресу http://guides.rubyonrails.org/migrations.html). Вы уже видели, что сценарий миграции базы данных в Yii 2 – это фактически маленькая программа, которая может использовать методы класса `\yii\db\Migration` для выполнения изменений в схеме реляционной базы данных в единообразной, независимой от производителя манере. Использование их имеет очевидные преимущества, по сравнению с выполнением сценариев SQL напрямую:

- вы не зависите от производителя базы данных. На самом деле вы можете даже заменить нижележащую базу данных в течение жизненного цикла приложения, безо всяких изменений в существующих миграциях, собранных с течением времени;

- внутри миграции вы находитесь на уровне приложения. Вы можете совершить произвольные проверки, перед тем как возиться с базой данных. Более того, в миграции вы можете выполнять произвольный код, вообще не обязательно связанный с базой данных.

Есть также дополнительная неочевидная возможность: если хотите, вы можете определить как процедуру обновления (upgrade), так и процедуру отката обновления (downgrade). В результате вы сможете отменить сделанные изменения и убрать запись о проделанном ранее обновлении из базы данных. Это полезно при подготовке изменений в очередном методе `up()`, потому что вы можете легко откатить только что сделанные изменения в базе данных и поправить определение метода `up()`. Правда, это будет работать, только если вы будете очень осторожны со своими изменениями в базе данных, так как некоторые изменения могут быть необратимыми. Эта возможность была предоставлена вам в каждом шаблоне миграции, который вы видели до этого момента:

```
class m140318_173202_add_auth_key_to_user extends \yii\db\Migration
{
    public function up()
    {
        $this->addColumn('user', 'auth_key', 'string UNIQUE');
    }

    public function down()
    {
        $this->dropColumn('user', 'auth_key');
    }
}
```

В то время как метод `up()` – это то, что вам обычно нужно, чтобы сделать изменение, которое вы хотите, метод `down()` существует, чтобы отменить эти изменения. Как уже было сказано, существуют необратимые изменения. Например, удаление колонки из таблицы. Для того чтобы полноценно откатить это изменение на реальной базе данных, нужно будет восстановить данные, которые хранились в этой колонке, но вы вряд ли захотите при удалении колонки сохранять её прежний набор данных только ради поддержки отката изменений. Для таких операций вы можете просто опустить метод `down()`, и при откате изменений эта миграция будет молча пропущена.

Если метод `up()` возвращает булево значение `false`, эта миграция считается неприменённой, и все возможные последующие миграции

отменяются. Это полезно, если вы делаете изменения, которые требуют какой-либо ручной подготовки от оператора. Условное выражение в миграции проверит, нужна ли подготовка, и остановит миграцию, если нужно. Затем, после того как проблема будет решена, миграция пройдёт как обычно.

Метод `down()` обладает той же возможностью. Если он возвращает булево значение `false`, то текущий откат изменений вместе со всеми возможными последующими будет отменён. Это может быть использовано в случае, если у вас есть какие-то по-настоящему необратимые изменения в методе `up()`, после которых уже некуда возвращаться. Шаблон сценария миграции по умолчанию в Yii 2 подготавливает именно такой метод `down()`.

Хотя откат изменений особенно полезен для облегчения исправлений возможных ошибок в методе `up()`, он также может использоваться для отката состояния базы данных на определённый момент в истории разработки. Это значит, что если кто-то обнаружит баг в коде работы с базой данных, появившийся только после версии приложения, скажем, x, а самая свежая версия приложения, скажем, x+5, тогда вы можете посмотреть на историю фиксаций изменений в вашей системе контроля версий, найти последний сценарий миграции, существующий в версии кода x, и откатить миграции до этого момента. После этого вы откатываете базу кода на пять версий ранее, используя систему контроля версий, и получаете состояние приложения в точности, как оно было в версии x. Это намного менее агрессивно, чем откатывание кода на версию x, уничтожение текущей базы данных и создание её заново с нуля при помощи только методов `up()`, но вам нужна серьёзная дисциплина, для того чтобы всегда делать только обратимые или безвредные необратимые изменения в миграциях.

За исключением этой сложности с возможностью отката изменений, в самой концепции миграций вообще нет ничего сложного. Использование их – с другой стороны, совсем иная тема.

Мы широко использовали миграции в течение этой книги, поэтому вы уже должны были привыкнуть к самой команде `./yii migrate` (которая является сокращённой формой команды `./yii migrate/up`, и в следующем разделе мы объясним, почему). В дополнение к этому вызов команды `./yii help migrate` выдаст вам список всех возможных вариантов вызова `./yii migrate`.

Для того чтобы знать, какие миграции применены к базе данных, а какие – (ещё) нет, Yii 2 создаёт и сопровождает за вашей спиной специальную таблицу в базе данных, название которой контролиру-

ется параметром `\yii\console\controllers\MigrateController::$migrationTable`. Каждая запись в этой таблице хранит название использованного класса миграции и время его использования. По умолчанию эта таблица называется `migration`, и если вы собираетесь использовать это название для одной из своих таблиц, вам следует поменять данную настройку команды `MigrateController`.

Для такого фундаментального изменения будет неудобно передавать именованный параметр `--migrationTable` в каждый вызов `./yii migrate`, поэтому лучше использовать вместо этого настройки приложения. Так как это контроллер, а не компонент, вам нужно для этого использовать настройку `controllerMap` консольного приложения следующим образом:

```
'controllerMap' => [
   'migrate' => [
      'class' => 'yii\console\controllers\MigrateController',
      'migrationTable' => 'my_custom_migrate_table',
   ],
]
```

Таким же образом вы можете переопределить любое другое свойство консольных контроллеров. Эта техника также применима и к веб-контроллерам.

Обратите внимание, что это перекрытие параметров имеет важный побочный эффект: мы фактически объявляем идентификатор для вызова определённого контроллера с предустановленными параметрами. Ничто не останавливает нас от того, чтобы объявить один и тот же контроллер несколько раз под разными идентификаторами и с разными настройками. Например, с контроллером `HashController`, который мы обсуждали ранее, мы могли сделать следующее:

```
'controllerMap' => [
   'silentHash' => [
      'class' => 'app\commands\HashController',
      'interactive' => false
   ]
]
```

Это дало бы нам возможность вызова `./yii silentHash` без необходимости делать дополнительное определение класса.

Другое важное свойство – это свойство `migrationPath`, которое определяет каталог, в котором `MigrationController` будет искать (и создавать) классы миграций. По умолчанию это подкаталог `migrations`

корневого каталога приложения, и это в точности то, что мы решили использовать в главе 2 (чудесным образом). Мы использовали это свойство в главе 6, когда устанавливали схему RBAC в нашей базе данных.

Вы также можете использовать свойство db, которое должно быть либо экземпляром \yii\db\Connection, либо строковым идентификатором компонента приложения, который является экземпляром \yii\db\Connection. Используя это свойство, вы можете выполнять миграции на других базах данных. Манипулируя свойствами migrationPath и db в настройке controllerMap консольного приложения, вы можете управлять несколькими различными базами данных внутри одного и того же приложения. По умолчанию эта настройка имеет значение db, что является идентификатором подключения к базе данных по умолчанию.

И наконец, существует настройка templateFile, с которой мы поиграем в следующем разделе.

Вы можете прочитать в документации по MigrateController, что в нём есть два дополнительных метода: safeUp() и safeDown(), – которые делают то, что делают методы up() и down() соответственно, за исключением того, что делают они это в транзакциях. Однако существует очень важная особенность: используя MySQL версии как минимум до 5.5, вы можете с тем же успехом забыть об этих методах, потому что любая команда из языка определения данных будет автоматически зафиксирована (см. описание этого явления здесь: http://dev.mysql.com/doc/refman/5.5/en/implicit-commit.html). Таким образом, следующий код создаст таблицы first и second, даже если транзакция должна провалиться из-за исключения:

```
public function safeUp()
{
    $this->createTable('first',
        ['id' => 'pk', 'name' => 'string']);
    $this->createTable('second',
        ['id' => 'pk', 'value' => 'int']);
    throw new \LogicException;
    $this->createTable('third',
        ['id' => 'pk', 'value' => 'date']);
}
```

Это документированное поведение, но важность этого момента, возможно, недостаточно выделена. Итог в том, что вам просто вообще не нужно использовать эти методы, если ваша нижележащая база данных – MySQL. Более того, вам никогда не нужно переопределять одновременно и метод up(), и метод safeUp() в одном и том же классе миграции, так как метод safeUp() вызывается родительской реализацией up(). То же самое относится к методам down() и safeDown().

Создание преднастроенных шаблонов для миграций базы данных

Вот как выглядит шаблон по умолчанию для миграций базы данных в Yii 2 на момент написания этой главы:

```
use yii\db\Schema;

class <?= $className ?> extends \yii\db\Migration
{
   public function up()
   {

   }

   public function down()
   {
      echo "<?= $className ?> cannot be reverted.\n";

      return false;
   }
}
```

Давайте представим, что мы – осторожная и дисциплинированная команда разработчиков, у которой превосходная документация на все файлы исходного кода. Мы используем способную РСУБД, так что транзакции для нас – норма (не будем называть никаких названий, так как религиозные войны не входят в наши намерения), и мы достаточно строги с нашими изменениями в схеме базы данных, поэтому они почти всегда обратимы. В подобном случае нам больше подойдёт такой шаблон:

```
/**
 * TODO: Объяснение миграции.
 */
class <?= $className ?> extends \yii\db\Migration
{
   public function safeUp()
   {
      // TODO: содержимое процедуры миграции.
   }

   public function safeDown()
   {
      // TODO: содержимое процедуры отката миграции.
   }
}
```

Изменения следующие:

- внесли блок самодокументации для объяснения миграции;
- использовали транзакционные версии методов `up()` и `down()`;
- ясно отметили места, которые должны быть заполнены реальным кодом;
- убрали `return false` из процедуры отката изменений, так как наши миграции чаще будут обратимыми, нежели необратимыми.

Для того чтобы не раздувать базу кода ещё одним подкаталогом, давайте разместим этот шаблон в файле `views/layouts/migration.php`, так как подкаталог `layouts` – самое логичное место для этого. Не забудьте, что этот файл – это сценарий PHP, который будет обработан средой исполнения PHP и выведен так же, как любой другой сценарий PHP. Поэтому в дополнение к предыдущему коду файл `views/layouts/migration.php` должен также содержать следующие строчки в самом верху:

```
<?php
/**
 * Шаблон для миграций.
 * Свойство под названием `MigrateController.templateView` контролирует, какой шаблон должен использоваться.
 */
echo "<?php\n";
?>
```

В общем случае предоставление пояснений к каждому файлу исходного кода является хорошим стилем, и мы также обязаны вывести директиву обработки `<?php` в начале файла, который должен получиться в результате обработки шаблона (не забудьте, мы ведём речь о сценарии PHP, выводом которого является другой сценарий PHP).

Присоединить этот шаблон к нашему приложению, чтобы исходный код всех будущих классов миграции был основан на нём, очень просто: нам нужно использовать настройку `controllerMap.migrate.templateFile` в конфигурации консольного приложения:

```
'controllerMap' => [
    'migrate' => [
        'class' => 'yii\console\controllers\MigrateController',
        'templateFile' => '@app/views/layouts/migration.php'
    ]
]
```

Мы уже показывали этот приём в предыдущем разделе. Стоит отметить, что, к сожалению, хотя Yii и автоматически включает соответствие идентификатора `migrate` и класса `MigrateController`, нам всё равно нужно ещё раз указывать класс контроллера при переопределении настроек для этого идентификатора. Заметьте также, что настройка `templateFile` принимает еще псевдонимы путей, что очень полезно.

После всех этих приготовлений все миграции, созданные впоследствии, будут выглядеть так, как описано выше, достаточно отличаясь от стандартного вида. Как мы уже открыли ранее, определяя несколько различных идентификаторов контроллеров в настройке `controllerMap`, мы можем, если нужно, настроить несколько различных вызовов `MigrateController` с различными шаблонами.

Итоги

На этом всё, друзья.

Эта глава, последняя в книге, разбирала наивысший слой Yii 2: обслуживание самой базы кода. Здесь мы обсудили три темы:

- как динамически собирать конфигурацию приложения так, чтобы разработчики могли легко развёртывать его на любую машину;
- как делать свои собственные консольные команды для получения возможности делать некоторые трюки в помощь разработке;
- как изменить внешний вид автоматически сгенерированных сценариев миграции.

Мы также изучили, как реализованы в Yii 2 миграции базы данных и консольные команды в целом. И не будем забывать о том, что теперь мы знаем ещё о двух вспомогательных классах: `ArrayHelper` и `Console`.

Если честно, мы прошли мимо многих тем, и много информации было просто не представлено, потому что являлось бы копированием без изменений из уже существующей документации Yii. Для того чтобы ещё больше развить беглость в обращении с этим фреймворком, вам будет полезно действительно глубоко изучить следующие продвинутые возможности, помимо кратких упоминаний о них в этой книге:

- на слое данных: валидаторы, `ActiveQuery` и `\yii\db\Command`;
- на слое отображения: встроенные виджеты, всё пространство имён `yii\rest` целиком, поддержка загрузки файлов, методы виджета `ActiveForm` для отрисовки полей ввода, а также, на удив-

ление, класс `\yii\captcha\Captcha`, являющийся законченным решением для добавления капчи в HTML-формы;

- промежуточные компоненты: контейнер внедрения зависимостей в пространстве имён `yii\di`, мьютексы в пространстве имён `yii\mutex`, множество вспомогательных классов, а также поддержка интернационализации приложения.

Документация по Yii прекрасна и всеобъемлюща, и блоки самодокументации в исходном коде больше похожи на определяющий справочник. В этой книге мы постарались покрыть темы, которые либо не очевидны из этих источников информации, либо, по какой-либо причине, не были выявлены из нижележащего исходного кода.

Приложение 1

Настройка развёртывания с использованием Vagrant

Так как достаточно важно, чтобы вы могли разрабатывать приложение локально, на своей рабочей станции, давайте использовать проект Vagrant для настройки и установки локальной цели развёртывания.

Если совсем коротко, то Vagrant (см. http://www.vagrantup.com/) – это набор инструментов для администрирования виртуальных машин из командной строки. Конечным результатом настройки будет следующее:

1. Вы включаете свою рабочую станцию.
2. Переходите в корневой каталог вашего проекта.
3. Выполняете команду `vagrant up` в командной строке.
4. Ждёте немного и затем открываете в браузере URL http://localhost:8888/.
5. Ваше веб-приложение вам там отвечает.
6. Вы выполняете команду `vagrant halt` в командной строке.

Ваше веб-приложение больше не доступно и более не тратит никаких ресурсов.

Все зависимости приложения, включая веб-сервер и СУБД, находятся внутри виртуальной машины, которой набор инструментов Vagrant неявным образом управляет. Ничто не оказывает влияния на вашу хост-систему.

Vagrant поддерживает нескольких поставщиков виртуализации, так что у нас не будет никаких проблем, если мы воспользуемся про-

ектом Virtualbox (см. https://www.virtualbox.org/), так как обе эти технологии являются свободными для использования проектами с открытым исходным кодом.

Планирование

Так как мы будем настраивать виртуальную машину, нам нужно в деталях продумать наш стек LAMP с самого начала.

Нам нужно установить четыре вещи:

1) PHP 5.4+;
2) Apache 2.4+ (это современный стандарт в любом случае);
3) MySQL 5.5+ (нам нужна база данных, так как управление некоторыми данными – предметная область нашего приложения);
4) веб-сайт, настроенный для доступа снаружи виртуальной машины.

Более того, нам нужен сценарий установки, который автоматизирует необходимые приготовления базовой системы, включая установку всех упомянутых компонентов.

Развёртывание благодаря Vagrant будет крайне упрощено, так как он просто делает каталог, содержащий файлы проекта (тот каталог, откуда вы будете вызывать `vagrant up`), общим с виртуальной машиной с точкой доступа по адресу `/vagrant`. Косая черта имеет значение, так как это папка под названием `vagrant` прямо в корневом каталоге файловой системы виртуальной машины. В результате база кода всегда будет синхронизирована между целью развёртывания и рабочей станцией, так что вы можете использовать любые инструменты, какие вы используете для разработки и сопровождения кода, и он будет прозрачно и непрерывно отправляться на цель развёртывания.

Vagrant прячет управление виртуальной машиной за концепцией «коробки» («box»). «Коробка» – это специальным образом подготовленный образ виртуальной машины в любом формате, поддерживаемом Vagrant. Необходимо, чтобы «коробку» можно было скачать откуда-то. Веб-сайт по адресу http://www.vagrantbox.es/ содержит ссылки на множество таких подготовленных образов.

Однако есть «коробки», которые Vagrant распознаёт и может установить сразу, без дополнительных настроек. Одна из них называется `precise64` и является образом Ubuntu 12.04. Для простоты мы воспользуемся именно этой коробкой.

Начальная настройка

До того, как мы углубимся в объяснения, давайте создадим необходимую конфигурацию для Vagrant. Создайте файл под названием `Vagrantfile` в корневом каталоге проекта и напишите в нём следующий код:

```
Vagrant.configure("2") do |config|

# Какую коробку мы будем использовать как основу
  config.vm.box = "hashicorp/precise64"

# Справочная информация, так как Vagrant и так знает,
# откуда достать коробку "precise64".
#  config.vm.box_url = "http://files.vagrantup.com/precise64.box"

# Что делать с базовой коробкой в качестве первоначальной настройки
  config.vm.provision :shell, :path => "bootstrap/01-prepare-precise64.sh"
  config.vm.provision :shell, :path => "bootstrap/02-configure-app-for-
precise64.sh"
  config.vm.provision :shell, :path => "bootstrap/03-prepare-application.
sh"

# Как сделать веб-приложение в коробке видимым снаружи:
# опубликовать порт 80 в виртуальной машине как порт 8888 на хосте.
  config.vm.network "forwarded_port", guest: 80, host: 8888

end
```

Это код на языке Ruby (см. `https://www.ruby-lang.org/`). Если хотите, вы можете писать в нём произвольные выражения, до тех пор, пока в нём также находится вызов `Vagrant.configure`.

С этим файлом на своём месте вы можете наконец совершать магию команды `vagrant up`. Когда вы выполните эту команду впервые, на неподготовленной машине и базе кода, Vagrant сделает следующее:

1) скачает коробку из предоставленного `box_url`. В нашем случае Vagrant уже знает, где находится коробка под названием `hashicorp/precise64`;
2) распакует коробку и зарегистрирует её как виртуальную машину в среде Virtualbox. Распакованная коробка будет сохранена в домашнем каталоге текущего пользователя и будет заново использована во всех последующих вызовах `vagrant up`;

3) запустит виртуальную машину, используя обычные средства Virtualbox. Если вам захочется, вы даже можете открыть пользовательский интерфейс управления Virtualbox и увидеть в списке среди прочих машин (если таковые имеются) виртуальную машину под управлением Vagrant;
4) запустит сценарии подготовки («provision scripts»), то есть выполнит всё, перечисленное в настройке `config.vm.provision`. В нашем случае подготовка разделена на три сценария, которые должны быть выполнены точно по порядку;
5) осуществит проброс портов согласно тому, как мы указали.

Он также может сделать ещё некоторые вещи, но мы заинтересованы только в этих этапах.

При всех последующих вызовах команды `vagrant up` Vagrant будет просто запускать виртуальную машину и пробрасывать порты. Как уже было сказано, исходный код будет постоянно обновляться в виртуальной машине, согласно изменениям на базовой.

Очевидно, что Vagrant предоставляет крайне полезное окружение для локальной разработки. Самая сложная часть – правильно составить сценарии подготовки.

Тонкая настройка виртуальной машины

Как указано в `Vagrantfile`, мы собираемся иметь три сценария подготовки, которые должны выполняться в определённом порядке. Давайте в том же порядке их рассмотрим.

В пакете кода, прилагающемся к этой книге, эти сценарии находятся в подкаталоге `bootstrap` с теми же именами, что и в вышеуказанном примере кода. В этом же подкаталоге находится также набор других файлов; это различные файлы конфигурации для программ, которые Vagrant установит согласно сценариям подготовки.

Подготовка гостевой ОС

Мы используем Ubuntu 12.04, в репозиториях которой нет PHP 5.4. С другой стороны, у нас появляется окружение, в котором доступен `apt-get`, что означает, что мы можем относительно легко установить абсолютно всё, что нам может понадобиться.

Мы не покажем полное содержимое файла `01-prepare-precise64.sh`, так как он довольно длинный и скучный. Лучше посмотрите сами в этот файл в подкаталоге `bootstrap`. Вот что этот сценарий делает, по порядку:

1) добавляет специальный PPA (Personal Package Archive, репозиторий пакетов в стиле Ubuntu), содержащий последние версии PHP;
2) обновляет базу пакетов системы `apt-get`;
3) устанавливает последние версии Apache 2, PHP 5.4, vim, новейший MySQL (и сервер, и клиент), git (он требуется для запуска Composer из виртуальной машины) и пакет CURL для PHP 5 (он в наши дни необходим практически для всего, что связано с управлением библиотеками PHP);
4) устанавливает виртуальный буфер экрана X, среду исполнения Java и браузер Firefox, чтобы было можно выполнять приёмочные тесты непосредственно изнутри коробки Vagrant;
5) удаляет виртуальный хост по умолчанию, созданный Apache при установке;
6) включает модуль mod_rewrite для Apache, так как он необходим для создания красивых URL в Yii 2;
7) настраивает Apache так, чтобы он выполнялся под учётной записью Vagrant (которая так и называется, `vagrant`), чтобы он мог писать в подкаталог `web` нашего проекта.

Делает небольшую дополнительную настройку, чтобы убрать стандартное предупреждение Apache о том, что имя сервера по умолчанию не определено.

По сути, этот сценарий настраивает платформу, не само приложение. Однако, так как мы для простоты будем использовать пароль суперпользователя для доступа к базе данных, а базу данных мы создаём здесь, то этот же пароль придётся упомянуть и на следующем этапе подготовки.

Чтобы комфортабельно работать с Vagrant поверх Virtualbox, вам следует знать про плагин VirtualBox Guest Additions. Vagrant полагается на этот плагин при осуществлении своей магии синхронизации базы кода, и подвох в том, что этот плагин должен быть установлен как на хост-, так и на гостевой машине, *и их версии должны в точности совпадать*. Скорее всего, Virtualbox Guest Additions на вашей хост-машине будет новее, чем тот, что установлен в гостевой «коробке». В результате Vagrant просто выругается, и ничего хорошего не произойдёт. Чтобы разрешить эту проблему, команда Vagrant предоставляет нам свой собственный плагин под названием vagrant-vbguest. Для его установки нужно выполнить команду `vagrant plugin install vagrant-vbguest`. После этого при каждом вызове `vagrant up` этот плагин будет сверять версии Guest Additions и устанавливать корректную версию на гостевую машину, если нужно (это, правда, может занять значительное количество времени).

Подготовка базы данных и веб-сервера

Второй уровень подготовки – это мост между платформой и приложением. Он довольно короткий:

```
# Отдельно определяем настройки БД
# ЗАМЕТЬТЕ, что пароль уже был указан ранее в предыдущем сценарии под-
готовки!
DB_USER=root
DB_PASS=mysqlroot
DB_NAME=crmapp

# Создаём БД
# ЗАМЕТЬТЕ отсутствие пробела между флагом`-p` и паролем!
mysql -u ${DB_USER} -p${DB_PASS} -e "create database if not exists
`${DB_NAME}` default character set utf8";
mysql -u ${DB_USER} -p${DB_PASS} -e "create database if not exists ${DB_
NAME}_test default character set utf8 default collate utf8_unicode_ci";

# Копируем заранее составленный конфиг Apache из базы кода в каталог
файлов настроек Apache.
cp -f /vagrant/bootstrap/frontend.apache2.conf /etc/apache2/sites-enabled/
# Перезапускаем Apache, чтобы опубликовать этот виртуальный хост.
/etc/init.d/apache2 restart
```

Всё, что мы здесь делаем, – это создаём базы данных для самого приложения и для функциональных тестов, затем копируем заранее подготовленные настройки Apache из подкаталога `bootstrap` в папке проекта в то место, которое Apache ожидает. Этот файл настроек содержит определение виртуального хоста, основанное на портах, что и послужило причиной удаления конфигурации по умолчанию на предыдущем уровне (мы фактически заменили хост по умолчанию на наш собственный, сохранив порт доступа неизменным).

Подготовка приложения

Последний сценарий подготовки наиболее сложен по смыслу. Вот его содержимое на последнем этапе разработки примера приложения, после *главы 13*:

```
# Переходим в корневой каталог приложения
cd /vagrant

# Устанавливаем все предзависимости, включая Yii 2
php composer.phar install --prefer-dist
```

```
# Копируем подготовленный фрагмент конфигурации в дерево конфигурации
cp bootstrap/local.php config/overrides/

# Копируем подготовленный фрагмент конфигурации для тестов
cp bootstrap/test.php config/

# Инициализируем таблицы для поддержки RBAC
./yii migrate --migrationPath='@yii/rbac/migrations' --interactive=0

# Инициализируем базу данных в целом
./yii migrate –interactive=0
```

Здесь мы делаем следующее:

1) устанавливаем все зависимости, управляемые Composer, включая сам Yii 2;
2) копируем конфигурацию для этой конкретной цели развёртывания в то место, которого ожидает наше приложение (см. подробности в *главе 13*);
3) запускаем встроенную в Yii миграцию, которая подготавливает таблицы для менеджера RBAC, основанного на базе данных (см. подробности в *главе 6*);
4) запускаем все миграции, собранные в течение этой книги.

Все эти три сценария относительно безвредны, и в результате вы можете запускать подготовку вручную, без опасности сломать что-либо. Это делается вызовом команды `vagrant provision`. Это бывает полезно, например если нужно быстро перезагрузить настройки Apache.

Использование виртуальной машины в качестве локальной цели развёртывания

В случае если вы ещё это не прочитали в документации, вы останавливаете виртуальную машину, управляемую Vagrant, вызывая команду `vagrant halt` из корневого каталога, который содержит файл `Vagrantfile`. Если вы хотите полностью избавиться от машины, то вызываете `vagrant destroy`. Это может быть полезно, когда вы сами будете подготавливать «коробку» для работы.

Автоматический обмен кодом между гостевой и хост-машиной избавляет вас от необходимости вручную развёртывать базу кода. Фактически ваше действие развёртывания сокращается до запуска миграций время от времени.

Вы получаете доступ к локальной цели развёртывания, выполняя команду `vagrant ssh` в корневом каталоге проекта, после чего вы можете делать там всё, что захотите. С теми настройками MySQL, которые были показаны выше, вы можете получить доступ к базе данных внутри виртуальной машины, выполнив:

```
$ mysql -u root -pmysqlroot crmapp
```

Не забывайте, что ваша база кода расположена не в домашнем каталоге пользователя `vagrant`, куда вы попадаете после вызова `vagrant ssh`, а в каталоге `/vagrant`, так что выполнение `cd /vagrant` сразу после входа в гостевую систему легко может стать вашей привычкой.

Наиболее сложная часть – это наборы автоматических тестов, аккуратно составленных на протяжении этой книги.

Как было сказано ранее в *главе 3*, выполнение приёмочных тестов слишком много раз (в нашем случае более 10 раз) переполнит таблицы в CRUD-интерфейсах, которые настроены так, чтобы показывать только 20 элементов на странице. После этого тесты перестанут выполняться успешно.

Так что от вас требуется вручную очищать базу данных после приёмочных тестов, так как у них нет никакой возможности убирать за собой самостоятельно.

В пакете кода, прилагающемся к этой книге, находится сценарий под названием `reset_database.sh`, который вам в этой задаче поможет. Он содержит всего четыре функциональные строчки кода:

```
# Уничтожить и заново создать базу данных
mysql -u root -pmysqlroot -e "drop database if exists crmapp; create
database crmapp default character set utf8 default collate utf8_unicode_
ci";

# Восстановить таблицы RBAC в пустой базе данных
./yii migrate --interactive=0 --migrationPath='@yii/rbac/migrations'

# Запустить все наши миграции
./yii migrate --interactive=0

# Создать дамп данных для Codeception
mysqldump -u root -pmysqlroot crmapp > tests/_data/dump.sql
```

В результате выполнение этого сценария полностью перестроит вашу БД в чистое состояние. Не используйте что-то похожее на этот сценарий в приложении, развёрнутом на «боевом» сервере!

Второй вспомогательный сценарий называется `selenium.sh`. Этот сценарий запускает локальный экземпляр сервера Selenium (см. http://docs.seleniumhq.org/), настроенного таким образом, что вы можете запускать приёмочные тесты прямо оттуда, из «коробки» Vagrant. Это не так надёжно, как использование реального удалённого подключения к отдельной цели развёртывания (потому что соединение устанавливается к локальной машине, в обход DNS и прочих вещей вроде прокси-серверов). Однако это удобно, поскольку вы сможете использовать приёмочные тесты сразу после запуска виртуальной машины, без предварительной подготовки хост-машины. Запускайте этот сценарий в отдельной командной строке, а затем запускайте приёмочные тесты в другой.

Третий вспомогательный сценарий называется `minify_assets.sh`, и он просто выполняет корректную команду `./yii asset`, которая была описана в *главе 8*, в разделе про минификацию материалов.

В целом пакет кода, прилагаемый к этой книге, настроен таким образом, чтобы вы могли запустить «коробку» Vagrant командой `vagrant up` и затем сразу же иметь возможность выполнить полный набор тестов, используя одну команду:

```
$ ./cept run
```

Этот сценарий «`cept`» – сокращение для вызова настоящего исполняемого файла Codeception в подкаталоге `vendor`.

Приложение 2

Пример Active Form

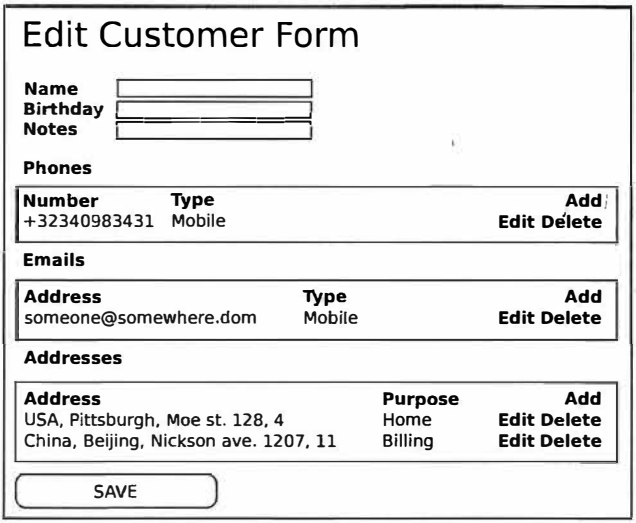
Когда мы в главе 11 делали пользовательский интерфейс в стиле Yii, мы сфокусировали наше внимание только на виджете `GridView`, так как он был темой главы. Но в любом веб-приложении есть другая важная часть – HTML-формы.

Yii 2 предоставляет очень функциональный и удобный в использовании виджет под названием `ActiveForm`, который может полуавтоматически создавать для нас HTML-формы на основе экземпляра `ActiveRecord`, описывающего модель, с которой мы работаем. Здесь мы просто продолжим начатое в предыдущей главе.

Создание формы редактирования клиента

Наша модель предметной области, описывающая клиента, физически разделена на четыре таблицы в базе данных. Перед нами встаёт серьёзная проблема разработки удобного интерфейса для редактирования этой конструкции.

У нас нет ни нужного объёма книги, ни намерения, необходимых для реализации какого-нибудь напичканного JavaScript богатого интерфейса, даже если он, несомненно, будет более отзывчивым и визуально приятным. Давайте поступим в духе старой школы и сделаем веб-интерфейс, традиционный для мира статических HTML-страниц. Вот его набросок:



Таким образом, телефоны, почтовые адреса и адреса электронной почты будут представлены в виде таблиц, с кнопками «Добавить», «Редактировать» и «Удалить». Эти таблицы будут вести себя точно так же, как пользовательский интерфейс, использованный на маршрутах `/user/index` и `/services/index`, которые мы сделали при помощи автоматического генератора CRUD Gii. Кнопки «Добавить» и «Редактировать» будут переносить нас на страницы добавления/редактирования, соответствующие подчинённой модели, которую мы ходим добавить/отредактировать, и после нажатия на «Сохранить» на этих страницах мы будем перенесены обратно на страницу формы редактирования клиента.

ВОЗМОЖНОСТЬ: Active Query

В чём заключается понятие «активного запроса» (**Active Query**), который мы уже несколько раз использовали в течение этой книги, иногда явно, иногда неявно? Это предметно-ориентированный язык (DSL) для формулирования запросов к базе данных с целью получения активных записей, которые, возможно, ещё и связаны друг с другом. С одной стороны, он скрывает сложности сборки правильного SQL, а с другой – скрывает сложности сборки экземпляров активных записей из сырых данных, полученных от БД. Эта концепция реализована в виде класса `\yii\db\ActiveQuery`, который является расширением класса `\yii\db\Query`.

В то время как обычный Query возвращает данные из БД в виде ассоциативных массивов, ActiveQuery специально построен таким образом, что возвращает перечислимые коллекции активных записей, что полезно, когда вы на уровне выше, чем сырые данные из БД. Конечно же, создание полного экземпляра ActiveRecord для каждой записи, возвращённой запросом, – дорогая операция, так что это удобство идёт в комплекте с ухудшением производительности. Однако обычно гораздо разумнее в первую очередь думать об архитектуре и настраивать производительность позднее, когда нужно, потому что если вы не используете активных записей или паттерна Репозиторий поверх активных записей, вы будете использовать API для прямого доступа к БД, и это в долгосрочной перспективе намного сложнее сопровождать.

Класс ActiveQuery в конечном счёте *очень* большой. Он наследует от класса Query, который использует особенность \yii\db\QueryTrait, и сам использует две дополнительные особенности, \yii\db\ActiveQueryTrait и \yii\db\ActiveRelationTrait. Можно написать целую книгу только ради объяснения деталей использования класса ActiveQuery в Yii 2. Вам настоятельно рекомендуется взглянуть на документацию и исходный код этого класса и исследовать его самостоятельно.

Нам не нужна вся функциональность ActiveQuery. В качестве простого выразительного примера вот как вы можете получить клиентов, которых можно на этой неделе поздравить с днём рождения, но только тех, которые были зарегистрированы менеджером, авторизованным в данный момент:

```
    $week_ago = (new DateTime)->sub(new DateInterval('P1W'))->format(
'Y-m-d' );
    $current_user = Yii::$app->user->id;
    $customers = CustomerRecord::find()
        ->where(
            ['and', 'created_by=:current_user', 'birth_date>=:week_ago'],
            compact('current_user', 'week_ago')
        )->all();
```

Что этот код делает? Вот что:

- Вызов к ActiveRecord.find() возвращает экземпляр ActiveQuery, сконфигурированный для поиска активных записей класса CustomerRecord.
- Его метод where() настраивает ActiveQuery на то, чтобы отобрать только записи, соответствующие двум нашим условиям.
- Наконец, метод all() возвращает нам массив активных записей.

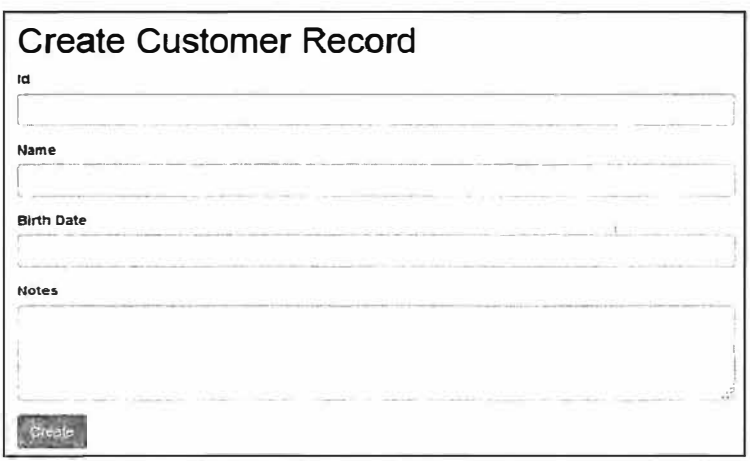
Встроенную в PHP функцию под названием `compact()` мы уже давным-давно упоминали.

Это – обычный способ использования `ActiveQuery`: мы «впадаем» в «режим запроса», вызывая метод `find()` класса `ActiveRecord`, и когда мы заканчиваем сцеплять все нужные нам методы, мы «вытаскиваем» наши активные записи, «возвращаясь» из DSL `ActiveQuery`.

Второй метод использования `ActiveQuery` – это создание вручную и передача его в методы, которые его ожидают. Одно из таких мест – это экземпляры `DataProvider`, которые мы описывали ранее в главах 2 и 11.

Настройка автоматически созданной формы

Давайте посмотрим, как в данный момент выглядит форма создания записи о клиенте, открыв страницу `/customer-records` и нажав на большую зелёную кнопку **Create Customer Record** над таблицей:



Как она реализована? Отслеживая маршрут `/customer-records/create`, мы оказываемся в файле представления `views/customer-records/create.php`, который вызывает файл `_form.php` из того же каталога. Этот файл – то, что управляет отрисовкой данной формы создания. Можно заметить, что файлы `update.php` и `create.php` выглядят практически одинаково и используют одну и ту же форму из сценария `_form.php`. Это сделано намеренно, так как в идеологии Yii 2 создание и обновление активных записей – очень похожие действия.

Если вы откроете тот файл представления, то увидите достаточно простую структуру, игнорируя код HTML:

- мы начинаем ActiveForm вызовом `$form = ActiveForm::begin();`
- затем отрисовываем поля ввода вызовами `$form->field($model, $attr)->textInput($setup);`
- потом делаем немного магии, для того чтобы показать либо кнопку **Create**, либо кнопку **Update** отправки формы, с правильными классами CSS;
- затем мы завершаем ActiveForm вызовом `ActiveForm::end()` (обратите внимание на то, что мы вызываем статический метод класса, а не метод объекта `$form`).

Вначале давайте изменим раскладку этой формы на более «горизонтальную», согласно нашему наброску. Так как у нас уже встроено расширение `yii2-bootstrap`, сделать это крайне просто. Просто нужно объявить, что наша форма принадлежит не классу `\yii\widgets\ActiveForm`, а классу `\yii\bootstrap\ActiveForm`. Это можно сделать в блоке предложений `use` вверху файла представления. Найдите следующее определение:

```
use yii\widgets\ActiveForm;
```

и замените его на такое определение:

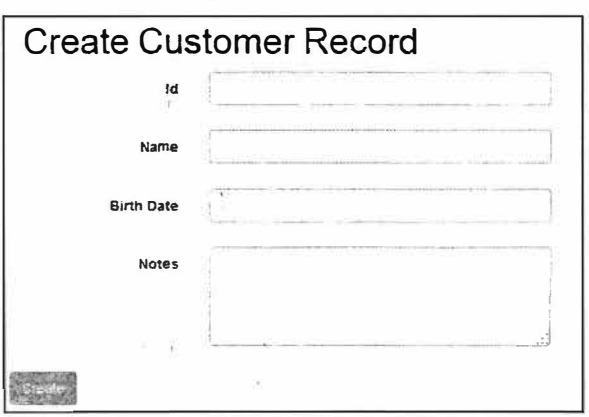
```
use yii\bootstrap\ActiveForm;
```

Подменив это объявление, нам даже не нужно будет ничего менять в коде самого представления.

После этого нам нужно модифицировать вызов `ActiveForm::begin()`, добавив настройку `layout` в конфигурацию виджета:

```
<?php $form = ActiveForm::begin(['layout' => 'horizontal']);?>
```

На этом всё, вот как теперь выглядит наша форма:



Нам определённо не нужна возможность редактировать идентификатор клиента, так что нам надо удалить следующую строку:

```
<?= $form->field($model, 'id')->textInput() ?>
```

Здесь нам нужно вставить важное защитное условие. Для того чтобы создать новую запись о клиенте, нам нужно удалить все подчинённые таблицы с телефонами, почтовыми адресами и адресами электронной почты, потому что для их отображения нам нужен идентификатор записи клиента, а он будет присвоен только после того, как запись будет сохранена в базе данных.

Вставьте следующие скобки `if-endif` в файл `_form.php`:

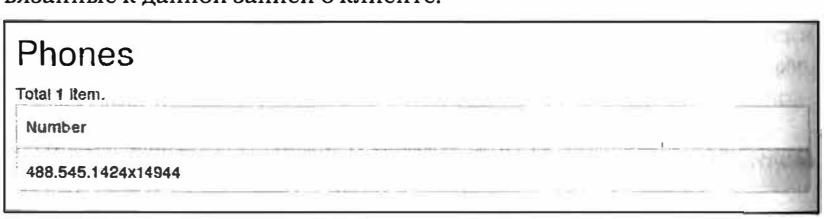
```
<?= $form->field($model, 'notes')->textarea(['rows' => 6]) ?>

<?php if (!$model->isNewRecord):?>
<!-- здесь будут подчинённые таблицы… -->
<?php endif?>

<div class="form-group">
```

В случае если запись о клиенте – новая, это предотвратит отрисовку таблиц, которые мы опишем в дальнейшем. Подразумевается, что все последующие примеры кода будут находиться внутри этих скобок!

Теперь вернёмся к теме этой главы и добавим GridView для записей о телефонах, связанных с редактируемым клиентом. Мы начнём с простого:

```
<h2>Phones</h2>
<?= \yii\grid\GridView::widget([
    'dataProvider' => new \yii\data\ActiveDataProvider([
        'query' => $model->getPhones(),
        'pagination' => false
    ]),
    'columns' => ['number']
]);?>
```

Это даст нам следующую таблицу, если у нас есть телефоны, привязанные к данной записи о клиенте:



Мы воспользуемся методом отношений `CustomerRecord.getPhones()`, созданным в главе 11, чтобы вернуть экземпляр `ActiveQuery` для поиска связанных экземпляров `PhoneRecord`.

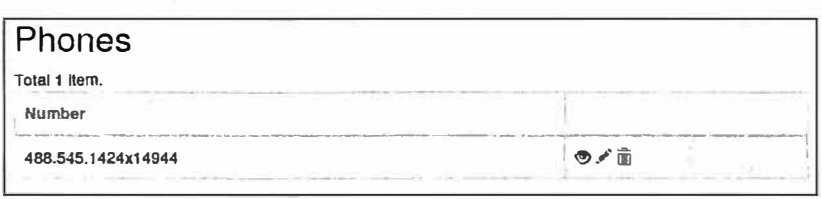
Нам не нужна разбивка на страницы в этой таблице, так как количество строчек, скорее всего, всё равно будет небольшим. Хотя виджет `GridView` и имеет настройку `pager`, она предназначена для настройки подчинённого виджета `\yii\widgets\LinkPager`, который отвечает только за отрисовку всех этих стрелочек и кнопочек с циферками для перемещения между страницами. Мы же, напротив, хотим отключить само *понятие* «разбивки на страницы» для этого списка телефонов. Поэтому нам нужно использовать настройку `pagination` в классе `ActiveDataProvider` на одну абстракцию глубже.

Нам не нужно ничего, кроме номера телефона, поэтому только одна колонка явно объявлена. Без явного объявления колонок на странице создания новой записи о клиенте эта `GridView` сломает весь интерфейс, потому что попытается получить какой-нибудь объект `PhoneRecord`, чтобы узнать, какие у него имеются поля, и не сможет этого сделать, потому что `CustomerRecord` ещё не присвоен идентификатор.

Теперь мы займёмся настоящей магией и привяжем эту таблицу к CRUD, который мы подготовили для класса `PhoneRecord` в главе 11. Для этого нам нужно добавить специальную колонку с кнопками, соответствующими действиям, которые можно совершить с соответствующими активными записями о телефонах:

```
<?= \yii\grid\GridView::widget([
    // ...
    'columns' => [
        'number',
        [
            'class' => \yii\grid\ActionColumn::className(),
        ]
    ]
]);?>
```

Вот как эта колонка будет выглядеть:



Если посмотреть, каким ссылкам соответствуют эти иконки, можно увидеть, что это действия `view`, `update` и `delete`, которые нам нужны, но они привязаны к текущему контроллеру; нам нужен контроллер `phones` вместо контроллера `customer-records`. Это на удивление просто решается, так как у класса `ActionColumn` есть настройка, которая явно указывает, к какому контроллеру она должна считать себя относящейся:

```
<?= \yii\grid\GridView::widget([
    // ...
    'columns' => [
        'number',
        [
            'class' => \yii\grid\ActionColumn::className(),
            'controller' => 'phones'
        ]
    ]
]);?>
```

Наконец, давайте добавим кнопку **Add Phone**. Согласно нашему наброску, она находится прямо в шапке колонки с кнопками:

```
<?= \yii\grid\GridView::widget([
    // ...
    'columns' => [
        'number',
        [
            'class' => \yii\grid\ActionColumn::className(),
            'controller' => 'phones',
            'header' => Html::a('Add New', ['phones/create']),
        ]
    ]
]);?>
```

Давайте также добавим иконку с изображением знака «плюс» рядом с надписью **Add New**:

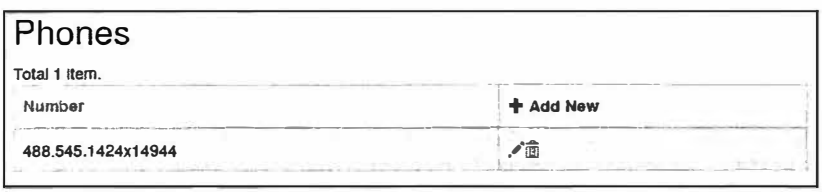
```
'header' => Html::a(
    '<i class="glyphicon glyphicon-plus"></i> Add New',
    ['phones/create']
),
```

Не то, чтобы это был очень чистый код, но, по крайней мере, мы сделали, что хотели.

Нам не нужна кнопка для просмотра записи о телефонном номере (та, что с символом глаза), поскольку сам номер и так показан в таблице, а других данных в этой записи нет. Посмотрите, как это делается при помощи свойства `ActionColumn.tempate`:

```
<?= \yii\grid\GridView::widget([
   // ...
   'columns' => [
      'number',
      [
         'class' => \yii\grid\ActionColumn::className(),
         // ...
         'template' => '{update}{delete}',
      ]
   ]
]);?>
```

Теперь таблица выглядит превосходно:



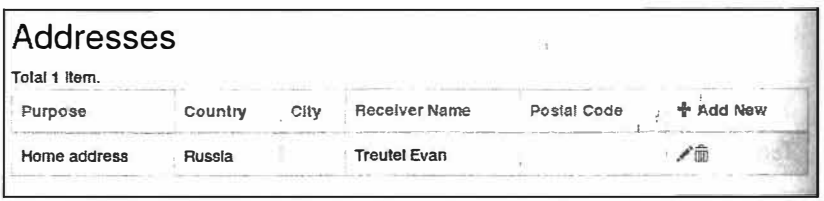
Ширина колонки может быть скорректирована применением некоторого количества CSS, о чём мы не будем беспокоиться.

Затем мы добавляем таблицу с почтовыми адресами, следующего вида:

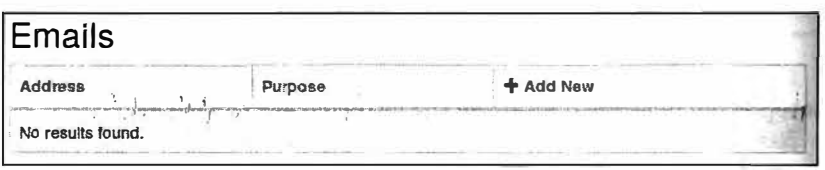
```
<h2>Addresses</h2>
<?= \yii\grid\GridView::widget([
   'dataProvider' => new \yii\data\ActiveDataProvider(
      ['query' => $model->getAddresses(), 'pagination' => false]
   ),
   'columns' => [
      'purpose',
      'country',
      'city',
      'receiver_name',
      'postal_code',
      [
         'class' => \yii\grid\ActionColumn::className(),
         'controller' => 'addresses',
         'template' => '{update}{delete}',
         'header' => Html::a(
            '<i class="glyphicon glyphicon-plus"></i> Add New',
            ['addresses/create']
         ),
```

```
        ],
    ],
]);?>
```

Части, отличающиеся от таблицы телефонов, выделены. Этот код должен выдать следующую таблицу:

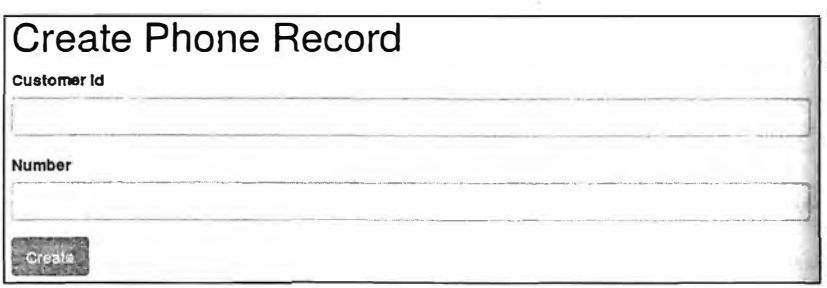


И таким же образом мы делаем таблицу с адресами электронной почты, которую вы можете собрать самостоятельно, если мы покажем вам вот этот конечный результат в качестве образца:



Передача идентификатора клиента в подчинённые модели

У нас есть небольшая проблема в нашем пользовательском интерфейсе. Давайте нажмём на эту новую кнопку **Add New**, которую мы только что создали для таблицы с телефонами:



Мы не передаём создаваемой записи о телефоне в идентификатор клиента! И нам вообще не нужно поле для его ввода, если мы будем

его передавать автоматически. Нам нужен какой-то способ передачи идентификатора главной модели в форму создания новой записи о подчинённой модели.

Простейшим способом будет модификация метода `SubmodelController.actionCreate()` таким образом, что он будет принимать дополнительный параметр. Давайте назовём его в обобщённом виде:

```
public function actionCreate($relation_id)
```

И затем в конфигурации для заголовка этой последней колонки с кнопками мы добавим к создаваемой ссылке параметр `relation_id`:

```
'header' => Html::a(
    '<i class="glyphicon glyphicon-plus"></i> Add New',
    ['phones/create', 'relation_id' => $model->id]
),
```

После объяснений в главе 12 должно быть понятно, что аргументы методов контроллеров, названия которых начинаются на `action`, становятся обязательными параметрами `GET` или `POST` запросов для соответствующих маршрутов, с теми же именами.

Затем мы можем корректно разместить данный «идентификатор отношения» `relation_id` в создаваемую запись:

```
public function actionCreate($relation_id)
{
    /** @var ActiveRecord $model */
    $model = new $this->recordClass;
    $model->customer_id = $relation_id;
// … остальной код ...
```

Однако, так как мы уже назвали входной аргумент в обобщённом виде, давайте абстрагируемся от концепции «клиента» и сделаем целевое поле также обобщённым:

```
    $model->{$this->relationAttribute} = $relation_id;
```

Это ещё один пример возможностей метапрограммирования в PHP, так как мы только что использовали строку, сохранённую в переменной, как имя свойства объекта (чей класс сам только что был выведен на основе строки, сохранённой в другой переменной). Конечно же, нам теперь придётся объявить это свойство:

```
/** @var string Название атрибута, который будет хранить данные ID-
отношения */
public $relationAttribute;
```

А также исправить определение контроллеров AddressesController, EmailsController и PhonesController, добавив туда следующее объявление:

```
public $relationAttribute = 'customer_id';
```

Вот пример вреда, причинённого излишним обобщением в вашем коде. То, что началось как вроде бы небольшое красивое обобщение, теперь стоит нам тройным дублированием кода. И мы также не можем выставить customer_id в качестве значения по умолчанию, потому что наш SubmodelController полностью отделён от предметной области, и если мы так сделаем, то разрушим абстракцию, что ещё хуже, чем простое дублирование кода.

Мы теперь можем удалить поля ввода для свойства customer_id из файлов представлений views/addresses/_form.php, views/emails/_form.php и views/phones/_form.php тем же образом, каким мы удалили поле id из формы обновления клиента.

Возвращение в форму редактирования клиента после редактирования подчинённой модели

Это, впрочем, ещё не всё. Когда мы обновляем или создаём новый телефон, адрес или адрес электронной почты, после нажатия на кнопку отправки формы нас перенаправляют на страницы /phone/view, /address/view или /email/view соответственно, что отнюдь не является тем, что нам нужно. Лучше, если нас вернут на ту страницу, где мы были до этого, то есть на страницу редактирования записи о клиенте. Это довольно просто сделать, используя инструменты, которые нам предоставляет Yii 2.

Вначале нам нужно сохранить URL страницы редактирования, сделав следующее в методе \app\controllers\CustomerRecordsController::actionUpdate():

```
$this->storeReturnUrl();
```

Это понятное человеку название для функции, которую мы определяем следующим образом:

```
private function storeReturnUrl()
{
    Yii::$app->user->returnUrl = Yii::$app->request->url;
}
```

Здесь мы сохраняем URL, который нам возвращает вызов \yii\web\Request.getUrl(), вызовом метода \yii\web\User.setReturnUrl(), испол-

зуя красивый синтаксис, предоставленный нам магическими методами `__get()` и `__set()` из Yii 2.

Затем мы снова переходим в контроллер `SubmodelController`, к методу `actionCreate()`, и находим следующее перенаправление:

```
return $this->redirect(['view', 'id' => $model->id]);
```

Нам нужно заменить его на следующее:

```
return $this->goBack();
```

Это и проще, и делает то, что нам нужно. Метод `\yii\web\Controller.goBack()` делает перенаправление на URL, сохранённый в `\yii\web\User.getReturnUrl()`, а это в точности то, что мы только что сохранили. Документация для этого свойства компонента `User` утверждает, что это URL, на который мы перенаправляем пользователя после успешного входа в систему, но на самом деле мы можем вызвать `goBack()` откуда угодно, так что этот `returnUrl` на самом деле общего назначения.

Нужно сделать ту же самую замену в методе `actionUpdate()`. Дополнительно внутри метода `actionDelete()` нужно заменить следующее перенаправление:

```
return $this->redirect(['index']);
```

на это перенаправление:

```
return $this->goBack();
```

Или иначе после нажатия на кнопку **Delete** возле любого телефона, адреса или адреса электронной почты и последующего подтверждения удаления система попытается перенаправить вас на несуществующее действие `actionIndex()` соответствующей модели.

Преднастроенное значение колонки адреса

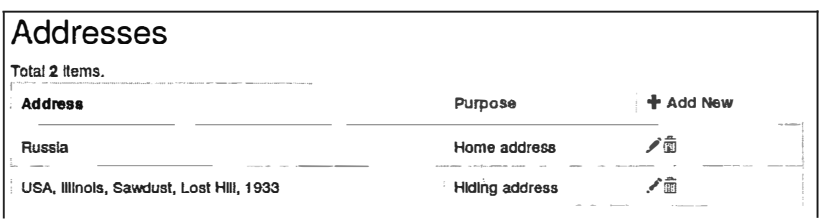
У нас осталось только одно различие между исходным наброском и текущим состоянием формы редактирования клиента: внешний вид таблицы почтовых адресов. Нам нужна одна колонка, в которой весь адрес будет записан в одну строчку.

Это решается очень просто, так как мы уже делали точно то же самое в главе 11. Воспользуемся же чудесным свойством `Column.value` вновь:

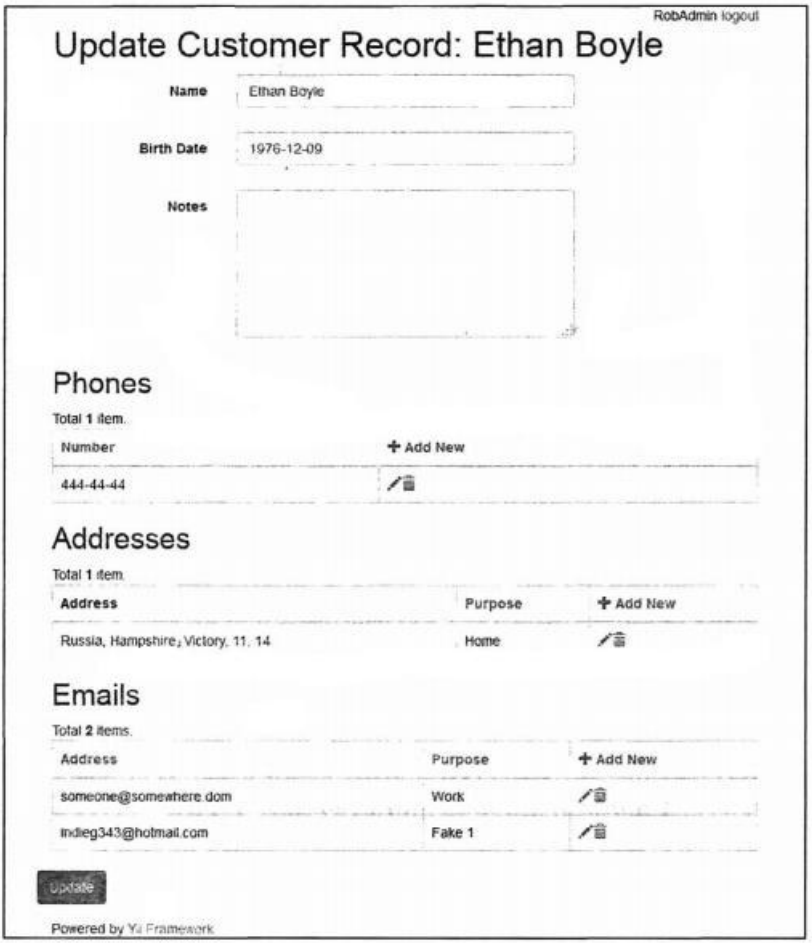
```
'columns' => [
   [
      'label' => 'Address',
```

```
            'value' => function ($model) {
               return implode(', ',
                  array_filter(
                     $model->getAttributes(
                        ['country', 'state', 'city', 'street',
'building', 'apartment'])));
               }
         ],
         'purpose',
         [
            // … здесь ActionColumn, на которую мы не обращаем внимания
…
         ],
      ],
```

Вот что мы получаем с таким определением колонки:



В итоге мы получаем форму, достаточно сильно похожую на тот набросок, который мы сделали в начале этого раздела.



Мы не рассмотрели подробно важную деталь в обработке активных форм: валидацию введённых данных. Не упускайте возможность узнать про валидацию форм в официальной документации здесь: http://www.yiiframework.com/doc-2.0/guide-input-validation.html, так как это очень полезная возможность Yii 2.