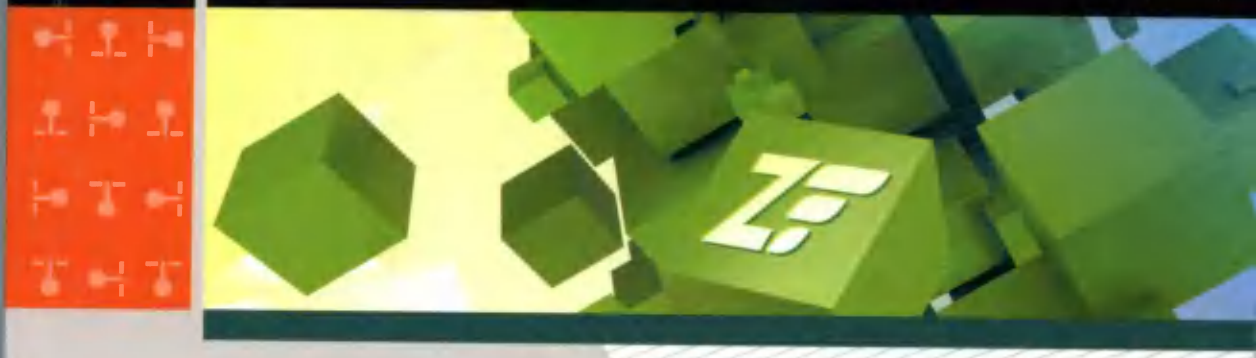
БИБЛИОТЕКА ПРОГРАММИСТА



Викрам Васвани

ZEND FRAMEWORK

Разработка веб-приложений на PHP

- Модель – представление – контроллер
- Маршрутизация
- Проверка входных данных
- Локализация и кэширование

Mc Graw Hill

ПИТЕР

Vikram Vaswani

Zend Framework:
A Beginner's Guide

New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

БИБЛИОТЕКА ПРОГРАММИСТА

Викрам Васвани

ZEND FRAMEWORK

Разработка веб-приложений на PHP

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2012

ББК 32.988.02-018
УДК 004.738.5
В19

Васвани В.
В19 Zend Framework: разработка веб-приложений на PHP. — СПб.: Питер, 2012. — 432 с.: ил.

ISBN 978-5-459-00826-5

Zend Framework основывается на простоте, лучших приемах объектно-ориентированного программирования, дружественной к корпорациям лицензионной политике и тщательно протестированном гибком коде. Zend Framework нацелен на разработку более безопасных, надежных и современных Web 2.0-приложений и служб и на использование общедоступных API

Из этой книги, автор которой хорошо разбирается в вопросе и прекрасно владеет техническим языком, вы узнаете, каким образом Zend Framework достигает поставленных целей. Здесь вы найдете подробные и понятные объяснения, а также законченные примеры, и мы надеемся, что впоследствии вы станете с удовольствием разрабатывать собственные приложения с помощью инструмента Zend Framework, фактически ставшего стандартом.

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-0071639392 англ.
ISBN 978-5-459-00826-5

Краткое оглавление

Оглавление

Об авторе

Викрам Васвани — основатель и генеральный директор консалтинговой компании Melonfire (http://www.melonfire.com), специализирующейся на открытых программных средствах и технологиях. На протяжении 12 лет Викрам работает с PHP и MySQL, занимаясь разработкой веб-приложений и менеджментом. За это время им было создано множество приложений на PHP для корпоративных локальных сетей, интернет-сайтов и клиентских приложений, решающих самые важные задачи.

Викрам является сторонником программного обеспечения с открытым исходным кодом, регулярно пишет статьи и руководства по PHP, MySQL, XML и связанным с ними инструментам, размещая их на сайтах Zend Developer Zone и IBM DeveloperWorks. Он написал хорошо известный цикл *«101 статья по PHP для начинающих»*, «MySQL: использование и администрирование» (СПб.: Питер, 2011), *«Как сделать всё что угодно с помощью PHP и MySQL»* (http://www.everythingphpmysql.com/), *«Программные решения на PHP»* (http://www.php-programming-solutions.com/) и *«PHP: руководство для начинающих»* (http://www.php-beginners-guide.com/).

Являясь стипендиатом Felix Scholarship в Оксфордском университете в Англии, Викрам успешно совмещает разработку веб-приложений с множеством других занятий. В свободное от обдумывания планов захвата мирового господства время он с удовольствием читает фантастику, смотрит фильмы, играет в сквош, ведет блоги и пристально следит за агентом Смитом. Узнайте больше об авторе на сайте http://www.zf-beginners-guide.com.

О научном редакторе

Райан Могер — ведущий разработчик компании Lupimedia (http://www.lupimedia.com/), занимающейся мультимедийным дизайном и специализирующейся на заказных системах управления контентом для дизайн-ориентированных веб-сайтов. Райан всей душой предан Zend Framework, его можно найти в IRC (#channel), где он отвечает на вопросы и дает советы. Свободное от пропаганды Zend Framework время Райан уделяет обязанностям отца, но часто сбегает на озера, чтобы порыбачить. О нем и его работе можно узнать на сайте http://www.rmauger.co.uk/.

Предисловие

За последние шесть лет экосистема PHP претерпела существенные изменения. До выхода **PHP 5** мы, разработчики, создавали свои проекты в основном для разового использования, и каждый из этих проектов отличался индивидуальностью; если новому проекту уделялось достаточно внимания, он оказывался лучше предыдущих — но гарантий не было. Несмотря на наличие средств и способов управления качеством и стандартами кода, они все еще были недостаточно развиты и не находили широкого применения. Идея использования PHP в качестве основы для стабильных приложений корпоративного уровня вызывала лишь насмешки, несмотря на тот факт, что он использовался в некоторых наиболее посещаемых веб-сайтах.

С выходом **PHP 5** мы стали уделять больше внимания разработке надежных приемов программирования. Пересмотренная и переработанная объектная модель стала прочной основой для создания объектов, пригодных для многократного использования. Такие программные средства, как **PHPUnit**, использовали эту объектную модель, чтобы упростить и сделать более надежными подходы к тестированию кода. Это, в свою очередь, привело к более пристальному изучению соответствия качества кода жизненному циклу приложений на PHP.

Именно в этой экосистеме начали появляться PHP-фреймворки. Хотя некоторые из них существовали и для **PHP 4**, в **PHP 5** эта идея приобрела популярность, и в результате начали развиваться несколько фреймворков. Их целью было обеспечить пользователей лучшими решениями и предоставить повторяемую и пригодную к многократному использованию структуру для разрабатываемых приложений.

К этим фреймворкам относится и Zend Framework. В соответствии с представленной на сайте информацией его цель заключается в следующем: «Следуя духу PHP, Zend Framework основывается на простоте, лучших приемах объектно-ориентированного программирования, дружественной к корпорациям лицензионной политике и тщательно протестированном гибком коде. Zend Framework нацелен на разработку более безопасных, надежных и современных Web 2.0-приложений и служб и на использование общедоступных API».

Из этой книги, автор которой хорошо разбирается в вопросе и прекрасно владеет техническим языком, вы узнаете, каким образом Zend Framework достигает поставленных целей. Здесь вы найдете подробные и понятные объяснения, а также законченные примеры, и мы надеемся, что впоследствии вы станете с удовольствием разрабатывать собственные приложения с помощью инструмента Zend Framework, фактически ставшего стандартом.

Мэтью Вайер О'Финни (Matthew Weier O'Phinney),
руководитель проекта Zend Framework

Благодарности

Zend Framework представляет собой сложное программное обеспечение, и написание книги о нем, как выяснилось в последние восемь месяцев, — *не самая* простая задача. К счастью, в этом процессе мне помогала группа энергичных людей, и за то, что эта книга оказалась в ваших в руках, я благодарен каждому из них.

В первую очередь я хочу сказать огромное спасибо моей жене, которая поддерживала меня на протяжении всего процесса и обеспечивала комфортную и свободную от стрессов рабочую обстановку. Я уверен, что эта книга никогда бы не увидела свет без ее помощи. Спасибо, малышка!

Команда редакторов и маркетологов в издательстве McGraw-Hill также заслуживает всячески похвал. Это моя шестая книга, выпущенная этим издательством, и я в очередной раз выражаю им свою признательность. Координатор Джоя Энтони (Joya Anthony), руководитель редакционного отдела Пэти Мон (Patty Mon), а также главные редакторы Джейн Браунлоу (Jane Brownlow) и Мэг Морин (Megg Morin) помогли рукописи пройти весь процесс подготовки и сыграли огромную роль в превращении мыслей и пикселей в безупречный и профессиональный продукт. Я хотел бы поблагодарить их за компетентность, самоотверженность и старание.

Отдельно хочется поблагодарить Райана Могера (Ryan Mauger), технического редактора этой книги. Райан лично проверил каждую строчку кода и использовал свои необъятные знания Zend Framework, чтобы убедиться в его корректности и правильности. Если вам когда-либо придется искать эксперта по PHP, вы не найдете никого лучше, чем он!

И наконец, хочется выразить отдельную благодарность тем, кто скрасил нелегкий процесс написания книги — это Патрик Куинлан, Ян Флеминг, Брайан Адамс, группа Stones, Питер О'Доннелл, журнал MAD Magazine, Скотт Адамс, FHM, Гэри Ларсон, канал VH1, Бритни Спирс, Джордж Майкл, Кайли Миноуг, «Баффи — истребительница вампиров», Фарах Малегам, Стивен Кинг, певица Shakira, Анахита Маркер, Джон Ле Карре, группа The Saturdays, Барри Уайт, Гвен Стефани, Роберт Крейс, Роберт Б. Паркер, Баз Лурманн, Stefy, Анна Курникова, Джон Коннолли, Wasabi, Omega, Pidgin, Кэл Эванс, Ling's Pavilion, Тонка со своим близнецом Бонкой, Дин Таи Фунг, канал HBO, Марк Твен, Тим Бартон, Хариш Камат, Мадонна, Джон Сэнфорд, герой комиксов «Железный человек», Tube, певица Dido, Квентин

Тарантино, Google.com, фильм «Матрица», Ли Чайлд, Майкл Коннелли, Антонио Прохиас, Квентин Тарантино, Альфред Хичкок, Вуди Ален, Kinokuniya, Перси Джексон, Дженнифер Хадсон, Mambo's and Tito's, Easyjet, Хамфри Богарт, Thai Pavilion, Wikipedia, Amazon.com, U2, The Three Stooges, Pacha, Оскар Уайлд, Хью Грант, Punch, Келли Кларксон, Скотт Туроу, дистрибутив Slackware Linux, Calvin и Hobbes, компания Blizzard Entertainment, Альфред Кропп, Otto, Пабло Пикассо, Popeye и Olive Oyl, Деннис Лехэйн, Trattoria, Dire Straits, Брюс Спрингстин, Дэвид Митчелл, сериал «Западное крыло», гитарист Карлос Сантана, Род Стюарт и все мои друзья, где бы они ни находились.

Вступление

Zend Framework — это, говоря словами Эрнеста Хемингуэя, «праздник, который всегда с тобой». Задуманный и реализованный как надежная и функциональная библиотека компонентов для разработчиков на PHP, он позволяет быстро и эффективно решать множество распространенных задач, связанных с разработкой приложений, таких как создание и проверка форм ввода, обработка XML, генерация динамических меню, разбивка данных на страницы, работа с веб-сервисами и многое-многое другое!

Следует отметить, что Zend Framework поднял искусство разработки на PHP на новый уровень, познакомив разработчиков с более стандартизированным и структурированным подходом к программированию. Результатом такого подхода стали более понятные, безопасные и легко поддерживаемые приложения. Именно в этом заключается одна из ключевых причин того, что все больше и больше разработчиков переходят со старого стиля «одноразового» программирования к новому стилю, основанному на использовании фреймворков.

Тем не менее для многих начинающих разработчиков на PHP Zend Framework кажется пугающим шагом в неизвестность. Паттерн Модель–Представление–Контроллер, архитектура со слабой связанностью и большое количество доступных компонентов способны сбить с толку разработчиков, привыкших к «обычному» процедурному программированию и считающих основанную на фреймворке разработку слишком сложной для понимания.

В неизвестном и пугающем мире эта книга станет вам путеводителем. Если вы — один из тех пользователей, которые слышали о Zend Framework и хотели бы узнать о нем побольше, эта книга для вас. В ней подробно рассмотрены наиболее важные возможности Zend Framework, такие как реализация паттерна Модель–Представление–Контроллер, маршрутизация, валидация ввода, интернационализация и кэширование, и показано их практическое использование. С этой книгой вы пройдете через процесс создания законченного веб-приложения с помощью Zend Framework, начиная с основ и постепенно добавляя более сложные функции, такие как разбиение данных на страницы и сортировка, аутентификация пользователей, обработка исключений, локализация и веб-сервисы. Она поможет вам с помощью Zend Framework поднять разработку на PHP на новый уровень.

Кому адресована эта книга

«Zend Framework: разработка веб-приложений на PHP» предназначена для пользователей, не знакомых с Zend Framework. Предполагается, что вы уже знаете основы программирования на PHP (включая новую объектную модель в PHP 5.x) и в некоторой степени знакомы с HTML, CSS, SQL, XML JavaScript. Если вы совсем не знаете PHP, то начните с руководства для начинающих — вы найдете его на сайте http://www.melonfire.com/community/columns/trog/, или приобретите руководство вроде *«Как сделать все что угодно с помощью PHP и MySQL»* (http://www.everythingphpmysql.com) или *«PHP: руководство для начинающих»* (http://www.php-beginners-guide.com), а потом вернитесь к этой книге.

Для работы с демонстрационным приложением, представленным в этой книге, вам потребуется установить интерпретатор **PHP 5.x**, работающий с веб-сервером Apache 2.2.x и сервером баз данных MySQL 5.x. И очевидно вам потребуется последняя версия Zend Framework. Инструкции по загрузке и настройке окружения для разработки на PHP приведены в приложении к этой книге, а глава 1 подробно описывает процесс установки Zend Framework.

О чем можно узнать из этой книги

Поскольку книга *«Zend Framework: разработка веб-приложений на PHP»* предназначена для пользователей, не знакомых с Zend Framework, мы начнем с объяснения базовых принципов и решения простых задач. Когда вы познакомитесь с основами разработки с использованием Zend Framework, мы перейдем к более сложным задачам, таким как интернационализация и оптимизация производительности и их возможные решения. Это также означает, что вы должны придерживаться установленного порядка глав в книге, так как в каждой последующей главе вам понадобятся знания, полученные в предыдущих главах.

Ниже приведен краткий обзор каждой главы:

- **Глава 1, «Введение в Zend Framework»**, знакомит с Zend Framework, объясняет преимущества разработки с использованием фреймворков и проводит через процесс создания нового проекта Zend Framework.
- **Глава 2, «Работа с моделями, представлениями, контроллерами и маршрутами»**, рассматривает основы паттерна Модель–Представление–Контроллер и знакомит с такими важными понятиями, как маршрутизация, глобальные макеты и модули.
- **Глава 3, «Работа с формами»**, знакомит с компонентом Zend_Form и объясняет, как программно создавать веб-формы, проводить их валидацию, защищать от атак и управлять сообщениями об ошибках.
- **Глава 4, «Работа с моделями»**, рассматривает роль моделей в приложении Zend Framework и знакомит с инструментарием Doctrine ORM и начальным загрузчиком Zend Framework.

- **Глава 5, «Работа с операциями CRUD»**, рассматривает способы интеграции моделей Doctrine с контроллерами Zend Framework для реализации четырех распространенных операций CRUD, добавление аутентификации в приложение и создание простой системы для входа и выхода.
- **Глава 6, «Индексация, поиск и форматирование данных»**, рассматривает индексацию и поиск данных, а также показывает, как можно добавить в приложение Zend Framework поддержку нескольких типов вывода.
- **Глава 7, «Разбиение на страницы, сортировка и загрузка данных на сервер»**, объясняет, как осуществлять разбиение на страницы и сортировку результатов запросов к базе данных, отфильтровывать и обрабатывать загруженные файлы, а также читать и записывать конфигурационные файлы в форматах INI и XML.
- **Глава 8, «Журналирование и отладка исключений»**, объясняет, как Zend Framework обрабатывает исключения уровня приложения и показывает, как можно добавить в приложение Zend Framework журналирование и фильтрацию исключений.
- **Глава 9, «Локализация приложений»**, рассматривает различные доступные в Zend Framework средства для создания локализованных многоязычных приложений, которые можно легко «переносить» в другие страны и регионы.
- **Глава 10, «Работа с новостными лентами и веб-сервисами»**, объясняет, как использовать Zend Framework для генерации и чтения новостных лент в форматах Atom и RSS, осуществлять доступ к сторонним веб-сервисам с помощью SOAP или REST и предоставлять разработчикам доступ к вашим приложениям с помощью REST.
- **Глава 11, «Работа с элементами пользовательского интерфейса»**, рассказывает, как улучшить навигацию по сайту посредством меню, навигационных цепочек и карт сайта, а также объясняет, как интегрировать Zend Framework и Dojo, на примерах всплывающего календаря и пóля формы с поддержкой автоматического дополнения, использующих технологию AJAX.
- **Глава 12, «Оптимизация производительности»**, рассматривает различные приемы измерения и улучшения производительности веб-приложений, включающие оценку производительности, стресс-тестирование, профилирование кода, кэширование и оптимизацию запросов.
- **Приложение, «Установка и настройка необходимого программного обеспечения»**, описывает процесс установки и настройки связки Apache/PHP/MySQL в Windows и Linux.

Соглашения

Для выделения особых рекомендаций в книге используются различные способы оформления.

ПРИМЕЧАНИЕ

Дополнительная информация по теме.

СОВЕТ

Способ или технический прием, помогающий улучшить результат.

ВНИМАНИЕ

Нечто, требующее особого внимания.

ВОПРОС ЭКСПЕРТУ

В: Часто задаваемый вопрос...

О: ...и ответ на него.

В приведенных в книге листингах кода жирным шрифтом обозначены команды, которые необходимо вводить в командной строке. Например, в следующем листинге

```
mysql> INSERT INTO movies (mtitle, myear) VALUES ('Rear Window', 1954);
Query OK, 1 row affected (0.06 sec)
```

выделенный жирным текст представляет собой запрос, который вы могли бы ввести в командной строке. Выделенные фрагменты можно использовать для тестирования приведенных в книге команд.

Сайт книги

Примеры, используемые в данной книге, можно скачать с сайта издательства «Питер» (www.piter.com) или найти на сайте http://www.zf-beginners-guide.com/. Архивы сгруппированы по главам и могут быть загружены и использованы для разработки в среде Zend Framework.

Введение в Zend Framework

Прочитав эту главу, вы:

- оцените преимущества разработки с использованием фреймворков;
- проникнитесь историей и уникальными достижениями Zend Framework;
- вникните в структуру приложения Zend Framework;
- установите Zend Framework и приступите к его использованию.

Можно без преувеличения сказать, что на сегодняшний день PHP является одним из самых популярных языков программирования в мире и миллионы разработчиков веб-приложений выбрали его в качестве своего основного инструмента. По последним статистическим данным, этот язык используется более чем 22 миллионами веб-сайтов и третьей частью всех веб-серверов на планете — это впечатляет, особенно если учесть, что PHP разработан и поддерживается исключительно добровольцами без какой-либо коммерческой поддержки!

Понять причины популярности PHP несложно. Он масштабируем, доступен и хорошо взаимодействует со сторонними программами. Он использует ясный и простой синтаксис и радует незапутанным кодом, делая несложным его изучение и использование и способствуя быстрой разработке приложений. И у него есть огромное преимущество перед коммерческими средствами разработки, поскольку он абсолютно бесплатен, применим для множества платформ и архитектур, включая UNIX, Microsoft Windows и Mac OS, и использует открытую лицензию.

Разработчики тоже довольны PHP. В исследовании десяти языков сценариев, проведенном в августе 2009 года компанией Evans Data Corporation, наиболее довольные пользователи были выявлены среди разработчиков на PHP (за ними с небольшим отставанием следуют пользователи Ruby и Python). В частности, PHP занял первые места по таким показателям, как кроссплатформенная совместимость, доступность и качество программных средств и производительность, и вторые — за простоту сопровождения и удобочитаемость кода, расширяемость, простоту использования и безопасность.

Для организаций и независимых разработчиков все эти факты означают следующее: использование PHP экономит деньги и время. Разработка приложений на PHP требует меньших денежных затрат, поскольку этот язык можно использовать в различных целях, не выплачивая лицензионные отчисления и не вкладывая средства в дорогостоящее оборудование или программное обеспечение. А вследствие доступности готовых надежных и проверенных элементов управления и расширений, которые разработчики могут использовать для добавления в язык новых функций, использование PHP также уменьшает и время разработки, при этом не принося в жертву качество.

Хотя на первый взгляд это и не очевидно, но хваленая простота использования PHP — одновременно и достоинство, и недостаток. Достоинство заключается в том, что в отличие от, скажем, C++ или Java, программы на PHP относительно легко читать и понимать, и данный факт подталкивает начинающих разработчиков к экспериментам и быстрому освоению языка, минуя стадию кропотливого изучения. Недостаток же в том, что присущее PHP отсутствие «строгости» дает этим разработчикам ложное ощущение безопасности и подталкивает их к написанию общедоступных приложений без понимания необходимых стандартов качества, безопасности и возможности повторного использования кода.

С учетом всего этого в последние несколько лет в рядах PHP-сообщества происходила напряженная и плодотворная работа по переходу от программирования в стиле «можно все» к более стандартизированному подходу, ориентированному на использование фреймворков. Этот подход не только упрощает первоначальную подготовку при написании PHP-приложения с нуля, но и позволяет создавать более ясный, последовательный и безопасный код. В этой главе и в оставшейся части книги вы познакомитесь с одним из таких фреймворков, Zend Framework, который предусматривает гибкий и масштабируемый подход к разработке PHP-приложений для серьезных разработчиков.

Обзор

По словам официального веб-сайта (http://framework.zend.com), Zend Framework — это «фреймворк с открытым исходным кодом, предназначенный для разработки веб-приложений и сервисов на PHP 5 [...], основанный на простоте, лучших приемах объектно-ориентированного программирования, дружественной по отношению к корпорациям лицензионной политике и тщательно протестированной кодовой базе». Он предоставляет полный набор инструментов для создания и развертывания основанных на PHP веб-приложений со встроенными API для распространенных функций, таких как безопасность, валидация ввода, кэширование данных, операции с базами данных и XML, а также интернационализация.

В отличие от других фреймворков Zend Framework использует архитектуру со «слабой связанностью». Это означает, что хотя сам фреймворк состоит из множества компонентов, они по большей части независимы и минимально связаны друг с другом. Такая архитектура помогает создавать «легкие» приложения, поскольку

разработчики могут использовать только те компоненты, которые нужны им для выполнения поставленной задачи. Например, разработчики, желающие добавить в свое приложение аутентификацию или кэширование, могут напрямую воспользоваться компонентами Zend_Auth и Zend_Cache, не задействуя остальную часть фреймворка.

Zend Framework также предоставляет законченную реализацию паттерна Модель–Представление–Контроллер (*MVC, Model-View-Controller*), позволяющего отделить бизнес-логику приложения от пользовательского интерфейса и моделей данных. Этот паттерн рекомендуется для приложений среднего и высокого уровня сложности и повсеместно используется при разработке веб-приложений, так как способствует повторному использованию кода и создает структуру кода, которой проще управлять. Реализация паттерна MVC в Zend Framework подробно рассматривается в главе 2.

Zend Framework создан и поддерживается компанией Zend Technologies, поставщиком коммерческого программного обеспечения, чьи основатели — Энди Гатманс (Andi Gutmans) и Зив Сураски (Zeev Suraski) осуществили первое крупное переписывание кода анализатора PHP, вышедшего в 1997 году как PHP 3.0. Первая версия Zend Framework, *v1.0*, была выпущена в июле 2007 года и содержала 35 основных компонентов, включая компоненты для кэширования, аутентификации, управления конфигурацией, доступа к базам данных, генерации лент RSS и Atom и локализации.

С того момента фреймворк прошел путь от версии *v1.0* до версии *v1.10*, содержащей теперь 65 компонентов, которые, помимо всего прочего, поддерживают формат сообщений Action Message Format (AMF) компании Adobe, API GData компании Google, а также веб-сервисы Amazon EC2 и SQS. Увеличению числа компонентов сопутствовало увеличение объема документации — руководство по Zend Framework v1.9 (примерно 2009 год) занимало 3.7 Мбайт по сравнению с руководством по Zend Framework v1.0 2007 года, занимавшим 780 Кбайт.

Несмотря на коммерческую деятельность компании Zend Technologies, Zend Framework является общедоступным проектом с открытым исходным кодом под лицензией BSD License, что позволяет использовать фреймворк в закрытых коммерческих продуктах без выплаты лицензионных отчислений. Эта «дружественная к бизнесу» лицензионная политика сделала Zend Framework популярным как среди корпоративных, так и среди одиночных пользователей. В число поклонников PHP входят стартапы, компании из списка Fortune 500, независимые разработчики и те, для кого PHP является хобби, — на момент написания книги Zend Framework был загружен более 10 миллионов раз и существовали более 400 проектов с открытым исходным кодом, основанных на нем или расширяющих его.

Энергичное и полное энтузиазма сообщество разработчиков обменивается исправлениями ошибок и советами на форумах и вики-страницах, а размещенные в сети учебник и справочное руководство оказывают дополнительную помощь. В сообществе также приветствуется определенная «отдача» фреймворку в виде отправки новых компонентов — при условии, что они соответствуют требованиям Zend касательно документации и модульного тестирования.

Возможности

Почему же использование Zend Framework является лучшей идей, нежели просто написание собственного кода? Ниже перечислены некоторые из причин.

Соответствие стандартам и лучшие приемы программирования

В отличие от некоторых других языков программирования PHP не навязывает общий стандарт написания кода. В результате стиль кода в различных приложениях на PHP существенно различается у разных разработчиков, что затрудняет поддержание согласованности проекта. Относительный недостаток «строгости» в PHP также приводит к появлению «некачественного» кода, что делает его уязвимым.

Zend Framework, напротив, объединяет в себе приемы программирования, считающиеся лучшими на сегодняшний день, предоставляет стандартную схему размещения файлов в файловой системе и обеспечивает встроенную поддержку для решения распространенных задач, возникающих при разработке приложений, таких как валидация и очистка входных данных. Следовательно, использование этого фреймворка как основы для проекта на PHP автоматически приводит к созданию более качественного кода и приложения, более устойчивого к проблемам безопасности. Кроме этого, в связи наличием хорошей документации Zend Framework разработчики, присоединившиеся к уже разрабатываемому проекту, смогут гораздо быстрее освоить его и начать вносить изменения.

Многократное использование

Zend Framework является полностью объектно-ориентированным и в полной мере использует новую объектную модель PHP 5.x. Такая архитектура, основанная на объектно-ориентированном программировании (ООП), способствует многократному использованию кода, позволяя разработчикам значительно уменьшить время, затрачиваемое на написание дублирующегося кода. Это особенно важно для веб-приложений с несколькими интерфейсами для доступа к данным. Предположим, что вы хотите добавить к уже существующим в вашем приложении функциям поиска интерфейс, использующий XML. Для этого необязательно переписывать логику какого-либо из существующих контроллеров, а весь процесс является прозрачным и простым.

Интернационализация

Так как Zend Framework предназначен для использования в разработке веб-приложений, было бы странным, если бы он не предоставлял всеобъемлющую поддержку интернационализации и локализации приложений. Компонент `Zend_Locale` позволяет осуществлять управление языковой настройкой, а компонент `Zend_Translate` делает возможной поддержку нескольких языков, содержащих на-

боры латинских, китайских и европейских символов. К другим полезным компонентам относятся Zend_Date и Zend_Currency, предназначенные для локализованного форматирования даты/времени и валюты.

Открытый исходный код

Zend Framework — проект с открытым исходным кодом. Хотя он и спонсируется компанией Zend Technologies, значительная часть разработки ведется командой добровольцев, которые занимаются исправлением ошибок и добавлением новой функциональности. Zend Technologies определяет направление проекта, а также группу «ведущих инженеров», принимающих решения о том, что следует включить в конечный продукт. Как упоминалось ранее, фреймворк можно использовать без уплаты лицензионных отчислений или приобретения дорогостоящего оборудования или программного обеспечения. Этот факт уменьшает затраты на разработку программ, не влияя на гибкость и надежность. Кроме того, открытость кода означает, что любой разработчик, где бы он ни находился, может изучать иерархию кода, выявлять ошибки и предлагать возможные пути их исправления; в результате мы имеем стабильный, надежный продукт, в котором любые ошибки исправляются по мере их нахождения — иногда в течение нескольких часов!

Поддержка со стороны сообщества

Ищете способы интеграции в ваш проект фотогалерей Flickr или данных Google Maps? Попробуйте компоненты Zend_Service_Flickr и Zend_Gdata. Хотите взаимодействовать с Flash-приложением, используя формат Adobe Action Message Format (AMF)? Добавьте компонент Zend_Amf. Требуется быстро интегрировать в приложение ленту RSS? Zend_Feed — все, что вам нужно.

Как показывают эти примеры, одной из приятных особенностей этого проекта является возможность использования творческого потенциала и изобретательности сотен разработчиков по всему миру. Zend Framework состоит из большого числа независимых компонентов, которыми пользуются разработчики для беспроблемного добавления новой функциональности в свои проекты на PHP. Применение этих компонентов представляет собой более эффективную с точки зрения времени и денежных затрат альтернативу написанию собственного кода.

Уникальные преимущества

Можно поспорить, что перечисленные выше возможности относятся ко всем PHP-фреймворкам, а не только к Zend Framework. Однако последний обладает рядом уникальных возможностей, которые дают ему определенные преимущества.

Слабая связанность

В отличие от многих других фреймворков, отдельные элементы которых сильно привязаны друг к другу, компоненты Zend Framework могут быть легко разделены и использованы по мере необходимости. Поэтому хотя Zend Framework, безусловно, содержит все, что вам требуется для создания современного совместимого с паттерном MVC приложения, он не заставляет вас это делать: вы с тем же успехом можете взять любой из отдельных компонентов библиотеки и интегрировать его в ваше приложение, не использующее MVC. Эта *слабая связанность* помогает уменьшить ресурсы, используемые приложением, и сохраняет гибкость по отношению к будущим изменениям.

Быстрая смена релизов

Команда Zend Framework придерживается агрессивной политики в отношении выпуска новых версий (в среднем от одной до трех в месяц). Кроме этих стабильных релизов существуют также предварительные версии и релиз-кандидаты, цель которых — дать сообществу представление о том, чего следует ожидать в готовящейся версии. Такая быстрая смена релизов нужна не только для продвижения проекта, но и для быстрого поиска и исправления ошибок. Кроме того, разработчики могут получить доступ к «передовому коду», хранящемуся в открытом Subversion-репозитории проекта.

Политика модульного тестирования

Учитывая слабую связанность компонентов Zend Framework, находящихся в постоянной разработке, для модульного тестирования важно обеспечить их корректную работу и стабильность кодовой базы на протяжении множества циклов релиза. Zend Framework придерживается строгой политики модульного тестирования, в соответствии с которой компоненты могут добавляться в состав фреймворка только в том случае, если к ним прилагается достаточно полный и работающий набор модульных тестов (написанных с использованием фреймворка для тестирования PHPUnit). Эта политика гарантирует, что между релизами будет поддерживаться обратная совместимость, а любые ухудшения будут легко заметны.

Средства генерации кода

Zend Framework содержит возможность «инструментальной обработки», позволяющую разработчикам минимальными усилиями настроить и запустить новый проект Zend Framework. Эта возможность, реализованная в виде сценария командной строки, построенного на основе компонента Zend_Tool, позволяет создать базовую иерархию файлов для нового проекта и добавить к ней первоначальный набор контроллеров, представлений и действий. Разработчики могут использовать этот сценарий в качестве удобного способа быстрого создания новых объектов проекта по мере продвижения разработки.

Доверие рынка

Zend Framework спонсируется Zend Technologies — одной из наиболее известных связанных с PHP компаний, занимающихся разработкой программного обеспечения. Компания выпускает несколько коммерческих продуктов для корпоративного использования и имеет длинный послужной список в области создания успешных и инновационных продуктов для разработчиков на PHP. Он включает в себя такие проекты, как Zend Server — сервер веб-приложений для программ на PHP, имеющих большое значение для бизнеса, а также Zend Studio — интегрированную среду разработки (IDE) для создания приложений на PHP. В списке клиентов Zend Technologies находятся IBM, McAfee, FOX Interactive Media, Lockheed Martin, SalesForce.com, NASA и Bell Canada; поддержка Zend Framework такой компанией незамедлительно гарантирует доверие на рынке и служит прекрасным инструментом для убеждения клиентов и/или ведущих менеджеров перейти к разработке с использованием фреймворков.

Возможность взаимодействия со сторонними приложениями

Компания Zend Technologies является одной из ведущих поставщиков корпоративных PHP-решений, что позволило ей получить широкую «промышленную» поддержку Zend Framework. Zend Framework содержит встроенную поддержку множества сторонних средств и технологий, включая Adobe Action Message Format (AMF), API Google Data, Dojo Toolkit, Microsoft CardSpace и веб-сервисы Amazon, Yahoo!, Twitter, Flickr, Technorati и Del.icio.us.

И это не все. Исторически одной из сильных сторон PHP была поддержка им широкого спектра различных баз данных, форматов файлов и протоколов. Zend Framework предоставляет общий API для доступа к базам данных MySQL, PostgreSQL, Oracle и Microsoft SQL Server (среди прочих) с помощью компонентов Zend_Db, а также содержит компоненты для отправки и получения почты по протоколам SMTP, IMAP и POP3, создания веб-сервисов с использованием протоколов SOAP и REST, кодирования и декодирования данных в формате JSON, анализа лент в форматах Atom и RSS, создания и работы с PDF-документами.

Варианты коммерческой поддержки

Zend Framework бесплатен — пользователи могут загружать и использовать его без каких-либо денежных затрат в соответствии с условиями лицензии BSD License, но при этом предполагается, что они будут решать возникшие вопросы самостоятельно с помощью средств, предоставляемых сообществом, таких как форумы и вики-страницы. Компаниям и отдельным пользователям, желающим получить более высокий уровень поддержки, Zend Technologies предлагает коммерческую поддержку и семинары, консультационные услуги инженеров Zend, а также закрытые инструменты для разработки и развертывания приложений на PHP, помогающие ускорить и оптимизировать разработку с использованием Zend Framework. Для многих коммерческих организаций возможность получения технической поддержки, хотя и платной, является основной причиной для выбора

Zend Framework в качестве средства разработки приложений. Существует также программа сертификации Zend Framework Certification, с помощью которой оцениваются навыки отдельно взятых разработчиков, использующих Zend Framework, и которая является показателем уровня их компетентности.

Исчерпывающая документация

Zend Framework содержит исчерпывающую документацию к шестидесяти с лишним компонентам, входящим в состав базового дистрибутива. В документацию включены руководство программиста объемом более тысячи страниц, руководство «для быстрого старта», нацеленное на опытных разработчиков, детальная документация по API, видеоуроки, вебинары и подкасты известных инженеров Zend. Такое обилие обучающих материалов может значительно уменьшить время обучения как новичков, так и опытных разработчиков, и фактически является одной из основных областей, в которых Zend Framework превосходит конкурирующие PHP-фреймворки.

Среда для приложений

Все приложения на основе Zend Framework также являются PHP-приложениями и могут выполняться в любой среде, пригодной для PHP. Как правило, эта среда состоит из трех компонентов:

- Операционная система, обычно Linux или Microsoft Windows.
- Веб-сервер, как правило Apache в Linux или Internet Information Services в Microsoft Windows, который перехватывает HTTP-запросы и либо обрабатывает их непосредственно, либо передает для выполнения интерпретатору PHP.
- Интерпретатор PHP, который осуществляет анализ и выполнение кода на PHP и возвращает результаты веб-серверу.

Иногда присутствует четвертый, *необязательный, но очень полезный* компонент:

- Система управления базами данных, например MySQL или PostgreSQL, которая используется для хранения данных приложения, принимает подключения со стороны PHP и модифицирует данные в базе данных или извлекает их.

Взаимодействие между перечисленными компонентами показано на рис. 1.1.

Стоит отметить, что среди разработчиков чрезвычайно популярна комбинация Linux/Apache/PHP/MySQL, иначе называемая *стек LAMP*. Популярность стека LAMP вызвана тем, что все его компоненты являются проектами с открытым исходным кодом и, как следствие, могут быть бесплатно загружены из Интернета. Как правило, также отсутствуют какие-либо выплаты за использование этих компонентов в личных или коммерческих целях и за разработку и распространение использующих их приложений. Однако если вы собираетесь разрабатывать коммерческие приложения, правильнее будет рассмотреть условия лицензирования для каждого из компонентов, которые можно найти на веб-сайте компонента или в архиве вместе с ним.

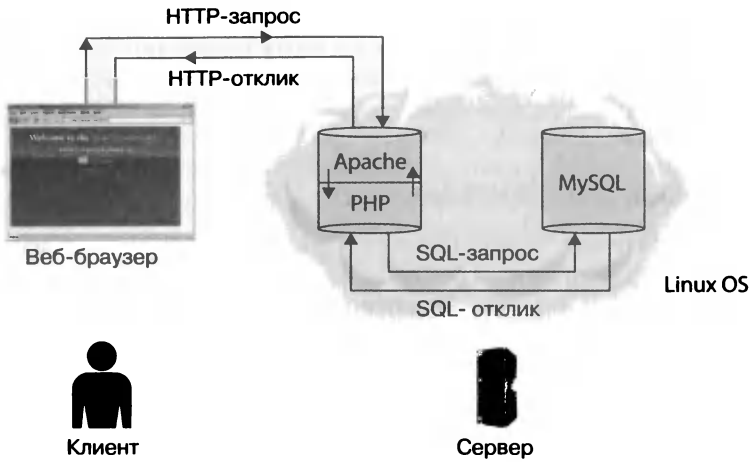


Рис. 1.1. Компоненты типичного окружения приложения на PHP

Установка Zend Framework

Теперь, когда вы кое-что знаете о Zend Framework, давайте приступим к созданию приложений с его использованием. В качестве обязательного первого шага убедитесь, что у вас установлена работающая среда разработки в виде Apache/PHP/MySQL. В приложении к этой книге приведены подробные инструкции по получению этих компонентов, их установке и проверке среды разработки на работоспособность, поэтому перелистните страницы и вернитесь, как только будете готовы.

Получилось? Следующий шаг — загрузка и установка Zend Framework в вашу среду разработки. Посетите официальный веб-сайт Zend Framework http://framework.zend.com/ и загрузите копию самого свежего релиза. Zend Technologies выпускает две версии пакета: «минимальную» версию, содержащую только стандартные библиотеки и средства командной строки, и «полную» версию, включающую в себя дополнительную документацию, примеры, модульные тесты и сторонние инструментарии. Рекомендуется использовать полную версию.

Как только вы загрузили архив с исходным кодом, извлеките его содержимое во временный каталог файловой системы.

```
shell> cd /tmp
shell> tar -xzvf ZendFramework-XX.tar.gz
```

В результате получится структура каталогов, похожая на ту, что изображена на рис. 1.2.

Из всех этих каталогов вам потребуются library/ и bin/. Каталог library/ содержит все компоненты Zend Framework, а каталог bin/ содержит средства командной строки, помогающие инициализировать новый проект и добавлять к нему объекты.

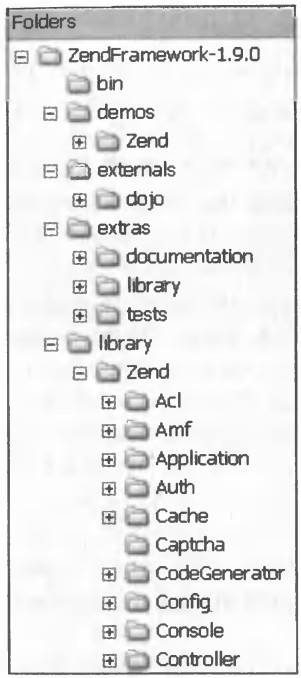


Рис. 1.2. Содержимое архива с Zend Framework

Теперь сделайте следующее:

Содержимое каталога library/ переместите в каталог, содержащийся в списке «путей поиска» PHP. В системах UNIX/Linux для этого хорошо подойдут /usr/local/lib/php или /usr/local/share/php. В Windows для этой цели используйте каталог установки PHP или PEAR, например C:\Program Files\PHP или C:\Program Files\PEAR. Заметьте, что если целевой каталог не содержится в списке «путей поиска» PHP, вы должны добавить его в этот список перед тем, как продолжить.

Содержимое каталога bin/ должно быть перемещено в каталог, входящий в список путей, по которым система осуществляет поиск исполняемых файлов. Если каталог, содержащий ваш исполняемый файл PHP, — обычно это /usr/local/bin в UNIX/Linux и C:\PHP в Windows — уже находится в этом списке, используйте его. В качестве альтернативы можно переместить содержимое bin/ в любой другой каталог, а затем добавить последний в список путей поиска исполняемых файлов.

ПРИМЕЧАНИЕ

Каталог bin/ содержит три сценария: zf.sh, интерфейс командной строки для UNIX/Linux; zf.bat, интерфейс командной строки для Windows; и zf.php, основной «рабочий» сценарий. В UNIX/Linux вам потребуется сделать сценарий zf.sh исполняемым с помощью команды chmod; возможно, вы также захотите переименовать его или создать для него псевдоним (alias), чтобы упростить доступ.

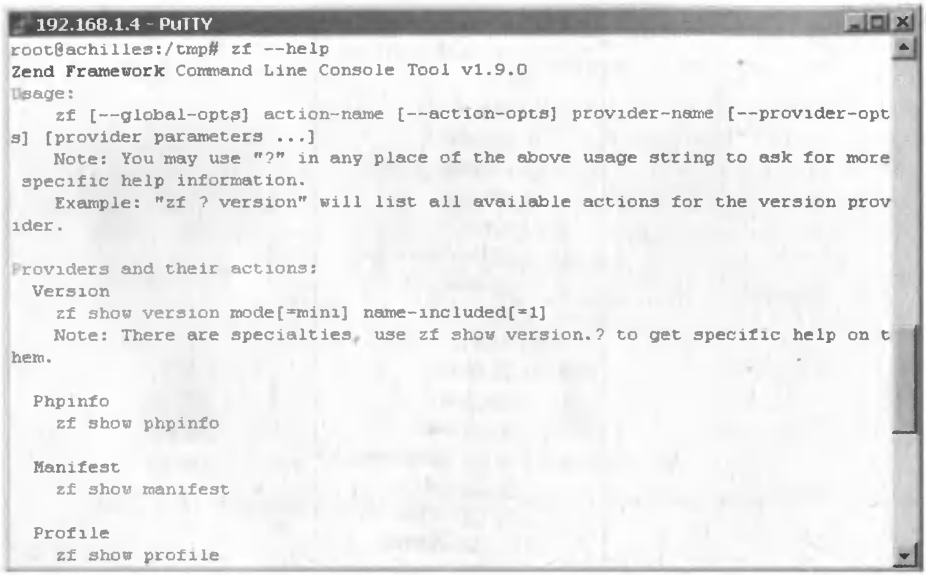


Рис. 1.3. Вывод команды zf --help

Ниже приведен список команд, использующихся для выполнения перечисленных действий:

```
shell> cd ZendFramework-XX
shell> mv library/* /usr/local/lib/php/
shell> mv bin/* /usr/local/bin/
shell> chmod +x /usr/local/bin/zf.sh
shell> ln -s /usr/local/bin/zf.sh /usr/local/bin/zf
```

Теперь вы можете вызывать сценарий zf из командной строки как в Linux, так и в Windows. Попробуйте сделать это, набрав в командной строке следующую команду:

```
shell> zf -help
```

Если все работает правильно, вы увидите список опций. На рис. 1.3 показан вывод приведенной команды в Linux.

Упражнение 1.1. Создание нового проекта

После того как вы установите Zend Framework и убедитесь в работоспособности сценария командной строки zf, вы готовы к созданию приложений с его помощью. Следующие разделы объяснят, как это сделать.

Выяснение требований к приложению

Перед погружением в написание кода стоит потратить несколько минут на то, чтобы разобраться в демонстрационном приложении, которое вы будете создавать на протяжении первой половины этой книги. Данное приложение — это веб-сайт вымышленного магазина, специализирующегося на продаже редких почтовых марок как любителям, так и профессиональным филателистам. Однако в отличие от других хобби-магазинов у этого есть интересная особенность: он выступает в роли интернет-агентства для поиска марок, позволяя коллекционерам загружать в основную базу данных изображения и описания марок, которые они могли бы продать, и давая покупателям возможность искать марки в этой базе по стране, году и ключевому слову. В случае совпадения магазин купит марку у продавца и перепродаст ее покупателю... разумеется, оставив себе награду за труды!

Модераторы сайта получат непосредственный доступ к спискам загруженных марок и смогут вручную подтверждать отображение подходящих марок в списке результатов поиска. Кроме того, они будут иметь доступ к простой системе управления содержимым для создания новостей и пресс-релизов; эта информация будет доступна как через веб-сайт, так и через RSS-ленту. И чтобы вы не скучали, для упрощения интеграции со сторонними приложениями база данных марок также будет доступна через интерфейс SOAP.

Звучит круто? Так и есть. И даже название у этого приложения классное: *Stamp Query and Research Engine* (система поиска и исследования марок), или *SQUARE*.

Демонстрационное приложение SQUARE отвечает всем распространенным требованиям, встречающимся в повседневной разработке приложений: статические страницы, формы ввода, загрузка изображений, защищенная паролем панель администрирования, сортировка данных и разбиение их на страницы, множество вариантов вывода и поиск по ключевым словам. Реализация этой функциональности требует понимания тонкостей обработки форм, валидации ввода, управления сессиями, аутентификации и безопасности, операций CRUD с базой данных, API веб-сервисов и интеграции со сторонними библиотеками. Таким образом, она должна стать хорошей отправной точкой на пути к пониманию разработки приложений с использованием Zend Framework.

Создание каталога приложения

Давайте приступим. Перейдите в корневой каталог документов веб-сервера (обычно **/usr/local/apache/htdocs** в UNIX/Linux или **C:\Program Files\Apache\htdocs** в Windows) и создайте новый подкаталог для приложения. По указанным в предыдущем разделе причинам назовите этот каталог **square/**.

```
Shell> cd /usr/local/apache/htdocs
Shell> mkdir square
```

На протяжении всей книги созданный каталог будет упоминаться как **$APP_DIR**.

Создание каркаса приложения

Следующим шагом будет инициализация приложения и создание основных файлов и каталогов, необходимых для простейшего приложения Zend Framework. Сценарий командной строки zf может автоматически сделать это за вас — просто перейдите в каталог $APP_DIR и выполните следующую команду (рис. 1.4):

```
shell> cd /usr/local/apache/htdocs/square
shell> zf create project
```

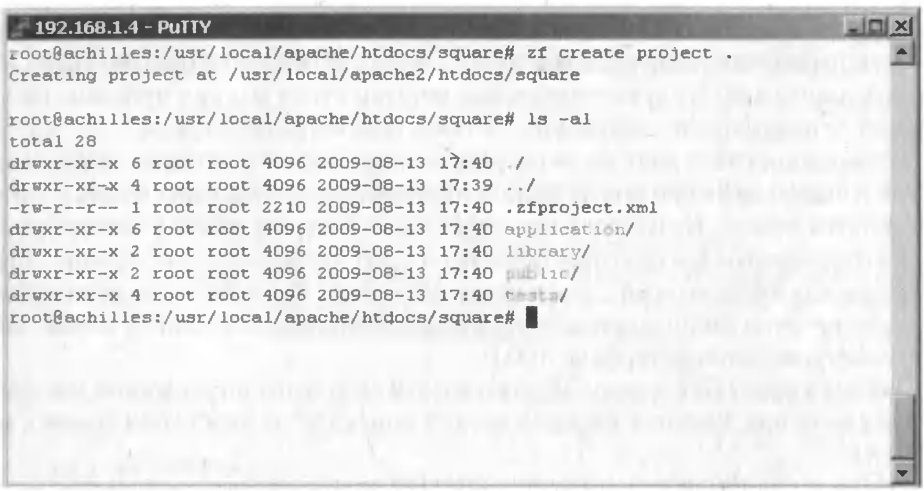


Рис. 1.4. Вывод команды zf create project

Теперь сценарий создаст контейнер для пустого приложения и заполнит его первоначальным набором файлов. Как только процесс завершится, вы увидите несколько новых подкаталогов в каталоге приложения (рис. 1.5).

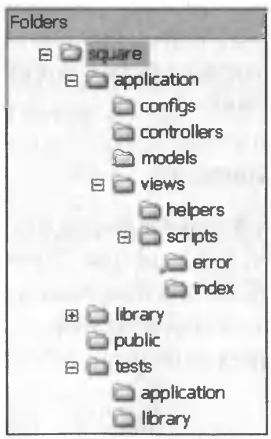


Рис. 1.5. Стандартная структура каталогов нового приложения Zend Framework

Это стандартная структура каталогов для приложений Zend Framework. Каждый каталог имеет свое предназначение, как указано в следующем списке:

- $APP_DIR/application/ — это основной каталог приложения, содержащий весь его код, включая контроллеры, представления и модели;
- $APP_DIR/library/ содержит сторонние библиотеки и классы, использующиеся приложением. Если вы решите объединить Zend Framework с вашим приложением (см. следующий раздел), фреймворк следует поместить именно сюда;
- $APP_DIR/public/ — здесь хранится общедоступное содержимое, такое как изображения и медиафайлы, таблицы стилей CSS, код на JavaScript и прочие статические ресурсы;
- $APP_DIR/tests/ содержит модульные тесты для приложения.

Добавление библиотек Zend Framework

На этом этапе вам потребуется принять важное решение. Вы должны решить, включить ли библиотеки Zend Framework в состав вашего приложения или предоставить их загрузку и установку пользователям. У каждого варианта есть плюсы и минусы:

- Если пользователям придется самостоятельно загружать библиотеки Zend Framework, они всегда будут иметь доступ к последней версии кода (и исправлениям ошибок). Однако этот процесс может испугать новичков, а в случае, если в новых библиотеках нарушена обратная совместимость с использованными библиотеками, могут появиться необычные и трудно отслеживаемые ошибки.
- Объединение библиотек Zend Framework с приложением гарантирует, что пользователи могут приступить к работе с приложением, не имея проблем с несовместимостью версий. Однако это также привязывает пользователей к конкретной версии Zend Framework, потенциально усложняя обновление до новых версий, обладающих дополнительной функциональностью или необходимыми исправлениями ошибок.

В этой книге я предполагаю, что библиотеки Zend Framework объединены с приложением. Поэтому скопируйте содержимое каталога library/ из архива Zend Framework в каталог $APP_DIR/library/, как вы уже делали ранее в этой главе, с помощью приведенной ниже команды. Настройки приложения по умолчанию предполагают, что поиск включаемых библиотек автоматически будет осуществляться в этом каталоге.

```
shell> cp -R /usr/local/lib/php/Zend library/
```

Определение настроек виртуальных хостов

Чтобы упростить доступ к приложению, определите новый виртуальный веб-хост и направьте его в общедоступный каталог приложения. Это необязательный, но рекомендованный шаг, так как он помогает эмулировать «настоящую» среду и пред-

ставляет ресурсы приложения (URL) в таком же виде, в каком пользователи видели бы их в общедоступной среде.

При использовании веб-сервера Apache вы можете создать для приложения именованный виртуальный хост, отредактировав конфигурационный файл Apache (**httpd.conf** или **httpd-vhosts.conf**) и добавив в него следующие строки:

```
NameVirtualHost *:80
<VirtualHost *:80>
        DocumentRoot "/usr/local/apache/htdocs/square/public"
        ServerName square.localhost
</VirtualHost>
```

Эти строки определяют новый виртуальный хост, **http://square.localhost/**, корневой каталог документов которого соответствует каталогу **$APP_DIR/public/**. Чтобы новые параметры вступили в силу, перезапустите веб-сервер. Обратите внимание, что если вы находитесь в сети, вам может потребоваться сообщить локальному DNS-серверу о появлении нового хоста.

Рис. 1.6. Стандартная главная страница приложения

После выполнения указанных шагов откройте веб-браузер и перейдите к только что настроенному виртуальному хосту, набрав в адресной строке URL **http://square.localhost**. Если вы видите страницу приветствия Zend Framework (рис. 1.6), похвалите себя, поскольку вы только что создали и запустили законченное (хотя и чрезвычайно простое) приложение Zend Framework!

Использование средства командной строки

Как показано в предыдущем разделе, сценарий командной строки zf позволяет вам выполнять несколько различных действий. Например, откройте командную строку и выполните следующую команду:

```
shell> zf show version
```

ВОПРОС ЭКСПЕРТУ

В: Могу ли я использовать Zend Framework на виртуальном хостинге, где у меня, вероятнее всего, ограничен или отсутствует контроль над глобальными конфигурационными параметрами PHP, такими как «путь поиска»?

О: Да, разумеется. Существует пара способов сделать это:

- Если вы просто хотите использовать в своем приложении классы Zend Framework, все, что вам требуется сделать, — это скопировать каталог library/Zend в ваш домашний каталог, а затем использовать в коде вашего приложения функцию ini_set(), чтобы динамически добавить новое расположение в путь поиска PHP.
- Если вы хотите использовать сценарий командной строки zf, вам также следует скопировать каталог bin/ в ваш домашний каталог (на один уровень с каталогом library/). После этого у вас должна появиться возможность вызывать сценарий zf как обычно, указывая полный путь к файлу сценария. Это сработает, так как если zf не сможет найти Zend Framework в пути поиска PHP, он произведет поиск в каталоге library/Zend, расположенном на один уровень выше собственного каталога, и в случае успеха будет использовать его. В качестве альтернативного варианта вы можете явно указать сценарию zf, где находится ваша копия Zend Framework, установив подходящее значение переменной окружения ZEND_TOOL_INCLUDE_PATH_PREPEND.

Эта команда отобразит номер версии установленного в данный момент релиза Zend Framework. На рис. 1.7 и 1.8 показаны результаты выполнения команды в Linux и Windows соответственно.

Выполните следующую команду, чтобы получить всю информацию, предоставляемую функцией phpinfo():

```
shell> zf show phpinfo
```

Сценарий командной строки zf также предоставляет быстрый и простой способ просмотра текущего «профиля» вашего приложения. Этот профиль содержит иерархический список текущего содержимого приложения, снабженный описаниями содержащихся в нем файлов, и позволяет взглянуть на приложение «с высоты птичьего полета» без необходимости вручную исследовать каждый каталог.

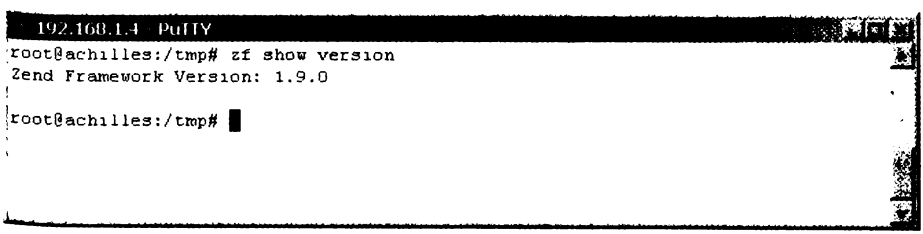


Рис. 1.7. Вывод команды zf show version в Linux

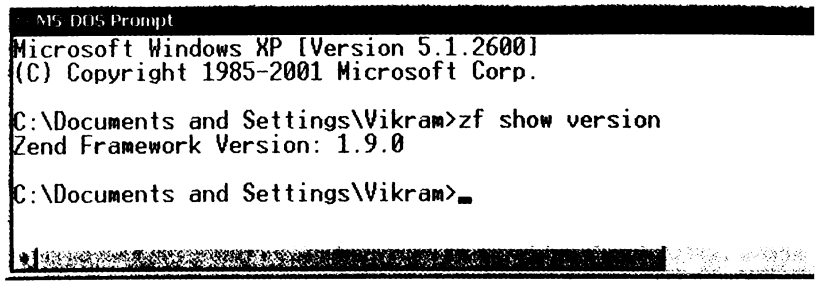


Рис. 1.8. Вывод команды zf show version в Windows

Перейдите в каталог $APP_DIR и запустите следующую команду:

```
shell> zf show profile
```

На рис. 1.9 показаны результаты выполнения команды в Linux.

Рис. 1.9. Пример профиля проекта

Выводы

В этой главе было дано представление о возможностях разработки с использованием Zend Framework с помощью знакомства с проектом и демонстрации некоторых уникальных возможностей и преимуществ этого фреймворка по сравнению с конкурирующими альтернативами. Вы прошли через процесс установки Zend Framework и использования средства командной строки для создания нового проекта. Эти базовые навыки пригодятся в следующей главе, в которой обсуждаются основные принципы разработки приложения и где вы начнете создавать веб-приложение, основанное на фреймворке.

Если вы хотите получить больше информации по изложенным в этой главе темам, используйте следующие ссылки:

- Официальный веб-сайт Zend Framework: http://framework.zend.com/.
- Вики-сайт сообщества Zend Framework: http://framework.zend.com/wiki/.
- Статистика использования Zend Framework: http://framework.zend.com/about/numbers.
- Конкретные примеры использования Zend Framework: http://framework.zend.com/about/casestudies.
- Компоненты Zend Framework: http://framework.zend.com/about/components.
- Средство командной строки Zend Framework: http://framework.zend.com/manual/en/zend.tool.framework.clitool.html.
- Структура файловой системы Zend Framework: http://framework.zend.com/wiki/display/ZFDEV/Choosing+Your+Application%27s+Directory+Layout.

  План разработки Zend Framework: http://framework.zend.com/roadmap.

Работа с моделями, представлениями, контроллерами и маршрутами

Прочитав эту главу, вы:

- познакомитесь с основами паттерна Модель–Представление–Контроллер;
- выясните, как приложение Zend Framework обрабатывает запросы URL;
- оцените преимущества модульного расположения каталогов;
- определите и примените глобальный шаблон к представлениям приложения;
- создадите собственные маршруты для ресурсов приложения;
- научитесь обрабатывать страницы со статическим содержимым.

В предыдущей главе вы получили небольшое представление о Zend Framework, пройдя через процесс установки фреймворка и запуска нового проекта. Теперь вам требуется добавить к каркасу приложения код, делающий его функциональным. В данной главе вы познакомитесь с фундаментальными принципами проектирования приложения Zend Framework и примените эти знания к задаче создания реального приложения. Поэтому давайте без лишних слов приступим к делу!

Базовые принципы

Для разработки приложение на PHP типичным является внедрение PHP-кода в один или несколько HTML-документов с помощью специальных разделителей. Это упрощает создание динамических веб-страниц, содержащих конструкции языков программирования, такие как переменные или вызовы функций; достаточно просто изменить значения переменных, внедренных в HTML-код, и содержимое страницы соответствующим образом изменится.

Однако каждый разработчик приложений знает, что за это удобство нужно платить. Описанный в предыдущем параграфе подход приводит к созданию сце-

нариев PHP, которые настолько тесно переплетены с HTML-кодом, что их сопровождение становится ночным кошмаром. Поскольку один и тот же физический файл содержит и элементы интерфейса, определенные посредством HTML, и код на PHP, разработчики и дизайнеры интерфейса должны согласовывать изменения, прежде чем вносить их. Наиболее распространенный пример: когда дизайнерам интерфейса требуется изменить внешний вид веб-приложения, разработчик должен следить за вносимыми изменениями, чтобы гарантировать целостность бизнес-логики.

Такой подход легко распознать по наиболее заметному признаку: злые и невыспавшиеся разработчики и дизайнеры толкаются возле компьютера и спорят о том, чья очередь пользоваться клавиатурой. Нет смысла говорить, что помимо нервотрепки и получающегося в результате неоптимального кода реализация подобного подхода потребует больше времени и денежных средств, чем необходимо для решения конкретной задачи. Именно здесь может помочь Zend Framework.

Приложения Zend Framework создаются в соответствии с общепринятым набором принципов, способствующих повторному использованию кода, простоте его сопровождения и масштабируемости. Одной из основ этого подхода является паттерн проектирования Модель–Представление–Контроллер (Model-View-Controller, MVC), отделяющий бизнес-логику приложения от пользовательского интерфейса и моделей данных и разрешающий работать с ними по отдельности. Паттерн MVC также способствует эффективной организации и разделению обязанностей приложения и позволяет независимо тестировать различные компоненты.

В следующих разделах рассказывается о ключевых компонентах паттерна MVC и приводятся примечания, связанные с реализацией этого паттерна в Zend Framework.

Модели

Каждое приложение, будь то простой запрос имени пользователя и пароля или многовалютная корзина в интернет-магазине, управляется данными. В паттерне MVC этот «слой данных» представлен одной или несколькими *моделями*, которые предоставляют функции для получения, сохранения, удаления и иных действий с данными приложения. Слой данных безразличен к способу их вывода: он имеет дело только с самими данными и не имеет никакого отношения к тому, как эти данные будут представлены пользователю. В сущности, он предоставляет логически независимый интерфейс для манипулирования данными приложения.

Чтобы проиллюстрировать это, рассмотрим простое веб-приложение, которое позволяет пользователям отправлять классифицированные объявления, связанные с подержанными автомобилями. В соответствии с паттерном MVC данные этого приложения — списки автомобилей — можно представить моделью Listing, которая предоставляет методы для взаимодействия с лежащими в ее основе данными. Эта модель не связана с визуальным отображением списков; напротив, она содержит функции, необходимые для доступа и модификации отдельных списков и их атрибутов в хранилище данных.

Ниже приведен пример того, как могла бы выглядеть такая модель:

```
<?php
class ListingModel
{
    public function __construct()
    {
            // конструктор
    }
    public function read($id)
    {
            // код для получения единственного списка по его идентификатору
    }
    public function find($criteria)
    {
            // код для получения списков, соответствующих заданным критериям
    }
    public function save()
    {
            // код для вставки или обновления списка
    }
    public function delete()
    {
            // код для удаления списка
    }
}
?>
```

Если данные приложения хранятся в базе данных, например MySQL, SQLite или PostgreSQL, модели могут использовать лежащий в ее основе *слой абстракции* для решения задач по управлению подключениями к базе и выполнению SQL-запросов. Zend Framework содержит слой абстракции для баз данных, **Zend_Db**, который предоставляет единообразный интерфейс к множеству различных СУБД (систем управления безами денных), а модели в Zend Framework обычно описываются с использованием паттерна Data Mapper. Также в приложение Zend Framework можно легко интегрировать сторонние модели, например, созданные с помощью средств объектно-реляционного отображения (Object-Relational Mapping, ORM), таких как Doctrine и Propel.

Представления

Тогда как модели занимаются только доступом и модификацией необработанных данных приложения, *представления* отвечают за то, как эти данные будут представлены пользователю. Представления можно считать *слоем пользовательского интерфейса*, ответственным за отображение данных, но неспособным получать к ним непосредственный доступ или изменять их. Представления могут получать данные от пользователя, но их «обязанности», опять же, ограничены созданием внешнего вида и поведения формы ввода; они не занимаются обработкой, очисткой или валидацией входных данных.

В вышеупомянутом приложении для систематизации объявлений представления отвечают за отображение текущих списков и за генерацию форм, используемых для ввода новых списков. Например, может существовать представление для отображения всех последних списков в определенной категории и представление для ввода новых списков. Во всех случаях контроллер и/или модель решает задачи получения и обработки данных, в то время как представление приводит эти данные к формату, приемлемому для отображения, а затем показывает их пользователю.

Ниже приведен пример представления, предназначенного для отображения последних списков:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <meta http-equiv="Content-Type" content="text/html;
        charset=utf-8"/>
        <base href="/" />
        </head>
    <body>
        <div id="header">
                    <div id="logo">
                            <img src="/images/logo.gif" />
                    </div>
        </div>

        <div id="content">
            <h1>Recent listings</h1>
            <?php foreach ($this->listings as $l): ?>
                <div class="listing">
                    <h3 class="listing_title">
                        <a href="/listing/view/<?php echo $l->id; ?>">
                                <?php echo $l->title; ?>
                        </a>
                    </h3>
                    <span class="listing_content">
                            <?php echo $l->content; ?>
                    </span>
                </div>
            <?php endforeach; ?>
        </div>
    </body>
</html>
```

Как показано в этом примере, представления описываются в виде сценариев PHP, содержащих HTML-код или разметку, необходимую для правильного формирования вывода и отображения его пользователю. Эти сценарии могут также содержать переменные-заполнители для динамических данных; значения этих заполнителей устанавливаются соответствующим контроллером и вставляются в представление при его формировании. В Zend Framework сценарии представ-

ления формируются компонентом Zend_View, который кроме этого предоставляет вспомогательные функции для экранирования вывода и набор «вспомогательных классов» для решения распространенных задач, связанных с представлением, таких как навигация, создание мета-тегов и генерация ссылок. Также довольно просто интегрировать в приложение Zend Framework сторонние механизмы шаблонизации, например Smarty или Savant, расширив абстрактный класс Zend_View_Interface.

Контроллеры

Контроллеры — это связующее звено между моделями и представлениями. С помощью моделей они вносят изменения в данные приложений, а затем вызывают представления для отображения результатов для пользователя. Контроллер может быть связан с несколькими представлениями и может использовать разные представления в зависимости от результата, который требуется отобразить в любой конкретный момент времени. Таким образом, контроллеры можно считать *слоем обработки*, реагирующим на действия пользователей и инициирующим изменения в состоянии приложения и отображение нового состояния для пользователя.

В вышеупомянутом приложении контроллеры отвечают за чтение и валидацию параметров запроса, сохранение и получение списков с помощью функций модели и выбор подходящих представлений для отображения деталей списка. Например, контроллер может перехватывать запрос на получение последних списков, запрашивать последние записи у модели ListingModel, выбирать подходящее представление, подставлять в него записи и формировать страницу. Например, если пользователь хочет добавить новый список, контроллер выбирает и формирует представление для ввода данных, проводит валидацию и очистку пользовательского ввода, добавляет очищенные входные данные в хранилище данных с помощью модели ListingModel, а затем выбирает и формирует другое представление, сообщающее об успешном завершении операции или об ошибке.

Ниже приведен пример контроллера, который объединяет приведенные выше модель и представление:

```
<?php
class ListingController
{
    function indexAction()
    {
        // код для инициализации модели
        // и получения данных
        $listing = new ListingModel();
        $matches = $listing->find(
            array(
            'date' => '-10 days',
            'status' => 'published'
             )
        );
        // код для инициализации представления
        // и заполнения его данными, полученными от модели
```

```
        $view = new ListingView();
        $view->listings = $matches;
        echo $view->render('recent.php');
    }
}
?>
```

Как показано в этом примере, контроллер является посредником, вызывающим методы модели для проведения операций с данными приложения и использующим представления для отображения результатов этих операций. В Zend Framework контроллеры являются наследниками класса Zend_Controller_Action и содержат методы, иначе называемые *действиями*, имеющие код, необходимый для взаимодействия с моделями и представлениями. Существует также «сверхконтроллер», *front-контроллер*, ответственный за перехват запросов пользователя и вызов подходящих контроллеров и методов-действий для их обслуживания (более подробно об этом рассказывается в следующем разделе).

Помимо трех ключевых компонентов, описанных выше, Zend Framework также вносит в процесс разработки приложения некоторые дополнительные идеи. Они описаны в следующих разделах.

Модули

По умолчанию все модели, контроллеры, действия и представления находятся в модуле default. Однако зачастую у вас может возникать желание сгруппировать модели, представления и контроллеры в неких «коробках», разделив их по различным функциональным областям вашего приложения. Это можно сделать с помощью *модулей*. Например, если ваше приложение содержит функции поиска, управления профилями пользователей и новости, вы можете создать отдельные модули search, profile и news и поместить в каждый из них соответствующие модели, представления и контроллеры.

Модули представляют собой удобный способ организации кода приложения или способ создания сторонних компонентов приложения, которые легко добавляются к существующей установке. Стандартный маршрутизатор Zend Framework полностью поддерживает работу с модулями, позволяя приступить к их использованию в приложении без написания собственного кода. Каждый модуль (за исключением модуля default) имеет свое пространство имен, это предотвращает возникновение коллизий между объектами или переменными.

Маршруты

Маршруты — это связующее звено между пользовательскими запросами и выполняемыми действиями. Когда пользователь делает запрос к приложению по определенному URL, front-контроллер перехватывает этот запрос и, основываясь на шаблоне URL, принимает решение о том, какие контроллер и действие должны быть вызваны для обработки запроса. Этот процесс *маршрутизации* запросов контроллерам является ключевой частью потока выполнения приложения и обладает

большими возможностями для настройки. Для сопоставления шаблона URL маршруты используют регулярные выражения и могут быть описаны с использованием синтаксиса XML- или INI-файлов.

По умолчанию Zend Framework содержит несколько стандартных маршрутов, подходящих для приложений малой или средней сложности. Эти стандартные маршруты предполагают, что URL в приложении имеют вид **/модуль/контроллер/действие**, и соответствующим образом перенаправляют пользовательские запросы. Например, запрос URL **http://application/auto/listing/index** может автоматически сопоставляться действию **ListingController::indexAction** в модуле **auto**. Обратите внимание, что имена контроллера и действия подчиняются определенным правилам использования регистра букв и именования.

ПРИМЕЧАНИЕ

Подсистема маршрутизации Zend Framework автоматически применяет значения по умолчанию к маршрутам, формат которых отличается от /модуль/контроллер/действие. Эти значения по умолчанию и их влияние на сопоставление запрашиваемых URL контроллерам и действиям рассматриваются в разделе «Взаимодействие компонентов».

Если стандартные маршруты, описанные в предыдущем параграфе, являются слишком ограниченными или недостаточно гибкими для вашего приложения, Zend Framework разрешает использование маршрутов, определенных пользователем. Они поддерживают (помимо всего прочего) необязательные и обязательные параметры, значения по умолчанию и объединение нескольких маршрутов в цепочки, а также позволяют подробно настроить маршрутизацию вашего приложения таким образом, что адреса URL не требуют отражения внутренней системы контроллеров, действий и модулей.

Макеты

В классическом значении макет определяет то, каким образом размещается совокупность элементов. В контексте приложения Zend Framework *макеты* можно считать шаблонами интерфейса, предоставляющими способ «украсить» содержимое приложения с помощью одного или нескольких стандартных интерфейсов, применяемых ко всему приложению. Макеты дают возможность поместить распространенные элементы интерфейса, такие как верхние и нижние колонтитулы или элементы навигации по сайту, в отдельные файлы, которые редактируются и поддерживаются независимо от конкретных представлений.

Довольно часто приложение имеет несколько макетов. Возможно, вы захотите предоставить пользователям один интерфейс, а администраторам — другой, или порадовать пользователей самостоятельным выбором темы интерфейса из предложенного множества тем. Макеты упрощают реализацию этих и других возможностей шаблонизации.

ПРИМЕЧАНИЕ

Работать с модулями, пользовательскими маршрутами и макетами вы начнете далее в этой главе.

Взаимодействие между компонентами

Если считать приложение Zend Framework цирком (зачастую эта аналогия более чем уместна), то front-контроллер — это инспектор манежа, который управляет всеми выступлениями и следит за тем, чтобы публика была довольна. Этот раздел более подробно рассматривает роль front-контроллера, демонстрируя его работу с помощью шагов, осуществляющихся при перехвате и выполнении запроса какого-либо ресурса приложения.

На рис. 2.1 показано движение запроса в типичном приложении Zend Framework.

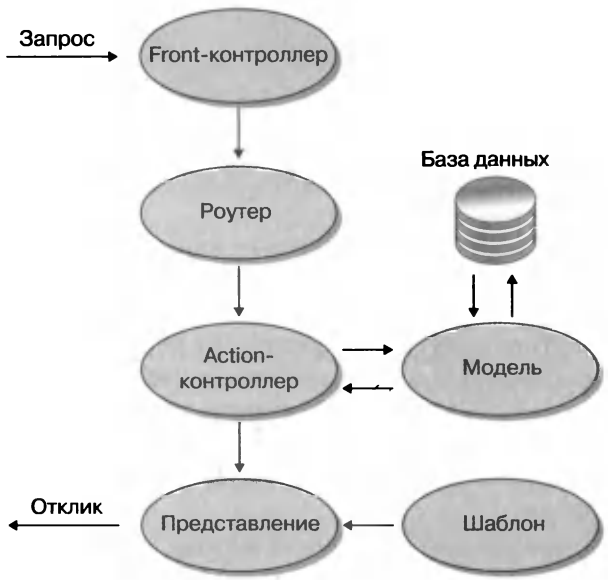


Рис. 2.1. Взаимодействие между моделями, представлениями и контроллерами

Поскольку перед контроллером ставится задача перехвата запросов клиента и направления их подходящему целевому объекту для формирования ответа, front-контроллер играет ключевую роль в движении запроса через приложение.

1. Когда приходит запрос, файл .htaccess веб-сервера автоматически приводит его к стандартному формату и передает сценарию index.php. Этот сценарий подготавливает окружение приложения, читает его конфигурационный файл и создает экземпляр front-контроллера.
2. Front-контроллер анализирует запрос и определяет основные компоненты URL. Затем он направляет запрос подходящему контроллеру. Для выполнения этой маршрутизации front-контроллер проверяет как стандартные, так и пользовательские маршруты и задействует методы сопоставления по шаблону для выбора цели запроса.
3. Если найдено совпадение, front-контроллер передает управление соответствующему контроллеру. Будучи вызванным, действие вносит изменения в состоя-

ние приложения, используя одну или несколько моделей. Оно также выбирает отображаемое представление и устанавливает в нем необходимые параметры. После завершения действия выбранное представление формирует страницу с его выводом, в случае необходимости помещая вывод в макет. После чего этот вывод возвращается клиенту.

4. Если ни один из определенных в приложении маршрутов не совпадает с запросом, выбрасывается исключение и вызываются контроллер и действие, предназначенные для обработки ошибки. В зависимости от параметров исключения действие формирует представление, содержащее сообщение об ошибке. Затем этот вывод возвращается клиенту.

ПРИМЕЧАНИЕ

Любые неперехваченные исключения, сгенерированные в процессе обработки запроса, вызовут контроллер, обрабатывающий ошибки, и соответствующее действие. В результате будет создано представление, содержащее сообщение об ошибке, возвращаемое сформировавшему запрос клиенту.

Как говорилось ранее, подсистема маршрутизации автоматически сопоставляет URL в формате /модуль/контроллер/действие соответствующим модулю, контроллеру и действию. Например, для доступа к действию ListingController::saveAction в модуле auto вам необходимо будет запросить URL http://application/auto/listing/save. Аналогично, для доступа к действию NewsController::editAction в модуле content вам потребуется запросить URL http://application/content/news/edit.

Если подсистема маршрутизации получает запрос, не соответствующий формату /модуль/контроллер/действие, она автоматически использует следующие параметры по умолчанию:

- Для запросов, в которых отсутствует имя модуля, подсистема маршрутизации автоматически использует модуль default.
- Для запросов, в которых отсутствует имя контроллера, подсистема маршрутизации автоматически использует IndexController выбранного модуля.
- Для запросов, в которых отсутствует имя действия, подсистема маршрутизации автоматически использует indexAction выбранного контроллера.

Чтобы лучше понять, как перечисленные параметры по умолчанию применяются на практике, рассмотрим следующие примеры:

- К действию ContactController::sendAction в модуле default можно обратиться как по URL http://application/default/contact/send, так и по URL http://application/contact/send.
- К действию PostController::indexAction в модуле default можно обратиться как по URL http://application/default/post/index, так и по URL http://application/post/.
- К действию NewsController::indexAction в модуле content можно обратиться как по URL http://application/content/news/index, так и по URL http://application/content/news.

Устройство стандартной главной страницы

Вооружившись всей полученной информацией, давайте вернемся к коду, сгенерированному средством командной строки zf в главе 1, и детально рассмотрим, каким образом генерируется стандартная страница приветствия Zend Framework. Мы начнем с самого начала, с запроса http://square.localhost/. В результате выполнения этого запроса создается стандартная главная страница приложения (рис. 2.2).

Рис. 2.2. Стандартная главная страница приложения

Как это происходит? Начнем с того, что при запросе URL http://square.localhost/ применяются маршруты по умолчанию, описанные в предыдущем разделе, и URL автоматически приводится к виду http://square.localhost/default/index/index. Затем подсистема маршрутизации Zend Framework перенаправляет этот запрос методу indexAction() контроллера IndexController в модуле default.

Эти контроллер и действие автоматически создаются сценарием командной строки zf и обычно сохраняются в файл $APP_DIR/application/controllers/IndexController.php. Ниже приведен пример этого файла:

```
<?php
class IndexController extends Zend_Controller_Action
{
    public function init()
    {
        /* Initialize action controller here */
    }
    public function indexAction()
    {
        // action body
    }
}
```

Здесь метод indexAction() представляет собой простую заглушку, не имеющую исполняемого кода. Когда этот метод завершает выполнение, он автоматически выбирает для формирования свое представление по умолчанию. По общепринятому соглашению это представление называется по имени контроллера и действия и располагается в соответствующем каталоге сценариев представлений; в данном случае это файл $APP_DIR/application/views/scripts/index/index.phtml. Откройте его, и вы увидите HTML-разметку, создающую вывод, показанный на рис. 2.2:

```
<style>
    a:link,
    a:visited
    {
        color: #0398CA;
    }

    span#zf-name
    {
        color: #91BE3F;
    }

    div#welcome
    {
        color: #FFFFFF;
        background-image: url(
        http://framework.zend.com/images/bkg_header.jpg);
        width: 600px;
        height: 400px;
        border: 2px solid #444444;
        overflow: hidden;
        text-align: center;
    }

    div#more-information
    {
        background-image: url(
          http://framework.zend.com/images/bkg_body-bottom.gif);
        height: 100%;
    }
</style>
    <div id="welcome">
    <h1>Welcome to the <span id="zf-name">Zend Framework!</span></h1>

    <h3>This is your project's main page</h3>

    <div id="more-information">
        <p><img src=
          "http://framework.zend.com/images/PoweredBy_ZF_4LightBG.png"/>
        </p>
        <p>
          Helpful Links: <br />
          <a href="http://framework.zend.com/">Zend Framework Website</a> |
          <a href="http://framework.zend.com/manual/en/">
          Zend Framework Manual</a>
        </p>
    </div>
</div>
```

Из этой разметки видно, что если вы корректно называете и размещаете свои контроллеры, действия и представления, вам не потребуется выполнять слишком много работы. Фреймворк автоматически найдет и выполнит файлы, используя маршруты по умолчанию и не требуя вашего вмешательства. По мере ознакомления с материалом книги вы узнаете больше о том, как использовать стандартные соглашения Zend Framework в своих интересах, уменьшив количество кода, который необходимо написать, чтобы настроить и запустить приложение.

Модульная структура каталогов

В предыдущем разделе вы видели, как сценарий командной строки **zf** создает структуру каталогов для вашего новбго приложения Zend Framework. В своем первоначальном виде эта структура содержит только каталоги, необходимые для запуска простейшего тестового приложения. По мере добавления в приложение новой функциональности вам также потребуется расширить эту базовую структуру и создать дополнительные каталоги для различных типов данных.

В качестве примера взгляните на рис. 2.3, на котором изображена полная структура каталогов приложения Zend Framework.

Каждый из показанных на рисунке каталогов имеет свое предназначение (табл. 2.1).

Таблица 2.1. Основные каталоги в приложении Zend Framework

Каталог	Описание
$APP_DIR/application	Основной каталог приложения
$APP_DIR/application/controllers	Глобальные контроллеры
$APP_DIR/application/views	Глобальные представления
$APP_DIR/application/models	Глобальные модели
$APP_DIR/application/configs	Глобальные конфигурационные параметры
$APP_DIR/application/layouts	Глобальные макеты
$APP_DIR/application/modules	Модули
$APP_DIR/library	Сторонние библиотеки и классы
$APP_DIR/public	Основной общедоступный каталог
$APP_DIR/public/css	Таблицы стилей CSS
$APP_DIR/public/js	Код на JavaScript
$APP_DIR/public/images	Изображения для приложения
$APP_DIR/tests	Модульные тесты
$APP_DIR/temp	Временные данные

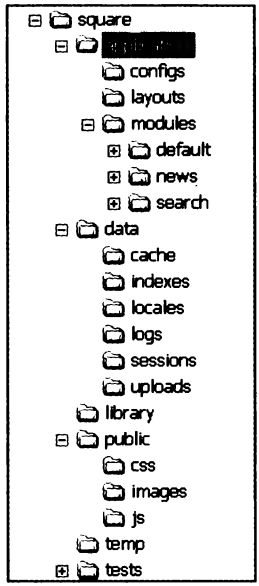


Рис. 2.3. Рекомендованная структура каталогов для приложения Zend Framework

Отдельного упоминания заслуживает каталог **$APP_DIR/application/modules**. Он предназначен для хранения модулей приложения, при этом каждый модуль представлен подкаталогом в каталоге **$APP_DIR/application/modules/**. Внутренняя структура каталога каждого модуля повторяет структуру глобального каталога **$APP_DIR/application/** (рис. 2.4).

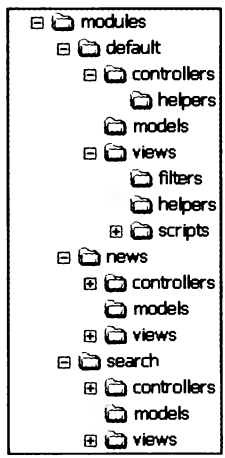


Рис. 2.4. Рекомендованная структура каталогов для модуля приложения Zend Framework

Таким образом, данная структура каталогов разделяет глобальные контроллеры, представления и модели приложения, хранящиеся в иерархии **$APP_DIR/application/**,

и контроллеры, представления и модели отдельных модулей, хранящиеся в иерархии $APP_DIR/application/modules/.

С точки зрения разработчика, выбор расположения кода вашего приложения — вещь сугубо субъективная. Не существует единственного «правильного» подхода, поэтому вы можете хранить свой код в глобальных каталогах, в каталогах модулей или же одновременно использовать оба варианта, в зависимости от того, какой подход будет соответствовать требованиям и структуре вашего приложения.

Тем не менее демонстрационное приложение SQUARE, описанное в этих главах, активно использует модули, и на протяжении всей книги мы будем рекомендовать использовать модули для разработки приложений Zend Framework. Тому есть несколько причин:

- Организация кода в виде модулей создает структурированную иерархию кода, поскольку все контроллеры, представления и модели, относящиеся к конкретной функции или набору функций, хранятся в одном дереве каталогов. Основанная на модулях структура каталогов также делает очевидными области логического разделения приложения, не требуя написания дополнительной документации, а кроме того, она упрощает сопровождение.
- Организация кода в виде модулей способствует созданию более надежного и расширяемого программного обеспечения. Модули могут быть сгруппированы в виде независимых пакетов, содержащих собственные контроллеры, представления и модели. Таким образом, модули можно считать компонентами для многократного использования, которые подключаются к существующему приложению для оперативного добавления в него новой функциональности.

При использовании модулей со стандартными маршрутами Zend Framework необходимо включать имя модуля в запрос URL в формате /модуль/контроллер/действие. Если подсистема маршрутизации получает запрос, не содержащий имени модуля, она автоматически осуществляет поиск в расположении, которое считается модулем default, — в глобальном каталоге приложения, $APP_DIR/application/.

Это поведение может создавать путаницу и непоследовательность в используемых в приложении URL, если приложение также содержит дополнительные модули в иерархии $APP_DIR/application/modules/. Следовательно, если вы придерживаетесь модульной структуры каталогов, правильнее будет явно создать каталог для модуля default и переместить контроллеры, представления и модели, находящиеся в иерархии $APP_DIR/application/*, в иерархию $APP_DIR/application/modules/default/*.

Упражнение 2.1. Использование модульной структуры каталогов

Как обсуждалось в предыдущем разделе, модульная структура каталогов способствует логичной организации и созданию более простой в сопровождении иерархии кода. В качестве предварительной подготовки к началу разработки приложения

в следующих разделах объясняется, как применить эту структуру к демонстрационному приложению, созданному в главе 1.

Создание модуля по умолчанию (default)

Первым шагом будет создание каталога $APP_DIR/application/modules и набора подкаталогов для модуля по умолчанию, а также его контроллеров и представлений. Средство командной строки zf не создает эти каталоги, поэтому их необходимо создать вручную.

```
shell> cd /usr/local/apache/htdocs/square/application
shell> mkdir modules
shell> mkdir modules/default
```

Теперь переместите существующие модели, контроллеры и представления из $APP_DIR/application/* в $APP_DIR/application/modules/default/*:

```
shell> mv controllers modules/default/
shell> mv views modules/default/
shell> mv models modules/default/
```

Обновление конфигурационного файла приложения

Следующим шагом будет добавление пути к каталогу модулей в глобальный конфигурационный файл приложения, $APP_DIR/application/configs/application.ini. Таким образом мы сообщаем подсистеме маршрутизации Zend Framework, как следует разрешать специфичные для модулей пути.

Чтобы произвести обновление, откройте конфигурационный файл приложения в текстовом редакторе и добавьте в раздел [production] следующие строки:

```
resources.frontController.moduleDirectory = APPLICATION_PATH "/modules"
resources.modules = ""
```

После выполнения этих шагов попробуйте открыть главную страницу приложения, используя URL http://square.localhost/ и http://square.localhost/default/index/index. Если все работает правильно, в обоих случаях вы должны увидеть главную страницу, показанную на рис. 2.2.

СОВЕТ

Независимо от того, разбиваете вы приложение на модули или нет, помните, что с помощью пользовательских маршрутов вы всегда можете перенаправить запросы URL конкретным модулям, контроллерам и действиям.

Основные макеты и пользовательские маршруты

Вы уже имеете достаточное прдставление о внутреннем устройстве приложения Zend Framework, чтобы приступить к написанию собственного кода. Следующие разделы помогут вам создать собственную страницу приветствия, провести настройки основного макета для приложения и проложить пользовательский маршрут к нему.

ВОПРОС ЭКСПЕРТУ

В: Как получается, что результатом обращения к разным URL http://square.localhost/ и http://square.localhost/default/index/index является одна и та же страница?

О: Подсистема маршрутизации Zend Framework автоматически применяет ряд стандартных настроек к маршрутам, не соответствующим стандартному формату /модуль/контроллер/действие. Проще говоря, при отсутствии имени модуля используется модуль default, а при отсутствии имен контроллера и действия используется метод indexAction() контроллера IndexController в выбранном модуле. В результате этих стандартных подстановок запрос http://square.localhost/ автоматически заменяется на http://square.localhost/default/index/index и направляется для выполнения действию IndexController::indexAction в модуле default.

В: Какие соглашения об именовании контроллеров, действий и представлений применяются в Zend Framework?

О: Zend Framework использует стиль CamelCase для названий контроллеров и действий. В именах контроллеров используется прописная первая буква (UpperCamelCase) и суффикс Controller (примеры: IndexController, StaticContentController, FunkyChickenController), а в именах действий — строчная первая буква (lowerCamelCase) и суффикс Action (примеры: indexAction, displayPostAction, redButtonAction). Для модулей, отличных от default, в именах контроллеров должен использоваться дополнительный префикс в виде названия модуля (примеры: News_IndexController, Catalog_EntryController).

Имена сценариев представлений основываются на названиях соответствующих контроллеров и действий. Обычно сценарий представления находится в каталоге, соответствующем имени контроллера (без суффикса Controller), в файле, имя которого соответствует имени действия (без суффикса Action). Следовательно, сценарий представления для действия clickAction в контроллере ExampleController может располагаться в файле /views/scripts/example/click.phtml.

Для разделения слов в имени контроллера или действия в пути к файлу сценария представления используются дефисы или точки. Следовательно, сценарий представления для действия displayItemAction в контроллере ShoppingCartController может располагаться в файле /views/scripts/shopping-cart/display-item.phtml.

В целях предотвращения возникновения ошибок Zend Framework также позволяет создавать пользовательские пространства имен, которые можно добавлять к названиям объектов или классов в качестве префикса. Чтобы соответствующие определения автоматически загружались по мере необходимости, эти пространства имен можно зарегистрировать в автозагрузчике Zend Framework. Пример вы увидите в главе 3.

бновление главной страницы приложения

так, хотя стандартная страница приветствия, созданная средством командной роки zf, выглядит довольно симпатично, она не является тем, что хотели бы идеть реальные пользователи. Как насчет добавления в нее собственного приетствия и какой-нибудь информации о приложении?

Вы уже знаете, что главная страница приложения обслуживается действием dexController::indexAction модуля default. В модульной структуре представление, ответствующее контроллеру и действию, находится в файле $APP_DIR/application/

modules/default/views/scripts/index/index.phtml. Откройте этот файл в текстовом редакторе и замените его содержимое следующей разметкой:

```
<h2>Welcome!</h2>
<p>Welcome to SQUARE, our cutting-edge Web search application for rare
stamps.</p>
<p>We have a wide collection of stamps in our catalog for your
browsing pleasure, and we also list <strong>hundreds of thousands of
stamps</strong> from individual collectors across the country. If
you find something you like, drop us a line and we'll do our best to
obtain it for you. Needless to say, all stamps purchased through us
come with a certificate of authenticity, and <strong>our unique 60-day
money-back guarantee</strong>.</p>
<p>The SQUARE application is designed to be as user-friendly as
possible. Use the links in the menu above to navigate and begin
searching.</p>c
```

Сохраните изменения и снова перейдите по адресу http://square.localhost/. Вы должны увидеть измененную главную страницу (рис. 2.5).

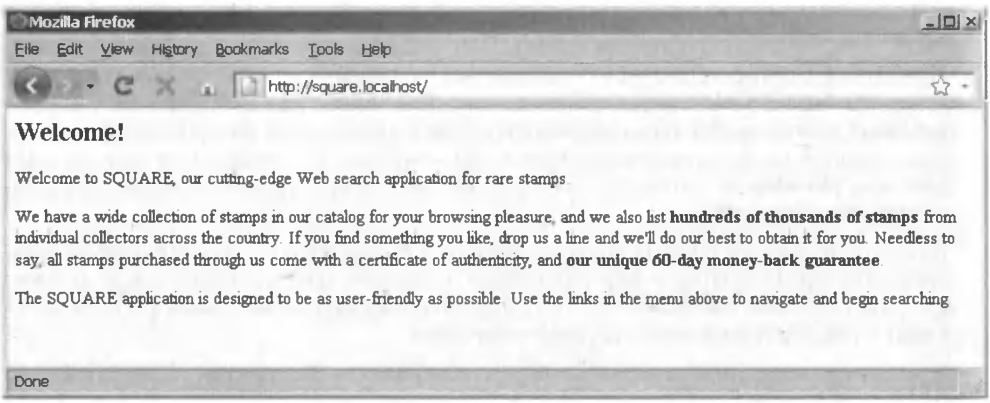


Рис. 2.5. Обновленная главная страница приложения

Установка основного макета

Теперь главная страница приложения отображает нужное содержимое, но она все еще далека от совершенства. Поэтому следующим этапом будет придание ей блеска путем добавления нескольких изображений и навигации. Эти элементы будут присутствовать на всех страницах приложения, а это значит, что хоть вы и можете добавлять их к каждому представлению, лучше размещать их в основном макете, в котором помещаются все представления. При использовании макета не только тратится меньше времени, но и упрощается внесение изменений по мере развития приложения, так как макет хранится в единственном файле, использующемся всеми представлениями.

Чтобы установить для приложения основной макет, выполните следующие шаги.

Создание файла-шаблона для макета

Для начала создайте каталог **$APP_DIR/application/layouts/**, который по умолчанию используется в рекомендованной структуре каталогов Zend Framework для хранения файлов макетов.

```
shell> cd /usr/local/apache/htdocs/square/application
shell> mkdir layouts
```

В этом каталоге создайте новый текстовый файл, содержащий следующую разметку:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8"/>
    <base href="/" />
    <link rel="stylesheet" type="text/css" href="/css/master.css" />
  </head>
  <body>
    <div id="header">
      <div id="logo">
        <img src="/images/logo.gif" />
      </div>

       <div id="menu">
        <a href="#">HOME</a>
        <a href="#">SERVICES</a>
         <a href="#">CONTACT</a>
      </div>
    </div>

    <div id="content">
      <?php echo $this->layout()->content ?>
    </div>

    <div id="footer">
      <p>
      Created with <a href="http://framework.zend.com/">
      Zend Framework</a>. Licensed under
      <a href="http://www.creativecommons.org/">Creative Commons</a>.
      </p>
    </div>
  </body>
</html>
```

Сохраните этот файл под именем **$APP_DIR/application/layouts/master.phtml**.

Вы, наверное, заметили, что основной макет использует два дополнительных файла — таблицу стилей CSS и изображение-логотип. Эти файлы должны находиться в общедоступной области приложения, чтобы подключившиеся клиенты

могли получить их по протоколу HTTP. Соответственно создайте каталоги **$APP_DIR/public/css/** и **$APP_DIR/public/images/** и скопируйте в них необходимые файлы, которые можно найти в архиве с дополнительными материалами к этой главе.

Обновление конфигурационного файла приложения

Следующим шагом будет добавление в глобальный конфигурационный файл, **$APP_DIR/application/configs/application.ini**, пути к каталогу с макетами и имени макета, используемого по умолчанию при формировании представлений. Откройте файл в текстовом редакторе и добавьте в секцию **[production]** следующие параметры:

```
resources.layout.layoutPath = APPLICATION_PATH "/layouts"
resources.layout.layout = master
```

Теперь при переходе на главную страницу приложения вы увидите новый макет во всем его великолепии, как показано на рис. 2.6.

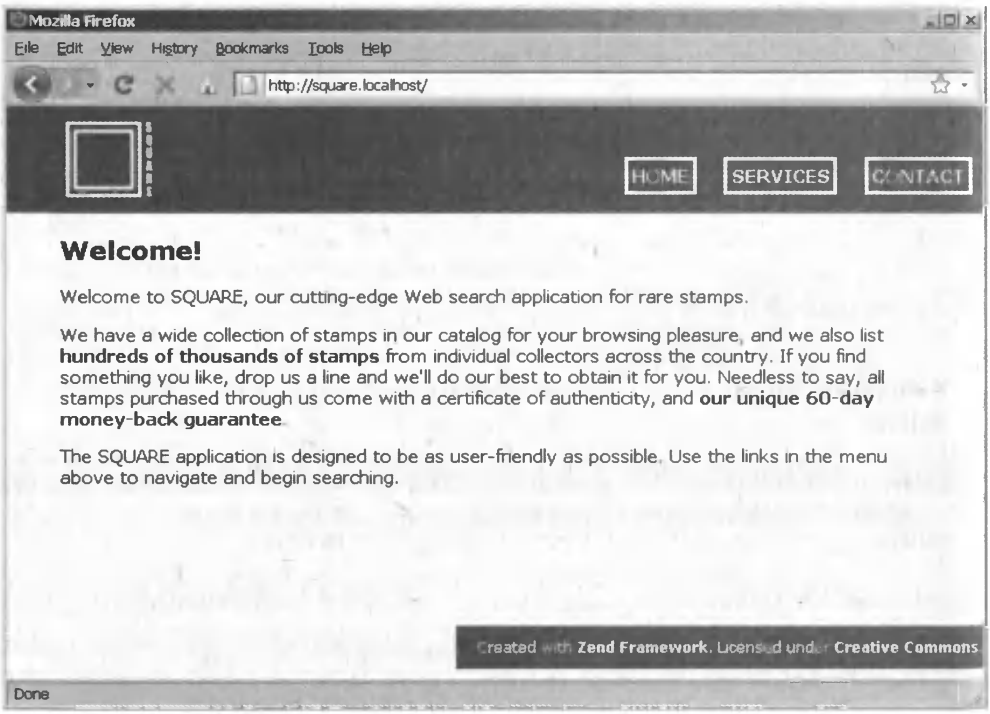


Рис. 2.6. Главная страница приложения, помещенная в пользовательский макет

Использование пользовательского маршрута

Пользовательские маршруты дают возможность устанавливать соответствие между URL приложения и конкретными модулями, контроллерами и действиями и особенно полезны, когда стандартных маршрутов Zend Framework становится недостаточно.

Для примера предположим, что вы хотите, чтобы URL /home перенаправлял пользователей на главную страницу приложения. Для создания пользовательского маршрута, устанавливающего соответствие между этим URL и действием IndexController::indexAction в модуле default, просто добавьте следующую маршрутную запись в секцию [production] конфигурационного файла приложения, $APP_DIR/application/configs/application.ini:

```
resources.router.routes.home.route = /home
resources.router.routes.home.defaults.module = default
resources.router.routes.home.defaults.controller = index
resources.router.routes.home.defaults.action = index
```

Маршрутная запись обычно содержит атрибуты route, module, controller и action. Атрибут route определяет шаблон URL, которому должен соответствовать маршрут, а атрибуты module, controller и action указывают, какие модуль, контроллер и действие должны использоваться для выполнения запроса. Кроме этого, каждый маршрут имеет уникальное имя (в данном случае home), использующееся в качестве сокращения при автоматической генерации URL внутри представлений.

После сохранения изменений попробуйте перейти на страницу http://square.localhost/home. Если все работает правильно, вы должны увидеть главную страницу приложения, показанную на рис. 2.6.

СОВЕТ

Имена маршрутов можно использовать во вспомогательном методе представления url(), чтобы автоматически генерировать URL внутри представлений. Например, вызов $view->url(array(), 'home') внутри сценария представления автоматически сгенерирует строку URL /home.

ВОПРОС ЭКСПЕРТУ

В: Почему макеты расположены на глобальном уровне, а не в модулях?

О: В веб-приложениях представления из различных модулей зачастую используют один и тот же общий макет. Поэтому стандартная структура каталогов Zend Framework предполагает размещение макетов на уровне приложения, то есть глобальном, в каталоге $APP_DIR/application/layouts/ вместо $APP_DIR/application/modules/.

Однако если ваше приложение организовано таким образом, что каждый модуль использует свой макет, вы можете переместить файлы макетов на уровень модулей и написать собственное программное дополнение, динамически изменяющее макет в зависимости от модуля, к которому осуществляется доступ. По ссылкам, приведенным в конце этой главы, можно найти более подробную информацию по реализации такой схемы.

Упражнение 2.2. Обработка статического содержимого

В предыдущих разделах ваши действия ограничивались изменением существующих контроллеров и представлений приложения. Однако по мере разработки вы столкнетесь с необходимостью создания новых контроллеров, действий и пред-

ставлений, содержащих дополнительную функциональность. Это не должно быть поводом для беспокойства, так как существует стандартный процесс, которому вы можете следовать при необходимости добавить в приложение Zend Framework новую функциональность.

Инструкции последующих разделов проведут вас через процесс добавления нового контроллера **StaticContentController** и соответствующих ему представлений, которые будут отвечать за обработку страниц со статическим содержимым, таких как **About Us** и **Services**.

Определение пользовательских маршрутов

Первым шагом будет определение базового маршрута для статических страниц. Предположим, что все такие страницы будут иметь URL вида **/content/xx**, где *xx* — переменная, обозначающая название страницы. Чтобы добавить пользовательский маршрут для обработки этих URL, внесите в конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**, следующее определение маршрута:

```
resources.router.routes.static-content.route = /content/:page
resources.router.routes.static-content.defaults.module = default
resources.router.routes.static-content.defaults.controller = static-content
resources.router.routes.static-content.defaults.action = display
```

Этот маршрут несколько сложнее того, что вы видели ранее, поскольку содержит переменную-заполнитель. Zend Framework поддерживает использование в маршрутах переменных-заполнителей, которые обозначаются двоеточием перед именем, и автоматически преобразует указанную часть URL в переменную, которую можно использовать в контроллере. Таким образом, если клиент запросил, например, URL **/content/hello-world**, подсистема маршрутизации автоматически выделит из этого URL часть **hello-world** и сохранит ее в переменной запроса с именем page.

Определение контроллера

Зачем же нужно фиксировать последнюю часть URL и сохранять ее в качестве переменной запроса? Ответ на этот вопрос приведен в следующем параграфе, но пока мы до него не добрались, давайте создадим в модуле **default** контроллер **StaticContentController**, который, согласно общепринятому правилу, необходимо поместить в файл **$APP_DIR/application/modules/default/controllers/StaticContentController.php**. Создайте этот файл и добавьте в него следующий код:

```
<?php
class StaticContentController extends Zend_Controller_Action
{
    public function init()
    {
    }
```

```
    // отображение статических представлений
    public function displayAction()
    {
      $page = $this->getRequest()->getParam('page');
      if (file_exists($this->view->getScriptPath(null) .
      "/" . $this->getRequest()->getControllerName() .
      "/$page." . $this->viewSuffix)) {
        $this->render($page);
      } else {
        throw new Zend_Controller_Action_Exception('Page not found', 404);
    }
  }
}
```

В этом контроллере доступно единственное действие, **displayAction()**, отвечающее за чтение переменной **page**, установленной подсистемой маршрутизации, посредством вызова метода **getParam()** объекта запроса. Затем осуществляется попытка найти сценарий представления, соответствующий значению этой переменной. Если подходящий сценарий найден, он используется для формирования страницы, которая отправляется клиенту. Если, например, клиент запросит URL **/content/hello-world**, вызов **$request->getParam('page')** вернет значение **hello-world** и действие попытается сформировать представление с именем **$APP_DIR/application/modules/default/views/scripts/static-content/hello-world.phtml**.

Если подходящее представление отсутствует, выбрасывается исключение **404**, передающееся стандартному обработчику ошибок, который преобразовывает его в удобочитаемую страницу с ошибкой и отображает эту страницу клиенту.

ВОПРОС ЭКСПЕРТУ

В: Почему ни одно из объявлений класса в этой главе не содержит закрывающий тег PHP?

О: В соответствии с принятыми в Zend Framework стандартами кодирования, файлы, содержащие только код на PHP, не должны включать в себя закрывающий тег ?>, поскольку это может привести к проблемам в HTTP-ответе в случае, если используемый вами текстовый редактор автоматически добавляет в конец файлов символ перевода строки. За подробной информацией по стандартам кодирования Zend Framework обратитесь к ссылкам, приведенным в конце этой главы.

Определение представления

Следующим (и последним) шагом будет создание представлений, соответствующих контроллерам и действиям, созданным на предыдущем шаге. В большинстве случаев имя представления будет основано на именах контроллера и действия. Однако в данном конкретном случае для отображения страницы используется *обобщенное действие*, которое получает имя представления из запроса URL в виде переменной. Поэтому чтобы определить статическую страницу **Services**, доступную по URL **/content/services**, создайте файл **$APP_DIR/application/modules/default/views/**

scripts/static-content/services.phtml и заполните его каким-нибудь содержимым, как показано в следующем примере:

```
<h2>Services</h2>
<p>We provide a number of services, including procurement, valuation
and mail-order sales. Please contact us to find out more.</p>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.</p>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.</p>
```

Аналогично, чтобы определить страницу **About Us**, доступную по URL **/content/about-us**, создайте файл с именем **$APP_DIR/application/modules/default/views/scripts/static-content/about-us.phtml** и заполните его каким-нибудь содержимым, как показано в следующем примере:

```
<h2>About Us</h2>
<p>We have been in the business of stamp procurement and sales since
1927, and are internationally known for our expertise and industry-wide network.
We encourage you to find out more about us using the links
above.</p>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.</p>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.</p>
```

Теперь если вы перейдете по URL **http://square.localhost/content/about-us** или **http://square.localhost/content/services**, вы увидите приведенные выше статические страницы (рис. 2.7).

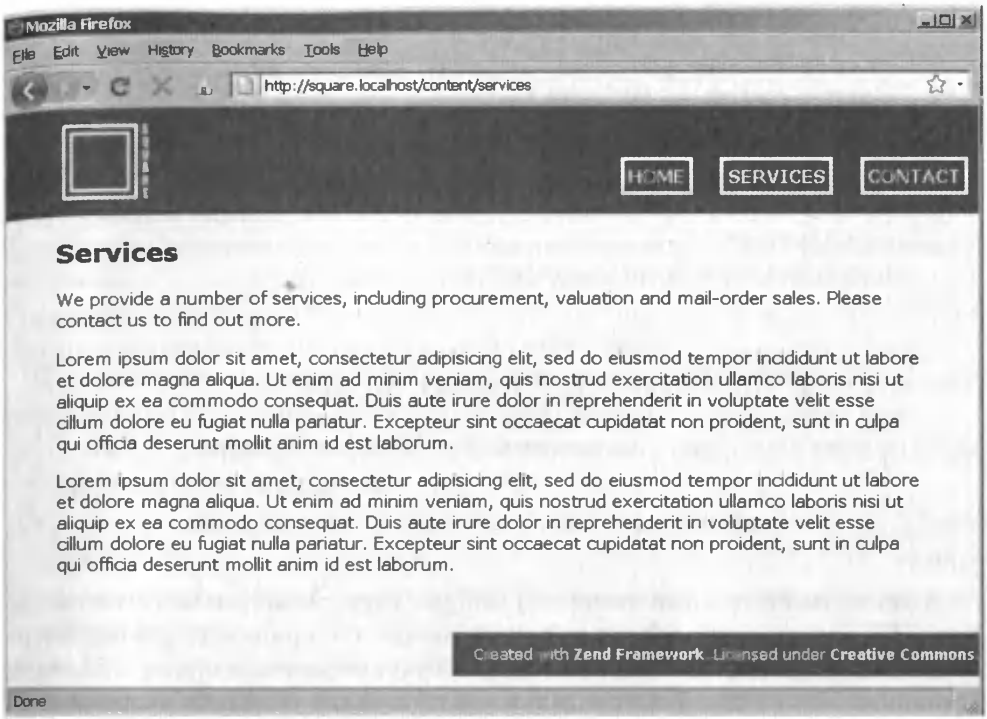


Рис. 2.7. Страница со статическим содержимым

Обновление основного макета

В качестве последнего шага воспользуйтесь вспомогательным методом url() и обновите навигационные ссылки в главном меню приложения, чтобы они указывали на новые статические страницы. Чтобы сделать это, внесите в основной макет $APP_DIR/application/layouts/master.phtml изменения, выделенные жирным:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8"/>
    <base href="/" />
    <link rel="stylesheet" type="text/css" href="/css/master.css" />
  </head>
  <body>
    <div id="header">
      <div id="logo">
        <img src="/images/logo.gif" />
      </div>
```

```
        <div id="menu">
        <a href="<?php echo $this->url(array(), 'home'); ?>">HOME</a>
          <a href="<?php echo $this->url(array('page' => 'services'),
        'static-content'); ?>">SERVICES</a>
      <a href="#">CONTACT</a>
    </div>
  </div>

    <div id="content">
      <?php echo $this->layout()->content ?>
    </div>

    <div id="footer">
      <p>Created with <a href="http://framework.zend.com/">
      Zend Framework</a>. Licensed under
      <a href="http://www.creativecommons.org/">Creative Commons</a>.
      </p>
    </div>
  </body>
</html>
```

Здесь вспомогательный метод **url()** используется для автоматической генерации URL для маршрутов **/home** и **/content/services**. Он принимает два параметра: массив пар «переменная-значение», которые будут вставлены в строку URL, и имя маршрута, а затем при формировании представления генерирует строки URL, соответствующие этим параметрам. Если вы снова зайдете на главную страницу приложения, то увидите, что ссылки **Home** и **Services** в главном меню теперь активны и при переходе по ним отображается правильное содержимое.

ПРИМЕЧАНИЕ

Стоит отметить, что на самом деле существует более простой способ обрабатывать статические страницы в приложении Zend Framework: вы можете просто поместить их в каталог $APP_DIR/public/ как обычные HTML-файлы и вручную проставить ссылки на них. Однако эти страницы не смогут использовать какие бы то ни было встроенные возможности Zend Framework, например глобальные макеты, маршруты, кэширование или средства безопасности, и по этой причине мы не рекомендуем использовать данный подход.

Выводы

Теперь, когда вы дошли до конца этой главы, вы гораздо лучше разбираетесь в том, что необходимо для приведения приложения Zend Framework в работоспособное состояние. Эта глава началась с введения в паттерн проектирования Модель–Представление–Контроллер, объясняющего, что представляют собой модели, представления и контроллеры и как они взаимодействуют друг с другом для создания приложения, разделенного на логические слои. В ней вы также познакомились с некоторыми важными дополнительными понятиями, такими как модули, маршруты, макеты и front-контроллер, и с основами маршрутизации и об-

работки пользовательских запросов в контексте приложения. Наконец, в этой главе все накопленные теоретические сведения были применены для решения реальной задачи и начался процесс настройки и улучшения простого тестового приложения, созданного в конце предыдущей главы.

Демонстрационное приложение SQUARE все еще находится в зачаточной стадии: у него есть настроенный макет, оно умеет работать с модулями и может обрабатывать статическое содержимое. Это может показаться не таким уж и существенным достижением, но реализация даже столь простой функциональности поможет вам понять базовые принципы разработки с использованием Zend Framework и паттерна MVC и создаст надежный фундамент для более легкого освоения материала, изложенного в следующих главах.

Если вы хотите получить больше информации по рассмотренным в этой главе темам, вам будут полезны следующие ссылки:

- Архитектура Модель–Представление–Контроллер в Википедии: http://en.wikipedia.org/wiki/Model-view-controller
- Основы реализации MVC в Zend Framework: http://framework.zend.com/docs/quickstart.
- Маршрутизатор Zend Framework: http://framework.zend.com/manual/en/zend.controller.router.html.
- Front-контроллер Zend Framework: http://framework.zend.com/manual/en/zend.controller.front.html.
- Механизм макетов Zend Framework: http://framework.zend.com/manual/en/zend.layout.html.
- Создание модульного приложения с использованием Zend Framework (Йерун Кеппенс (Jeroen Keppens)): http://www.amazium.com/blog/create-modular-application-with-zend
- Использование макетов на уровне модулей на вики-сайтах и форумах Zend Framework: http://framework.zend.com/wiki/display/ZFPROP/Zend_Layout и http://www.zfforums.com/zend-framework-components-13/model-view-controller-mvc-21/modules-layouts-2645.html
- Структура каталогов Zend Framework: http://framework.zend.com/wiki/display/ZFPROP/Zend+Framework+Default+Project+Structure+-+Wil+Sinclair
- Стандарты кодирования Zend Framework: http://framework.zend.com/manual/en/coding-standard.html

3

Работа с формами

Прочитав эту главу, вы:

- научитесь программно создавать формы и их элементы;
- научитесь производить фильтрацию и валидацию пользовательского ввода;
- защитите ваши формы от атак, основанных на подделке межсайтовых запросов (Cross-Site Request Forgery, CSRF) и спам-ботов;
- научитесь управлять внешним видом элементов формы и сообщений об ошибках;
- создадите работоспособную форму обратной связи.

В предыдущей главе вы узнали, каким образом Zend Framework реализует паттерн Модель–Представление–Контроллер, и заглянули «под капот» демонстрационного приложения, чтобы посмотреть, как оно работает. Затем вы начали расширять это приложение, перейдя на модульную структуру каталогов, добавив основной макет и создав пользовательские контроллеры, представления и маршруты для статического содержимого.

Итак, хотя вы, конечно, можете использовать Zend Framework для предоставления статического содержимого, это равносильно использованию бульдозера для сноса башни из пластмассовых блоков. Это, конечно, не запрещено, но на самом деле бульдозер предназначен для других целей, и для начала вас могут спросить, что он делает в вашей гостиной! То же самое и с Zend Framework — он предназначен для предоставления надежных, элегантных и расширяемых решений сложных задач, связанных с разработкой веб-приложений. Чем сложнее задача, тем больше она соответствует мощи и гибкости фреймворка... и тем интереснее будет с ней справляться!

Прочитав эту главу, вы узнаете, как Zend Framework может упростить решение одной из самых распространенных задач в разработке приложений — создания веб-форм и обработки пользовательского ввода. Используя эти знания, вы добавите в наше демонстрационное приложение SQUARE некоторую интерактивность, создав форму обратной связи. Поэтому давайте без промедления приступим к делу!

Основы работы с формами

Чтобы показать, как Zend Framework может помочь вам с созданием форм, будет достаточно небольшого, но наглядного примера. Если вы относитесь к большинству PHP-разработчиков, велики шансы, что на каком-то этапе своей карьеры вы писали сценарий для обработки форм наподобие следующего:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
  </head>

  <body>
  <h2>Create Item</h2>
  <?php
  if (!isset($_POST['submit'])) {
  // запрос POST не отправлялся, отображаем форму
  ?>
    <form method="post" action="/item/create">
      <table>
        <tr>
          <td>Item name:</td>
          <td><input type="text" name="name" size="30" /></td>
        </tr>

        <tr>
          <td>Item quantity:</td>
          <td><input type="text" name="qty" size="3" /></td>
        </tr>

        <tr>
          <td colspan="2">
            <input type="submit" name="submit" value="Submit" />
        </td>
      </tr>
    </table>
  </form>
<?php
} else {
  // отправлен запрос POST, проверяем входные данные
  if (trim($_POST['name']) == '') {
    die(' ERROR: Missing value - Item name ');
  }
  if (trim($_POST['qty']) == '') {
    die(' ERROR: Missing value - Item quantity ');
  }
  if ($_POST['qty'] <= 0) {
    die(' ERROR: Invalid value - Item quantity ');
  }
```

```
  // обрабатываем входные данные
    // например сохраняем их в базу данных
  // попытка подключения
  try {
    $pdo = new PDO('mysql:dbname=test;host=localhost', 'user', 'pass');

    // создаем и выполняем запрос INSERT
    $name = $pdo->quote($_POST['name']);
    $qty = $pdo->quote($_POST['qty']);
      $sql = "INSERT INTO shoppinglist (name, qty) VALUES ($name, $qty)";
      $pdo->exec($sql) or die("ERROR: " . implode(":", $pdo->errorInfo()));

      // закрываем подключение
      unset($pdo);

      // отображаем сообщение об успешной операции
      echo 'Thank you for your submission';
    } catch (Exception $e) {
      die("ERROR: " . $e->getMessage());
    }
    }
  ?>
  </body>
</html>
```

Здесь нет ничего хитроумного или сложного. Этот сценарий разбит на две части условным оператором, проверяющим значение переменной **$_POST**, чтобы определить, была ли форма отправлена. Первая его половина отображает форму ввода с двумя полями и кнопкой отправки, а вторая проверяет правильность формата введенных данных, затем экранирует их и добавляет в базу данных.

Внешний вид формы показан на рис. 3.1.

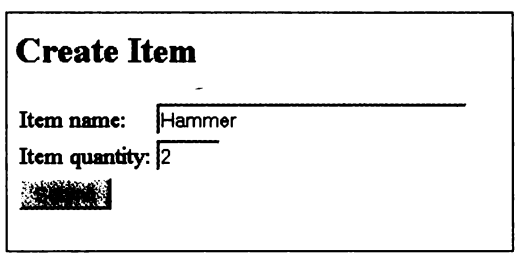


Рис. 3.1. Форма, созданная с помощью стандартной HTML-разметки

Хотя приведенный сценарий и весь подход в целом работают на практике, невозможно отрицать тот факт, что у них существует ряд проблем:

- Один и тот же файл сценария содержит как элементы интерфейса, описанные на HTML, так и бизнес-логику на PHP. Как говорилось в предыдущей главе, такое соседство не только неряшливо выглядит, но и осложняет сопровождение. Помимо этого, становится сложнее поддерживать согласованность форм,

так как код, необходимый для создания каждой из них, может подвергаться существенной модификации.

- Каждый раз, когда в первой половине сценария вы добавляете на форму новое поле, вам также необходимо добавить соответствующий набор проверок и сообщений об ошибках в его вторую половину. Это надоедающее и часто повторяющееся действие; взять хотя бы тот факт, что две первые проверки из предыдущего примера делают, в сущности, одно и то же.
- Не существует способа повторно использовать проверочные тесты из одной формы в других формах (если только вы с самого начала предусмотрительно не упаковали их в классы или функции). В результате вам часто приходится писать один и тот же код снова и снова, особенно при работе с формами, осуществляющими взаимосвязанные или схожие действия.

В состав Zend Framework входит набор компонентов с общим названием **Zend_Form**, которые решают перечисленные проблемы. В целях демонстрации рассмотрим следующий пример, где **Zend_Form** используется, чтобы создать сценарий, эквивалентный предыдущему:

```
<?php
class Form_Item_Create extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/item/create')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Item name:')
        ->setOptions(array('size' => '35'))
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addValidator('Alpha', true)
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем текстовое поле для ввода количества
    $qty = new Zend_Form_Element_Text('qty');
    $qty->setLabel('Item quantity:');
    $qty->setOptions(array('size' => '4'))
          ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addValidator('Int', true)
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем кнопку для отправки
    $submit = new Zend_Form_Element_Submit('submit');
```

```
    $submit->setLabel('Submit')
          ->setOptions(array('class' => 'submit'));

    // добавляем к форме элементы
    $this->addElement($name)
        ->addElement($qty)
        ->addElement($submit);
    }
  }

  class ExampleController extends Zend_Controller_Action
  {
  public function formAction()
  {
    $form = new Form_Item_Create;
    $this->view->form = $form;
    if ($this->getRequest()->isPost()) {
      if ($form->isValid($this->getRequest()->getPost())) {
        $values = $form->getValues();
        $pdo = new PDO('mysql:dbname=test;host=localhost', 'user', 'pass');
        $sql = sprintf("INSERT INTO shoppinglist (name, qty)
              VALUES ('%s', '%d')", $values['name'], $values['qty']);
        $pdo->exec($sql);
        $this->_helper->getHelper('FlashMessenger')
            ->addMessage('Спасибо');
        $this->_redirect('/index/success');
      }
    }
  }
}
```

На рис. 3.2 показано, как выглядит форма.

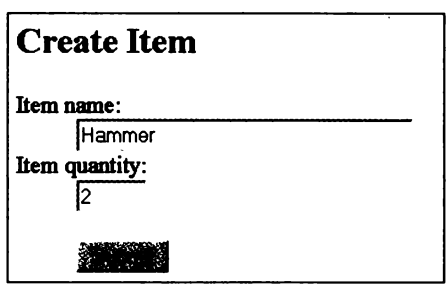


Рис. 3.2. Форма, созданная с использованием компонента Zend_Form

В коде, создающем приведенную на рис. 3.2 форму, вы наверняка заметили три особенности:

- В сценарии нет ни одной строчки HTML-кода. Форма и ее элементы представлены в виде объектов PHP и настраиваются с помощью методов этих объектов.

Такой подход обеспечивает согласованность и создает веб-форму, соответствующую стандартам.

- Для распространенных задач валидации и очистки ввода доступны встроенные валидаторы и фильтры. Они уменьшают количество выполняемой работы, помогают создавать более простой в сопровождении код и избегать повторений. Кроме того, для соответствия нестандартным требованиям валидаторы можно комбинировать и расширять.
- Валидаторы указываются одновременно с полями формы. Это позволяет форме «знать», какие данные поддерживаются каждым полем, и легко определять источник ошибок ввода. Для более строгой валидации одному полю можно назначить несколько валидаторов.

Отсюда вытекает, что Zend_Form предоставляет удобное, простое в сопровождении и расширяемое решение для создания форм ввода и валидации данных. В оставшейся части главы Zend_Form будет рассмотрен более подробно на примере практической задачи.

Создание форм и элементов формы

В предыдущем разделе вы узнали, что Zend_Form предоставляет объектно-ориентированный API для генерации форм и валидации пользовательского ввода. В соответствии с используемым в Zend_Form подходом *формы* представлены как экземпляры базового класса Zend_Form или объекты, наследующиеся от него. Этот базовый класс предоставляет несколько методов для управления функционированием формы, включая метод setAction(), устанавливающий URL для атрибута action (действие) формы, и метод setMethod(), устанавливающий атрибут method (метод отправки формы). Существует также универсальный метод setAttribs(), позволяющий задавать другие атрибуты формы. Ниже приведен пример использования этих методов:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/my/action')
        ->setAttribs(array(
            'class' => 'form',
            'id' => 'example'
          ))
        ->setMethod('post');
  }
}
```

Элементы формы добавляются путем создания экземпляров соответствующего класса Zend_Form_Element_*, установки свойств элементов с помощью методов класса и последующего прикрепления их к форме посредством метода addElement(). Ниже приведен пример добавления на форму поля для ввода текста и кнопки отправки:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/my/action')
        ->setAttribs(array(
          'class' => 'form',
          'id' => 'example'
         ))
        ->setMethod('post');

    // создаем текстовое поле для ввода названия
    $title = new Zend_Form_Element_Text('title');
    $title->setLabel('Title:')
        ->setOptions(array(
            'size' => '35'
         ));

    // создаем кнопку для отправки
    $submit = new Zend_Form_Element_Submit('submit', array(
      'label' => 'Submit',
      'class' => 'submit'
    ));

    // добавляем элементы к форме
    $this->addElement($title)
        ->addElement($submit);
    }
}
```

Объекты элементов можно настраивать, передавая значения в их конструкторы или используя именованные методы объектов. В предыдущем примере имя поля для ввода текста было передано в конструктор объекта, а для установки метки элемента и параметров отображения использовались методы setLabel() и setOptions() соответственно. Кнопка отправки была настроена непосредственно через конструктор объекта, которому в качестве второго аргумента передали массив параметров.

СОВЕТ

Вы можете добавлять описания к полям формы с помощью метода setDescription().

При желании вы можете создавать элементы формы с помощью метода createElement(), передавая ему в качестве первого аргумента тип элемента. Ниже приведен пример, эквивалентный предыдущему:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/my/action')
        ->setAttribs(array(
          'class' => 'form',
          'id' => 'example'
         ))
        ->setMethod('post');

    // создаем текстовое поле для названия
    $title = $this->createElement('text', 'title', array(
        'label' => 'Title:',
        'size' => 35,
    ));

    // создаем кнопку для отправки
    $submit = $this->createElement('submit', 'submit', array(
      'label' => 'Submit',
      'class' => 'submit'
  ));

    // добавляем элементы к форме
    $this->addElement($title)
        ->addElement($submit);
  }
}
```

СОВЕТ

Во многих фрагментах кода в этой главе вы встретите примеры *объединения методов в цепочку* (*method chaining*), которые похожи на вызов одного метода другим. Это пример «текучего интерфейса» (fluent interface) Zend Framework, предоставляющего удобный способ настройки объектов формы с минимумом дополнительного кода. Кроме этого, конечный результат значительно легче читать. Более подробно о текучих интерфейсах вы можете узнать по ссылкам в конце этой главы.

Работа с элементами форм

По умолчанию Zend Framework содержит определения 16 элементов форм, начиная от простых элементов для ввода текста и заканчивая более сложными списками с множественным выбором, поэтому полезно будет узнать о них больше. В табл. 3.1 приведен список этих элементов и соответствующих имен классов.

В следующих разделах все элементы рассматриваются более подробно.

Таблица 3.1. Классы элементов форм, входящие в состав Zend Framework

Класс элемента	Описание
Zend_Form_Element_Text	Поле ввода текста
Zend_Form_Element_Hidden	Скрытое поле
Zend_Form_Element_Password	Поле ввода пароля
Zend_Form_Element_Radio	Радиокнопка
Zend_Form_Element_Checkbox	Переключатель
Zend_Form_Element_MultiCheckbox	Группа взаимосвязанных переключателей
Zend_Form_Element_Select	Список для выбора (одиночного)
Zend_Form_Element_MultiSelect	Список для выбора (множественного)
Zend_Form_Element_Textarea	Поле ввода текста
Zend_Form_Element_File	Поле загрузки файла
Zend_Form_Element_Image	Изображение
Zend_Form_Element_Button	Кнопка
Zend_Form_Element_Hash	Уникальная строка (для идентификации сессии)
Zend_Form_Element_Captcha	CAPTCHA (для фильтрации спама)
Zend_Form_Element_Reset	Кнопка сброса
Zend_Form_Element_Submit	Кнопка отправки

Текстовые и скрытые поля

Поля для ввода текста и паролей, а также области для ввода больших объемов текста представлены классами **Zend_Form_Element_Text**, **Zend_Form_Element_Password** и **Zend_Form_Element_Textarea** соответственно, а скрытые поля формы представлены классом **Zend_Form_Element_Hidden**. Следующий пример демонстрирует использование этих элементов:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('First name:')
```

```
        ->setOptions(array('id' => 'fname'));

    // создаем поле для ввода пароля
    $pass = new Zend_Form_Element_Password('pass');
    $pass->setLabel('Password:')
         ->setOptions(array('id' => 'upass'));

    // создаем скрытое поле
    $uid = new Zend_Form_Element_Hidden('uid');
    $uid->setValue('49');

    // создаем текстовую область для комментария
    $comment = new Zend_Form_Element_Textarea('comment');
    $comment->setLabel('Comment:')
        ->setOptions(array(
          'id' => 'comment',
          'rows' => '10',
          'cols' => '30',
        ));

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($pass)
        ->addElement($uid)
        ->addElement($comment);
    }
}
```

Результат показан на рис. 3.3.

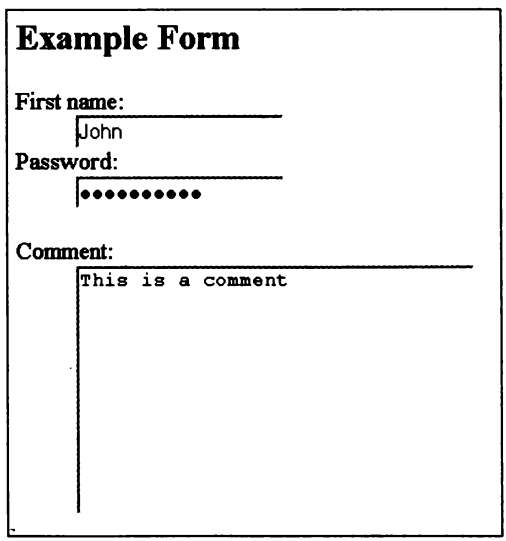


Рис. 3.3. Форма с текстовыми полями и скрытым полем

Радиокнопки и переключатели

Радиокнопки представлены классом **Zend_Form_Element_Radio**, а переключатели — классом **Zend_Form_Element_Checkbox**. Ниже приведен пример использования этих классов:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Name:')
        ->setOptions(array('id' => 'fname'));

    // создаем радиокнопки для выбора типа
    $type = new Zend_Form_Element_Radio('type');
    $type->setLabel('Membership type:')
        ->setMultiOptions(array(
            'silver' => 'Silver',
            'gold' => 'Gold',
            'platinum' => 'Platinum'
          ))
        ->setOptions(array('id' => 'mtype'));

    // создаем переключатель для подписки на новостную рассылку
    $subscribe = new Zend_Form_Element_Checkbox('subscribe');
    $subscribe->setLabel('Subscribe to newsletter')
        ->setCheckedValue('yes')
        ->setUncheckedValue('no');

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($type)
        ->addElement($subscribe);
    }
}
```

Метод **setMultiOptions()** объекта **Zend_Form_Element_Radio** принимает массив и использует его для формирования списка пунктов, доступных для выбора. Ключами массива являются значения формы, которые будут отправляться, а соответствующими значениями — «человекочитаемые» метки для каждого пункта. Аналогично, методы **setCheckedValue()** и **setUncheckedValue()** объекта **Zend_Form_Element_Checkbox** позволяют вам установить значения для выбранных и невыбранных элементов. По умолчанию этими значениями являются 1 и 0 соответственно.

Результат показан на рис. 3.4.

Рис. 3.4. Форма с радиокнопками и переключателем

Если вы хотите предоставить пользователю выбор из нескольких вариантов, класс Zend_Form_Element_MultiCheckbox подходит для этого лучше, чем класс Zend_Form_Element_Checkbox, поскольку предоставляет метод setMultiOptions(), позволяющий выбирать несколько элементов. После этого результирующий набор выбранных элементов форматируется и отправляется в виде массива. Ниже приведен пример использования перечисленных классов:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
         ->setMethod('post');

    // создаем текстовое поле для ввода названия
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Name:')
        ->setOptions(array('id' => 'fname'));

    // создаем радиокнопки для выбора типа
    $type = new Zend_Form_Element_Radio('type');
    $type->setLabel('Pizza crust:')
        ->setMultiOptions(array(
            'thin' => 'Thin',
            'thick' => 'Thick'
          ))
        ->setOptions(array('id' => 'type'));

    // создаем переключатель для выбора начинок
    $toppings = new Zend_Form_Element_MultiCheckbox('toppings');
    $toppings->setLabel('Pizza toppings:')
```

```
        ->setMultiOptions(array(
            'bacon' => 'Bacon',
            'olives' => 'Olives',
            'tomatoes' => 'Tomatoes',
            'pepperoni' => 'Pepperoni',
            'ham' => 'Ham',
            'peppers' => 'Red peppers',
            'xcheese' => 'Extra cheese',
          ));

    // добавляем к форме элементы
    $this->addElement($name)
        ->addElement($type)
        ->addElement($toppings);
    }
}
```

Результат показан на рис. 3.5.

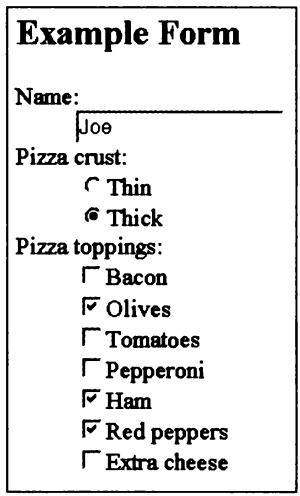


Рис. 3.5. Форма с радиокнопками и несколькими переключателями

Списки выбора

Списки с единичным и множественным выбором поддерживаются с помощью классов Zend_Form_Element_Select и Zend_Form_Element_MultiSelect. Как и класс Zend_Form_Element_MultiCheckbox, они предоставляют метод setMultiOptions(), который можно использовать для формирования списка доступных вариантов. Следующий пример демонстрирует использование обоих элементов:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Name:')
        ->setOptions(array('id' => 'fname'));

    // создаем новый список с единичным выбором страны отправления
    $from = new Zend_Form_Element_Select('from');
    $from->setLabel('Travelling from:')
        ->setMultiOptions(array(
            'IN' => 'India',
            'US' => 'United States',
            'DE' => 'Germany',
            'FR' => 'France',
            'UK' => 'United Kingdom'
          ));

    // создаем новый список с множественным выбором стран назначения
    $to = new Zend_Form_Element_MultiSelect('to');
    $to->setLabel('Travelling to:')
        ->setMultiOptions(array(
            'IT' => 'Italy',
            'SG' => 'Singapore',
            'TR' => 'Turkey',
            'DK' => 'Denmark',
            'ES' => 'Spain',
            'PT' => 'Portugal',
            'RU' => 'Russia',
            'PL' => 'Poland'
          ));

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($from)
        ->addElement($to);
    }
}
```

Результат показан на рис. 3.6.

Рис. 3.6. Форма со списком доступных вариантов

Поля для загрузки файла на сервер

Если вы хотите отправить через форму один или несколько файлов, вам потребуется класс Zend_Form_Element_File, который предоставляет поле для выбора файла. Ниже приведен пример его использования:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setEnctype('multipart/form-data')
        ->setMethod('post');

    // создаем поле загрузки файла с фотографией
    $photo = new Zend_Form_Element_File('photo');
    $photo->setLabel('Photo:')
        ->setDestination('/tmp/upload');

    // добавляем элементы к форме
    $this->addElement($photo);
  }
}
```

ВНИМАНИЕ

Помните, что для корректной обработки данных, загруженных через форму, вы должны установить для нее способ кодирования 'multipart/form-data'. Это можно сделать, вызвав метод setEnctype() объекта формы.

На рис. 3.7 показано поле загрузки файла.

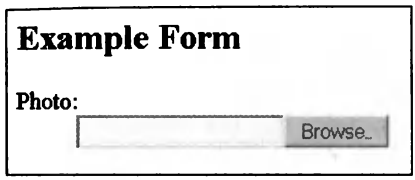


Рис. 3.7. Форма с полем загрузки файла

СОВЕТ

Если вы хотите загрузить несколько взаимосвязанных файлов, существует удобный метод setMultiFile(), который генерирует набор полей загрузки файла и избавляет вас от необходимости создания нескольких объектов Zend_Form_Element_File. Пример использования этого метода вы увидите в следующей главе.

Кнопки

Для каждой формы требуется кнопка отправки, а для некоторых — еще и кнопка сброса. Эти важные элементы формы представлены соответственно классами **Zend_Form_Element_Submit** и **Zend_Form_Element_Reset**, а в следующем листинге приведен пример их использования:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
         ->setMethod('post');

    // создаем текстовое поле для ввода названия
    $title = new Zend_Form_Element_Text('title');
    $title->setLabel('Title:')
         ->setOptions(array('size' => '35'));

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Submit');

    // создаем кнопку сброса
    $reset = new Zend_Form_Element_Reset('reset');
    $reset->setLabel('Cancel');

    // добавляем элементы к форме
    $this->addElement($title)
         ->addElement($submit)
         ->addElement($reset);
    }
}
```

Результат показан на рис. 3.8.

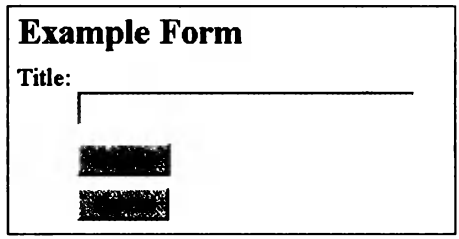


Рис. 3.8. Форма с кнопками отправки и сброса

Если вы ищете более общий вариант кнопки, вам подойдет класс **Zend_Form_Element_Button**, предоставляющий простую кнопку формы, которая может быть полезна для различных целей. С помощью класса **Zend_Form_Element_Image** можно создавать кнопки с изображениями; используйте метод **setImage()**, чтобы указывать для них исходное изображение. Ниже приведен пример такой кнопки:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода названия
    $title = new Zend_Form_Element_Text('title');
    $title->setLabel('Title:')
        ->setOptions(array('size' => '35'));

    // создаем кнопку отправки с изображением
    $submit = new Zend_Form_Element_Image('submit');
    $submit->setImage('/images/submit.jpg');

    // добавляем к форме элементы
    $this->addElement($title)
        ->addElement($submit);
    }
}
```

Результат показан на рис. 3.9.

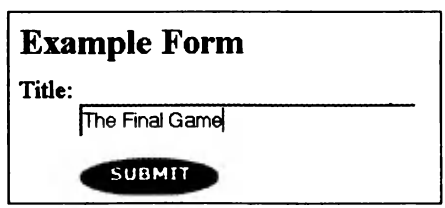


Рис. 3.9. Форма, содержащая кнопку с изображением

Поля Hash и CAPTCHA

Для обеспечения безопасности ввода Zend Framework содержит два «специальных» элемента формы: *Hash* и *CAPTCHA*. Они представлены классами Zend_Form_Element_Hash и Zend_Form_Element_Captcha соответственно.

Элемент Hash использует значение salt для генерации уникального ключа формы и записи его в хранилище сессий. При отправке формы переданное с ней значение хэша автоматически сравнивается со значением, находящимся в хранилище сессий. Если они совпадают, отправленная форма считается подлинной. Если обнаружено несоответствие, разумно предположить, что форма была перехвачена и используется для атаки, основанной на подделке межсайтовых запросов (CSRF).

Ниже приведен пример использования данного элемента:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода номера
    $cc = new Zend_Form_Element_Text('ccnum');
    $cc->setLabel('Credit card number:')
        ->setOptions(array('size' => '16'));

    // создаем текстовое поле для ввода количества
    $amount = new Zend_Form_Element_Text('amount');
    $amount->setLabel('Payment amount:')
        ->setOptions(array('size' => '4'));

    // создаем хэш
    $hash = new Zend_Form_Element_Hash('hash');
    $hash->setSalt('hf823hflw03j');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Submit');

    // добавляем элементы к форме
    $this->addElement($cc)
        ->addElement($amount)
        ->addElement($hash)
        ->addElement($submit);
    }
}
```

Элемент CAPTCHA автоматически генерирует поле ввода CAPTCHA, которое полезно для отсеивания автоматически отправленных форм. Множество сайтов использует CAPTCHA, чтобы уменьшить количество ложных регистраций и/или

спам-сообщений, поступающих через онлайн-формы. Несмотря на то что ручная генерация и проверка CAPTCHA является утомительным процессом, класс **Zend_Form_Element_Captcha** сводит его к добавлению нескольких строк кода к вашей форме. Пример:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени пользователя
    $name = new Zend_Form_Element_Text('username');
    $name->setLabel('Username:')
        ->setOptions(array('size' => '16'));

    // создаем поле ввода пароля
    $pass = new Zend_Form_Element_Password('password');
    $pass->setLabel('Password:')
        ->setOptions(array('size' => '16'));

    // создаем поле CAPTCHA
    $captcha = new Zend_Form_Element_Captcha('captcha', array(
            'captcha' => array(
            'captcha' => 'Figlet',
            'wordLen' => 5,
            'timeout' => 300,
          )
        ));
    $captcha->setLabel('Verification:');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Sign Up');

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($pass)
        ->addElement($captcha)
        ->addElement($submit);
    }
}
```

На рис. 3.10 показан результат.

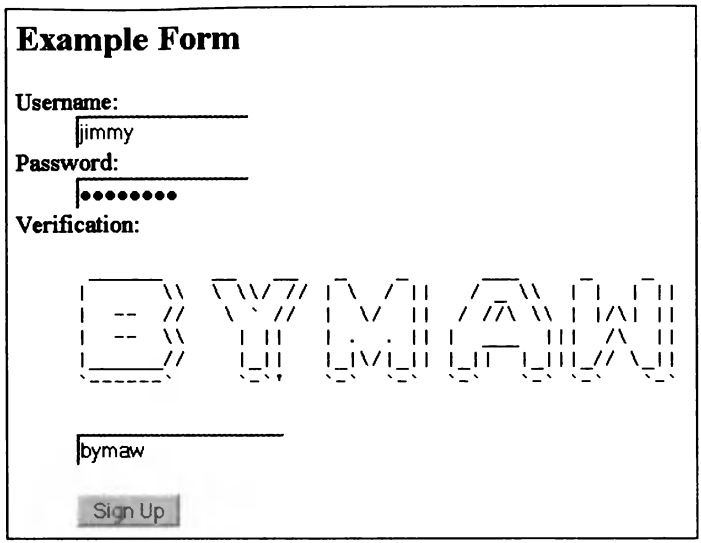


Рис. 3.10. Форма, содержащая CAPTCHA

ВОПРОС ЭКСПЕРТУ

В: Что такое CSRF-атака и как мне от нее защититься?

О: Как правило, когда пользователь посещает защищенный веб-сайт и подтверждает свои параметры доступа, создается пользовательская сессия, и при каждом запросе параметры доступа перепроверяются на основании данных из хранилища сессий. CSRF-атака включает в себя похищение подтвержденной пользовательской сессии и использование полного взаимного доверия, установившегося между пользователем и приложением, с целью незаметной передачи неавторизованных запросов через источники ввода данных, такие как веб-формы. Генерируя уникальное хэш-значение для каждой веб-формы и проверяя это значение при ее отправке, разработчик может усложнить проведение атак данного типа. Использование хэш-значения также дает (ограниченную) защиту от программ для автоматической рассылки спама («спам-ботов») и более удобно для пользователя, нежели CAPTCHA.

ПРИМЕЧАНИЕ

В состав Zend Framework входят несколько встроенных адаптеров для CAPTCHA, включая адаптеры для простых операций по изменению порядка символов в строках (Dumb) и для визуальных CAPTCHA (Image и Figlet). Далее в этой главе вы увидите другой пример CAPTCHA, использующего изображение.

Установка обязательных значений и значений по умолчанию

Вы можете сделать определенные элементы ввода обязательными, вызвав их метод `setRequired()` с аргументом `true`. Пример:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // инициализируем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Name:')
        ->setOptions(array('size' => '35'))
        ->setRequired(true);

    // создаем текстовое поле для ввода адреса электронной почты
    $email = new Zend_Form_Element_Text('email');
    $email->setLabel('Email address:');
    $email->setOptions(array('size' => '50'))
        ->setRequired(true);

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit',
        array('class' => 'submit')
    );
    $submit->setLabel('Sign Up');

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($email)
        ->addElement($submit);
    }
}
```

Когда вы используете метод **setRequired()** для поля ввода, **Zend_Form** автоматически добавляет к этому полю валидатор **NotEmpty**. Если при отправке формы поле оставлено пустым, возникает сообщение об ошибке. Результат показан на рис. 3.11.

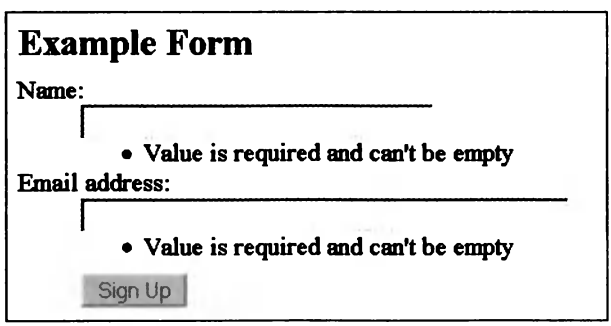


Рис. 3.11. Результат отправки формы с отсутствующими обязательными значениями

СОВЕТ

Вы можете заставить Zend_Form не добавлять валидатор NotEmpty к обязательным элементам, явно вызвав метод setAutoInsertNotEmptyValidator() с аргументом false. Более подробно валидаторы обсуждаются в следующем разделе.

ВОПРОС ЭКСПЕРТУ

В: Что такое CAPTCHA?

О: CAPTCHA, или *Completely Automated Public Turing test to tell Computers and Humans Apart* (полностью автоматизированный публичный тест Тьюринга для различения компьютеров и людей), — это распространенный тест вида вызов–ответ, использующийся, чтобы определять, находится на другом конце соединения человек или компьютер. В Интернете CAPTCHA обычно представлен искаженной последовательностью случайных алфавитно-цифровых символов и основан на том, что компьютер не сможет распознать искаженное изображение, в отличие от человека, обладающего более развитым восприятием. Такие тесты обычно добавляются к формам ввода (например, формам регистрации пользователя) и должны быть пройдены, чтобы введенные данные были обработаны приложением. CAPTCHA не обязательно должны быть визуальными; возможно также использование звуковых CAPTCHA, которые лучше всего подходят людям с нарушениями зрения.

Вы можете установить для элементов ввода значения по умолчанию, вызвав метод **setValue()** объекта элемента и передав ему нужное значение либо вызвав метод **setDefaults()** объекта формы и передав ему массив значений по умолчанию. Для полей ввода текста таким значением может быть любая строка; для радиокнопок и списков выбора им должен являться индекс выбранного элемента. Ниже приведен пример, демонстрирующий использование обоих методов:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Name:')
        ->setOptions(array('size' => '35'))
       ->setRequired(true)
        ->setValue('Enter your name');

    // создаем текстовое поле для ввода адреса электронной почты
    $email = new Zend_Form_Element_Text('email');
    $email->setLabel('Email address:');
    $email->setOptions(array('size' => '50'))
        ->setRequired(true)
        ->setValue('Enter your email address');
```

```
        // создаем радиокнопки для выбора типа
        $type = new Zend_Form_Element_Radio('type');
        $type->setLabel('Membership type:')
            ->setMultiOptions(array(
                'silver' => 'Silver',
                'gold' => 'Gold',
                'platinum' => 'Platinum'
            ));

        // создаем переключатель для подписки на новостную рассылку
        $subscribe = new Zend_Form_Element_Checkbox('subscribe');
        $subscribe->setLabel('Subscribe to newsletter')
            ->setCheckedValue('yes')
            ->setUncheckedValue('no');

        // создаем список выбора для страны отправления
        $from = new Zend_Form_Element_Select('from');
        $from->setLabel('Country:')
            ->setMultiOptions(array(
                'IN' => 'India',
                'US' => 'United States',
                'DE' => 'Germany',
                'FR' => 'France',
                'UK' => 'United Kingdom'
            ));

        // создаем кнопку отправки
        $submit = new Zend_Form_Element_Submit(
          'submit', array('class' => 'submit'));
        $submit->setLabel('Sign Up');

        // добавляем элементы к форме
        $this->addElement($name)
            ->addElement($email)
            ->addElement($type)
            ->addElement($from)
            ->addElement($subscribe)
            ->addElement($submit);

        // устанавливаем значения по умолчанию
        $this->setDefaults(array(
                'type' => 'platinum',
                'subscribe' => 'yes',
                'from' => 'FR',
            ));
    }
}
```

На рис. 3.12 показано, как выглядит результат.

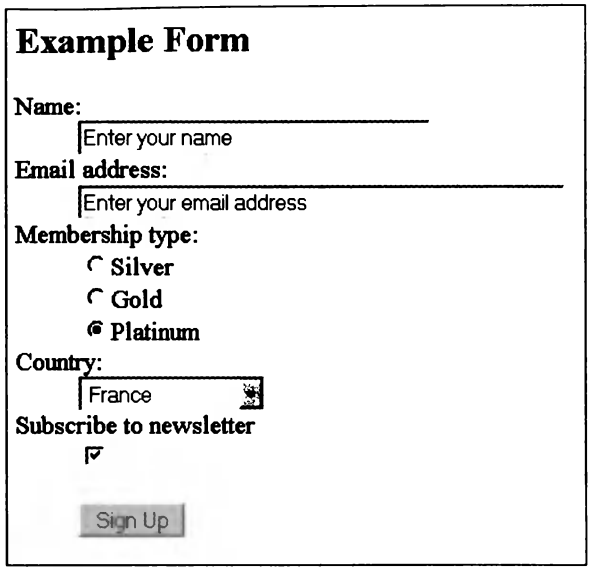


Рис. 3.12. Сгенерированная форма со значениями по умолчанию

Фильтрация и валидация входных данных форм

Поскольку вы разрабатываете веб-приложения, вам придется смириться с одним печальным фактом: в мире существуют люди, получающие удовольствие от поиска лазеек в вашем коде и злонамеренного их использования. Поэтому важнейшей задачей разработчика является обеспечение безопасности приложения, правильная фильтрация и валидация всех проходящих через него входных данных.

В следующих разделах рассматриваются доступные в Zend Framework средства фильтрации и валидации, а также примеры того, как их можно использовать с веб-формами, чтобы сделать приложение более безопасным.

Использование фильтров ввода

В большинстве случаев использование «дырок» в программном коде при обработке входных данных заключается в отправке вашему приложению искусно замаскированных значений, которые «обманывают» его, заставляя делать что-то, чего оно делать не должно. Распространенным примером этого типа эксплоитов является SQL-инъекция, при которой злоумышленник удаленно изменяет данные в вашей базе данных с помощью SQL-запроса, входящего в состав данных, вводимых через форму. Следовательно, одна из важнейших вещей, которые разработчик должен сделать, перед тем как использовать любые входные данные, переданные пользователем, — это их «очистка» путем удаления любых специальных символов.

В PHP существуют различные функции, помогающие разработчикам производить очистку входных данных. Например, функция **addslashes()** экранирует специальные символы (такие, как кавычки и обратные слеши), чтобы входные данные можно было безопасно вносить в базу данных, а функция **strip_tags()** удаляет из строки все HTML- и PHP-теги, оставляя в содержимом только символы ASCII. Также существует функция **htmlentities()**, обычно использующаяся для замены специальных символов, таких как «, &, < и >, соответствующими значениями HTML-сущностей, делая их безвредными.

Пример очистки входных данных формы функцией **htmlentities()**:

```
<?php
// определяем массив для очищенных данных
$sanitized = array();

// удаляемые теги из входных данных, переданных в запросе POST
if (isset($_POST['name']) && !empty($_POST['name'])) {
  $sanitized['name'] = htmlentities($_POST['name']);
}
// код для обработки //
?>
```

В том, что касается фильтрации пользовательского ввода, Zend Framework делает за вас значительную часть тяжелой работы. Компонент **Zend_Filter** предоставляет всеобъемлющий набор *фильтров ввода*, которые можно либо прикрепить к элементам формы методом **addFilter()**, либо использовать отдельно для очистки каких-либо входных данных по мере необходимости. Пример использования фильтра **HTMLEntities** для поля ввода текста:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем фому
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени пользователя
    // и отфильтровываем специальные символы
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Username:')
        ->setOptions(array('size' => '16'))
        ->addFilter('HtmlEntities');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Sign Up');

    // добавляем элементы к форме
    $this->addElement($name)
```

```
        ->addElement($submit);
    }
}
```

Вы также можете передать методу **addFilter()** экземпляр класса **Zend_Filter_***, как показано в следующем примере:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени пользователя
    // и отфильтровываем специальные символы
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Username:')
        ->setOptions(array('size' => '16'))
        ->addFilter(new Zend_Filter_HtmlEntities());

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Sign Up');

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($submit);
    }
}
```

Некоторые фильтры поддерживают дополнительные параметры, которые можно передать в метод **addFilter()** в виде массива или, при использовании экземпляра класса, в качестве аргументов конструктора объекта. Рассмотрим следующий пример, где с помощью фильтра **Alpha** осуществляется удаление из пользовательского ввода всех неалфавитных символов. Дополнительный параметр, переданный методу **addFilter()** в качестве второго аргумента, сохраняет пробельные символы (которые по умолчанию удаляются).

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
```

```
        // разрешаем использование алфавитных и пробельных символов
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:')
            ->setOptions(array('size' => '4'))
            ->setRequired(true)
            ->addFilter('Alpha', array('allowWhiteSpace' => true))
            ->addFilter('HtmlEntities');

        // создаем кнопку отправки
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Sign Up');

        // добавляем элементы к форме
        $this->addElement($name)
            ->addElement($submit);
    }
}
```

В табл. 3.2 приведены некоторые важные фильтры, входящие в состав Zend Framework, и их описания. Многие из этих фильтров вы встретите в этой и последующих главах.

СОВЕТ

Существует два способа добавить к элементу формы несколько фильтров: вызвав метод addFilter() несколько раз, передавая при каждом вызове названия разных фильтров, или воспользовавшись методом addFilters() и передав ему массив, содержащий список названий фильтров.

Таблица 3.2. Фильтры ввода, входящие в состав Zend Framework

Название фильтра	Описание
Alnum	Удаляет из аргумента символы, не являющиеся алфавитными или числовыми
Alpha	Удаляет из аргумента неалфавитные символы
Digits	Удаляет из аргумента нечисловые символы
Int	Возвращает целочисленное значение аргумента
Dir	Возвращает часть аргумента, содержащую название каталога
BaseName	Возвращает часть аргумента, содержащую имя файла
RealPath	Возвращает абсолютный путь к файлу, указанному в аргументе
StringToLower	Переводит аргумент в строку в нижнем регистре
StringToUpper	Переводит аргумент в строку в верхнем регистре
StringTrim	Удаляет из аргумента ведущие и концевые пробелы
StripNewlines	Удаляет из аргумента символы разрыва строки

Название фильтра	Описание
HtmlEntities	Преобразует специальные символы в аргументе в соответствующие HTML-сущности
StripTags	Удаляет из аргумента HTML- и PHP-код
Encrypt	Возвращает зашифрованную версию аргумента
Decrypt	Возвращает расшифрованную версию аргумента
NormalizedToLocalized	Возвращает аргумент в стандартном виде
LocalizedToNormalized	Возвращает аргумент в локализованном виде
Callback	Вызывает определенный пользователем фильтр с указанным аргументом
LowerCase	Переводит содержимое загруженного файла в нижний регистр
UpperCase	Переводит содержимое загруженного файла в верхний регистр
Rename	Переименовывает загруженный файл

Использование валидаторов ввода

Фильтрация ввода — лишь часть головоломки. Помимо этого, чрезвычайно важно произвести валидацию пользовательского ввода, чтобы убедиться в корректности его формата перед использованием его в вычислениях или сохранением в хранилище данных приложения. Неправильная валидация входных данных может не только привести к значительному повреждению и потере данных, но и поставить гордого разработчика в неудобное положение.

Чтобы показать важность валидации ввода, рассмотрим простой пример: онлайн-калькулятор ипотеки, который позволяет пользователю ввести желаемую сумму займа, срок и процентную ставку. Предположим, что в приложении отсутствует какая-либо валидация ввода. Также предположим, что пользователь решил ввести в поле строку 'десять' вместо числа 10.

Несложно угадать, что произойдет дальше. Приложение произведет несколько внутренних вычислений, завершающихся попыткой разделить общую сумму выплат на указанный срок. Так как последний в данном случае является строкой, PHP приведет ее к числу 0, что выльется в ошибку деления на ноль. Множество ужасных ошибок, возникших в результате, заставят покраснеть даже самого искушенного разработчика; и что более важно, если некорректный ввод в его первоначальном виде будет также сохранен в базу данных, ошибка будет возникать при проведении вычислений для каждой записи. Умножьте это хотя бы на несколько сотен записей, содержащих те же ошибки и разбросанных по базе данных, и вы сразу увидите, каким образом отсутствие подходящей валидации ввода может нанести существенный урон приложению.

В PHP имеются различные функции, помогающие разработчикам в решении задач валидации ввода. Например, функция is_numeric() проверяет, является ли

значение числом, а функции ctype_alpha() и ctype_alnum() можно использовать для проверки наличия в строках алфавитных и алфавитно-цифровых символов. Существует также функция filter_var(), которую можно использовать для проверки правильности адресов электронной почты и URL, и функция preg_match(), позволяющая использовать регулярные выражения для валидации по шаблону. Пример использования этих функций:

```
<?php
// определяем массив для корректных данных
$valid = array();

// проверяем, является ли возраст числом
if (is_numeric(trim($_POST['age']))) {
  $valid['age'] = trim($_POST['age']);
} else {
  die ('ERROR: Age is not a number.');
}

// проверяем корректность имени
if (isset($_POST['firstname']) && ctype_alpha($_POST['firstname'])) {
  $valid['firstname'] = trim($_POST['firstname']);
} else {
  die ('ERROR: First name not present or invalid.');
}

// проверяем корректность адреса электронной почты
if (isset($_POST['email'])
  && filter_var($_POST['email'], FILTER_VALIDATE_EMAIL)) {
  $valid['email'] = trim($_POST['email']);
} else {
  die ('ERROR: Email address not present or invalid.');
}
// код для обработки //
?>
```

Как и в случае с фильтрами, Zend Framework содержит множество встроенных *валидаторов ввода* под общим названием Zend_Validate, которые можно либо добавлять к элементам формы с помощью метода addValidator(), либо просто использовать. Специфичные для валидатора параметры можно передать в качестве третьего аргумента метода addValidator() в виде ассоциативного массива пар «ключ-значение», как показано в следующем примере:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');
```

```
    // создаем текстовое поле для ввода возраста
    // оно должно содержать только целочисленные значения от 0 до 100
    $age = new Zend_Form_Element_Text('age');
    $age->setLabel('Age:')
        ->setOptions(array('size' => '4'))
        ->setRequired(true)
        ->addValidator('Int')
        ->addValidator('Between', false, array(1,100));

    // создаем текстовое поле для ввода имени
    // оно должно содержать только алфавитные и пробельные символы
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('First name:')
        ->setOptions(array('size' => '16'))
        ->setRequired(true)
        ->addValidator('Alpha', false, array('allowWhiteSpace' =>
true));

    // создаем текстовое поле для ввода адреса электронной почты
    // оно должно содержать корректный адрес электронной почты
    $email = new Zend_Form_Element_Text('email');
    $email->setLabel('Email address:')
        ->setOptions(array('size' => '16'))
        ->setRequired(true)
        ->addValidator('EmailAddress');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Sign Up');

    // добавляем элементы к форме
    $this->addElement($age)
        ->addElement($name)
        ->addElement($email)
        ->addElement($submit);
    }
}
```

Как и в случае с фильтрами, валидаторы могут указываться в виде экземпляров соответствующего класса **Zend_Validate_***, а их параметры можно передавать в конструкторы объектов. Этот подход продемонстрирован в следующем примере:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');
```

```
        // создаем текстовое поле для ввода возраста
        // оно должно содержать только целочисленные значения от 0 до 100
        $age = new Zend_Form_Element_Text('age');
        $age->setLabel('Age:')
            ->setOptions(array('size' => '4'))
            ->setRequired(true)
            ->addValidator(new Zend_Validate_Int())
            ->addValidator(new Zend_Validate_Between(1,100));

        // создаем текстовое поле для ввода имени
        // оно должно содержать только алфавитные и пробельные символы
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('First name:')
            ->setOptions(array('size' => '16'))
            ->setRequired(true)
            ->addValidator(new Zend_Validate_Alpha(true));

        // создаем текстовое поле для ввода адреса электронной почты
        // оно должно содержать корректный адрес электронной почты
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:')
            ->setOptions(array('size' => '16'))
            ->setRequired(true)
            ->addValidator(new Zend_Validate_EmailAddress());

        // создаем кнопку отправки
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Sign Up');

        // добавляем элементы к форме
        $this->addElement($age)
            ->addElement($name)
            ->addElement($email)
            ->addElement($submit);
    }
}
```

На рис. 3.13 показан результат попытки отправить через форму некорректные данные.

В табл. 3.3 приведены некоторые важные валидаторы, доступные в Zend Framework, и их описания. Многие из этих валидаторов вы встретите в этой и последующих главах.

Таблица 3.3. Валидаторы ввода, входящие в состав Zend Framework

Название валидатора	Описание
NotEmpty	Возвращает false, если аргумент пуст
StringLength	Возвращает false, если длина аргумента не соответствует указанным минимальной и максимальной длине

Название валидатора	Описание
InArray	Возвращает false, если аргумент не находится в указанном массиве
Identical	Возвращает false, если аргумент не совпадает с указанным значением
Alnum	Возвращает false, если аргумент содержит символы, отличные от буквенно-цифровых
Alpha	Возвращает false, если аргумент содержит символы, отличные от алфавитных
Int	Возвращает false, если аргумент не является целым числом
Float	Возвращает false, если аргумент не является числом с плавающей точкой
Hex	Возвращает false, если аргумент не является шестнадцатеричным значением
Digits	Возвращает false, если аргумент содержит символы, отличные от цифровых
Between	Возвращает false, если значение аргумента не находится в указанном числовом диапазоне
GreaterThan	Возвращает false, если значение аргумента не превышает указанное
LessThan	Возвращает false, если значение аргумента не меньше указанного
Date	Возвращает false, если аргумент не является корректной датой
EmailAddress	Возвращает false, если аргумент не соответствует стандартным соглашениям об адресах электронной почты
Hostname	Возвращает false, если аргумент не соответствует стандартным соглашениям об именах хостов
Ip	Возвращает false, если аргумент не соответствует стандартным соглашениям об IP-адресах
Regex	Возвращает false, если аргумент не соответствует указанному регулярному выражению
Barcode	Возвращает false, если аргумент не является корректным штрихкодом
Ccnum	Возвращает false, если аргумент не соответствует алгоритму Луна, связанному с соглашениями о нумерации кредитных карт
Iban	Возвращает false, если аргумент не является корректным номером IBAN
Exists	Возвращает false, если аргумент не является корректным файлом
Count	Возвращает false, если количество загруженных файлов выходит за пределы, определенные аргументом
Size	Возвращает false, если размер загруженного файла выходит за пределы, определенные аргументом

продолжение ↛

Таблица 3.3 (продолжение)

Название валидатора	Описание
FilesSize	Возвращает false, если общий размер загруженных файлов выходит за пределы, определенные аргументом
Extension	Возвращает false, если расширение загруженного файла не соответствует указанным в аргументе
MimeType	Возвращает false, если тип MIME загруженного файла не соответствует указанным в аргументе
IsCompressed	Возвращает false, если загруженный файл не является сжатым файлом архива
IsImage	Возвращает false, если загруженный файл не является изображением
ImageSize	Возвращает false, если размеры загруженного изображения выходят за пределы, определенные аргументом
Crc32, Md5, Sha1, Hash	Возвращает false, если содержимое загруженного файла не соответствует значению хеша, указанного в аргументе (поддерживаются алгоритмы хеширования crc32, md5 и sha1)
ExcludeExtension	Возвращает false, если расширение загруженного файла совпадает с одним из перечисленных в аргументе
ExcludeMimeType	Возвращает false, если тип MIME загруженного файла совпадает с одним из перечисленных в аргументе
WordCount	Возвращает false, если количество слов в загруженном файле выходит за пределы, определенные аргументом
Db_RecordExists	Возвращает false, если определенная запись отсутствует в базе данных и таблице, заданных аргументом
Db_NoRecordExists	Возвращает false, если определенная запись присутствует в базе данных и таблице, заданных аргументом

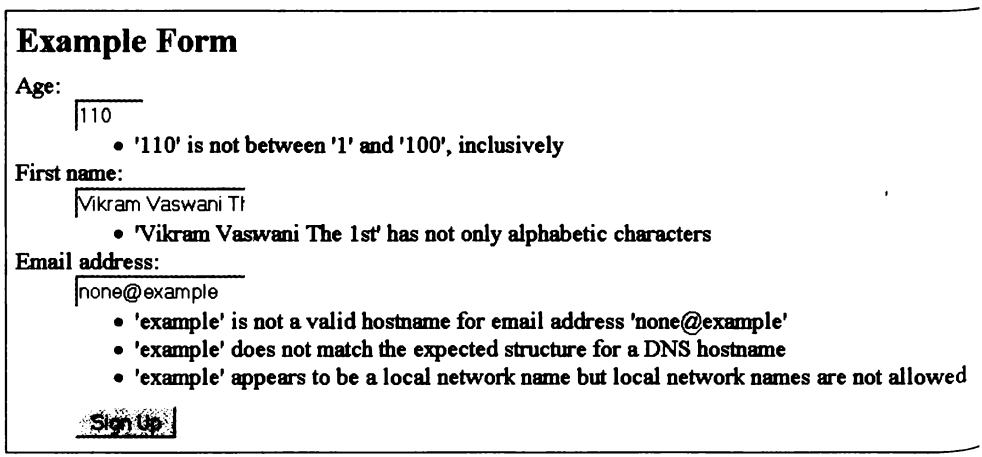


Рис. 3.13. Результат отправки формы с некорректными входными значениями

ВОПРОС ЭКСПЕРТУ

В: Я уже произвожу валидацию формы посредством JavaScript. Зачем делать это еще раз с помощью PHP?

О: Использование языков вроде JavaScript или VBScript для валидации ввода на стороне клиента — распространенная практика. Однако такая валидация ненадежна — если пользователь отключит JavaScript в клиентской программе, весь клиентский код перестанет работать. Именно поэтому хорошей идеей является объединение валидации на стороне клиента (которая быстрее) с валидацией на стороне сервера (которая более безопасна).

Использование цепочек валидаторов и фильтров

К одним из самых интересных возможностей компонентов Zend_Filter и Zend_Validate относится поддержка ими *объединения в цепочку*, или *накопления*. В сущности, это означает возможность добавления к одному элементу ввода нескольких фильтров и валидаторов, которые будут автоматически запускаться по очереди при отправке формы. В следующем примере показано создание цепочки из четырех фильтров:

```
<?php
    // создаем текстовое поле для ввода имени
    // отфильтровываем теги, HTML-сущности и пробельные символы
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('First name:')
        ->setOptions(array('size' => '16'))
        ->setRequired(true)
        ->addFilter('StripTags')
        ->addFilter('HTMLEntities')
        ->addFilter('StringTrim')
        ->addFilter('StringToLower');
?>
```

Здесь первый фильтр удаляет из входных данных HTML- и PHP-теги, второй кодирует HTML-сущности, третий удаляет ведущие и концевые пробельные символы, а четвертый переводит результат в нижний регистр. Фильтры применяются к входному значению в порядке их следования в цепочке.

Цепочки валидаторов работают аналогичным образом и обладают дополнительным свойством: цепочка валидаторов может быть настроена так, что ошибка при выполнении любого из валидаторов приведет к завершению всей цепочки и выдаче сообщения об ошибке. Это поведение определяется вторым аргументом метода addValidator(). Если он имеет значение true, то в случае ошибки при выполнении соответствующего валидатора цепочка прерывается. Рассмотрим это на следующем примере:

```
<?php
    // создаем текстовое поле для ввода возраста
    // должно содержать только целочисленные значения от 0 до 100
    $age = new Zend_Form_Element_Text('age');
    $age->setLabel('Age:')
        ->setOptions(array('size' => '4'))
```

```
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addValidator('Int', true)
        ->addValidator('Between', true, array(1,100));
?>
```

В данном случае ошибка любого из валидаторов приведет к разрыву цепочки, и оставшиеся валидаторы выполняться не будут. Например, если входное значение не является целочисленным, цепочка валидации завершится с сообщением об ошибке, сгенерированным валидатором Int, и валидатор Between выполнен не будет. Сравните приведенный выше листинг со следующим:

```
<?php
    // создаем текстовое поле для ввода возраста
    // должно содержать только целочисленные значения от 0 до 100
    $age = new Zend_Form_Element_Text('age');
    $age->setLabel('Age:')
        ->setOptions(array('size' => '4'))
        ->setRequired(true)
        ->addValidator('NotEmpty', false)
        ->addValidator('Int', false)
        ->addValidator('Between', false, array(1,100));
?>
```

В этой версии, даже в случае ошибки в одном из валидаторов, оставшиеся продолжат выполняться, а сообщения об ошибках, сгенерированые последующими валидаторами, будут добавлены в стек сообщений. На рис. 3.14 и 3.15 показаны различия в поведении приведенных выше листингов.

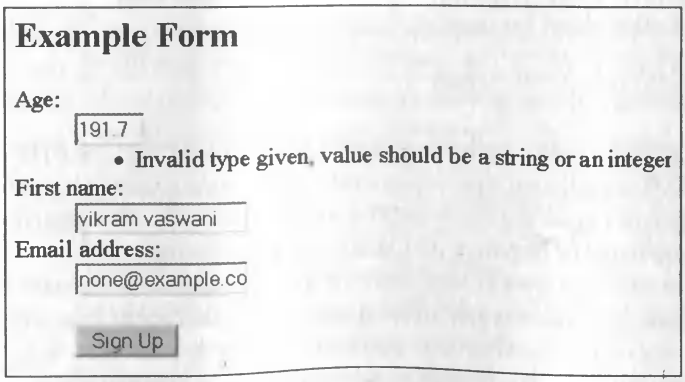


Рис. 3.14. Цепочка валидаторов, разорванная при первой ошибке

СОВЕТ

Если встроенные в Zend Framework фильтры и валидаторы не удовлетворяют вашим требованиям, помните, что вы всегда можете написать собственные. В руководстве Zend Framework приведены примеры того, как это можно сделать.

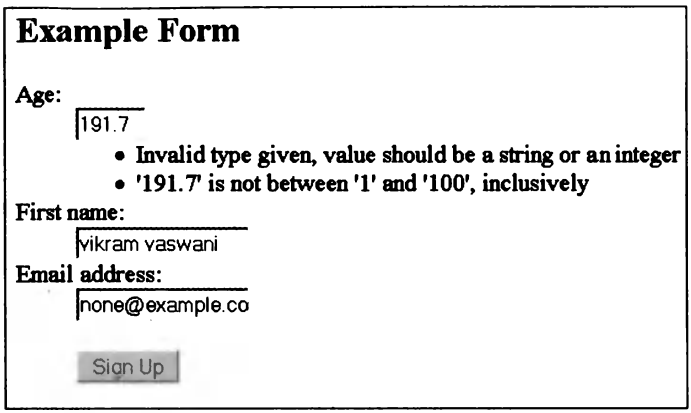


Рис. 3.15. Цепочка валидаторов, обработанная полностью

Получение и обработка данных форм

Чтобы получать и обрабатывать в сценарии контроллера, вы можете использовать несколько методов Zend_Form для получения и обработки формы ввода после ее отправки:

- Метод **isValid()** проверяет, являются ли переданные данные корректными. Он принимает массив входных значений и возвращает логическое значение **true** или **false** в зависимости от того, подчиняются ли эти значения правилам валидации, определенным с помощью вызовов **addValidator()**.
- Если входные данные некорректны, метод **getMessages()** возвращает список сообщений об ошибках, сгенерированных в процессе валидации. Этот список можно обработать и отобразить при повторной генерации формы, чтобы сообщить пользователю, что пошло не так.
- Если входные данные корректны, можно использовать метод **getValues()**, чтобы получить корректные отфильтрованные значения для дальнейшей обработки. Входные значения возвращаются в виде элементов ассоциативного массива, где ключ соответствует имени элемента, а значение — соответствующему входному значению. Существует также метод **getUnfilteredValues()**, возвращающий исходные неотфильтрованные входные данные, введенные пользователем.

СОВЕТ

Метод isValid() автоматически проверяет значения CAPTCHA и Hash, не требуя от вас написания какого-либо кода.

Следующий листинг показывает, как эти методы используются в контексте сценария контроллера:

```
<?php
class ExampleController extends Zend_Controller_Action
```

```
{
  public function formAction()
  {
    $form = new Form_Example;
    $this->view->form = $form;

    // проверяем запрос
    // запускаем валидаторы
    if ($this->getRequest()->isPost()) {
     if ($form->isValid($this->getRequest()->getPost())) {
        // данные корректны: получаем отфильтрованные корректные значения
        // делаем что-нибудь, например сохраняем их в базу данных или
        // записываем в файл
        // отображаем представление, сообщающее об успехе
        $values = $form->getValues();
        $this->_redirect('/form/success');
      } else {
        // данные некорректны: получаем массив с сообщениями об ошибках
        // для ручной обработки (если она требуется)
        // повторно отображаем форму вместе с ошибками
        $this->view->messages = $form->getMessages();
      }
    }
  }
}
?>
```

Упражнение 3.1. Создание формы обратной связи

Получив все предварительные знания, давайте посмотрим, как применить их в нашем приложении. В следующем разделе все ваши знания будут использованы для создания формы отправки запроса по электронной почте для приложения SQUARE. Эта форма будет предлагать пользователю ввести сообщение, а при отправке формы она создаст на основе введенных данных сообщение электронной почты и отправит его на рассмотрение администраторам сайта.

Определение формы

Для начала давайте обдумаем требования к форме ввода. Они просты: на самом деле понадобятся только три поля, в которые пользователь должен будет ввести свое имя, адрес электронной почты и сообщение. Необходимо провести валидацию этих значений, особенно адреса электронной почты, для подтверждения подлинности сообщения, и тем самым дать администраторам возможность отвечать на поступающие по почте вопросы. Для фильтрации автоматических сообщений

и уменьшения процента спама добавим визуальный тест CAPTCHA, что, как говорилось ранее, довольно просто сделать с помощью Zend_Form.

Пример получившейся в результате определения формы:

```
<?php
class Square_Form_Contact extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/contact/index')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Name:')
        ->setOptions(array('size' => '35'))
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addValidator('Alpha', true)
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем текстовое поле для ввода адреса электронной почты
    $email = new Zend_Form_Element_Text('email');
    $email->setLabel('Email address:');
    $email->setOptions(array('size' => '50'))
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addValidator('EmailAddress', true)
        ->addFilter('HtmlEntities')
        ->addFilter('StringToLower')
        ->addFilter('StringTrim');

    // создаем текстовое поле для сообщения
    $message = new Zend_Form_Element_Textarea('message');
    $message->setLabel('Message:')
        ->setOptions(array('rows' => '8','cols' => '40'))
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем CAPTCHA
    $captcha = new Zend_Form_Element_Captcha('captcha', array(
       'captcha' => array(
            'captcha' => 'Image',
            'wordLen' => 6,
            'timeout' => 300,
            'width' => 300,
```

```
            'height' => 100,
            'imgUrl' => '/captcha',
            'imgDir' => APPLICATION_PATH . '/../public/captcha',
            'font' => APPLICATION_PATH .
                      '/../public/fonts/LiberationSansRegular.ttf',
            )
        ));
    $captcha->setLabel('Verification code:');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Send Message')
        ->setOptions(array('class' => 'submit'));

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($email)
        ->addElement($message)
        ->addElement($captcha)
        ->addElement($submit);
    }
}
```

С большей частью приведенного кода вы уже должны быть знакомы. Форма содержит два элемента ввода текста для имени пользователя и адреса его электронной почты, одну текстовую область для тела сообщения и элемент CAPTCHA для проверки. К полям для ввода имени и тела сообщения прикреплены валидаторы **Alpha** и **NotEmpty**, а для проверки адреса электронной почты используется валидатор **EmailAddress**. Все поля фильтруются с помощью валидатора **HTMLEntities**, а адрес электронной почты дополнительно переводится в нижний регистр валидатором **StringToLower**.

Также стоит рассмотреть параметры, передаваемые экземпляру **Zend_Form_Element_Captcha**. В отличие от примера из предыдущего раздела, это определение генерирует более сложный CAPTCHA, динамически накладывая случайную последовательность символов на искаженный фон. Такой тип CAPTCHA часто используется в веб-формах, чтобы заблокировать автоматическую отправку форм роботами, многие их которых используют алгоритмы оптического распознавания образов (Optical Character Recognition, OCR), способные «прочитать» символы на однородном фоне. К параметрам, передаваемым экземплярам класса, относятся размеры изображения для CAPTCHA, путь для сохранения сгенерированного CAPTCHA, количество символов и файл шрифта, который будет использоваться для накладываемого текста.

Вы, вероятно, заметили, что предыдущий пример использует пользовательский шрифт и сохраняет сгенерированные CAPTCHA в указанном каталоге. Следовательно, создайте также каталоги **$APP_DIR/public/captcha/** и **$APP_DIR/public/fonts/** и скопируйте в них необходимые файлы из архива с дополнительными материалами к этой главе.

ВНИМАНИЕ

Если вы используете защищенные авторским правом шрифты, запрещенные к распространению, вы должны переместить каталог $APP_DIR/public/fonts/ за пределы корневого каталога документов сервера, например в $APP_DIR/application/fonts/, чтобы гарантировать, что к шрифтам не будет общего доступа через веб-браузер. Выполняя это действие, не забудьте также изменить путь к каталогу в коде приложения.

ПРИМЕЧАНИЕ

В данном примере для CAPTCHA используется шрифт Liberation Sans, являющийся частью набора шрифтов, переданных сообществу корпорацией Red Hat Inc. в 2007 году под лицензией GNU General Public License. Пользователям разрешается использовать, модифицировать, копировать и распространять эти шрифты на условиях лицензии GNU GPL.

Использование пользовательского пространства имен

Определение из предыдущего раздела использует пользовательское пространство имен, «Square», которое добавляется перед именем класса. Этот подход рекомендован для любых пользовательских объектов или библиотек, которые вы можете создать для приложения, так как помогает избежать коллизий имен между вашими определениями и другими определениями, которые могут в нем существовать. Дополнительное преимущество заключается в том, что если вы зарегистрируете ваше пространство имен в автозагрузчике Zend Framework и поместите определения в нужной части структуры каталогов приложения, Zend Framework будет автоматически находить их и при необходимости загружать во время выполнения.

С учетом сказанного, сохраните определение класса из предыдущего листинга в файл $APP_DIR/library/Square/Form/Contact.php, а затем добавьте в конфигурационный файл приложения $APP_DIR/application/configs/application.ini следующую директиву, чтобы зарегистрировать пространство имен «Square» в автозагрузчике:

```
autoloaderNamespaces[] = "Square_"
```

ВНИМАНИЕ

Если для отделения пространства имен от оставшейся части имени класса в ваших классах используется нижнее подчеркивание, вы должны добавить его при регистрации пространства имен в автозагрузчике Zend Framework.

Определение пользовательского маршрута

Настало подходящее время, чтобы добавить пользовательский маршрут для новой формы. Откройте конфигурационный файл приложения в текстовом редакторе и добавьте в него следующее определение маршрута:

```
resources.router.routes.contact.route = /contact
resources.router.routes.contact.defaults.module = default
resources.router.routes.contact.defaults.controller = contact
resources.router.routes.contact.defaults.action = index
```

В главе 2 вы познакомились с этим определением: оно создает маршрут, согласно которому запросы к приложению по URL /contact будут обрабатываться действием ContactController::indexAction из модуля default.

Определение контроллеров и представлений

Следующим шагом будет определение вышеупомянутого действия ContactController::indexAction. Как правило, контроллер должен находиться в файле $APP_DIR/application/modules/default/controllers/ContactController.php и выглядеть следующим образом:

```
<?php
class ContactController extends Zend_Controller_Action
{
  public function init()
  {
    $this->view->doctype('XHTML1_STRICT');
  }

  public function indexAction()
  {
    $form = new Square_Form_Contact();
    $this->view->form = $form;

    if ($this->getRequest()->isPost()) {
      if ($form->isValid($this->getRequest()->getPost())) {
        $values = $form->getValues();
        $mail = new Zend_Mail();
        $mail->setBodyText($values['message']);
        $mail->setFrom($values['email'], $values['name']);
        $mail->addTo('info@square.example.com');
        $mail->setSubject('Contact form submission');
        $mail->send();
        $this->_helper->getHelper('FlashMessenger')
           ->addMessage('Thank you. Your message was successfully
sent.');
        $this->_redirect('/contact/success');
    }
  }
}

public function successAction()
{
    if ($this->_helper->getHelper('FlashMessenger')->getMessages()) {
      $this->view->messages =
        $this->_helper->getHelper('FlashMessenger')->getMessages();
    } else {
       $this->_redirect('/');
    }
  }
}
```

Большая часть тяжелой работы в данном случае выполняется методом indexAction(), который создает объект рассмотренного ранее класса Square_Form_Contact и прикрепляет его к представлению. При отправке формы для валидации переданных пользователем данных используется метод isValid() этого объекта. Если данные корректны, создается экземпляр компонента Zend_Mail и с помощью его методов на основе полученных данных создается сообщение электронной почты, которое отправляется на указанный адрес. После отправки сообщения управление передается методу successAction(), который формирует представление, сообщающее об успехе операции.

Это происходит, если все идет хорошо... однако не стоит загадывать наперед, а поэтому подумайте, что может случиться при возникновении каких-либо проблем. Если входные данные некорректны, метод isValid() вернет false и форма будет отображена повторно, на этот раз с сообщениями о причинах ошибки(-ок). Кроме этого, Zend_Form автоматически заполнит форму исходными данными, чтобы пользователю не пришлось вводить их повторно. С другой стороны, если данные корректны, но произошла ошибка в процессе генерации и отправки сообщения, Zend_Mail выбросит исключение PHP, которое будет перехвачено и обработано стандартным обработчиком ошибок приложения.

ПРИМЕЧАНИЕ

Для корректной работы метода send() объекта Zend_Mail в системе должен присутствовать агент доставки сообщений (например, sendmail), должным образом настроенный через файл php.ini. В противном случае доставка сообщения завершится с ошибкой, а метод send() выбросит исключение.

В этом контроллере также используется новое средство, вспомогательный класс FlashMessenger, который является маленьким полезным «помощником», упрощающим отображение пользователю статусных сообщений. Сообщения можно добавлять в объект FlashMessenger с помощью его метода addMessage(); они хранятся в сессии до тех пор, пока не будут получены вызовом метода getMessages(), после чего происходит их удаление из сессии. Это делает FlashMessenger удобным местом для временного хранения сообщений в промежуток времени между окончанием операции и формированием следующего представления, поэтому вы будете часто встречать его использование в этой книге.

ВНИМАНИЕ

Вы могли заметить, что метод init() в контроллере устанавливает для типа документа значение XHTML 1.0 Strict. Это сделано потому, что по умолчанию Zend_Form не генерирует XHTML-разметку в правильном формате. Установка указанного типа документа заставляет фреймворк использовать правильный формат.

Вам также потребуется пара представлений: для формы ввода и для сообщения об успехе. Ниже приведено представление для ввода данных, которое следует поместить в файл $APP_DIR/application/modules/default/views/scripts/contact/index.phtml:

```
<h2>Contact</h2>
<?php echo $this->form; ?>
```

Второе представление предназначено для вывода сообщения об успешной отправке и должно быть помещено в файл **$APP_DIR/application/modules/default/views/scripts/contact/success.phtml**:

```
<h2>Success</h2>
<?php echo implode($this->messages); ?>
```

Обновление основного макета

Все, что осталось сделать, — обновить навигационные ссылки в главном меню приложения с помощью вспомогательного метода **url()**, чтобы отразить появление новой формы запроса. Для их обновления внесите в файл основного макета, **$APP_DIR/application/layouts/master.phtml**, выделенные жирным шрифтом изменения:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8"/>
  <base href="/" />
    <link rel="stylesheet" type="text/css" href="/css/master.css" />
  </head>
  <body>
    <div id="header">
      <div id="logo">
        <img src="/images/logo.gif" />
      </div>

      <div id="menu">
        <a href="<?php echo $this->url(array(), 'home'); ?>">HOME</a>
        <a href="<?php echo $this->url(array('page' => 'services'),
          'static-content'); ?>">SERVICES</a>
        <a href="<?php echo $this->url(array(), 'contact');
?>">CONTACT</a>
      </div>
    </div>

    <div id="content">
      <?php echo $this->layout()->content ?>
    </div>

    <div id="footer">
      <p>Created with <a href="http://framework.zend.com/">
      Zend Framework</a>. Licensed under
      <a href="http://www.creativecommons.org/">Creative Commons
      </a>.</p>
    </div>
  </body>
</html>
```

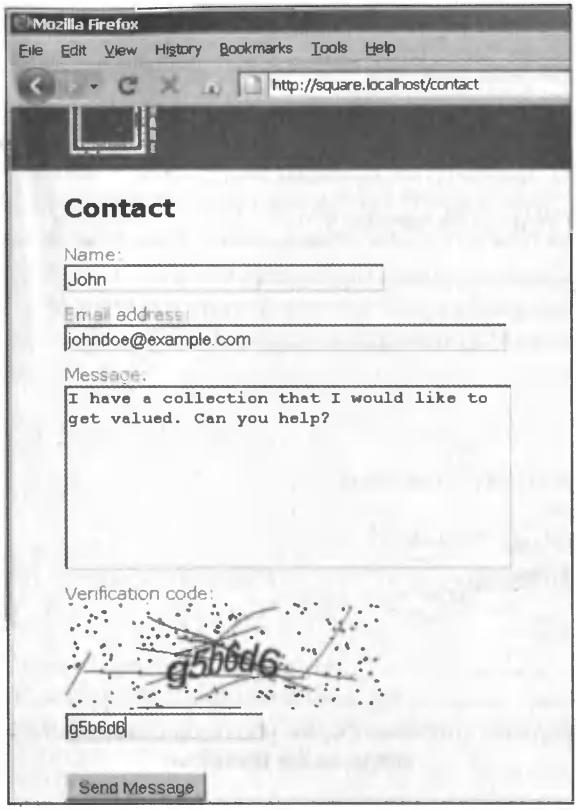


Рис. 3.16. Форма обратной связи SQUARE

Если вы сейчас перейдете по URL http://square.localhost/contact, вы должны увидеть форму, показанную на рис. 3.16.

Рис. 3.17. Результат успешной отправки формы обратной связи SQUARE

Введите значения в поля формы и отправьте ее; если все пойдет хорошо, вы должны увидеть сообщение об успехе, показанное на рис. 3.17.

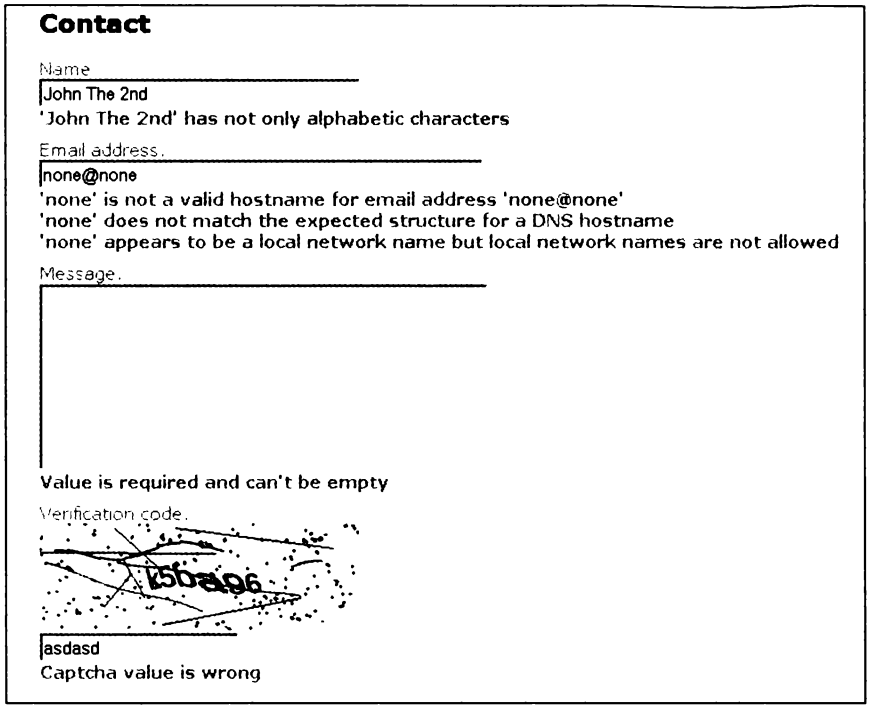


Рис. 3.18. Результат отправки формы обратной связи SQUARE с ошибочно введенными данными

Вы также можете попробовать отправить форму с некорректными значениями. Встроенные валидаторы перехватят ваш ввод и повторно отобразят форму, добавив к ней сообщения об ошибках (рис. 3.18).

ПРИМЕЧАНИЕ

Не забудьте заменить адрес получателя сообщения в действии ContactController::indexAction собственным адресом электронной почты, иначе вы никогда не получите сообщений, сгенерированных компонентом Zend_Mail.

Настройка внешнего вида формы

В данный момент вы уже хорошо понимаете, каким образом в Zend Framework создаются формы, и имеете некоторое представление о доступных средствах, помогающих защитить ваше приложение от некорректных или злонамеренных входных данных. В этом разделе поверхностно рассматриваются некоторые средства, которые помогут улучшить внешний вид и поведение форм, сделать их более понятными и информативными.

Использование пользовательских сообщений об ошибках

Каждый валидатор входных данных содержит набор стандартных сообщений об ошибках, соответствующих типу данных, для которых производится валидация. Чаще всего стандартные сообщения предоставляют пользователю достаточно информации, чтобы он мог найти и исправить свои ошибки. Однако могут возникнуть ситуации, в которых эти сообщения следует изменить, чтобы сделать их более наглядными и удобными для пользователя.

Сделать это несложно, поскольку каждый валидатор можно настроить на отображение пользовательских сообщений об ошибках с помощью ключа **'messages'** в массиве параметров, передаваемых методу **addValidator()**. Рассмотрим его использование на примере:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Name:')
        ->setOptions(array('size' => '35'))
        ->setRequired(true)
        ->addValidator('Alpha', false, array(
            'messages' => array(
              Zend_Validate_Alpha::INVALID
                 => "ERROR: Invalid name",
              Zend_Validate_Alpha::NOT_ALPHA
                 => "ERROR: Name cannot contain non-alpha characters",
              Zend_Validate_Alpha::STRING_EMPTY
                 => "ERROR: Name cannot be empty"
            )
        ))
      ->addFilter('StringTrim');

    $validator = $name->getValidator('Alpha');

    // создаем текстовое поле для ввода адреса электронной почты
    $email = new Zend_Form_Element_Text('email');
    $email->setLabel('Email address:');
    $email->setOptions(array('size' => '50'))
        ->setRequired(true)
        ->addValidator('EmailAddress', true, array(
            'messages' => array(
              Zend_Validate_EmailAddress::INVALID
```

```
                => "ERROR: Invalid email address",
            Zend_Validate_EmailAddress::INVALID_FORMAT
                => "ERROR: Invalid email address",
            Zend_Validate_EmailAddress::INVALID_HOSTNAME
                => "ERROR: Invalid hostname format",
            Zend_Validate_EmailAddress::INVALID_LOCAL_PART
                => "ERROR: Invalid username format",
            Zend_Validate_EmailAddress::LENGTH_EXCEEDED
                => "ERROR: Email address too long"
          )
        ))'
      ->addFilter('StringTrim');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Sign Up');

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($email)
        ->addElement($submit);
    }
}
```

В данном примере стандартные сообщения об ошибках для каждого случая ввода некорректных данных переопределены пользовательскими сообщениями, указанными в качестве значения для ключа **'messages'** в массиве, который передается методу **addValidator()**. Константы, расположенные в этом массиве слева, можно узнать, посмотрев исходный код соответствующего валидатора. Новые сообщения будут генерироваться, добавляться в стек ошибок и отображаться на форме каждый раз, когда соответствующий элемент не пройдет валидацию. Пример вывода показан на рис. 3.19.

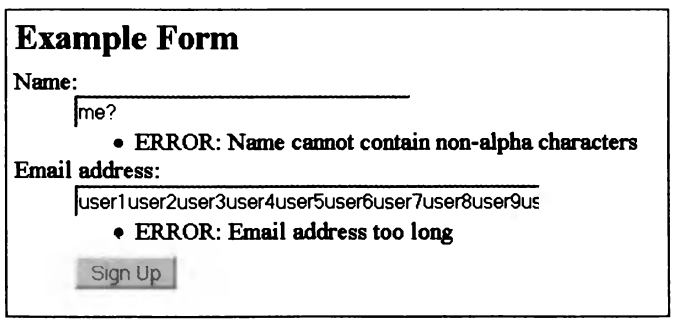


Рис. 3.19. Форма с пользовательскими сообщениями об ошибках

СОВЕТ

Вы также можете передевать валидатору пользовательские сообщения об ошибках, используя его метод setMessages().

Использование групп отображения

Как правило, хорошей идеей является группирование элементов формы, которые так или иначе связаны друг с другом. Такой подход более корректен и обеспечивает лучшую читаемость, чем неупорядоченный набор элементов, собранных вместе без какого-либо четкого деления на категории. В Zend_Form эта задача решается путем использования *групп отображения*.

Группы отображения добавляются на форму с помощью метода addDisplayGroup() объекта формы; метод принимает два аргумента: массив, содержащий названия элементов в группе, и название самой группы. Элементы должны быть предварительно добавлены на форму методом addElement(). Впоследствии можно указать название для группы отображения с помощью метода setLegend(). При формировании страницы группы отображения и комментарии к ним представляются соответственно элементами <fieldset> и <legend>.

Наглядный пример:

```
<?php
class Form_Example extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('Name:')
        ->setOptions(array('size' => '35'))
        ->setRequired(true)
        ->addFilter('StringTrim');

    // создаем текстовое поле для ввода адреса электронной почты
    $email = new Zend_Form_Element_Text('email');
    $email->setLabel('Email address:');
    $email->setOptions(array('size' => '50'))
        ->setRequired(true)
        ->addValidator('EmailAddress', true)
        ->addFilter('StringTrim');

    // создаем текстовое поле для ввода номера телефона
    $tel = new Zend_Form_Element_Text('tel');
    $tel->setLabel('Telephone number:');
    $tel->setOptions(array('size' => '50'))
        ->setRequired(true);

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Sign Up');
```

```
        // добавляем элементы к форме
        $this->addElement($name)
            ->addElement($email)
            ->addElement($tel);

        // добавляем группу отображения
        $this->addDisplayGroup(
            array('name', 'email', 'tel', 'address'),
            'contact'
        );
        $this->getDisplayGroup('contact')
            ->setLegend('Contact Information');
        $this->addElement($submit);
    }
}
```

Результат показан на рис. 3.20.

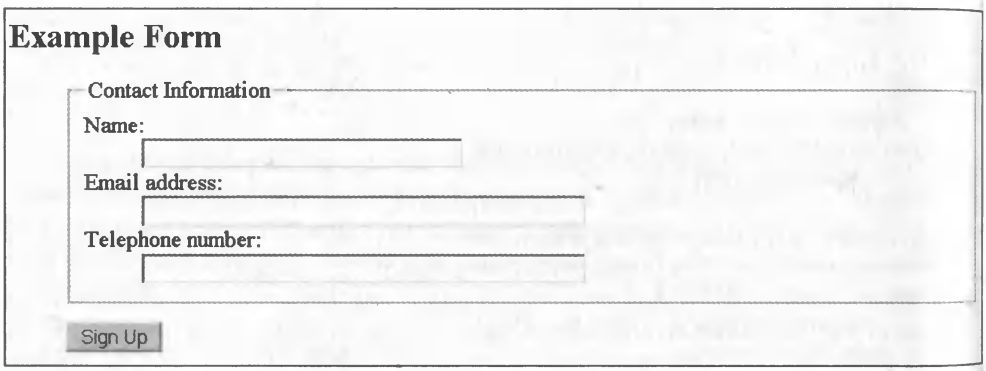


Рис. 3.20. Форма с отображением групп

Использование декораторов

При формировании формы Zend_Form автоматически помещает каждый ее элемент в набор HTML-блоков, управляющих размещением и внешним видом элемента. Эти блоки HTML-разметки называются *декораторами*, их можно модифицировать с помощью CSS или отдельно настраивать для каждого элемента, чтобы внести радикальные изменения в формируемый код элементов формы.

Как и в случае с фильтрами и валидаторами, в состав Zend Framework входят несколько стандартных декораторов. В табл. 3.4 перечислены те из них, с которыми вам предстоит столкнуться.

Стандартные декораторы Zend_Form используют следующую разметку:

- Метки форм помещаются в элементы <dt>...</dt>.
- Поля ввода помещаются в элементы <dd>...</dd>.
- Ошибки валидации представляются как элементы списков и помещаются в теги <ul>....

Таблица 3.4. Часто используемые декораторы форм

Название декоратора	Описание
Form	Управляет разметкой вокруг формы
FormElements	Управляет разметкой вокруг полей формы
HtmlTag	Управляет разметкой вокруг полей формы
ViewHelper	Управляет вспомогательным классом представления
Errors	Управляет разметкой вокруг ошибок валидации (для каждого поля)
FormErrors	Управляет разметкой вокруг ошибок валидации (общих)
Description	Управляет разметкой вокруг описаний полей
Label	Управляет разметкой вокруг меток полей
Fieldset	Управляет разметкой вокруг наборов полей

На рис. 3.21 представлен HTML-код одной из таких форм, а на рис. 3.22 — получившаяся в результате страница.

```
<html>
  <head>
  </head>
  <body>
    <h2>Example Form</h2>
    <form enctype="application/x-www-form-urlencoded" action="/sandbox/example/form" method="post"><dl class="z
<dt id="name-label"><label for="name" class="required">Name:</label></dt>
<dd id="name-element">
<input type="text" name="name" id="name" value="" size="35" />
<ul class="errors"><li>Value is required and can't be empty</li></ul></dd>
<dt id="email-label"><label for="email" class="required">Email address:</label></dt>
<dd id="email-element">
<input type="text" name="email" id="email" value="" size="50" />
<ul class="errors"><li>Value is required and can't be empty</li></ul></dd>
<dt id="subscribe-label"><label for="subscribe" class="optional">Subscribe to newsletter:</label></dt>
<dd id="subscribe-element">
<input type="hidden" name="subscribe" value="no" /><input type="checkbox" name="subscribe" id="subscribe" value
<dt id="submit-label"> </dt><dd id="submit-element">
<input type="submit" name="submit" id="submit" value="Sign Up" class="submit" /></dd></dl></form>  </body>
</html>
```

Рис. 3.21. Исходный код формы, использующей стандартные декораторы

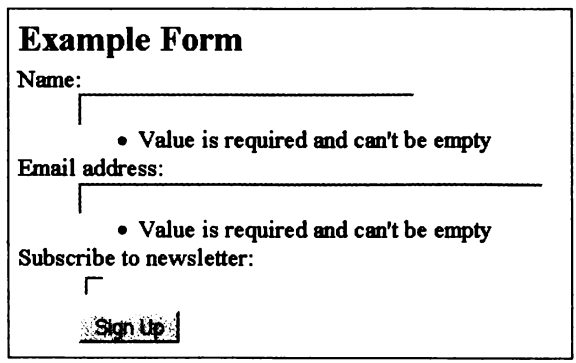


Рис. 3.22. Внешний вид формы, представленной на рис. 3.21

Зачастую такое расположение элементов не вписывается в пользовательский интерфейс приложения. Чтобы изменить его, используйте один из следующих методов:

- Метод **addDecorators()** принимает массив имен декораторов и добавляет соответствующие декораторы к элементу. Если декоратор уже существует, его настройки будут заменены на указанные в методе **addDecorators()**.
- Метод **clearDecorators()** удаляет из элемента все существующие декораторы.
- Метод **setDecorators()** принимает массив имен декораторов, удаляет из элемента все существующие декораторы и устанавливает для него новый набор декораторов.

СОВЕТ

Чтобы отключить стандартные декораторы для конкретной формы или ее элемента, добавьте к массиву параметров этой формы или элемента ключ 'disableLoadDefaultDecorators'. Обратите внимание, что после отключения стандартных декораторов вам тем не менее придется вернуть декоратор ViewHelper, воспользовавшись методом addDecorators() или setDecorators(), так как он создает базовую разметку для формы и ее элементов.

Чтобы продемонстрировать, каким образом можно использовать перечисленные декораторы, рассмотрим следующий пример:

```
<?php
class Form_Example extends Zend_Form
{
  public $formDecorators = array(
    array('FormElements'),
    array('Form'),
   );

    public $elementDecorators = array(
    array('ViewHelper'),
    array('Label'),
array('Errors'),
  );

    public $buttonDecorators = array(
    array('ViewHelper'),
    array('HtmlTag', array('tag' => 'p'))
  );

    public function init()
{
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post')
        ->setDecorators($this->formDecorators);

    // создаем текстовое поле для ввода имени
```

```
        $name = new Zend_Form_Element_Text('name');
        $name->setLabel('Name:')
            ->setOptions(array('size' => '35'))
            ->setRequired(true)
            ->setDecorators($this->elementDecorators);

        // создаем текстовое поле для ввода адреса электронной почты
        $email = new Zend_Form_Element_Text('email');
        $email->setLabel('Email address:');
        $email->setOptions(array('size' => '50'))
            ->setRequired(true)
            ->setDecorators($this->elementDecorators);

        // создаем переключатель для подписки на новостную рассылку
        $subscribe = new Zend_Form_Element_Checkbox('subscribe');
        $subscribe->setLabel('Subscribe to newsletter:')
            ->setCheckedValue('yes')
            ->setUncheckedValue('no')
            ->setDecorators($this->elementDecorators);

        // создаем кнопку отправки
        $submit = new Zend_Form_Element_Submit(
                'submit', array('class' => 'submit'));
        $submit->setLabel('Sign Up')
            ->setDecorators($this->buttonDecorators);

        // добавляем элементы к форме
        $this->addElement($name)
            ->addElement($email)
            ->addElement($subscribe)
            ->addElement($submit);
    }
}
```

Код, приведенный в данном листинге, убирает стандартные декораторы с помощью метода **setDecorators()**, тем самым удаляя элементы **<dt>...</dt>** и **<dd>...</dd>** вокруг меток и полей ввода, и вместо этого настраивает внешний вид элементов с помощью стилей CSS, которые обеспечивают более точный контроль.

ПРИМЕЧАНИЕ

Декораторы сами по себе являются обширной темой для обсуждения, и в этой книге невозможно рассмотреть их во всех подробностях. Материала, представленного в предыдущем разделе, достаточно, чтобы помочь вам разобраться с относительно простыми декораторами, используемыми в книге. Для более детального изучения обратитесь к ссылкам в конце этой главы.

Как показано в примере, декораторы предлагают простое решение проблемы приведения разметки формы в соответствие с остальным пользовательским интерфейсом вашего приложения. А если вы решите, что стандартные декораторы не предоставляют достаточного контроля, вы всегда сможете определить собственную

разметку формы, создав класс пользовательского декоратора (для этого нужно расширить абстрактный класс Zend_Form_Decorator_Abstract) и воспользовавшись им. Ссылки на статьи, затрагивающие этот вопрос, приведены в конце главы.

Выводы

Эта глава была посвящена формам: их созданию, валидации и обработке. В ней вы познакомились с тремя наиболее важными компонентами Zend Framework – Zend_Form, Zend_Validate и Zend_Filter – и увидели, каким образом их можно использовать для организации безопасных запросов и обработки пользовательского ввода. Затем полученные знания были применены к демонстрационному приложению SQUARE, и вы прошли через процесс регистрации нового пространства имен для пользовательских объектов, создания формы отправки вопроса по электронной почте и передачи полученных данных в виде электронного сообщения с помощью компонента Zend_Mail. И наконец, в этой главе были кратко рассмотрены некоторые средства, помогающие настроить внешний вид созданных вами форм, такие как группы отображения, декораторы и пользовательские сообщения об ошибках; в процессе знакомства с книгой вы еще не раз встретите эти средства.

Если вы хотите получить больше информации по изложенным в этой главе темам, вам будут полезны следующие ссылки:

- Информация об атаках CSRF и XSS в Википедии: http://en.wikipedia.org/wiki/Cross-site_request_forgery и
- http://en.wikipedia.org/wiki/Cross-site_scripting.
- Информация о CAPTCHA в Википедии: http://en.wikipedia.org/wiki/Captcha.
- Информация о текучих интерфейсах в Википедии: http://en.wikipedia.org/wiki/Fluent_interface.
- Автозагрузчик Zend Framework: http://framework.zend.com/manual/en/zend.loader.autoloader.html.
- Компонент Zend_Form: http://framework.zend.com/manual/en/zend.form.html.
- Компонент Zend_Filter: http://framework.zend.com/manual/en/zend.filter.html.
- Компонент Zend_Validate: http://framework.zend.com/manual/en/zend.validate.html.
- Компонент Zend_Mail: http://framework.zend.com/manual/en/zend.mail.html.
- Создание пользовательских декораторов (Мэтью Вайер О'Финни (Matthew Weier O'Phinney)): http://weierophinney.net/matthew/archives/213-From-the-inside-out-How-to-layerdecorators.html.

4 Работа с моделями

Прочитав эту главу, вы:

- узнаете все о роли моделей в паттерне Модель–Представление–Контроллер;
- изучите преимущества подхода «толстая модель, тонкий контроллер»;
- интегрируете инструментарий объектно-реляционного отображения (Object Relational Mapping, ORM) Doctrine в Zend Framework;
- создадите модели, используя генератор моделей Doctrine;
- используете модели Doctrine для получения записей из базы данных MySQL;
- выясните, как осуществлять фильтрацию ввода с помощью компонента Zend_Filter_Input;
- настроите процесс запуска приложения с помощью начального загрузчика Zend Framework;

В предыдущем разделе вы прошли ускоренный курс по основам создания, валидации и обработки форм с помощью Zend Framework. Однако введенные на форме данные не растворяются в пустоте; они должны куда-то попадать. Обычно под «куда-то» подразумевается база данных, например MySQL, SQLite или PostgreSQL, поэтому добавление поддержки базы данных в приложение становится основной задачей разработчика.

Однако база данных — лишь часть головоломки: для взаимодействия с ней вам также потребуются *модели*. Они логически не зависят от представлений и контроллеров и служат в качестве *слоя данных* приложения, предоставляя все необходимые функции для манипуляций с его данными. Zend Framework упрощает написание моделей, соответствующих требованиям вашего приложения; кроме того, вы можете интегрировать модели, созданные популярными сторонними инструментами, такими как Doctrine или Propel.

Эта глава улучшит ваши навыки работы с Zend Framework, показав, как создавать и добавлять модели приложение, а затем использовать их для получения данных из базы данных MySQL. Вы также познакомитесь с Doctrine, мощным (и свободным) инструментом отображения данных, который поможет значительно упростить эти задачи.

Модели

К этому моменту вы уже хорошо знакомы с представлениями и контроллерами, которые соответственно представляют собой *слой пользовательского интерфейса* и *слой обработки* в приложении, использующем MVC. Модели представляют *слой данных*, ответственный как за предоставление канала, через который можно получить доступ к данным, так и за поддержание и соблюдение любых бизнес-правил, связанных с этими данными. Например, при создании системы учета книг для онлайн-библиотеки вы можете использовать модель Transaction, которая будет не только записывать информацию о взятых и возвращенных книгах, но и содержать правила, запрещающие читателю иметь на руках более пяти книг единовременно или автоматически рассчитывающие плату за просроченный возврат и списывающие ее со счета соответствующего пользователя. Аналогично, при создании приложения для продажи авиабилетов вы можете использовать модель Ticket, которая будет не только сохранять и получать записи о приобретении билетов, но и автоматически увеличивать или уменьшать стоимость билета в зависимости от загруженности конкретного рейса.

Пример простой модели:

```
<?php
// модель для обработки данных о клиентах
class MemberModel
{
  protected $db;
  public $id;
  public $name;
  public $age;
  public $type;
  private $rate;

  // конструктор
  // инициализируем подключение к базе данных
  public function __construct()
  {
    $this->db = new PDO('mysql:dbname=db;host=localhost', 'user', 'pass');
    $this->db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  }

  // получаем запись о клиенте по его идентификатору
  public function fetch($id)
  {
    $id = $this->db->quote($this->id);
    $rs = $this->db->query("SELECT * FROM member WHERE id = $id");
    return $rs->fetchAll(PDO::FETCH_ASSOC);
  }

  // получаем записи обо всех клиентах
  public function fetchAll()
```

```
{
  $rs = $this->db->query("SELECT * FROM member");
  return $rs->fetchAll(PDO::FETCH_ASSOC);
}

// добавляем новую запись о клиенте
public function save()
{
  // фильтруем входные данные
  $f = array();
  $f['name'] = htmlentities($this->name);
  $f['age'] = htmlentities($this->age);
  $f['type'] = htmlentities($this->type);

  // проверяем возраст
  if ($f['age'] < 18) {
    throw new Exception('Member under 18');
  }

  // автоматически вычисляем скидку в зависимости от статуса клиента
  switch ($f['type']) {
    case 'silver':
      $f['rate'] = 0;
      break;
    case 'gold':
      $f['rate'] = 0.10;
      break;
    case 'platinum':
      $f['rate'] = 0.25;
      break;
  }

  $this->db->exec(
    'INSERT INTO member (Name, Age, Type, DiscountRate) VALUES(' .
    $this->db->quote($f['name']) . ', ' .
    $this->db->quote($f['age']) . ', ' .
    $this->db->quote($f['type']) . ', ' .
    $this->db->quote($f['rate']) . ')'
  );

  return $this->db->lastInsertId();
}
}
?>
```

Как видите, для взаимодействия с базой данных MySQL эта модель использует слой абстракции PHP — PDO. Она предоставляет несколько методов для получения записей из базы данных и сохранения их в базу. В ней также содержатся некоторые бизнес-правила (автоматический подсчет скидки, основанный на статусе клиента) и некоторые правила валидации ввода (проверка того, что клиент

не моложе 18 лет). Как и у всех хороших моделей, ее начальной и конечной целью являются данные; модель не содержит сведений о том, как эти данные форматируются и отображаются.

Пример возможного использования этой модели в приложении:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function saveAction()
  {
    if ($this->getRequest()->isPost()) {
      if ($form->isValid($this->getRequest()->getPost())) {
        $model = new MemberModel;
        $model->name = $form->getValue('name');
        $model->age = $form->getValue('age');
        $model->type = $form->getValue('type');
        $id = $model->save();
        $this->view->message = "Record saved with ID: $id";
      }
    }
  }
}
```

Паттерны для моделей

Для работы с моделями существуют два часто используемых паттерна: **DataMapper** и **ActiveRecord**. В предыдущем примере продемонстрирован паттерн **ActiveRecord**, в котором класс модели непосредственно соответствует таблице в базе данных и предоставляет для работы с записями этой таблицы такие методы, как **save()**, **update()** и **delete()**. Очевидно, что в этом паттерне модели и соответствующие им таблицы базы данных тесно связаны друг с другом, и внесение изменений в одну из них требует внесения изменений в другую.

Паттерн **DataMapper** несколько отличается от **ActiveRecord**, так как не требует наличия отношения 1:1 между классом модели и соответствующей таблицей базы данных. Вместо этого в нем используется промежуточный слой (**mapper** — отображающий класс), с помощью которого решается задача отображения данных из класса модели на поля таблицы. В этом случае именно отображающий класс предоставляет методы **save()**, **update()** и **fetch()** и осуществляет преобразования, необходимые для корректного отображения элементов класса на поля в базе данных. Данный паттерн обладает большей гибкостью и большим разнообразием возможных конфигураций, нежели **ActiveRecord**; кроме этого, отделение данных от их хранилища приводит к созданию более удобных для восприятия и простых в сопровождении моделей. Однако реализовать паттерн **DataMapper** сложнее, чем **ActiveRecord**.

Как правильно выбрать паттерн? Простого ответа на этот вопрос не существует, потому что он равносилен вопросу о том, какой сорт мороженого лучше. Паттерн **ActiveRecord**, по определению, предполагает, что для хранения данных вы исполь-

зуете базу данных, а модели, основанные на этом паттерне, тесно связаны с ее структурой. Такой сценарий подойдет для маленьких проектов или для проектов, в которых основная функциональность по большей части соответствует стандартному набору команд SELECT, INSERT, UPDATE и DELETE. Паттерн DataMapper позволяет явно разделить бизнес-правила уровня приложения и хранилище данных, и именно сама модель, а не лежащая в основе база данных, используется для определения данных. Как следствие, вы не ограничены только лишь базой данных; класс отображения может столь же легко отображать данные на другие типы хранилищ, такие как обычные файлы, XML или LDAP. Этот паттерн подойдет для проектов со сложными структурами данных и/или пользовательскими форматами для их хранения или же для проектов, основная функциональность которых требует более сложных взаимодействий между сущностями приложения.

Границы модели

Зачастую сложно провести черту между тем, что должно находиться в модели, и тем, что должно находиться в контроллере. Например, для описанного ранее примера с банковскими операциями можно утверждать, что валидация, относящаяся к суммам и местоположениям, должна осуществляться в контроллере, а не в модели. Однако в целом сообщество разработчиков приняло подход «толстая модель, тонкий контроллер», рекомендованный различными гуру, включая Мартина Фоулера (Martin Fowler), Джеймиса Бака (Jamis Buck) и Криса Хартджеса (Chris Hartjes). Этот подход предлагает по возможности размещать бизнес-логику в моделях, а не в контроллерах, и обладает рядом преимуществ:

- Инкапсулируя ключевые бизнес-правила в объектах и методах объектов, пригодных для многократного использования, «толстые» модели уменьшают дублирование и помогают разработчикам придерживаться принципа «Не повторяй самого себя» (Don't Repeat Yourself, DRY). Впоследствии для согласованного применения этих правил в приложении можно использовать цепочки наследования. Кроме того, определение моделей с использованием принципов ООП дает возможность наращивать функциональность базовых моделей по мере необходимости, а также разграничивать открытые и закрытые атрибуты моделей в более крупных приложениях.
- Самодостаточные «толстые» модели, инкапсулирующие все бизнес-правила, необходимые для их корректной работы, являются более переносимыми, чем «тонкие» модели, которые полагаются на то, что за соблюдение бизнес-правил будут отвечать контроллеры приложения. Поскольку основная часть бизнес-логики приложения находится в модели, а не в контроллере, упрощается, например, переход на другой фреймворк, так как можно просто переместить модель в новую систему, и она будет работать как обычно, не требуя существенной модификации.
- Когда большая часть тяжелой работы выполняется «толстыми» моделями, контроллеры обычно содержат лишь несколько строк узкоспециализирован-

ного кода. Это упрощает их чтение и понимание, а также повышает общую производительность (поскольку контроллеры обычно вызываются чаще, чем модели). Помимо этого, упрощается сопровождение кода: например, если вам потребуется изменить бизнес-правила для определенной сущности в приложении, необходимо будет обновить только модель для этой сущности, а не все контроллеры, которые ее используют.

- При подходе «толстая модель, тонкий контроллер» отладка и тестирование становятся более прозрачными, потому что обычно очень легко определить, является ли источником ошибки модель или же контроллер, и исправить эту ошибку, оказав минимальное влияние на остальную часть приложения.

Важно отметить, что в состав Zend_Framework не входит компонент **Zend_Model** или набор стандартных моделей, которые вы могли бы «присоединить» к своему приложению; вам придется создавать собственные модели. Zend Framework включает в себя несколько инструментов, способных помочь в решении этой задачи: например, слой абстракции баз данных **Zend_Db_Adapter** и интерфейс для операций с таблицами **Zend_Db_Table**. Однако вы также можете создать собственные модели для приложения с помощью сторонних средств, таких как **Doctrine** или Propel, и интегрировать их в свое приложение Zend Framework. В сущности, этому и посвящен следующий раздел.

ВОПРОС ЭКСПЕРТУ

В: Является ли «модель» простым набором предварительно подготовленных операторов SQL?

О: Определенно нет. Многие ошибочно полагают, что модели являются просто представлениями таблиц базы данных в виде объектов, содержащих методы, соответствующие SQL-операторам **INSERT**, **SELECT**, **UPDATE** и **DELETE**. На самом деле модели — это нечто большее, чем просто оболочка вокруг языка структурированных запросов (Structured Query Language, SQL). Они могут (и должны) обеспечивать соблюдение специфичных для приложения бизнес-правил, производить вычисления, осуществлять преобразования, применять фильтры и выполнять любые другие требуемые операции с данными. Запомните, что чем интеллектуальнее ваши модели, тем меньше работы остается контроллерам и тем более переносимым и простым в сопровождении является ваше приложение.

Установка Doctrine

Хотя вы, конечно, можете создавать модели для вашего приложения «с нуля», используя входящие в состав Zend Framework компоненты, существует более простой путь: использовать средство объектно-реляционного отображения (Object Relational Mapping, ORM) для автоматической генерации моделей. Такие средства могут автоматически анализировать существующую базу данных и создавать классы моделей, соответствующие содержащимся в ней таблицам, используя либо паттерн **ActiveRecord**, либо паттерн **DataMapper**. После этого не составит труда добавить автоматически сгенерированные модели в приложение и расширить их под конкретные требования.

Для PHP доступно большое количество бесплатных средств ORM с открытым исходным кодом, таких как Propel, Doctrine, RedBean и Qcodo. В этой книге используется средство Doctrine, которое чрезвычайно популярно среди разработчиков по причине простоты использования, гибкости и наличия обширной документации. В состав Doctrine входят слой абстракции баз данных, поддерживающий большинство распространенных систем управления реляционными базами данных (Relational Database Management System, RDBMS), модифицированный вариант SQL — DQL (Doctrine Query Language, язык запросов Doctrine), а также инструменты для автоматической генерации классов моделей на основе существующей схемы базы данных. Эти классы моделей реализуют паттерн **ActiveRecord** и полностью совместимы с объектной моделью PHP 5.3.x.

Чтобы начать работать с Doctrine, загрузите и установите его последнюю версию в ваше окружение разработки. Для этого найдите на официальном веб-сайте Doctrine свежий релиз, а затем распакуйте содержимое загруженного архива во временный каталог:

```
shell> cd /tmp
shell> tar -xzvf Doctrine-XX.tgz
```

В результате у вас должна получиться структура каталогов, похожая на ту, что показана на рис. 4.1. Каталог **lib/** содержит все компоненты Doctrine, а каталог **tests/** — варианты тестирования (test cases).

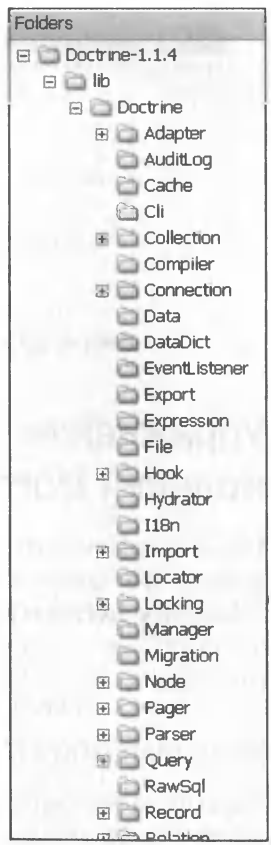


Рис. 4.1. Содержимое архива с релизом Doctrine

Содержимое каталога **lib/** необходимо переместить в расположение, входящее в список «путей поиска» PHP. В системах UNIX/Linux для этого подходит **/usr/local/lib/php** или **/usr/local/share/php**. В Windows можно воспользоваться каталогом установки PHP или PEAR, например, **C:\Program Files\PHP** или **C:\Program Files\PHP\PEAR**. Обратите внимание, что если целевой каталог не входит в список «путей поиска» PHP, вы должны добавить его туда, перед тем как продолжить.

Примеры команд, которые можно использовать для выполнения перечисленных задач:

```
shell> cd Doctrine-XX
shell> mv lib /usr/local/lib/php/Doctrine
```

Теперь вы можете использовать **Doctrine** в сценариях PHP. Чтобы удостовериться в этом, создайте следующий простой сценарий, не забыв заменить параметры доступа на те, что используются в вашей системе:

```
<?php
// загружаем файл с основным классом Doctrine
include_once 'Doctrine/Doctrine.php';
spl_autoload_register(array('Doctrine', 'autoload'));

// создаем менеджер Doctrine
```

```
$manager = Doctrine_Manager::getInstance();

// создаем подключение к базе данных
$conn = Doctrine_Manager::connection(
  'mysql://root@localhost/test', 'doctrine');

// получаем и выводим список баз данных
$databases = $conn->import->listDatabases();
print_r($databases);
?>
```

Сохраните этот сценарий под именем **/tmp/Doctrine-test.php** и выполните его в командной строке с помощью интерфейса командной строки PHP (CLI, command-line interface):

```
shell> cd /tmp/
shell> php ./doctrine-test.php
```

Вы увидите вывод, показанные на рис. 4.2.

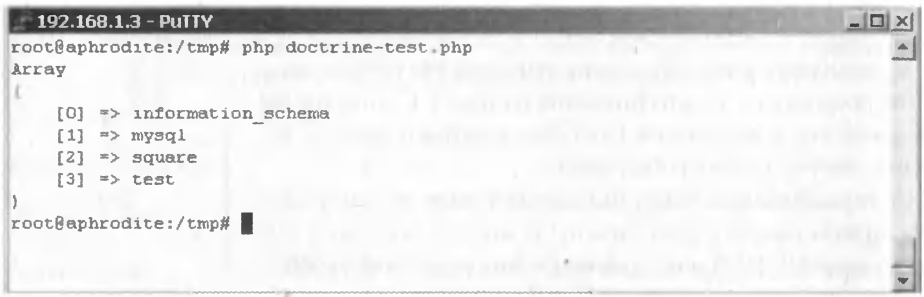


Рис. 4.2. Список доступных баз данных, возвращаемый Doctrine

Упражнение 4.1. Создание и интеграция моделей Doctrine

После того как **Doctrine** будет установлено и готово к работе, вы с его помощью можете приступать к созданию моделей. Эти модели впоследствии можно будет добавить к демонстрационному приложению и настроить таким образом, что они будут автоматически загружаться по мере необходимости. В следующих разделах рассказано, как это сделать.

Инициализация базы данных приложения

Первым шагом будет создание базы данных MySQL для приложения. Для этого перейдите в командную строку, запустите клиент командной строки **MySQL** и создайте новую базу данных, выполнив следующие команды:

```
shell> mysql -u root -p
Enter password: *****
```

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 5.1.30-community MySQL Community Server (GPL)

mysql> CREATE DATABASE square;
Query OK, 0 rows affected (0.00 sec)
mysql> USE square;
Database changed
```

Добавим несколько таблиц к базе данных, это показано в следующем коде:

```
mysql> CREATE TABLE IF NOT EXISTS country (
    -> CountryID INT(11) NOT NULL AUTO_INCREMENT,
    -> CountryName VARCHAR(255) NOT NULL,
    -> PRIMARY KEY (CountryID)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE IF NOT EXISTS grade (
    -> GradeID INT(11) NOT NULL AUTO_INCREMENT,
    -> GradeName VARCHAR(255) NOT NULL,
    -> PRIMARY KEY (GradeID)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE IF NOT EXISTS type (
    -> TypeID INT(11) NOT NULL AUTO_INCREMENT,
    -> TypeName VARCHAR(255) NOT NULL,
    -> PRIMARY KEY (TypeID)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE IF NOT EXISTS item (
    -> RecordID INT(11) NOT NULL AUTO_INCREMENT,
    -> RecordDate DATE NOT NULL,
    -> SellerName VARCHAR(255) NOT NULL,
    -> SellerEmail VARCHAR(255) NOT NULL,
    -> SellerTel VARCHAR(50) DEFAULT NULL,
    -> SellerAddress TEXT,
    -> Title VARCHAR(255) NOT NULL,
    -> `Year` INT(4) NOT NULL,
    -> CountryID INT(4) NOT NULL,
    -> Denomination FLOAT NOT NULL,
    -> TypeID INT(4) NOT NULL,
    -> GradeID INT(4) NOT NULL,
    -> SalePriceMin FLOAT NOT NULL,
    -> SalePriceMax FLOAT NOT NULL,
    -> Description TEXT NOT NULL,
    -> DisplayStatus TINYINT(1) NOT NULL,
    -> DisplayUntil DATE DEFAULT NULL,
    -> PRIMARY KEY (RecordID)
```

```
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE IF NOT EXISTS user (
    -> RecordID INT(4) NOT NULL AUTO_INCREMENT,
    -> Username VARCHAR(10) NOT NULL,
    -> Password TEXT NOT NULL,
    -> PRIMARY KEY (RecordID)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE IF NOT EXISTS log (
    -> RecordID int(11) NOT NULL AUTO_INCREMENT,
    -> LogMessage text NOT NULL,
    -> LogLevel varchar(30) NOT NULL,
    -> LogTime varchar(30) NOT NULL,
    -> Stack text,
    -> Request text,
    -> PRIMARY KEY (RecordID)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.09 sec)
```

В табл. 4.1 приведено краткое описание этих таблиц.

Таблица 4.1. Краткое описание базы данных приложения

Название таблицы	Таблица содержит
item	Список марок, доступных для продажи, а также контактные данные продавца
user	Список авторизованных пользователей
country	Основной список стран
grade	Основной список состояний марок
type	Основной список типов марок
log	Журнал ошибок

ВОПРОС ЭКСПЕРТУ

В: Если для названий сущностей в Zend Framework принято использовать стиль CamelCase с прописной первой буквой, то почему для названий базы данных и таблиц MySQL в этой главе используются только строчные буквы?

О: На уровне файловой системы базы данных и таблицы MySQL представлены каталогами и файлами соответственно. Некоторые файловые системы (например, используемые в Windows) игнорируют различия в регистре букв в именах файлов, а другие (например, используемые в Linux) учитывают их. Поэтому, чтобы упростить перемещение баз данных и таблиц между различными операционными системами, рекомендуется называть все базы данных и таблицы, используя только буквы в нижнем регистре, цифровые символы и символы подчеркивания. Разумеется, стоит также избегать использования в качестве имен баз данных зарезервированные ключевые слова MySQL.

Чтобы лучше понять взаимосвязь между таблицами, посмотрите на диаграмму сущность-связь (entity-relationship) базы данных MySQL, используемой в этой главе (рис. 4.3).

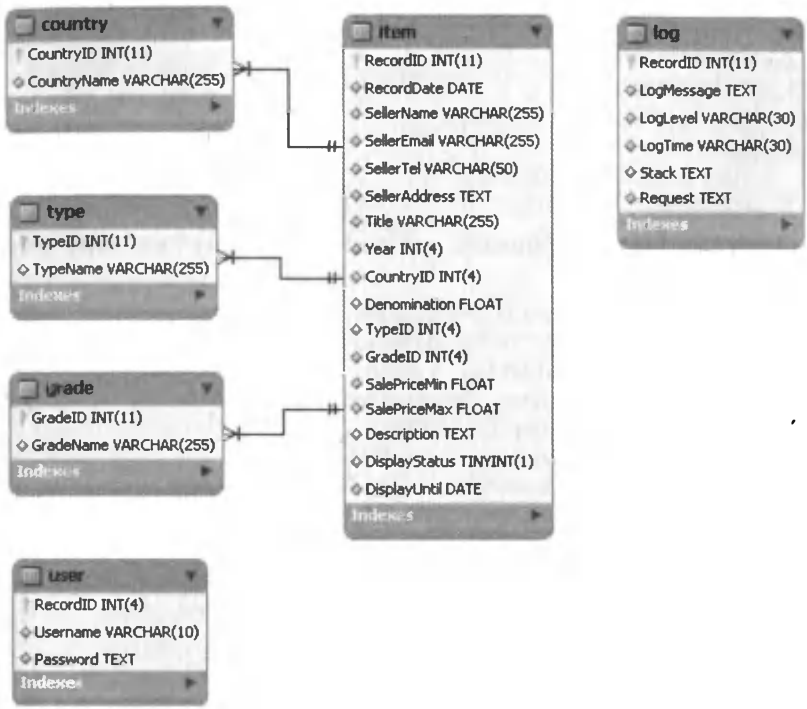


Рис. 4.3. Диаграмма сущность-связь для базы данных приложения

Инициализируем основные таблицы несколькими записями:

```
mysql> INSERT INTO country (CountryID, CountryName) VALUES
    -> (1, 'United States'),
    -> (2, 'United Kingdom'),
    -> (3, 'India'),
    -> (4, 'Singapore'),
    -> (5, 'Germany'),
    -> (6, 'France'),
    -> (7, 'Italy'),
    -> (8, 'Spain'),
    -> (9, 'Hungary');
Query OK, 9 rows affected (0.09 sec)
Records: 9 Duplicates: 0 Warnings: 0

mysql> INSERT INTO grade (GradeID, GradeName) VALUES
    -> (1, 'Very Fine'),
    -> (2, 'Fine'),
    -> (3, 'Good'),
    -> (4, 'Average'),
```

```
    -> (5, 'Poor');
Query OK, 5 rows affected (0.09 sec)
Records: 5 Duplicates: 0 Warnings: 0

mysql> INSERT INTO type (TypeID, TypeName) VALUES
    -> (1, 'Commemorative'),
    -> (2, 'Decorative'),
    -> (3, 'Definitive'),
    -> (4, 'Special'),
    -> (5, 'Other');
Query OK, 5 rows affected (0.09 sec)
Records: 5 Duplicates: 0 Warnings: 0
```

Для проверки вы также можете добавить в таблицу item пару фиктивных записей.

```
mysql> INSERT INTO item (RecordID, RecordDate, SellerName,
    -> SellerEmail, SellerTel, SellerAddress, Title,
    -> Year, CountryID, Denomination, TypeID, GradeID,
    -> SalePriceMin, SalePriceMax, Description,
    -> DisplayStatus, DisplayUntil) VALUES
    -> (1, '2009-12-06', 'John Doe', 'john@example.com',
    -> '+123456789102', '12 Green House, Red Road, Blue City',
    -> 'Himalayas - Silver Jubilee', 1958, 3, 5.00, 1, 2, 10, 15,
    -> 'Silver jubilee issue. Aerial view of snow-capped.
    -> Himalayan mountains. Horizontal orange stripe across
    -> top margin. Excellent condition, no marks.', 0, NULL);
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO item (RecordID, RecordDate, SellerName,
    -> SellerEmail, SellerTel, SellerAddress, Title,
    -> Year, CountryID, Denomination, TypeID, GradeID,
    -> SalePriceMin, SalePriceMax, Description,
    -> DisplayStatus, DisplayUntil) VALUES
    -> (2, '2009-10-05', 'Susan Doe', 'susan@example.com',
    -> '+198765432198', '1 Tiger Place, Animal City 648392',
    -> 'Britain - WWII Fighter', 1966, 2, 1.00, 1, 4, 1.00, 2.00,
    -> 'WWII Fighter Plane overlaid on blue sky. Cancelled',
    -> 0, NULL);
Query OK, 1 row affected (0.02 sec)
```

В целях безопасности создайте отдельную учетную запись в MySQL и разрешите ей доступ только к созданной выше базе данных. Сделайте это с помощью следующей команды:

```
mysql> GRANT ALL ON square.* TO square@localhost IDENTIFIED BY 'square';
Query OK, 1 row affected (0.00 sec)
```

Теперь ваша база данных настроена и вы готовы приступить к созданию моделей на ее основе!

Генерация моделей Doctrine

В состав Doctrine входит мощный генератор моделей, который может «читать» существующую базу данных и создавать на ее основе набор классов моделей. Чтобы увидеть его в действии, создайте и запустите следующий сценарий PHP:

```
<?php
// загружаем файл с основным классом Doctrine
include_once 'Doctrine.php';
spl_autoload_register(array('Doctrine', 'autoload'));

// создаем менеджер Doctrine
$manager = Doctrine_Manager::getInstance();

// создаем подключение к базе данных
$conn = Doctrine_Manager::connection(
  'mysql://square:square@localhost/square', 'doctrine');

// автоматически генерируем модели
Doctrine::generateModelsFromDb('/tmp/models',
  array('doctrine'),
  array('classPrefix' => 'Square_Model_')
);
?>
```

Этот сценарий устанавливает соединение с базой данных, созданной в предыдущем разделе, а затем использует метод **generateModelsFromDb()**, чтобы динамически сгенерировать модели, соответствующие таблицам базы. Эти модели будут сохранены в каталоге, указанном в качестве первого аргумента метода. Также обратите внимание, что в массиве, передаваемом в качестве третьего аргумента, содержится параметр, который указывает Doctrine на то, что к каждому имени класса модели должен добавляться определенный пользователем префикс. Полезно иметь модели, соответствующие требованиям подсистемы автоматической загрузки Zend Framework.

Выполните этот сценарий в командной строке с помощью PHP CLI:

```
shell> cd /tmp
shell> php ./doctrine-models-generate.php
```

Если все пойдет хорошо, Doctrine создаст в указанном каталоге набор моделей. Вы должны увидеть что-то наподобие рис. 4.4.

Теперь созданные модели следует скопировать в каталог приложения. Для этого перейдите в каталог **$APP_DIR** и выполните следующие команды:

```
shell> mkdir library/Square/Model
shell> cp /tmp/models/* library/Square/Model
shell> cp /tmp/models/generated/* library/Square/Model
```

По умолчанию генератор моделей Doctrine использует название класса модели в качестве основы для имени соответствующего файла. Например, класс модели с именем Square_Model_Item будет сохранен в файл Square_Model_Item.php. К сожалению, такой формат не подходит для автозагрузчика Zend Framework, который

ожидает, что класс с именем Square_Model_Item будет сохранен как Square/Model/Item.php. Для решения этой проблемы потребуется вручную переименовать каждый созданный файл с классом модели, удалив автоматически добавленный Doctrine префикс. Пример:

```
shell> cd library/Square/Model
shell> mv Square_Model_BaseCountry.php BaseCountry.php
```

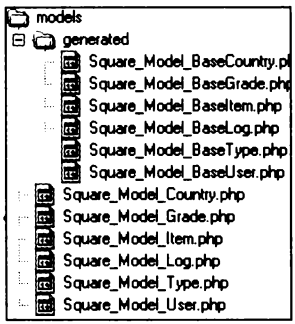


Рис. 4.4. Автоматически сгенерированные модели Doctrine в их первоначальном виде

СОВЕТ

Если вы используете Doctrine v1.2, вы можете передать методу generateModelsFromDb() дополнительный параметр 'classPrefixFiles'. Он определяет, требуется ли добавлять к именам файлов моделей префикс в виде имени класса. Установка значения 'false' уменьшит количество работы по интеграции моделей Doctrine в ваше приложение, поскольку больше не понадобится переименовывать каждый файл модели для соответствия соглашениям Zend Framework.

Заметьте, что Doctrine отображает каждую таблицу базы данных на два класса моделей: базовый класс и производный класс. Базовый класс расширяет класс Doctrine_Record, содержит определение таблицы и предоставляет (через наследование) все методы для реализации распространенных операций с базой данных. Производный класс — это класс-«заглушка», унаследованный от базового класса, который служит в качестве контейнера для любых пользовательских методов, которые вы захотите добавить в модель.

Установка отношений между моделями

Итак, хотя Doctrine и может создавать модели, соответствующие отдельным таблицам и предоставляющие методы для выполнения распространенных операций над ними, у него нет алгоритмов для автоматического обнаружения отношений между таблицами. Поскольку на практике эти отношения зачастую чрезвычайно важны, разработчик приложения должен определить их, тем самым связав модели вместе.

Doctrine предоставляет возможность создавать все типы отношений, доступные в RDBMS: один-к-одному, один-ко-многим, многие-ко-многим и ссылка на саму себя, — используя лишь два метода: hasOne() и hasMany(). Оба этих метода принима-

ют два аргумента: имя другой модели, участвующей в отношении, и массив опций, определяющих параметры отношения. Чтобы гарантировать, что эти отношения будут автоматически загружаться при создании экземпляра модели, указанные методы необходимо использовать в методе **setUp()**.

Чтобы разобраться, как это работает, вернитесь на несколько страниц назад к рис. 4.3, на котором показаны отношения между различными таблицами базы данных. Из рис. 4.3 понятно, что существует отношение 1:n между основными таблицами **country**, **grade** и **type**, с одной стороны, и таблицей **item** — с другой. Это отношение использует поля внешних ключей **item.CountryID**, **item.GradeID** и **item.TypeID** и может быть выражено в терминах **Doctrine** путем добавления в класс модели **Item** следующего кода:

```
<?php
class Square_Model_Item extends Square_Model_BaseItem
{
    public function setUp()
    {
        $this->hasOne('Square_Model_Grade', array(
                'local' => 'GradeID',
                'foreign' => 'GradeID'
            )
        );

        $this->hasOne('Square_Model_Country', array(
                'local' => 'CountryID',
                'foreign' => 'CountryID'
            )
        );

        $this->hasOne('Square_Model_Type', array(
                'local' => 'TypeID',
                'foreign' => 'TypeID'
            )
        );
    }
}
```

В данном случае метод **setUp()** берет на себя задачу автоматического определения отношений внешнего ключа между моделью **Item** и моделями **Country**, **Grade** и **Type**. Это может оказаться удобным при объединении таблиц в запросе **Doctrine** (как вы вскоре увидите).

Автозагрузка Doctrine

Все, что нам осталось сделать, — это обеспечить взаимодействие между приложением Zend Framework и **Doctrine**. Для этого существуют несколько способов, но рекомендуется производить инициализацию **Doctrine** в файле начальной загрузки приложения и настройку его для «отложенной загрузки» (lazy load) моделей по мере необходимости. Приверженцами данного подхода являются ведущие разработчики

Zend Framework, такие как Мэтью Вайер О'Финни (Matthew Weier O'Phinney) и Эрик Леклерк (Eric LeClerc). Ссылки на их работы вы найдете в конце главы.

В первую очередь необходимо решить, будете ли вы включать библиотеки **Doctrine** в состав вашего приложения или предоставите пользователям возможность самостоятельно загрузить и установить их. Чтобы разобраться с плюсами и минусами обоих вариантов, вернитесь к главе 1, где требовалось принять аналогичное решение для библиотек Zend Framework. Основываясь на нашем предыдущем решении, предположим, что библиотеки **Doctrine** будут поставляться вместе с приложением. Поэтому скопируйте содержимое каталога **lib/** из архива **Doctrine** в каталог **$APP_DIR/library/Doctrine** с помощью следующей команды:

```
shell> cp -R /usr/local/lib/php/Doctrine library/
```

Следующим шагом будет добавление в конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**, имени источника данных (DSN, Data Source Name), которое будет использоваться **Doctrine** для подключения к созданной ранее базе данных MySQL. Чтобы обновить конфигурационный файл, откройте его в текстовом редакторе и добавьте в раздел **[production]** следующие строки:

```
doctrine.dsn = "mysql://square.square@localhost/square"
```

В качестве последнего шага необходимо инициализировать **Doctrine**, добавив нужный код в файл начальной загрузки приложения. Как следует из названия, этот компонент автоматически вызывается при каждом запросе для осуществления «начальной загрузки» (bootstrap) требуемых ресурсов приложения. Он определен как класс, расширяющий класс **Zend_Application_Bootstrap_Bootstrap**, и находится в файле **$APP_DIR/application/Bootstrap.php**.

Откройте этот файл в текстовом редакторе и добавьте в него новый метод **_initDoctrine()**, как показано ниже:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initDoctrine()
    {
        require_once 'Doctrine/Doctrine.php';
        $this->getApplication()
            ->getAutoloader()
            ->pushAutoloader(array('Doctrine', 'autoload'), 'Doctrine');

       $manager = Doctrine_Manager::getInstance();
       $manager->setAttribute(
           Doctrine::ATTR_MODEL_LOADING,
           Doctrine::MODEL_LOADING_CONSERVATIVE
        );

        $config = $this->getOption('doctrine');
        $conn = Doctrine_Manager::connection($config['dsn'], 'doctrine');
        return $conn;
    }
}
```

В системе начальной загрузки Zend Framework любой защищенный метод, который принадлежит классу Bootstrap и содержит в названии префикс _init*, считается ресурсным методом и автоматически выполняется начальным загрузчиком. В приведенном листинге определен ресурсный метод _initDoctrine(), который начинается с включения основного класса Doctrine и передачи автозагрузчику информации о том, как загружать классы Doctrine. После этого метод создает экземпляр класса Doctrine_Manager, настраивает отложенную загрузку Doctrine, читает из конфигурационного файла приложения DSN и устанавливает соединение с базой данных.

Так как этот метод будет автоматически выполняться при каждом запросе, модели Doctrine можно будет использовать в любом контроллере приложения. Теперь Doctrine можно использовать!

Работа с моделями Doctrine

Перед тем как продолжить, сделаем небольшое отступление, касающееся устройства моделей Doctrine. Как говорилось ранее, модели Doctrine реализуют паттерн ActiveRecord, в соответствии с которым каждый экземпляр класса модели представляет собой запись в базе данных, а каждое свойство экземпляра — поле этой записи. Модели расширяют базовый класс Doctrine_Record, который предоставляет несколько методов, призванных упростить работу, связанную с добавлением, изменением, удалением и поиском записей. В следующих разделах эти аспекты описаны более подробно.

Получение записей

Существует несколько различных способов получать записи с помощью модели Doctrine. Если у вас есть первичный ключ записи, используйте метод find():

```
<?php
$member = Doctrine::getTable('Member')->find(1);
var_dump($member->toArray());
?>
```

Если вы хотите получить все доступные записи, воспользуйтесь методом findAll():

```
<?php
$members = Doctrine::getTable('Members')->findAll();
var_dump($members->toArray());
?>
```

Doctrine содержит собственный вариант SQL, который называется *Doctrine Query Language* (DQL, язык запросов Doctrine) и предоставляет удобный интерфейс для создания и выполнения запросов к базе данных. Он обеспечивает альтернативную и более гибкую возможность получения записей из базы. Рассмотрим следующий простой пример, который получает все записи о клиентах в виде массива:

```
<?php
$q = Doctrine_Query::create()
```

```
            ->from('Member m');
$result = $q->fetchArray();
?>
```

Использование фильтров

Класс **Doctrine_Query** имеет текучий интерфейс. Это означает, что очень просто добавлять к приведенному запросу новые уровни. Предположим, к примеру, что вы хотите получить записи, соответствующие только активным учетным записям клиентов. Это условие можно добавить в запрос с помощью метода **where()**:

```
<?php
$q = Doctrine_Query::create()
        ->from('Member m')
        ->where('m.Status = ?', 1);
$result = $q->fetchArray();
?>
```

Точно так же с помощью метода **addWhere()** к запросу можно добавлять несколько фильтров. Ниже приведен пример, который получает записи об активных клиентах, проживающих в Великобритании:

```
<?php
$q = Doctrine_Query::create()
        ->from('Member m')
        ->where('m.Status = ?', 1)
        ->addWhere('m.Location = ?', 'UK');
$result = $q->fetchArray();
?>
```

Группирование и упорядочивание результатов

С помощью методов **groupBy()** и **orderBy()** вы можете группировать и упорядочивать результирующий список записей. Пример группирования и упорядочивания клиентов по стране:

```
<?php
$q = Doctrine_Query::create()
        ->from('Member m')
        ->where('m.Status = ?', 1)
        ->groupBy('m.Location')
        ->orderBy('m.Location DESC');
$result = $q->fetchArray();
?>
```

Объединение таблиц

DQL также можно использовать для объединения таблиц с использованием методов **leftJoin()** и **innerJoin()**. Пример создания левого объединения:

```
<?php
$q = Doctrine_Query::create()
```

```
        ->from('Member m')
        ->leftJoin('m.Country c')
        ->leftJoin('m.OrgType o')
        ->where('o.DisplayStatus = 1');
$result = $q->execute();
?>
```

И пример внутреннего объединения:

```
<?php
$q = Doctrine_Query::create()
        ->from('Member m')
        ->innerJoin('m.Country c');
$result = $q->execute();
?>
```

Поля, использующиеся при объединении, устанавливаются Doctrine автоматически на основании отношений, определенных в методе setUp() исходной модели.

СОВЕТ

При использовании DQL иногда требуется проверить автоматически созданный код определенного запроса. Doctrine дает возможность сделать это с помощью метода getSql(), который возвращает фактический SQL-код запроса для последующего анализа.

Добавление, обновление и удаление записей

Добавление новой записи в базу данных через модель Doctrine — крайне простая процедура: вам нужно всего лишь создать экземпляр класса модели, установить значения полей через его свойства и вызвать его метод save(). Пример:

```
<?php
$member = new Member;
$member->FirstName = 'Jack';
$member->LastName = 'Frost';
$member->Email = 'jack@example.com';
$member->JoinDate = '2009-11-11';
$member->Status = 1;
$id = $member->save();
?>
```

ВОПРОС ЭКСПЕРТУ

В: Поддерживает ли DQL правые объединения?

О: Нет. В данный момент DQL поддерживает только левые и внутренние объединения. Но это не должно вас останавливать, так как левые и правые объединения взаимозаменяемы, все зависит от того, с какой стороны вы находитесь. Для иллюстрации рассмотрим два следующих эквивалентных запроса:

SELECT * FROM c LEFT JOIN a USING (id);

SELECT * FROM a RIGHT JOIN c USING (id);

Для обновления существующей записи получите ее с помощью метода **find()** или DQL-запроса, обновите нужные свойства и сохраните запись в базу данных посредством метода **save()**. Пример:

```
<?php
$member = Doctrine::getTable('Member')->find(1);
$member->FirstName = 'John';
$member->Status = 2;
$member->save();
?>
```

Для более точного контроля вы можете обновлять записи, используя метод **update()**, принимающий DQL-запрос. Приведенный ниже пример эквивалентен предыдущему:

```
<?php
$q = Doctrine_Query::create()
      ->update('Member m')
      ->set('m.FirstName', '?', 'John')
      ->set('m.Status', '?', '2')
      ->addWhere('m.RecordID = ?', 1);
$q->execute();
?>
```

Аналогичным образом вы можете удалять записи, либо вызывая метод **delete()** модели, либо используя DQL-запрос. В следующем листинге показаны оба этих подхода:

```
<?php
// использование методов модели
$member = Doctrine::getTable('Member')->find(11);
$member->delete();

// использование DQL
$q = Doctrine_Query::create()
      ->delete('Member m')
      ->addWhere('m.RecordID = ?', 11);
$q->execute();
?>
```

Хотя полное рассмотрение пакета **Doctrine** выходит за рамки этой книги, приведенных примеров достаточно, чтобы дать вам некоторое представление об основах операций с моделями и сформировать базис для понимания материала, представленного в следующих разделах. За более подробной информацией и дополнительными примерами обратитесь к ссылкам, приведенным в конце этой главы.

Упражнение 4.2. Получение записей из базы данных

Теперь, когда вы познакомились с работой моделей Doctrine, попробуем применить эти знания на практике. В следующем разделе мы будем использовать модели для получения из базы данных определенных списков марок, их форматирования и отображения в контексте демонстрационного приложения SQUARE.

Создание нового модуля

До настоящего момента все примеры, которые вы видели, создавались в модуле default. Но одно из преимуществ использования модульной структуры каталогов заключается в том, что она позволяет вам группировать логически связанные контроллеры и тем самым создавать иерархию кода, которая лучше организована и более проста в сопровождении. Поэтому создайте новый модуль с именем catalog, который будет содержать все контроллеры и представления, относящиеся к добавлению, получению, поиску и модификации каталога продаваемых марок.

Для создания нового модуля catalog перейдите в каталог $APP_DIR/application/ и выполните следующие команды:

```
shell> mkdir modules/catalog
shell> mkdir modules/catalog/controllers
shell> mkdir modules/catalog/views
shell> mkdir modules/catalog/views/scripts
```

Определение пользовательского маршрута

Теперь давайте определим маршрут для URL, связанных с отображением. Для простоты будем полагать, что все эти URL имеют форму /catalog/item/display/xx, где *xx* — переменная, указывающая на идентификатор записи. Чтобы создать пользовательский маршрут для обработки таких URL, добавьте в конфигурационный файл приложения, $APP_DIR/application/configs/application.ini, следующее определение маршрута:

```
resources.router.routes.catalog-display.route = /catalog/item/display/:id
resources.router.routes.catalog-display.defaults.module = catalog
resources.router.routes.catalog-display.defaults.controller = item
resources.router.routes.catalog-display.defaults.action = display
```

Определение контроллера

Теперь определим контроллер и действие, соответствующие определению маршрута из предыдущего раздела. Согласно общепринятому правилу, контроллер будет размещен в файле $APP_DIR/application/modules/catalog/controllers/ItemController.php. Пример:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
```

```
{
  public function init()
{
}

// действие для отображения элемента каталога
public function displayAction()
{
  // устанавливаем фильтры и валидаторы для входных данных,
  // полученных в запросе GET
$filters = array(
    'id' => array('HtmlEntities', 'StripTags', 'StringTrim')
  );
  $validators = array(
    'id' => array('NotEmpty', 'Int')
  );

  // проверяем корректность входных данных
  // получаем запрошенную запись
  // добавляем ее к представлению
  $input = new Zend_Filter_Input($filters, $validators);
  $input->setData($this->getRequest()->getParams());
  if ($input->isValid()) {
    $q = Doctrine_Query::create()
            ->from('Square_Model_Item i')
            ->leftJoin('i.Square_Model_Country c')
            ->leftJoin('i.Square_Model_Grade g')
            ->leftJoin('i.Square_Model_Type t')
            ->where('i.RecordID = ?', $input->id);

      $result = $q->fetchArray();
      if (count($result) == 1) {
        $this->view->item = $result[0];
      } else {
        throw new Zend_Controller_Action_Exception('Page not found', 404);
      }
    } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }
}
```

ПРИМЕЧАНИЕ

Для предотвращения коллизий объектов и переменных в Zend Framework каждый модуль (кроме модуля default) имеет собственное пространство имен. Это можно увидеть в классе Catalog_ItemController, где для создания пользовательского пространства имен к имени каждого модуля добавляется префикс в виде имени контроллера.

В приведенном примере метод displayAction() сначала получает значение входной переменной $_GET['id'], а затем вставляет это значение в запрос Doctrine, который пытается найти в базе данных подходящую запись. Если в качестве результата запрос возвращает единственную запись, результат отправляется в представление в виде ассоциативного массива. Если подходящие записи отсутствуют или их несколько, генерируется исключение 404, передаваемое стандартному обработчику ошибок, который формирует страницу с сообщением об ошибке и отображает ее пользователю.

В листинге также используется новый компонент, Zend_Filter_Input. Как следует из названия, он предоставляет альтернативный подход к фильтрации и валидации ввода и очень полезен в ситуациях, когда пользовательский ввод поступает не через форму (как в примере, где идентификатор записи передается в виде параметра запроса URL). С Zend_Filter_Input можно использовать все фильтры и валидаторы, о которых говорилось в главе 3.

Как это работает? Конструктор класса Zend_Filter_Input принимает три аргумента — массив фильтров, массив валидаторов и массив входных данных, которые будут проверяться, — и применяет содержимое первых двух аргументов к третьему. Вы также можете задать массив входных данных отдельно, используя метод setData(). Например, в предыдущем листинге переменная $_GET['id'] сначала обрабатывается фильтрами HTMLEntities, StripTags и StringTrim, а затем проверяется, является ли она целым числом. Как и в случае с Zend_Form, существует метод isValid(), который возвращает логическое значение true или false в зависимости от того, валидны входные данные или нет; метод можно использовать в качестве условной «оболочки» бизнес-логики действия.

Определение представления

Предполагая, что идентификатор запрашиваемой записи корректен и подходящая запись найдена в базе данных, контроллер формирует страницу на основе соответствующего представления, которое, как правило, должно храниться в файле $APP_DIR/application/modules/catalog/views/scripts/item/display.phtml. Пример:

```
<h2>View Item</h2>
<h3>
  FOR SALE:
  <?php echo $this->escape($this->item['Title']); ?> -
  <?php echo $this->escape($this->item['Year']); ?> -
  <?php echo $this->escape($this->item['Square_Model_Grade']
['GradeName']); ?>
</h3>

<div id="container">
  <div id="record">
    <table>
      <tr>
```

```
    <td class="key">Title:</td>
    <td class="value">
        <?php echo $this->escape($this->item['Title']); ?>
    </td>
  </tr>
  <tr>
    <td class="key">Type:</td>
    <td class="value">
      <?php echo $this->escape(
        $this->item['Square_Model_Type']['TypeName']); ?>
      </td>
  </tr>
  <tr>
    <td class="key">Year:</td>
    <td class="value">
        <?php echo $this->escape($this->item['Year']); ?>
    </td>
  </tr>
  <tr>
    <td class="key">Country:</td>
    <td class="value">
      <?php echo $this->escape(
        $this->item['Square_Model_Country']['CountryName']); ?>
    </td>
  </tr>
  <tr>
    <td class="key">Denomination:</td>
    <td class="value">
        <?php echo $this->escape(
         sprintf('%01.2f', $this->item['Denomination'])); ?>
     </td>
  </tr>
  <tr>
      <td class="key">Grade:</td>
      <td class="value">
        <?php echo $this->escape(
          $this->item['Square_Model_Grade']['GradeName']); ?>
     </td>
   </tr>
   <tr>
      <td class="key">Sale price:</td>
      <td class="value">
        $<?php echo $this->escape($this->item['SalePriceMin']); ?> -
        $<?php echo $this->escape($this->item['SalePriceMax']); ?>
      </td>
    </tr>
    <tr>
      <td class="key">Description:</td>
      <td class="value">
        <?php echo $this->escape($this->item['Description']); ?>
      </td>
```

```
      </tr>
    </table>
  </div>
</div>
```

Здесь нет ничего сложного. Этот сценарий представления просто изменяет формат записи, полученной моделью на предыдущем этапе, и отображает запись в удобочитаемом виде. Обратите внимание, что в коде используется метод **escape()**, который автоматически экранирует вывод представления перед отображением его пользователю.

Чтобы посмотреть сценарий в действии, откройте URL **http://square.localhost/catalog/item/display/1** в вашем браузере. Если все работает правильно, контроллер получит из базы данных запись с идентификатором **#1** (как вы помните, мы вручную добавили эту запись при инициализации базы данных), передаст ее представлению и сформирует страницу. Результат должен быть похож на тот, что представлен на рис. 4.5.

View Item

FOR SALE: Himalayas - Silver Jubilee - 1958 - Fine

Title:	Himalayas - Silver Jubilee
Type:	Commemorative
Year:	1958
Country:	India
Denomination:	5.00
Grade:	Fine
Sale price:	$10 - $15
Description:	Silver jubilee issue. Aerial view of snow-capped Himalayan mountains. Horizontal orange stripe across top margin. Excellent condition, no marks.

Рис. 4.5. Результат успешного получения записи базы данных

Для проверки вы можете попробовать запросить тот же URL, но с указанием некорректного или несуществующего идентификатора. Вы должны увидеть ошибку «Page not found» (страница не найдена) или «Invalid input» (некорректный ввод) (рис. 4.6).

ВОПРОС ЭКСПЕРТУ

В: Зачем экранировать вывод перед его отображением?

О: Существует общепринятое правило, в соответствии с которым вы не должны доверять каким бы то ни было данным, полученным из внешнего источника. Злоумышленник всегда может внедрить в эти данные вредоносное содержимое, и если вы воспользуетесь ими без предварительной очистки, то можете подвергнуть риску своих пользователей. Распространенным примером такого типа атаки является «межсайтовый скриптинг» (cross-site scripting), при котором атакующий может получить доступ к конфиденциальным пользовательским данным, добавив к вашим веб-страницам вредоносный код на JavaScript или код HTML-формы. Поэтому всегда подвергайте вывод процедуре очистки, прежде чем отображать его пользователю.

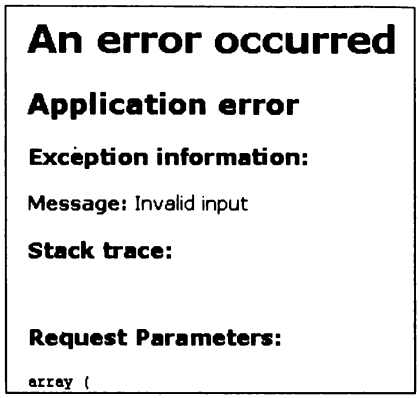


Рис. 4.6. Результат неудачной попытки получения записи

Выводы

Тогда как предыдущие главы посвящались представлениям и контроллерам, основной темой этой главы были модели: что они собой представляют, как работают и какую роль играют в приложении, основанном на MVC. Хотя в состав Zend Framework не входит отдельный компонент для моделей, довольно легко создать свои собственные модели или интегрировать в приложение Zend Framework сторонние, например сгенерированные пакетом ORM Doctrine. В этой главе был продемонстрирован процесс и показано, как создавать модели Doctrine и настраивать их для работы в контексте приложения Zend Framework. В ней также был представлен компонент Zend_Filter_Input, который является полезным средством для фильтрации и валидации ввода по мере необходимости, а также класс Bootstrap, предоставляющий инфраструктуру для инициализации ресурсов приложения на этапе выполнения.

Демонстрационное приложение SQUARE тоже стало значительно умнее. У него есть база данных для постоянного хранения информации и контроллер, который может взаимодействовать с базой для получения и отображения записей. Что более важно, в приложении теперь присутствует набор надежных и расширяемых моделей. Они не только упростят доступ к данным и управление ими, но также обеспечат основу для усовершенствованной функциональности, которая обсуждается в следующих главах этой книги.

Если вы хотите получить больше информации по изложенным в этой главе темам, вам будут полезны следующие ссылки:

- Официальный веб-сайт и руководство Doctrine: http://www.doctrine-project.org/ и http://www.doctrine-project.org/documentation/manual/1_1/en.
- Введение в модели Doctrine:http://www.doctrine-project.org/documentation/manual/1_1/en/introduction-to-models.

- Информация, касающаяся представления отношений в базе данных в моделях Doctrine: http://www.doctrine-project.org/documentation/manual/1_1/en/defining-models.
- Примеры запросов с использованием Doctrine: http://www.doctrine-project.org/documentation/manual/1_0/en/working-with-models.
- Информация о паттерне ActiveRecord на страницах Википедии: http://en.wikipedia.org/wiki/Active_record_pattern.
- Информация об объектно-реляционном отображении на страницах Википедии: http://en.wikipedia.org/wiki/Object-relational_mapping.
- Информация об атаках CSRF и XSS на страницах Википедии: http://en.wikipedia.org/wiki/Cross-site_request_forgery и http://en.wikipedia.org/wiki/Cross-site_scripting.
- Компонент Zend_Filter_Input: http://framework.zend.com/manual/en/zend.filter.input.html.
- Класс Bootstrap: http://framework.zend.com/manual/en/zend.application.theory-of-operation.html.
- Паттерн DataMapper (Мартин Фоулер (Martin Fowler)): http://martinfowler.com/eaaCatalog/dataMapper.html.
- Anemic Domain Model (Мартин Фоулер (Martin Fowler)): http://martinfowler.com/bliki/AnemicDomainModel.html.
- Мысли о подходе «толстые модели, тонкие контроллеры» (Джеймис Бак (Jamis Buck) и Крис Хартджес (Chris Hartjes)): http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model и http://www.littlehart.net/atthekeyboard/2007/04/27/fat-models-skinny-controllers/.
- Автозагрузка Doctrine и моделей Doctrine в контексте приложения Zend Framework (Мэтью Вайер О'Финни (Matthew Weier O'Phinney) и Эрик Леклерк (Eric Leclerc)): http://weierophinney.net/matthew/archives/220-Autoloading-Doctrine-and-Doctrineentities-from-Zend-Framework.html и http://www.danceric.net/2009/06/06/doctrine-orm-and-zend-framework/.

Работа с операциями CRUD

5

Прочитав эту главу, вы:

- познакомитесь с четырьмя операциями CRUD;
- изучите различные техники обработки ввода дат через веб-формы;
- создадите несколько шаблонных макетов и реализуете автоматическое переключение между ними;
- создадите простую систему входа/выхода;
- ограничите доступ к административным действиям с помощью аутентификации пользователей.

Вероятно, вы уже знакомы с акронимом *CRUD*. Он обозначает четыре основные операции — создание (Create), чтение (Read), обновление (Update) и удаление (Delete), — которые можно осуществить с данными приложения. Большинство приложений, в которых реализовано постоянное хранение данных, также предоставляют интерфейс, позволяющий пользователям выполнять эти четыре типа операций с данными. Модели являются интерфейсом к данным приложения, поэтому представление о моделях будет неполным без обсуждения того, как их можно использовать для добавления в приложение функций CRUD.

Данная глава посвящена именно этому аспекту разработки приложений и показывает, как реализовать функциональность CRUD в контексте демонстрационного приложения SQUARE. В ней представлен компонент Zend_Auth и показано, как добавить аутентификацию пользователя для прикладных действий. Также в этой главе приведена дополнительная информация о практическом использовании моделей, контроллеров, представлений и макетов. Итак, давайте приступим!

Упражнение 5.1. Создание записей в базе данных

Вам уже известно, как создавать и обрабатывать веб-формы, используя Zend Framework, а в предыдущей главе вы увидели, насколько просто интегрировать модель Doctrine в контроллер Zend Framework. Теперь давайте соединим все это вместе и создадим веб-форму, позволяющую продавцам вводить информацию о марках и добавлять ее в базу данных приложения.

Определение формы

Для начала определим форму для ввода данных, которая станет основой для записи в каталоге. Ниже приведено определение формы, которые необходимо поместить в файл **$APP_DIR/library/Square/Form/ItemCreate.php**:

```
<?php
class Square_Form_ItemCreate extends Zend_Form
{
  public function init()
  {
  // инициализируем форму
  $this->setAction('/catalog/item/create')
       ->setMethod('post');

  // создаем текстовое поле для ввода имени
  $name = new Zend_Form_Element_Text('SellerName');
  $name->setLabel('Name:')
       ->setOptions(array('size' => '35'))
       ->setRequired(true)
       ->addValidator('Regex', false, array(
           'pattern' => '/^[a-zA-Z]+[A-Za-z\'\-\. ]{1,50}$/'
         ))
       ->addFilter('HtmlEntities')
       ->addFilter('StringTrim');

  // создаем текстовое поле для ввода адреса электронной почты
  $email = new Zend_Form_Element_Text('SellerEmail');
  $email->setLabel('Email address:');
  $email->setOptions(array('size' => '50'))
       ->setRequired(true)
       ->addValidator('EmailAddress', false)
       ->addFilter('HtmlEntities')
       ->addFilter('StringTrim')
       ->addFilter('StringToLower');

  // создаем текстовое поле для ввода номера телефона
  $tel = new Zend_Form_Element_Text('SellerTel');
  $tel->setLabel('Telephone number:');
```

```
$tel->setOptions(array('size' => '50'))
      ->addValidator('StringLength', false, array('min' => 8))
      ->addValidator('Regex', false, array(
          'pattern' => '/^\+[1-9][0-9]{6,30}$/',
          'messages' => array(
            Zend_Validate_Regex::INVALID =>
              '\'%value%\' does not match international number
                format +XXYYZZZZ',
            Zend_Validate_Regex::NOT_MATCH =>
              '\'%value%\' does not match international number
               format +XXYYZZZZ'
          )
        ))
      ->addFilter('HtmlEntities')
      ->addFilter('StringTrim');

// создаем текстовое поле для ввода адреса
$address = new Zend_Form_Element_Textarea('SellerAddress');
$address->setLabel('Postal address:')
      ->setOptions(array('rows' => '6','cols' => '36'))
      ->addFilter('HtmlEntities')
      ->addFilter('StringTrim');

// создаем текстовое поле для ввода названия марки
$title = new Zend_Form_Element_Text('Title');
$title->setLabel('Title:')
      ->setOptions(array('size' => '60'))
      ->setRequired(true)
      ->addFilter('HtmlEntities')
     ->addFilter('StringTrim');

// создаем текстовое поле для ввода года выпуска марки
$year = new Zend_Form_Element_Text('Year');
$year->setLabel('Year:')
      ->setOptions(array('size' => '8', 'length' => '4'))
      ->setRequired(true)
      ->addValidator('Between', false, array(
          'min' => 1700, 'max' => 2015))
      ->addFilter('HtmlEntities')
      ->addFilter('StringTrim');

// создаем список выбора для страны
$country = new Zend_Form_Element_Select('CountryID');
$country->setLabel('Country:')
      ->setRequired(true)
      ->addValidator('Int')
      ->addFilter('HtmlEntities')
      ->addFilter('StringTrim')
      ->addFilter('StringToUpper');
foreach ($this->getCountries() as $c) {
```

```
  $country->addMultiOption($c['CountryID'], $c['CountryName']);
}

// создаем текстовое поле для ввода номинала марки
$denomination = new Zend_Form_Element_Text('Denomination');
$denomination->setLabel('Denomination:')
          ->setOptions(array('size' => '8'))
          ->setRequired(true)
          ->addValidator('Float')
          ->addFilter('HtmlEntities')
          ->addFilter('StringTrim');

// создаем радиокнопку для выбора типа марки
$type = new Zend_Form_Element_Radio('TypeID');
$type->setLabel('Type:')
      ->setRequired(true)
      ->addValidator('Int')
      ->addFilter('HtmlEntities')
      ->addFilter('StringTrim');
foreach ($this->getTypes() as $t) {
  $type->addMultiOption($t['TypeID'], $t['TypeName']);
}
$type->setValue(1);

// создаем список выбора для состояния марки
$grade = new Zend_Form_Element_Select('GradeID');
$grade->setLabel('Grade:')
      ->setRequired(true)
      ->addValidator('Int')
      ->addFilter('HtmlEntities')
      ->addFilter('StringTrim');
foreach ($this->getGrades() as $g) {
  $grade->addMultiOption($g['GradeID'], $g['GradeName']);
};

// создаем текстовое поле для ввода минимальной цены
$priceMin = new Zend_Form_Element_Text('SalePriceMin');
$priceMin->setLabel('Sale price (min):')
          ->setOptions(array('size' => '8'))
          ->setRequired(true)
          ->addValidator('Float')
          ->addFilter('HtmlEntities')
         ->addFilter('StringTrim');

// создаем текстовое поле для ввода максимальной цены
$priceMax = new Zend_Form_Element_Text('SalePriceMax');
$priceMax->setLabel('Sale price (max):')
      ->setOptions(array('size' => '8'))
      ->setRequired(true)
      ->addValidator('Float')
```

```
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

// создаем текстовое поле для ввода описания марки
$notes = new Zend_Form_Element_Textarea('Description');
$notes->setLabel('Description:')
        ->setOptions(array('rows' => '15','cols' => '60'))
        ->setRequired(true)
        ->addFilter('HtmlEntities')
        ->addFilter('StripTags')
        ->addFilter('StringTrim');

// создаем поле CAPTCHA для проверки
    $captcha = new Zend_Form_Element_Captcha('Captcha', array(
      'captcha' => array(
      'captcha' => 'Image',
      'wordLen' => 6,
      'timeout' => 300,
      'width' => 300,
      'height' => 100,
      'imgUrl' => '/captcha',
      'imgDir' => APPLICATION_PATH . '/../public/captcha',
      'font' => APPLICATION_PATH .
              '/../public/fonts/LiberationSansRegular.ttf',
      )
  ));

// создаем кнопку отправки
$submit = new Zend_Form_Element_Submit('submit');
$submit->setLabel('Submit Entry')
        ->setOrder(100)
      ->setOptions(array('class' => 'submit'));

// добавляем элементы к форме
$this->addElement($name)
        ->addElement($email)
        ->addElement($tel)
      ->addElement($address);

// создаем группу отображения для информации о продавце
   $this->addDisplayGroup(
          array('SellerName', 'SellerEmail', 'SellerTel',
  'SellerAddress'), 'contact');
  $this->getDisplayGroup('contact')
        ->setOrder(10)
        ->setLegend('Seller Information');

// добавляем элементы к форме
$this->addElement($title)
```

```
      ->addElement($year)
      ->addElement($country)
      ->addElement($denomination)
      ->addElement($type)
      ->addElement($grade)
      ->addElement($priceMin)
      ->addElement($priceMax)
      ->addElement($notes);

// создаем группу отображения для информации о марке
$this->addDisplayGroup(
  array('Title', 'Year', 'CountryID', 'Denomination',
        'TypeID', 'GradeID', 'SalePriceMin', 'SalePriceMax',
        'Description'), 'item');
$this->getDisplayGroup('item')
      ->setOrder(20)
      ->setLegend('Item Information');

// добавляем элемент к форме
$this->addElement($captcha);

// создаем группу отображения для CAPTCHA
$this->addDisplayGroup(array('Captcha'), 'verification');
$this->getDisplayGroup('verification')
      ->setOrder(30)
      ->setLegend('Verification Code');

// добавляем элемент к форме
  $this->addElement($submit);


public function getCountries() {
  $q = Doctrine_Query::create()
      ->from('Square_Model_Country c');
  return $q->fetchArray();


public function getGrades() {
  $q = Doctrine_Query::create()
       ->from('Square_Model_Grade g');
  return $q->fetchArray();
}

Public function getTypes() {
  $q = Doctrine_Query::create()
      ->from('Square_Model_Type t');
  return $q->fetchArray();
}
```

С большей частью этого кода вы должны быть знакомы по главе 3. Приведенная форма содержит набор полей для ввода текста и списков выбора, практически полностью соответствующих полям таблицы item, созданной ранее в той же главе. Для всех полей используются фильтры ввода HTMLEntities и StringTrim, а многие из них также содержат валидаторы. В частности, обратите внимание на использование валидатора RegEx для проверки имен и номеров телефонов на основе пользовательского шаблона. Для удобства использования элементы формы объединены в группы отображения, как было показано в главе 3, а в некоторых случаях используются еще и пользовательские сообщения об ошибках.

Однако самой интересной частью приведенного кода является использование моделей Doctrine для заполнения списков выбора. Заметьте, что класс содержит три вспомогательных метода, getCountries(), getGrades() и getTypes(); они используют модели Doctrine, созданные в главе 4, для получения списков вариантов из соответствующих основных таблиц. Затем эти варианты добавляются к элементам формы с помощью метода addMultiOption().

Определение контроллеров и представлений

URL для вызова действия, соответствующего определенной в предыдущем разделе форме, имеет вид /catalog/item/create. С учетом стандартных соглашений об именовании, принятых в Zend Framework, он соответствует действию ItemController::createAction в модуле catalog. Этот контроллер был создан в главе 4, и теперь к нему необходимо добавить новое действие. Пример кода:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
{
  public function createAction()
  {
    // генерируем форму ввода
    $form = new Square_Form_ItemCreate;
    $this->view->form = $form;

    // проверяем корректность введенных данных
    // если они корректны, заполняем модель
    // присваиваем некоторым из полей значения по умолчанию
    // сохраняем в базу данных
    if ($this->getRequest()->isPost()) {
      if ($form->isValid($this->getRequest()->getPost())) {
        $item = new Square_Model_Item;
        $item->fromArray($form->getValues());
        $item->RecordDate = date('Y-m-d', mktime());
        $item->DisplayStatus = 0;
        $item->DisplayUntil = null;
        $item->save();
        $id = $item->RecordID;
        $this->_helper->getHelper('FlashMessenger')->addMessage(
           'Your submission has been accepted as item #' . $id .
```

```
            '. A moderator will review it and, if approved,
            it will appear on the site within 48 hours.');
        $this->_redirect('/catalog/item/success');
    }
  }
}

 public function successAction()
 {
   if ($this->_helper->getHelper('FlashMessenger')->getMessages()) {
     $this->view->messages = $this->_helper
       ->getHelper('FlashMessenger')->getMessages();
   } else {
     $this->_redirect('/');
   }
 }
}
```

Если введенные данные корректны, метод **createAction()** создает экземпляр модели **Item** и заполняет его, используя данные, отправленные через веб-форму. Здесь также выполняются все необходимые модификации входных данных: например, присваивание записи статуса отображения «скрытая», а затем запись сохраняется в базу данных путем вызова метода **save()** модели, который формирует и выполняет требуемый запрос **INSERT**. Любые ошибки, возникшие в этом процессе, будут представлены в виде исключений **Doctrine** и переданы стандартному обработчику исключений для их решения.

В случае успешного сохранения записи к вспомогательному классу **FlashMessenger** добавляется соответствующее сообщение и управление передается методу **successAction()**, который формирует страницу на основе представления об успешной операции. Как правило, сценарий этого представления размещается в файле **$APP_DIR/application/modules/catalog/views/scripts/item/success.phtml**. Пример:

```
<h2>Success</h2>
<?php echo implode($this->messages); ?>
```

ВОПРОС ЭКСПЕРТУ

В: Зачем в предыдущем примере вы осуществляете перенаправление на **successAction()**, вместо того чтобы после сохранения записи просто сформировать страницу с сообщением об успехе в методе **createAction()**?

О: Такой подход рекомендуется использовать после любой успешной операции **POST**, чтобы избежать так называемой «проблемы двойной отправки». Предположим, что страница представления об успехе формируется методом **createAction()**; тогда если после формирования страницы пользователь нажмет в браузере кнопку **Обновить/Перезагрузить**, на сервер будут дважды отправлены запросы **POST** с одинаковыми данными. Однако сохраняя сообщение об успехе во вспомогательном классе **FlashMessenger** и осуществляя перенаправление на **successAction()** для формирования страницы, мы полностью избавляемся от данной проблемы; в этом случае перезагрузка страницы или снова вызовет представление об успехе (без повторной отправки данных), или перенаправит пользователя на главную страницу.

Аналогично, сценарий представления размещается в файле **$APP_DIR/application/modules/catalog/views/scripts/item/create.phtml** и выглядит следующим образом:

```
<h2>Add Item</h2>
<?php echo $this->form; ?>
```

Все готово! Чтобы увидеть результат в действии, откройте URL **http://square.localhost/catalog/item/create**, и перед вами должна появиться форма, похожая на ту, что показана на рис. 5.1.

Заполните форму и отправьте ее. Если введенные данные пройдут валидацию, будет создана новая запись, которая добавится в таблицу **item**, а затем представление об успешном завершении операции сформирует страницу. Результат успешной отправки показан на рис. 5.2.

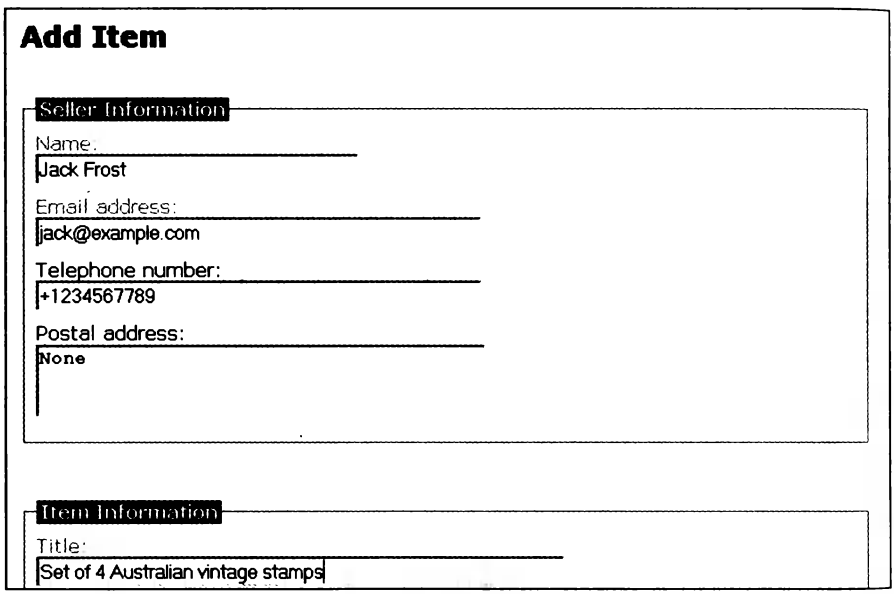


Рис. 5.1. Форма для добавления нового элемента в каталог SQUARE

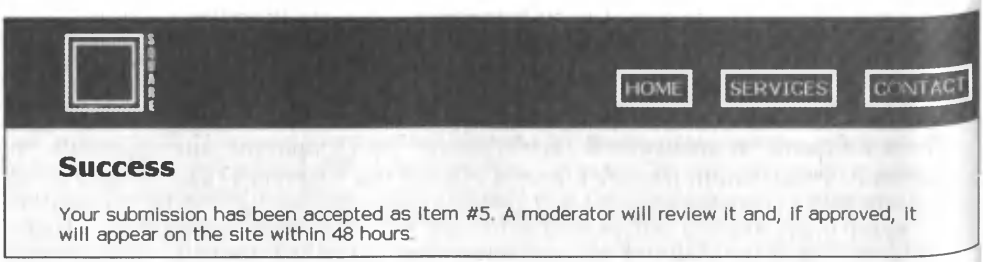


Рис. 5.2. Результат успешного добавления нового элемента в каталог SQUARE

Чтобы убедиться в том, что запись была успешно добавлена, вы можете обратиться к действию **ItemController::displayAction**, созданному в главе 4, по URL

http://square.localhost/catalog/item/display/xx, где *xx* — идентификатор добавленной записи.

Если введенные данные неверны, форма будет сформирована повторно и появятся сообщения об ошибках, информирующее о том, что именно пошло не так. Результат показан на рис. 5.3.

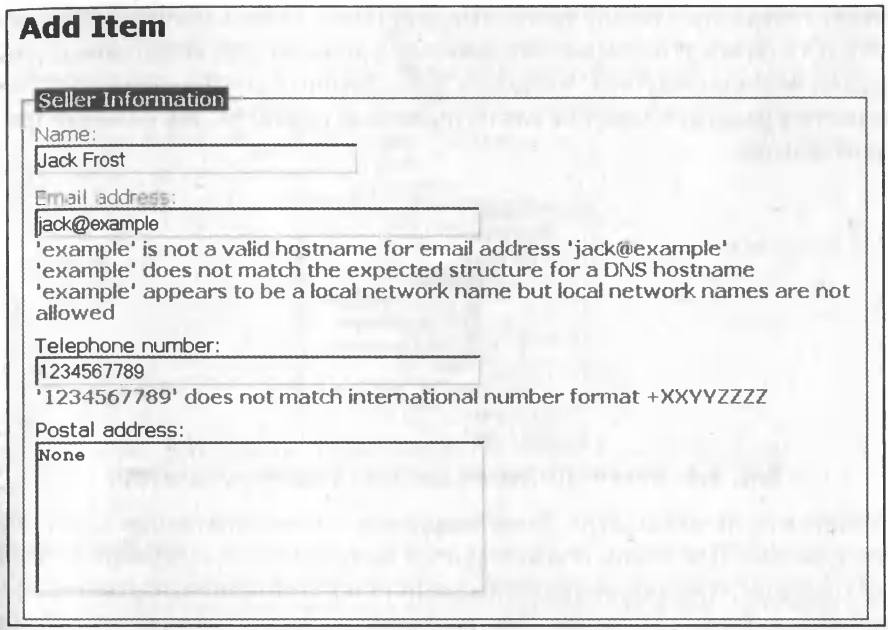


Рис. 5.3. Результат отправки формы с ошибочно введенными значениями

Работа с административными действиями

К настоящему моменту достигнута одна из основных целей демонстрационного приложения SQUARE — дать продавцам возможность загружать списки доступных марок. Однако пока что эти марки недоступны для публичного просмотра, поскольку администраторы приложения должны вручную проверить их и либо подтвердить отображение, либо удалить марки из базы данных. И здесь в дело вступает панель администрирования SQUARE.

Панель администрирования SQUARE — это раздел приложения, зарезервированный для его администраторов. Он предоставляет интерфейс, дающий администраторам возможность просматривать, обновлять и удалять записи каталога, и поэтому является отличным местом для демонстрации того, как можно реализовать оставшиеся операции CRUD. Но перед тем как погрузиться в код, давайте разберемся в способах обработки административных действий в приложении Zend Framework. Здесь существуют три аспекта, подлежащие рассмотрению: *структура*, *маршрутизация* и *макет*.

Структура

В модульном приложении существуют несколько способов структурирования административных действий:

- Создать в модуле **default** единственный контроллер, где будут находиться все административные действия приложения. При таком подходе этот контроллер может содержать административные действия для всех модулей приложения (рис. 5.4). За исключением очень маленьких приложений, использовать данный подход не рекомендуется, поскольку единственный контроллер может быстро вырасти в размерах, повлияв как на производительность, так и на простоту сопровождения.

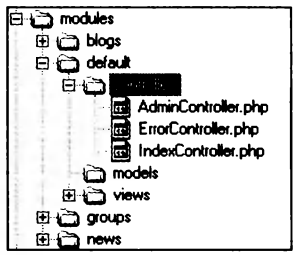


Рис. 5.4. Административные действия в одном контроллере

- Создать отдельный модуль, содержащий все административные действия для приложения. При таком подходе новый модуль может содержать отдельные контроллеры для административных действий приложения, по одному на каждый модуль (рис. 5.5). Это распространенный подход, но он тоже не особо прост в сопровождении, поскольку, как видно из рис. 5.5, в конечном итоге одному контроллеру придется управлять информацией, использующейся несколькими контроллерами в каждом модуле. Данный подход не идеален и при создании независимых модулей, поскольку действия, связанные с каждым модулем, не находятся в одном каталоге файловой системы.

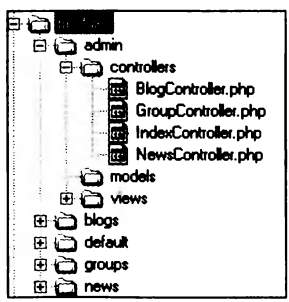


Рис. 5.5. Административные действия в отдельном модуле

- Создать в каждом модуле отдельный контроллер, содержащий административные действия для этого модуля. При таком подходе каждый модуль содержит как минимум два контроллера: один для общедоступных действий и другой

для административных (рис. 5.6). Этот подход рекомендуется использовать по разным причинам: он целесообразен как с логической, так и с физической точки зрения, поскольку все действия для конкретного модуля находятся в одном каталоге, его достаточно просто сопровождать, и он является достаточно гибким.

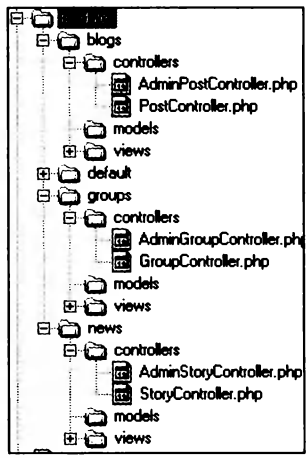


Рис. 5.6. Административные контроллеры в каждом модуле

Маршрутизация

Для сохранения согласованности необходимо привести административные маршруты к такому же виду, что и у соответствующих им общедоступных маршрутов. Например, если общедоступный маршрут для действия имеет вид **/article/edit/5**, соответствующий административный маршрут может быть представлен в форме **/admin/article/edit/5**. С помощью префикса **/admin** можно обозначить административную сущность маршрута, одновременно обеспечив прозрачный и согласованный интерфейс для функциональности приложения. Эта согласованность особенно важна, если, например, впоследствии вы решите также предоставить доступ к функциям приложения посредством REST или SOAP.

В отличие от некоторых других фреймворков Zend Framework не содержит встроенных административных маршрутов. Следовательно, их необходимо настроить вручную, либо указав их в конфигурационном файле приложения, либо расширив базовый маршрутизатор, чтобы заставить его учитывать префиксы. Пример первого подхода приведен ниже, а по ссылкам в конце этой главы вы сможете найти примеры второго:

```
resources.router.routes.list.route = /admin/articles/index
resources.router.routes.list.defaults.module = system
resources.router.routes.list.defaults.controller = articles
resources.router.routes.list.defaults.action = list

resources.router.routes.view.route = /admin/articles/view/:id
resources.router.routes.view.defaults.module = system
```

```
resources.router.routes.view.defaults.controller = articles
resources.router.routes.view.defaults.action = display

resources.router.routes.update.route = /admin/articles/edit/:id
resources.router.routes.update.defaults.module = system
resources.router.routes.update.defaults.controller = articles
resources.router.routes.update.defaults.action = update

resources.router.routes.delete.route = /admin/articles/delete/:id
resources.router.routes.delete.defaults.module = system
resources.router.routes.delete.defaults.controller = articles
resources.router.routes.delete.defaults.action = delete
```

Макет

По умолчанию все представления будут использовать основной макет, определенный в конфигурационном файле приложения. Однако зачастую клиенты требуют придать административным представлениям другой внешний вид, либо в эстетических целях, либо чтобы визуально подчеркнуть факт перехода пользователей к другому разделу приложения.

Реализовать это требование в Zend Framework несложно: создайте новый макет для административных представлений, а затем переключайтесь на него в отдельных действиях по мере необходимости. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function adminDeleteAction()
  {
    $this->_helper->layout->setLayout('admin');
    // остальная часть действия
  }
}
```

При работе с большим количеством действий изменение макета может стать утомительным (не говоря уже о том, что оно может превратиться в кошмар для тех, кто будет его сопровождать). Поэтому альтернативным подходом является анализ URL запроса и автоматическое переключение макета в методах init() или preDispatch() контроллера перед тем, как сформировать страницу в представлении. Более подробно данный подход обсуждается в следующем разделе.

Упражнение 5.2. Вывод списка, удаление и обновление записей базы данных

Получив всю исходную информацию, давайте приступим к созданию панели администрирования SQUARE. Предположим, что администраторам требуется только

просматривать, редактировать и удалять записи из каталога. В следующих разделах будет подробно рассмотрена каждая из этих функций.

Установка административного макета

Первым шагом будет определение нового основного макета для административных представлений. Поскольку основная цель этого макета — обеспечение возможности отличить административный интерфейс от общедоступного, нет необходимости тщательно его прорабатывать. Ниже приведен пример того, как он может выглядеть:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
   <head>
     <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
     <base href="/" />
     <link rel="stylesheet" type="text/css" href="/css/master.css" />
     <link rel="stylesheet" type="text/css" href="/css/admin.css" />
     <script src="/js/form.js"></script>
   </head>
   <body>
     <div id="header">
       <div id="logo">
         <img src="/images/logo-admin.gif" />
       </div>

     <div id="menu">
       <a href="<?php echo $this->url(
       array(), 'admin-catalog-index'); ?>">CATALOG</a>
     </div>
   </div>

     <div id="content">
       <?php echo $this->layout()->content ?>
   </div>

     <div id="footer">
       <p>Created with <a href="http://framework.zend.com/">
       Zend Framework</a>. Licensed under
       <a href="http://www.creativecommons.org/">Creative Commons
     </a>.</p>
   </div>
 </body>
</html>
```

Сохраните этот шаблон макета в каталог макетов приложения под именем `$APP_DIR/application/layouts/admin.phtml`, как вы делали в главе 2.

Вы, вероятно, заметили, что макет использует три дополнительных файла: таблицу стилей CSS, файл с кодом на JavaScript и изображение с логотипом. Эти

файлы должны быть размещены в общедоступной части приложения, чтобы подключившиеся клиенты могли получить их по протоколу HTTP. Файлы ресурсов находятся в архиве с дополнительными материалами для этой главы.

Следующим шагом будет настройка приложения, позволяющая фреймворку автоматически переключать стандартный макет на административный, когда маршрутизатор обнаружит в URL префикс **/admin**. Как уже говорилось, проще всего это сделать в методе **preDispatch()** контроллера, автоматически вызывающегося перед выполнением каждого действия.

ВОПРОС ЭКСПЕРТУ

В: Для чего нужен метод **preDispatch()**?

О: Метод **preDispatch()** — это специальный метод-«заглушка», который автоматически выполняется контроллером перед вызовом запрашиваемого действия. Он особенно полезен для предварительной обработки, ведения журнала и выполнения других операций с содержимым запроса, перед тем как он будет передан конкретному действию. Если вам интересно, существует также метод **postDispatch()**, который автоматически выполняется контроллером после вызова запрошенного действия.

Код для автоматического переключения макета следует добавить к контроллеру **Catalog_AdminItemController** в файле **$APP_DIR/application/modules/catalog/controllers/AdminItemController.php**:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
    // действие для обработки административных URL
    public function preDispatch()
    {
      // проверяем URL на предмет наличия префикса /admin
      // устанавливаем административный макет
      $url = $this->getRequest()->getRequestUri();
    $this->_helper->layout->setLayout('admin');
  }
}
```

Этот код проверяет текущий запрос и с помощью метода **setLayout()** переключает макет на административный.

Определение пользовательских маршрутов

Теперь необходимо задать пользовательские маршруты для выполнения административных действий. На данном этапе нужно определиться с форматом этих маршрутов, поэтому предположим, что:

1. URL для получения общего списка будет иметь вид **/admin/catalog/item/index**.
2. Все URL для отображения элементов будут иметь вид **/admin/catalog/item/display/xx**, где *xx* — переменная, обозначающая идентификатор записи.

3. Все URL для обновления записей будут представлены в форме **/admin/catalog/item/update/xx**, где *xx* — переменная, обозначающая идентификатор записи.
4. Все URL для удаления записей будут иметь вид **/admin/catalog/item/delete/xx**, где *xx* — переменная, обозначающая идентификатор записи.

Для создания пользовательских маршрутов, соответствующих перечисленным URL, добавьте в конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**, следующие определения маршрутов:

```
resources.router.routes.admin-catalog-index.route = /admin/catalog/item/index
resources.router.routes.admin-catalog-index.defaults.module = catalog
resources.router.routes.admin-catalog-index.defaults.controller = admin.item
resources.router.routes.admin-catalog-index.defaults.action = index

resources.router.routes.admin-catalog-display.route = /admin/catalog/item/
display/:id
resources.router.routes.admin-catalog-display.defaults.module = catalog
resources.router.routes.admin-catalog-display.defaults.controller = admin.item
resources.router.routes.admin-catalog-display.defaults.action = display

resources.router.routes.admin-catalog-update.route = /admin/catalog/item/
update/:id
resources.router.routes.admin-catalog-update.defaults.module = catalog
resources.router.routes.admin-catalog-update.defaults.controller = admin.item
resources.router.routes.admin-catalog-update.defaults.action = update
resources.router.routes.admin-catalog-update.defaults.id = ""

resources.router.routes.admin-catalog-delete.route = /admin/catalog/item/delete
resources.router.routes.admin-catalog-delete.defaults.module = catalog
resources.router.routes.admin-catalog-delete.defaults.controller = admin.item
resources.router.routes.admin-catalog-delete.defaults.action = delete

resources.router.routes.admin-catalog-success.route = /admin/catalog/item/success
resources.router.routes.admin-catalog-success.defaults.module = catalog
resources.router.routes.admin-catalog-success.defaults.controller = admin.item
resources.router.routes.admin-catalog-success.defaults.action = success
```

Определение действия и представления для вывода списка

Теперь, когда все настроено, приступим к написанию кода действия. Взгляните на метод **Catalog_AdminItemController::indexAction**, который получает из базы данных список записей, используя модель **Doctrine**, и добавляет их к представлению:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
  // действие для отображения списка элементов каталога
  public function indexAction()
  {
    $q = Doctrine_Query::create()
```

```
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Type t');
    $result = $q->fetchArray();
    $this->view->records = $result;
  }
}
```

Затем представление преобразует полученную информацию в табличную форму для удобства чтения. Сценарий представления может выглядеть следующим образом (как обычно, он должен находиться в файле **$APP_DIR/application/modules/catalog/views/scripts/admin-item/index.phtml**):

```
<h2>List Items</h2>
<?php if (count($this->records)): ?>
<div id="records">
  <form method="post" action="
    <?php echo $this->url(array(), 'admin-catalog-delete'); ?>">
    <table>
      <tr>
        <td></td>
        <td class="key">
            Item ID
        </td>
        <td class="key">
            Title
        </td>
        <td class="key">
            Denomination
        </td>
        <td class="key">
            Country
        </td>
        <td class="key">
            Grade
        </td>
        <td class="key">
            Year
        </td>
        <td></td>
        <td></td>
      </tr>
      <?php foreach ($this->records as $r):?>
      <tr>
<td><input type="checkbox" name="ids[]"
    value="<?php echo $r['RecordID']; ?>" style="width:2px" />
        </td>
        <td><?php echo $this->escape($r['RecordID']); ?></td>
        <td><?php echo $this->escape($r['Title']); ?></td>
```

```
        <td><?php echo $this->escape(sprintf('%1.2f',
        $r['Denomination'])); ?></td>
        <td><?php echo $this->escape(
        $r['Square_Model_Country']['CountryName']); ?></td>
        <td><?php echo $this->escape(
        $r['Square_Model_Grade']['GradeName']); ?></td>
        <td><?php echo $this->escape($r['Year']); ?></td>
        <td><a href="<?php echo $this->url(array('id' =>
$r['RecordID']),
        'admin-catalog-display'); ?>">Display</a></td>
      <td><a href="<?php echo $this->url(array('id' =>
$r['RecordID']),
        'admin-catalog-update'); ?>">Update</a></td>
       </tr>
       <?php endforeach; ?>
       <tr>
         <td colspan="10"><input type="submit" name="submit"
         style="width:150px" value="Delete Selected" /></td>
      </tr>
    </table>
  </form>
</div>
<?php else: ?>
No records found
<?php endif; ?>
```

В этом представлении стоит отметить несколько важных моментов:

- Перед формированием всех полей их значения экранируются. Как уже говорилось в главе 4, это основная мера предосторожности, которую вы должны принять при работе с выходными значениями, чтобы уменьшить риск атак вида CSRF и XSS.
- Тег **<table>** находится внутри тега **<form>**. Это необходимо для работы с переключателями, расположенными рядом с каждой записью и позволяющими удалять несколько записей одновременно. Обратите внимание на то, что эта форма создается в сценарии представления вручную, а не с помощью **Zend_Form**, поскольку в **Zend_Form** не существует простого способа создать форму, содержащую таблицу.
- Рядом с каждой записью находятся ссылки **Update** и **Display**. Они соответствуют маршрутам, определенным в предыдущем разделе, и динамически генерируются во время выполнения программы с помощью вспомогательного метода представления **url()**. Заметьте, что идентификатор записи добавляется в запрос URL в виде параметра GET.

Чтобы посмотреть, как работает это представление, откройте браузер и перейдите по адресу **http://square.localhost/admin/catalog/item/index**. Вы должны увидеть сводную таблицу, похожую на ту, что показана на рис. 5.7.

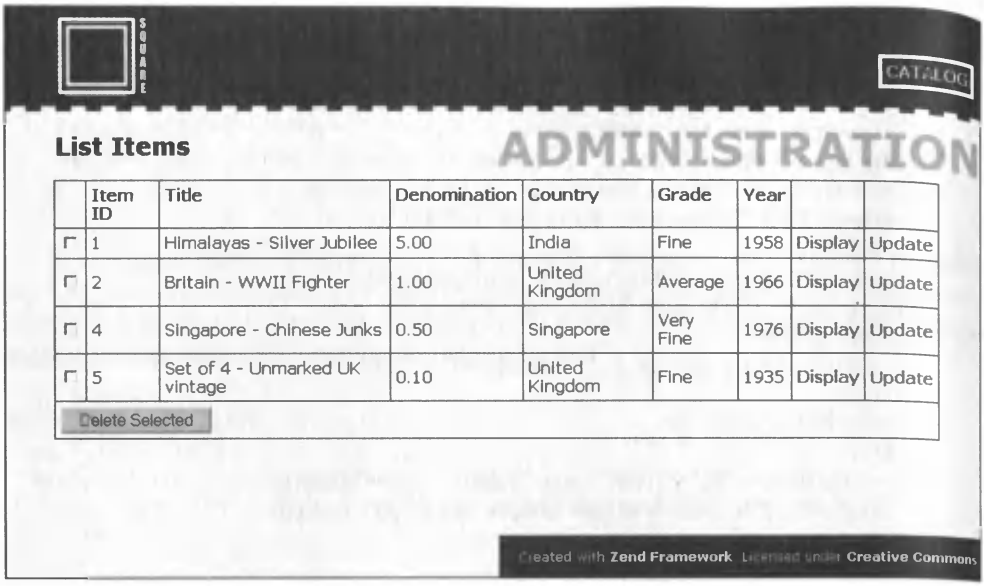


Рис. 5.7. Сводная страница каталога в панели администрирования SQUARE

Определение действия для удаления

Когда пользователь отмечает для удаления одну или несколько записей, их идентификаторы объединяются в массив и отправляются действию **Catalog_AdminItem Controller::deleteAction** в запросе POST. Метод **deleteAction()** извлекает выбранные идентификаторы и использует модель **Doctrine** для формирования запроса **DELETE**, который удаляет записи из базы данных. Пример соответствующего кода:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
  // действия для удаления элементов каталога
  public function deleteAction()
  {
    // устанавливаем фильтры и валидаторы для данных, полученных в запросе POST
    $filters = array(
      'ids' => array('HtmlEntities', 'StripTags', 'StringTrim')
    );
    $validators = array(
    'ids' => array('NotEmpty', 'Int')
    );
    $input = new Zend_Filter_Input($filters, $validators);
    $input->setData($this->getRequest()->getParams());

    // проверяем корректность введенных данных
    // читаем массив идентификаторов записей
```

```
    // удаляем записи из базы данных
    if ($input->isValid()) {
      $q = Doctrine_Query::create()
            ->delete('Square_Model_Item i')
            ->whereIn('i.RecordID', $input->ids);
      $result = $q->execute();

      $this->_helper->getHelper('FlashMessenger')
        ->addMessage('The records were successfully deleted.');
      $this->_redirect('/admin/catalog/item/success');
    } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }

  // действие, выполняемое при успешном завершении операции
  public function successAction()
  {
    if ($this->_helper->getHelper('FlashMessenger')->getMessages()) {
      $this->view->messages = $this->_helper
         ->getHelper('FlashMessenger')->getMessages();
    } else {
      $this->_redirect('/admin/catalog/item/index');
    }
  }
}
```

Чтобы проверить работу приведенного кода, выберите для удаления одну или несколько записей, отметив соответствующие переключатели на странице со списком, а затем отправьте форму, используя кнопку **Delete selected**.

Определение формы для обновления

Зачастую администраторы могут обновлять значения бóльшего количества полей записи, чем обычные пользователи. Например, в приложении SQUARE, кроме стандартных полей описания элементов, администраторы также могут обновлять статус их отображения и определять период, в течение которого элементы будут видимы в общедоступном каталоге.

Zend_Form, с его объектно-ориентированным подходом к созданию форм, особенно хорош для выполнения данного требования. Используя такие принципы ООП, как наследование и расширяемость, можно дополнить базовый класс формы и унаследовать от него различные производные классы форм, добавляя и удаляя элементы с помощью методов **addElement()** и **removeElement()** класса **Zend_Form**.

Для иллюстрации рассмотрим класс формы **ItemUpdate**, унаследованный от класса **ItemCreate**, созданного в начале этой главы. Производный класс содержит многие элементы базового класса; однако поскольку он предназначен для использования администраторами, из него удалены некоторые ненужные элементы

(например, CAPTCHA) и добавлены новые (поля для ввода статуса отображения и дат). Пример кода:

```
<?php
class Square_Form_ItemUpdate extends Square_Form_ItemCreate
{
  public function init()
  {
    // инициализируем базовую форму
    parent::init();

    // устанавливаем действие для формы (чтобы использовать текущий URL,
    // используйте значение false)
    $this->setAction('/admin/catalog/item/update');

    // удаляем ненужные элементы
    $this->removeElement('Captcha');
    $this->removeDisplayGroup('verification');

    // создаем скрытое поле для идентификатора элемента
    $id = new Zend_Form_Element_Hidden('RecordID');
    $id->addValidator('Int')
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем список выбора для статуса отображения элемента
    $display = new Zend_Form_Element_Select('DisplayStatus',
      array('onChange' =>
        "javascript:handleInputDisplayOnSelect('DisplayStatus',
           'divDisplayUntil', new Array('1'));"));
    $display->setLabel('Display status:')
        ->setRequired(true)
        ->addValidator('Int')
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');
    $display->addMultiOptions(array(
      0 => 'Hidden',
      1 => 'Visible'
    ));

    // создаем скрытое поле для даты отображения элемента
    $displayUntil = new Zend_Form_Element_Hidden('DisplayUntil');
    $displayUntil->addValidator('Date')
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем списки выбора для даты отображения элемента
    $displayUntilDay = new Zend_Form_Element_Select('DisplayUntil_day');
    $displayUntilDay->setLabel('Display until:')
        ->addValidator('Int')
```

```
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim')
        ->addFilter('StringToUpper')
        ->setDecorators(array(
            array('ViewHelper'),
            array('Label', array('tag' => 'dt')),
            array('HtmlTag',
              array(
                'tag' => 'div',
                'openOnly' => true,
                'id' => 'divDisplayUntil',
                'placement' => 'prepend'
              )
            ),
        ));
for($x=1; $x<=31; $x++) {
  $displayUntilDay->addMultiOption($x, sprintf('%02d', $x));
}

  $displayUntilMonth = new Zend_Form_Element_Select('DisplayUntil_month');
  $displayUntilMonth->addValidator('Int')
      ->addFilter('HtmlEntities')
      ->addFilter('StringTrim')
      ->setDecorators(array(
          array('ViewHelper')
      ));
  for($x=1; $x<=12; $x++) {
    $displayUntilMonth->addMultiOption(
  $x, date('M', mktime(1,1,1,$x,1,1)));
  }

  $displayUntilYear = new Zend_Form_Element_Select('DisplayUntil_year');
  $displayUntilYear->addValidator('Int')
      ->addFilter('HtmlEntities')
      ->addFilter('StringTrim')
      ->setDecorators(array(
          array('ViewHelper'),
          array('HtmlTag',
              array(
                'tag' => 'div',
                  'closeOnly' => true
              )
            ),
        ));
  for($x=2009; $x<=2012; $x++) {
    $displayUntilYear->addMultiOption($x, $x);
  }

  // добавляем элементы к форме
  $this->addElement($id)
```

```
            ->addElement($display)
            ->addElement($displayUntil)
            ->addElement($displayUntilDay)
            ->addElement($displayUntilMonth)
            ->addElement($displayUntilYear);

        // создаем группу отображения для статуса
        $this->addDisplayGroup(
          array('DisplayStatus', 'DisplayUntil_day',
                'DisplayUntil_month', 'DisplayUntil_year',
                'DisplayUntil'), 'display');
        $this->getDisplayGroup('display')
            ->setOrder(25)
            ->setLegend('Display Information');
        }
}
```

Сохраните это определение формы в файле **$APP_DIR/library/Square/Form/ItemUpdate.php**.

Обработка ввода дат

Несколько слов о полях для ввода дат, которые содержит эта форма. Как вы, возможно, помните из главы 3, в состав **Zend_Form** входит валидатор **Date**, который можно использовать для проверки корректности даты. Если вы посмотрите на приведенное выше определение формы, то увидите, что для ввода даты используются три списка выбора — день, месяц и год. Также существует скрытое поле ввода для внутреннего использования, в котором хранится результирующее значение даты, составленное на основе этих списков и связанное с вышеупомянутым валидатором **Date**.

Вероятно, вы думаете, что это слишком сложно, и вы правы. Почему? Дело в том, что при стандартных настройках валидатор **Date** ожидает на входе значение в виде строки в формате **ГГГГ-ММ-ДД**, и если строка не соответствует этому формату, возникает ошибка валидации. В реальности пользователи часто ошибаются при ручном вводе дат в виде строк. Наиболее распространенные ошибки включают в себя выбор другого формата (например, **ДД-ММ-ГГГГ**), пропуск ведущих нулей для значений дней и месяцев, состоящих из одной цифры (например, ввод 9 вместо 09), или использование другого разделителя (например, **ГГГГ/ММ/ДД**).

С учетом сказанного, для ввода даты обычно безопаснее и удобнее предоставлять пользователю список выбора или графический календарь, которые меньше зависят от его предпочтений, а затем программно переводить введенные данные в подходящий формат. В следующем разделе вы увидите, что именно этим и занимается метод **updateAction()**. Он использует входные данные из трех списков выбора, чтобы создать результирующее значение даты в формате **ГГГГ-ММ-ДД**, а затем присваивает это значение скрытому полю, после чего валидация осуществляется как обычно.

Так как скрытое поле ввода связано с валидатором Date, некорректная дата не пройдет валидацию и валидатор Date сгенерирует сообщение об ошибке, которое пользователь увидит при повторном отображении формы. Если дата корректна, входные данные пройдут валидацию и оставшаяся часть кода действия выполнится как обычно.

СОВЕТ

Существуют и другие подходы к обработке полей ввода даты. Один из них, предложенный ведущим разработчиком Zend Framework Мэтью Вайер О'Финни (Matthew Weier O'Phinney), предполагает расширение базового класса Zend_Form_Element для создания составного элемента ввода даты (ссылка на описание этого подхода приведена в конце главы). Другая техника, которая подробно обсуждается в главе 11, задействует графический календарь в сочетании с полем ввода текста для непосредственной записи выбранной даты в форму в требуемом формате.

Вы, наверное, заметили, что приведенное определение формы также использует декоратор HtmlTag. Как объяснялось в главе 3, он управляет разметкой вокруг полей формы. В данном конкретном случае он используется для помещения трех списков выбора даты внутрь элемента с помощью выборочного использования атрибутов декоратора 'openOnly' и 'closeOnly'. Этот используется пользовательской JavaScript-функцией handleInputDisplayOnSelect(), которая автоматически скрывает или показывает набор списков выбора даты в зависимости от статуса отображения элемента.

Определение действия и представления для обновления

При обновлении записей обычно используются два отдельных запроса:

1. Первый запрос — *чтение* — осуществляется, когда пользователь выбирает запись для обновления. Метод updateAction() должен прочитать из базы данных выбранную запись, заполнить веб-форму текущими значениями полей и отобразить ее пользователю.
2. Второй запрос — *запись* — осуществляется, когда пользователь отправляет форму. Метод updateAction() должен получить отправленные значения и провести их фильтрацию и валидацию. Если значения корректны, действие должно внести новую версию записи в базу данных.

Действие Catalog_AdminItemController::updateAction выполняет обе эти функции. Пример:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
  // действие для изменения отдельного элемента каталога
  public function updateAction()
  {
    // генерируем форму ввода
    $form = new Square_Form_ItemUpdate;
    $this->view->form = $form;
```

```
if ($this->getRequest()->isPost()) {
  // если получен запрос POST,
  // проверяем корректность входных данных,
  // получаем текущую запись,
  // обновляем значения и заменяем хранящиеся в базе данных
  $postData = $this->getRequest()->getPost();
  $postData['DisplayUntil'] = sprintf('%04d-%02d-%02d',
    $this->getRequest()->getPost('DisplayUntil_year'),
    $this->getRequest()->getPost('DisplayUntil_month'),
    $this->getRequest()->getPost('DisplayUntil_day')
  );
  if ($form->isValid($postData)) {
    $input = $form->getValues();
    $item = Doctrine::getTable('Square_Model_Item')
          ->find($input['RecordID']);
    $item->fromArray($input);
    $item->DisplayUntil =
      ($item->DisplayStatus == 0) ? null : $item->DisplayUntil;
    $item->save();
    $this->_helper->getHelper('FlashMessenger')
      ->addMessage('The record was successfully updated.');
    $this->_redirect('/admin/catalog/item/success');
  }
} else {
  // если получен запрос GET,
  // устанавливаем фильтры и валидаторы для входных данных из этого запроса,
  // проверяем корректность последних,
  // получаем запрошенную запись,
  // заполняем форму
  $filters = array(
    'id' => array('HtmlEntities', 'StripTags', 'StringTrim')
  );
  $validators = array(
    'id' => array('NotEmpty', 'Int')
  );
  $input = new Zend_Filter_Input($filters, $validators);
  $input->setData($this->getRequest()->getParams());
  if ($input->isValid()) {
  $q = Doctrine_Query::create()
    ->from('Square_Model_Item i')
    ->leftJoin('i.Square_Model_Country c')
    ->leftJoin('i.Square_Model_Grade g')
    ->leftJoin('i.Square_Model_Type t')
    ->where('i.RecordID = ?', $input->id);
$result = $q->fetchArray();
if (count($result) == 1) {
  // выполняем преобразования для списков выбора даты
  $date = $result[0]['DisplayUntil'];
```

```
            $result[0]['DisplayUntil_day'] = date('d', strtotime($date));
            $result[0]['DisplayUntil_month'] = date('m', strtotime($date));
            $result[0]['DisplayUntil_year'] = date('Y', strtotime($date));
            $this->view->form->populate($result[0]);
        } else {
            throw new Zend_Controller_Action_Exception('Page not found', 404);
        }
            } else {
                throw new Zend_Controller_Action_Exception('Invalid input');
            }
        }
    }
}
```

В общих чертах метод updateAction() состоит из двух фрагментов, разделенных проверкой условия:

1. Если запрос является запросом POST, действие получает входные данные и осуществляет необходимые преобразования, чтобы создать составную строку даты в формате ГГГГ-ММ-ДД, как говорилось в предыдущем разделе. Затем оно использует метод isValid() объекта Zend_Form, чтобы проверить корректность отправленных данных. Если они корректны, действие использует метод find() Doctrine, чтобы получить выбранную запись в виде экземпляра класса Item, и устанавливает новые значения для различных полей записи, основываясь на введенных данных. После этого с помощью метода save() осуществляется сохранение записи в базу данных и управление передается методу successAction().
2. Если запрос является запросом GET, действие получает идентификатор записи из строки запроса URL и выполняет запрос Doctrine для получения соответствующей записи из базы данных. Затем оно осуществляет обратные преобразования входных значений даты и использует метод populate() объекта Zend_Form, чтобы заполнить форму ввода текущими значениями полей записи.

Ниже приведен соответствующий сценарий представления, который необходимо сохранить в файле $APP_DIR/application/modules/catalog/views/scripts/admin-item/update.phtml:

```
<h2>Update Item</h2>
<?php echo $this->form; ?>

<script>
handleInputDisplayOnSelect('DisplayStatus',
  'divDisplayUntil', new Array('1'));
</script>
```

Чтобы проверить работу сценария, используйте ссылку Update на странице со списком марок для выбора записи или откройте в браузере соответствующий URL, например http://square.localhost/admin/catalog/item/update/1. Вы должны увидеть форму, которая уже заполнена текущим содержимым записи (рис. 5.8).

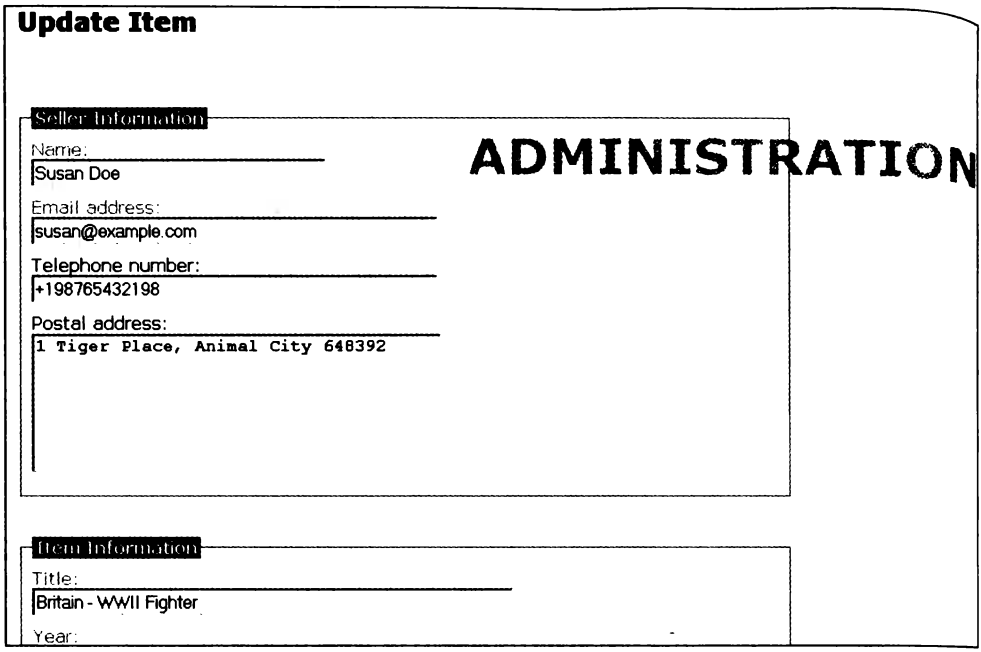


Рис. 5.8. Форма для обновления существующей записи в панели администрирования SQUARE

Обратите внимание, что эта форма содержит дополнительные поля, позволяющие управлять статусом отображения элемента (рис. 5.9). Если отображение элемента включено, у вас также есть возможность определить дату, до которой элемент будет оставаться видимым.

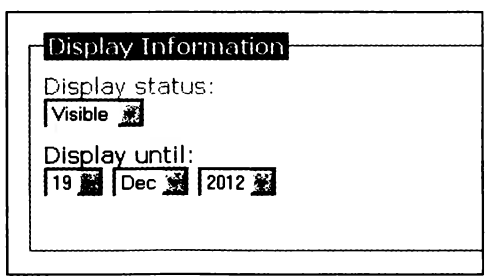


Рис. 5.9. Дополнительные опции для определения видимости элемента в панели администрирования SQUARE

Внесите изменения и отправьте форму. Если введенные данные пройдут валидацию, запись будет обновлена, а вы увидите сообщение об успехе. Вы можете проверить корректность обновления, перейдя по соответствующему URL для отображения записи, например, http://square.localhost/catalog/item/display/1.

Обновление действия для отображения

Осталось внести последнее изменение. В главе 4 вы создали действие displayAction(), принимающее идентификатор записи и получающее из базы данных соответствующую запись, используя запрос Doctrine. Однако этот запрос не учитывал текущий статус отображения элемента. Поскольку теперь существует механизм, с помощью которого администраторы могут изменять данный статус, неплохо было бы модифицировать displayAction() и добавить в него учет статуса отображения для отображения только тех элементов, которые доступны для публичного просмотра.

Для решения этой задачи измените действие ItemController::displayAction, расположенное в файле $APP_DIR/application/modules/catalog/controllers/ItemController.php, добавив в запрос Doctrine условие, проверяющее видимость элемента:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
{
  // действие для отображения элемента каталога
  public function displayAction()
  {
    // устанавливаем фильтры и валидаторы для входных данных,
    // полученных в запросе GET
    //
    // проверяем корректность входных данных,
    // получаем запрошенную запись,
    // добавляем ее к представлению
      if ($input->isValid()) {
      $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Type t')
        ->where('i.RecordID = ?', $input->id)
        ->addWhere('i.DisplayStatus = 1')
        ->addWhere('i.DisplayUntil >= CURDATE()');
    $result = $q->fetchArray();
    if (count($result) == 1) {
      $this->view->item = $result[0];
    } else {
    throw new Zend_Controller_Action_Exception('Page not found', 404);
      }
    } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }
}
```

После этого изменениям через публичный интерфейс станут доступны только элементы, отмеченные как видимые. Любая попытка получить доступ к другим элементам приведет к исключению 404, которое будет передано стандартному обработчику исключений для отображения сообщения пользователю.

ПРИМЕЧАНИЕ

Вы можете спросить, почему эта глава не содержит информации о чтении отдельных записей (буква *R* в аббревиатуре CRUD)? Этот аспект уже был рассмотрен в главе 4 в разделе «Получение записей из базы данных», где использовалась модель Doctrine для чтения и отображения записей базы данных по их идентификатору. Панель администрирования, рассмотренная в этой главе, содержит похожую функцию, которую вы можете найти в архиве с дополнительными материалами к этой главе.

Добавление аутентификации пользователей

В данный момент разработка панели администрирования завершена, и администраторы могут получать, обновлять и удалять записи через веб-браузер. Однако эти действия еще не защищены, и любой пользователь может получить к ним доступ по их URL. Очевидно, что данный факт угрожает безопасности, поэтому следующим шагом будет реализация механизма, позволяющего приложению отличать *непривилегированных пользователей* от *администраторов* и разрешать доступ к административным функциям только последним.

Zend Framework содержит компонент, специально разработанный для решения задач аутентификации пользователей. Этот компонент, который называется Zend_Auth, предоставляет API и набор адаптеров для сопоставления учетных данных пользователя с различными источниками данных, включая SQL-совместимые СУБД, текстовые файлы, каталоги LDAP и провайдеры OpenID. Zend_Auth сохраняет идентификационную информацию пользователей, прошедших аутентификацию, в хранилище сессий, поэтому ее в любое время можно получить или перепроверить в контроллере или действии.

ПРИМЕЧАНИЕ

По умолчанию Zend_Auth поддерживает только хранилища сессий. Однако вы можете добавить поддержку других механизмов хранения, просто расширив класс Zend_Auth_Storage_Interface. Более подробную информацию и примеры можно найти по ссылкам, приведенным в конце этой главы.

Чтобы понять, как это работает на практике, рассмотрим следующий пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function authenticateAction()
  {
    // генерируем форму входа
    $form = new Form_Example_Auth;
    $this->view->form = $form;
    if ($this->getRequest()->isPost()) {
      if ($form->isValid($this->getRequest()->getPost())) {
        // получаем учетные данные из пользовательского ввода,
        // инициализируем адаптер аутентификации
        // выполняем аутентификацию и проверяем результат
        $values = $form->getValues();
```

```
        $adapter = new Zend_Auth_Adapter_DbTable($this->db);
        $adapter->setIdentity($values['username'])
                ->setCredential($values['password']);
        $auth = Zend_Auth::getInstance();
        $result = $auth->authenticate($adapter);
        if ($result->isValid()) {
        $this->_helper->getHelper('FlashMessenger')
                ->addMessage('You were successfully logged in.');
          $this->_redirect('/admin/index');
        } else {
          $this->view->message =
             'You could not be logged in. Please try again.';
        }
      }
    }
  }
}
```

В примере используется адаптер аутентификации по базе данных **DbTable**, который инициализируется экземпляром класса **Zend_Db_Adapter_Abstract**. Этому адаптеру передается набор учетных данных пользователя, а затем используется синглтон **Zend_Auth** для аутентификации с использованием этих данных. Метод **Zend_Auth::authenticate** возвращает объект **Zend_Auth_Result**; его метод **isValid()** можно использовать, чтобы узнать об успешности аутентификации.

Zend_Auth также содержит несколько дополнительных методов, которые можно вызывать из контроллеров и действий, чтобы определить, существует ли аутентифицированная сессия. Например, метод **hasIdentity()** возвращает логическое значение, определяющее, существует ли сессия для аутентифицированного пользователя, а метод **getIdentity()** возвращает идентификационную информацию этого пользователя. Методы можно вызывать в действиях, чтобы определить, прошел ли пользователь аутентификацию и, следовательно, можно ли разрешить выполнение действия.

Ниже приведен пример, в котором действие **protectedAction()** доступно только аутентифицированным пользователям:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function protectedAction()
  {
    // проверяем, аутентифицирован ли пользователь
    // если нет, перенаправляем его на страницу входа,
    // если да, продолжаем выполнение действия
    if (!Zend_Auth::getInstance()->hasIdentity()) {
        $this->_redirect('/admin/login');
    } else {
      $model = new MemberModel;
      $this->view->data = $model->fetchAll();
    }
  }
}
```

Расширив базовый класс Zend_Auth_Adapter_Interface, можно создать пользовательские адаптеры для аутентификации, которые будут сверять учетные данные с другими источниками данных. Производный класс должен как минимум предоставлять конструктор, принимающий учетные данные пользователя, и метод authenticate(), который сверяет эти данные с провайдером аутентификации. Этот метод возвращает экземпляр класса Zend_Auth_Result, содержащий константу, определяющую, была ли аутентификация успешной. Пример пользовательского адаптера аутентификации вы увидите в следующем разделе.

Упражнение 5.3. Создание системы входа/выхода

Zend_Auth относится к числу компонентов, работу которых трудно объяснить на словах и проще понять на практическом примере. Поэтому давайте используем этот компонент на практике, создав простую систему входа/выхода для ограничения доступа к панели администрирования SQUARE. Этот процесс описан в следующих разделах.

Определение пользовательских маршрутов

Как обычно, начнем с определений маршрутов. Создайте маршруты для входа в панель администрирования и выхода из нее, добавив следующие определения в конфигурационный файл приложения, $APP_DIR/application/configs/application.ini:

```
resources.router.routes.login.route = /admin/login
resources.router.routes.login.defaults.module = default
resources.router.routes.login.defaults.controller = login
resources.router.routes.login.defaults.action = login

resources.router.routes.login-success.route = /admin/login/success
resources.router.routes.login-success.defaults.module = default
resources.router.routes.login-success.defaults.controller = login
resources.router.routes.login-success.defaults.action = success

resources.router.routes.logout.route = /admin/logout
resources.router.routes.logout.defaults.module = default
resources.router.routes.logout.defaults.controller = login
resources.router.routes.logout.defaults.action = logout
```

Определение формы входа

Следующим шагом будет определение самой формы входа. Она необязательно должна быть очень сложной: все, что на самом деле понадобится, — это поле для ввода имени пользователя, поле для ввода пароля и кнопка отправки. Ниже приведено определение формы, которое следует сохранить в файле $APP_DIR/library/Square/Form/Login.php:

```
<?php
class Square_Form_Login extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/admin/login')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $username = new Zend_Form_Element_Text('username');
    $username->setLabel('Username:')
        ->setOptions(array('size' => '30'))
        ->setRequired(true)
        ->addValidator('Alnum')
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем текстовое поле для ввода пароля
    $password = new Zend_Form_Element_Password('password');
    $password->setLabel('Password:')
        ->setOptions(array('size' => '30'))
        ->setRequired(true)
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Log In')
        ->setOptions(array('class' => 'submit'));

    // добавляем элементы к форме
    $this->addElement($username)
        ->addElement($password)
        ->addElement($submit);
    }
}
```

Определение адаптера аутентификации

Форма из предыдущего раздела отправляет данные **LoginController::loginAction** в модуле **default**. Перед тем как начать писать код этого действия, уделим несколько минут созданию адаптера аутентификации, способного сверить переданные имя пользователя и пароль с таблицей пользователей, созданной в главе 4.

Как уже говорилось, создать пользовательский адаптер аутентификации с помощью **Zend_Auth** несложно. Достаточно всего лишь расширить базовый класс **Zend_Auth_Adapter_Interface** и реализовать в производном классе метод **authenticate()**, который будет осуществлять требуемую аутентификацию. Более важный вопрос —

зачем вообще создавать пользовательский адаптер, если **Zend_Auth** уже содержит адаптер для аутентификации по базе данных?

Причина заключается в эффективности и согласованности. Во-первых, адаптер **Zend_Auth_Adapter_DbTable** работает только с экземпляром класса **Zend_Db** и не может быть непосредственно использован с моделью **Doctrine**. В свою очередь, формат параметров подключения к базе данных, используемый **Zend_Db**, отличается от того, что используется в **Doctrine**. Поэтому для непосредственного использования **Zend_Auth_Adapter_DbTable** потребуется либо указывать в конфигурационном файле одни и те же параметры в двух различных форматах (в этом случае при изменении, скажем, пароля базы данных придется обновлять оба набора параметров), либо писать код для «перевода» DSN-строки **Doctrine** в формат массива, используемый **Zend_Db** (этот подход утомителен и может стать причиной ошибок из-за большого числа возможных вариантов DSN).

Определенное значение имеет и согласованность. До настоящего момента весь доступ к базе данных осуществлялся через модели **Doctrine**. Переход к использованию **Zend_Db** для запросов, связанных с аутентификацией, может сбить с толку; кроме того, он неэффективен, поскольку для работы с базой данных приложению понадобится загружать два набора библиотек вместо одного. С учетом всех этих факторов имеет смысл создать пользовательский адаптер аутентификации, использующий **Doctrine**, вместо того чтобы задействовать адаптер **Zend_Auth_Adapter_DbTable**.

Обосновав наш выбор, приступим к написанию кода. Взгляните на следующий листинг, в котором создается основанный на **Doctrine** адаптер, соответствующий соглашениям, принятым для **Zend_Auth**:

```
<?php
class Square_Auth_Adapter_Doctrine implements Zend_Auth_Adapter_Interface
{
  // массив, содержащий запись об аутентифицированном пользователе
  protected $_resultArray;

  // конструктор
  // принимает имя пользователя и пароль
  public function __construct($username, $password)
  {
    $this->username = $username;
    $this->password = $password;
  }

  // основной метод аутентификации
  // запрашивает базу данных на предмет наличия подходящих учетных данных
  // возвращает экземпляр Zend_Auth_Result с кодом успеха/неудачи
  public function authenticate()
  {
    $q = Doctrine_Query::create()
        ->from('Square_Model_User u')
        ->where('u.Username = ? AND u.Password = PASSWORD(?)',
```

```
            array($this->username, $this->password)
        );
    $result = $q->fetchArray();
    if (count($result) == 1) {
      return new Zend_Auth_Result(
        Zend_Auth_Result::SUCCESS, $this->username, array());
    } else {
      return new Zend_Auth_Result(
        Zend_Auth_Result::FAILURE, null,
            array('Authentication unsuccessful')
      );
    }
}

    // возвращает результирующий массив, представляющий собой запись
    // об аутентифицированном пользователе
    // если требуется, удаляет из записи указанные поля
    public function getResultArray($excludeFields = null)
    {
      if (!$this->_resultArray) {
        return false;
        }
        if ($excludeFields != null) {
          $excludeFields = (array)$excludeFields;
            foreach ($this->_resultArray as $key => $value) {
              if (!in_array($key, $excludeFields)) {
            $returnArray[$key] = $value;
        }
    }
        return $returnArray;
      } else {
        return $this->_resultArray;
      }
    }
}
```

Сохраните этот класс в файл $APP_DIR/library/Square/Auth/Adapter/Doctrine.php.

Основной частью данного класса является метод **authenticate()**, который формирует и выполняет запрос Doctrine для проверки корректности переданных имени пользователя и пароля. Метод возвращает экземпляр класса Zend_Auth_Result, содержащий три свойства: код результата, обозначающий успех или неудачу, имя пользователя (или другой уникальный ключ идентификации) — в случае успеха, и массив, содержащий сообщения об ошибках, в случае неудачи.

СОВЕТ

Помимо простого обозначения успеха или неудачи существуют еще несколько кодов результата. Они могут быть использованы для предоставления более конкретной информации о причинах ошибки аутентификации: например, связана она с именем пользователя или с паролем, было ли найдено более одной подходящей записи о пользователе или произошла какая-то другая ошибка. Полный список кодов можно найти в документации к Zend Framework.

Метод **authenticate()** просто сообщает, был пользователь аутентифицирован или нет, а метод **getResultArray()** возвращает полную запись об аутентифицированном пользователе в виде массива. В качестве необязательного аргумента этот метод принимает список полей, которые требуется исключить, и убирает их из возвращаемого массива. Этот полезный метод стоит включить в ваш адаптер, так как он позволяет осуществлять быстрый доступ к пользовательской записи и упрощает ее сохранение в сессии для использования в других действиях. Пример приведен в следующем разделе.

Определение действия и представления для входа

После передачи основной работы адаптеру аутентификации создание действия **LoginController::loginAction** не представляет труда. Ниже приведен код, который следует сохранить в файл **$APP_DIR/application/modules/default/controllers/LoginController.php**:

```
<?php
class LoginController extends Zend_Controller_Action
{
  public function init()
  {
    $this->_helper->layout->setLayout('admin');
  }

    // действие для входа
    public function loginAction()
    {
      $form = new Square_Form_Login;
      $this->view->form = $form;
      // проверяем корректность введенных данных,
      // производим аутентификацию с использованием адаптера,
      // сохраняем запись о пользователе в сессии,
      // перенаправляем на первоначально запрошенный URL, если таковой имеется
      if ($this->getRequest()->isPost()) {
        if ($form->isValid($this->getRequest()->getPost())) {
          $values = $form->getValues();
          $adapter = new Square_Auth_Adapter_Doctrine(
            $values['username'], $values['password']
        );
        $auth = Zend_Auth::getInstance();
        $result = $auth->authenticate($adapter);
        if ($result->isValid()) {
          $session = new Zend_Session_Namespace('square.auth');
          $session->user = $adapter->getResultArray('Password');
          if (isset($session->requestURL)) {
            $url = $session->requestURL;
            unset($session->requestURL);
            $this->_redirect($url);
          } else {
            $this->_helper->getHelper('FlashMessenger')
```

```
                ->addMessage('You were successfully logged in.');
            $this->_redirect('/admin/login/success');
            }
          } else {
            $this->view->message =
              'You could not be logged in. Please try again.';
        }
      }
    }
  }

public function successAction()
{
    if ($this->_helper->getHelper('FlashMessenger')->getMessages()) {
      $this->view->messages = $this->_helper
            ->getHelper('FlashMessenger')
            ->getMessages();
     } else {
      $this->_redirect('/');
    }
  }
}
```

ВОПРОС ЭКСПЕРТУ

В: Почему LoginController сделан частью модуля default, а не модуля catalog?

О: Модуль catalog выступает в роли контейнера для контроллеров, действий и представлений, связанных с управлением каталогом. Действия для входа и выхода не относятся к управлению каталогом, поэтому их лучше разместить на глобальном уровне, то есть на уровне приложения. Таким образом, с точки зрения логики модуль default подходит для них больше.

Действие loginAction() считывает учетные данные, переданные через форму входа, и использует адаптер аутентификации, созданный на предыдущем этапе, для сопоставления их с базой данных приложения. Если аутентификация завершается с ошибкой, действие осуществляет повторное отображение формы, добавляя сообщение об ошибке. В случае успеха метод сохраняет запись о пользователе в сессии, а затем производит поиск первоначально запрошенного URL (об этом чуть позже) в пространстве имен последней. Если URL найден, клиент перенаправляется по нему; если нет — вызывается действие successAction() и соответствующее представление формирует страницу.

Ниже приведен вариант сценария представления для входа, находящийся в файле $APP_DIR/application/modules/default/views/scripts/login.phtml:

```
<h2>Login</h2>
<div style="color:red; font-weight:bolder">
   <?php echo $this->message; ?>
</div>
<?php echo $this->form; ?>
```

Определение действия для выхода

Дополнением к действию **loginAction()** является действие **logoutAction()**, которое уничтожает сессию аутентифицированного пользователя и удаляет его идентификационную информацию с помощью метода **Zend_Auth::clearIdentity**. Пример:

```
<?php
class LoginController extends Zend_Controller_Action
{
  public function logoutAction()
  {
    Zend_Auth::getInstance()->clearIdentity();
    Zend_Session::destroy();
    $this->_redirect('/admin/login');
  }
}
```

Защита административных действий

Последним шагом будет добавление проверок, связанных с аутентификацией, в действия, которые вы хотите защитить. Так как все административные действия, которые у нас в данный момент есть, находятся в одном контроллере, **Catalog_AdminItemController**, эти проверки удобно добавить в метод **preDispatch()** контроллера. Ниже приведен пример обновленного метода:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
  // действие для обработки административных URL
  public function preDispatch()
  {
    // устанавливаем административный макет
    // проверяем, аутентифицирован ли пользователь,
    // если нет, перенаправляем его на страницу входа
    $url = $this->getRequest()->getRequestUri();
    $this->_helper->layout->setLayout('admin');
    if (!Zend_Auth::getInstance()->hasIdentity()) {
      $session = new Zend_Session_Namespace('square.auth');
      $session->requestURL = $url;
      $this->_redirect('/admin/login');
    }
  }
}
```

В результате этого изменения при получении маршрутизатором запроса на выполнение административного действия метод **preDispatch()** сначала проверит, прошел ли пользователь аутентификацию, воспользовавшись для этого методом **Zend_Auth::hasIdentity()**. Если идентификационной информации пользователя не существует, метод вернет **false**. В этом случае URL запроса будет сохранен в сессии, а клиент будет перенаправлен на форму входа.

После успешного входа URL запроса будет извлечен из сессии в методе loginAction() (о котором говорилось в предыдущем разделе) и клиент будет перенаправлен по этому URL. Снова будет вызван метод preDispatch(), однако в этот раз, поскольку личность пользователя была подтверждена, hasIdentity() вернет true и запрошенное действие будет обработано как обычно.

Обновление основного макета

Как показано в предыдущем разделе, метод hasIdentity() объекта Zend_Auth предоставляет удобный способ проверки статуса аутентификации пользователя. Этот факт можно использовать в основном макете для выборочного отображения некоторых элементов меню в зависимости от того, вошел пользователь в систему или нет. Ниже приведен пример, который обновляет административный макет в файле $APP_DIR/application/layouts/admin.phtml, с тем чтобы отображать в главном меню элемент Logout для вошедших пользователей и элемент Login для невошедших. Изменения в макете выделены жирным шрифтом.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
   <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
   <base href="/" />
   <link rel="stylesheet" type="text/css" href="/css/master.css" />
   <link rel="stylesheet" type="text/css" href="/css/admin.css" />
   <link rel="stylesheet" type="text/css" href="/css/yui/calendar.css" />
   <script src="/js/form.js"></script>
 </head>
 <body>
   <div id="header">
     <div id="logo">
       <img src="/images/logo-admin.gif" />
     </div>

     <div id="menu">
       <?php if (Zend_Auth::getInstance()->hasIdentity()): ?>
       <a href="<?php echo $this->url(array(),
       'admin-catalog-index'); ?>">CATALOG</a>
       <a href="<?php echo $this->url(array(), 'logout'); ?>">LOGOUT</a>
       <?php else: ?>
      <a href="<?php echo $this->url(array(), 'login'); ?>">LOGIN</a>
      <?php endif; ?>
     </div>
   </div>

   <div id="content">
     <?php echo $this->layout()->content ?>
   </div>
```

```
    <div id="footer">
        <p>Created with <a href="http://framework.zend.com/">
        Zend Framework</a>. Licensed under
        <a href="http://www.creativecommons.org/">Creative Commons
      </a>.</p>
    </div>
  </body>
</html>
```

Все готово! Чтобы убедиться в этом, перейдите на сводную страницу по адресу http://square.localhost/admin/catalog/item/index. Вы должны увидеть форму входа (рис. 5.10).

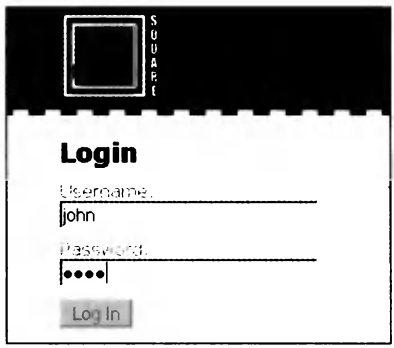


Рис. 5.10. Форма входа в панель администрирования SQUARE

После отправки через форму корректных учетных данных вы будете автоматически перенаправлены на сводную страницу. Ваш статус вошедшего пользователя также отразится в главном меню — там появится новая ссылка Logout (рис. 5.11).

Рис. 5.11. Меню в панели администрирования SQUARE после входа в систему

ВОПРОС ЭКСПЕРТУ

В: Могу ли я использовать Zend_Auth, чтобы разрешать или запрещать пользователям доступ к ресурсам приложения в зависимости от их ролей или привилегий?

О: Нет. Как указано в руководстве к Zend Framework, Zend_Auth связан с *аутентификацией пользователя* (проверкой того, что пользователь является тем, кем он представился), а не *контролем доступа пользователей* (проверкой того, что у пользователя есть права для доступа к определенным ресурсам приложения). На практике это означает, что после прохождения процедуры аутентификации пользователь имеет доступ к любым защищенным действиям; система не позволяет более детально определять, у каких пользователей и к каким действиям есть доступ. Если в вашем приложении требуется управлять доступом к определенным действиям, основываясь на ролях пользователей, следует рассмотреть возможность использования компонента Zend_Acl.

Выводы

Эта глава была посвящена двум наиболее распространенным задачам, которые вам предстоит решать при создании веб-приложений: реализации операций CRUD и аутентификации пользователей. Основываясь на техниках, рассмотренных в предыдущих главах, вы научились получать, создавать, обновлять и удалять записи базы данных, используя парадигму MVC. В данной главе вы также познакомились с компонентом Zend_Auth и разобрались с тем, как использовать его для ограничения доступа к определенным действиям с помощью простой системы аутентификации пользователей.

К концу главы демонстрационное приложение SQUARE обладает весьма существенной функциональностью. Теперь пользователи могут самостоятельно добавлять продаваемые марки в каталог, а администраторы могут просматривать списки, подтверждать отображение марок, изменять их или удалять через полноценную панель администрирования. Доступ к административным функциям ограничен простой системой входа/выхода, и в приложении присутствует даже пара приятных мелочей, таких как автоматическое перенаправление на запрошенную страницу после входа и главное меню, которое «знает», вошел пользователь или нет.

Реализация этих функций дала вам некоторое понимание того, как можно объединять разные компоненты Zend Framework, чтобы удовлетворять реальные требования. Не забывайте, что более подробную информацию о рассмотренных в этой главе темах вы можете получить, посетив следующие ссылки:

- Компонент Zend_Controller: http://framework.zend.com/manual/en/zend.controller.html.
- Компонент Zend_Layout: http://framework.zend.com/manual/en/zend.layout.html.
- Компонент Zend_Auth: http://framework.zend.com/manual/en/zend.auth.html.
- Реализация более сложного адаптера аутентификации Doctrine (Джейсон Эйзенменгер (Jason Eisenmenger) и Дэвид Вернер (David Werner)): http://framework.zend.com/wiki/pages/viewpage.action?pageId=3866950.
- Создание простого расширяемого CRUD с использованием Zend Framework (Райан Могер (Ryan Mauger)): http://www.rmauger.co.uk/2009/06/creating-simple-extendible-crud-using-zend-framework/.
- Создание составных элементов для ввода даты (Мэтью Вайер О'Финни (Matthew Weier O'Phinney)): http://weierophinney.net/matthew/archives/217-Creating-composite-elements.html.
- Более подробная информация о создании системы входа/выхода (Мэтью Вайер О'Финни (Matthew Weier O'Phinney)): http://weierophinney.net/matthew/archives/165-Login-and-Authentication-with-Zend-Framework.html.
- Улучшение базового маршрутизатора (Майкл Шекоски (Michael Sheakoski)): http://framework.zend.com/wiki/display/ZFUSER/MJS_Controller_PathRouter+-+An+enhanced+RewriteRouter?focusedCommentId=9437448.
- Использование базы данных в качестве хранилища идентификационной информации для Zend_Auth (Бранко Эйзел (Branko Ajzele)): http://inchoo.net/zend/zend-authentication-component-zend_auth-database-storage-class/.

Индексация, поиск и форматирование данных

Прочитав эту главу, вы:

- научитесь динамически создавать и выполнять поисковые запросы к базе данных;
- научитесь хранить и индексировать различные типы данных;
- создадите механизм полнотекстового поиска на основе компонента Zend_Search_Lucene;
- научитесь переключать тип вывода во время выполнения;
- представите один и тот же вывод в различных форматах, в том числе в формате XML.

Вы когда-нибудь видели губку в ведре с водой? Первые минуту-две, будучи по большей части сухой, губка легко и беззаботно плавает на поверхности. Но впитывая все больше и больше воды, она постепенно погружается, пока не станет совсем мокрой и готовой к использованию. Дружественное пользователям веб-приложение очень похоже на губку. После запуска оно представляет собой почти пустой каркас и ожидает, когда мир заметит его. Привлекая пользователей, приложение начинает впитывать данные, и в нем находят свое отражение положительные особенности Сети: чем больше людей его используют, тем более ценным оно является для остальных.

К чему все это? Дело в том, что по мере накопления вашим приложением пользовательского содержимого все более и более важными становятся возможность осуществления поиска по этому содержимому и представление его в различных форматах, чтобы пользователи могли использовать его в различных целях (например, наполняя его содержимым с других сервисов). Недостаточно просто предоставить список содержимого базы данных; необходимо также предложить пользователям разные способы поиска данных и фильтрации результатов, а также разные форматы их представления.

Именно этому посвящена данная глава. На следующих страницах вы узнаете, как добавить в веб-приложение возможность поиска, и увидите примеры как поиска по фильтрам, так и полнотекстового поиска. Также будет рассмотрен

ContextSwitch — вспомогательный класс Zend Framework, — предоставляющий гибкую и расширяемую систему для обработки нескольких типов вывода, включая XML и JSON.

Упражнение 6.1. Поиск и фильтрация записей базы данных

В предыдущей главе вы создали интерфейс, дающий пользователями возможность загружать элементы в каталог товаров, а администраторам — проверять эти элементы и выставлять их в открытый доступ. В этом разделе результаты, полученные в предыдущей главе, будут использованы для создания интерфейса, позволяющего пользователям осуществлять навигацию по подтвержденным элементам, а также их фильтрацию по различным критериям.

Определение формы поиска

Для нашего демонстрационного примера максимально упростим форму поиска, предположив, что пользователи будут искать марки только по трем критериям: *году*, *цене* и *состоянию*. Ниже приведено определение формы поиска:

```
<?php
class Square_Form_Search extends Zend_Form
{
 public $messages = array(
       Zend_Validate_Int::INVALID =>
           '\'%value%\' is not an integer',
       Zend_Validate_Int::NOT_INT =>
          '\'%value%\' is not an integer'
 );

 public function init()
 {
     // инициализируем форму
     $this->setAction('/catalog/item/search')
       ->setMethod('get');

   // устанавливаем декораторы формы
   $this->setDecorators(array(
     array('FormErrors',
       array('markupListItemStart' => '', 'markupListItemEnd' => '')),
     array('FormElements'),
     array('Form')
   ));

   // создаем текстовое поле для ввода года
   $year = new Zend_Form_Element_Text('y');
   $year->setLabel('Year:')
```

```
        ->setOptions(array('size' => '6'))
        ->addValidator('Int', false,
          array('messages' => $this->messages))
       ->addFilter('HtmlEntities')
       ->addFilter('StringTrim');

    // создаем текстовое поле для ввода цены
    $price = new Zend_Form_Element_Text('p');
    $price->setLabel('Price:')
        ->setOptions(array('size' => '8'))
        ->addValidator('Int', false,
           array('messages' => $this->messages))
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем список выбора для состояния
    $grade = new Zend_Form_Element_Select('g');
    $grade->setLabel('Grade:')
        ->addValidator('Int', false,
          array('messages' => $this->messages))
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim')
        ->addMultiOption('', 'Any');
  foreach ($this->getGrades() as $g) {
    $grade->addMultiOption($g['GradeID'], $g['GradeName']);
  };

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Search')
        ->setOptions(array('class' => 'submit'));

    // добавляем элементы к форме
    $this->addElement($year)
        ->addElement($price)
        ->addElement($grade)
        ->addElement($submit);

    // устанавливаем декораторы для элементов
    $this->setElementDecorators(array(
      array('ViewHelper'),
      array('Label', array('tag' => '<span>'))
    ));
    $submit->setDecorators(array(
      array('ViewHelper'),
    ));
  }

  public function getGrades()
```

```
  {
    $q = Doctrine_Query::create()
        ->from('Square_Model_Grade g');
    return $q->fetchArray();
  }
}
```

Это довольно стандартное определение формы, с которым вы уже знакомы: два поля для ввода текста, список выбора и кнопка отправки, помещенные в метод init(). Обратите внимание, что стандартный декоратор Label был изменен для помещения метки каждого элемента внутрь тега (вместо используемого по умолчанию <dt>), тем самым гарантируя, что элементы формы будут выводиться горизонтально, а не вертикально.

Стоит отметить еще один момент. До сих пор во всех предыдущих примерах ошибки валидации автоматически отображались пользователю под соответствующими элементами ввода. Однако зачастую в требованиях к приложению или пользовательскому интерфейсу указывается, что эти ошибки должны отображаться вместе, в одном блоке в верхней части формы, а не под отдельными элементами. Реализовать это несложно — существует встроенный декоратор FormErrors, который сделает все за вас, — но вам необходимо отключить декоратор Errors, по умолчанию прикрепленный к каждому элементу. Приведенное определение формы выполняет оба действия; оно добавляет декоратор FormErrors с помощью метода setDecorators() объекта Zend_Form, а затем, используя метод setElementDecorators(), удаляет декоратор Errors из списка, применяющегося к каждому элементу.

Разобравшись с определением формы, сохраните его в файле $APP_DIR/library/Square/Form/Search.php.

Определение контроллера и представления

Следующим шагом будет определение действия и представления для интерфейса навигации и поиска. Добавьте в класс Catalog_ItemController в файле $APP_DIR/modules/catalog/controllers/ItemController.php приведенный ниже метод searchAction():

```
<?php
class Catalog_ItemController extends Zend_Controller_Action

 public function searchAction()
 {
   // генерируем форму ввода
   $form = new Square_Form_Search;
   $this->view->form = $form;

   // проверяем корректность ввода,
   // генерируем базовый запрос
   if ($form->isValid($this->getRequest()->getParams())) {
     $input = $form->getValues();
     $q = Doctrine_Query::create()
```

```
            ->from('Square_Model_Item i')
            ->leftJoin('i.Square_Model_Country c')
            ->leftJoin('i.Square_Model_Grade g')
            ->leftJoin('i.Square_Model_Type t')
            ->where('i.DisplayStatus = 1')
            ->addWhere('i.DisplayUntil >= CURDATE()');

        // добавляем критерии к базовому запросу
        if (!empty($input['y'])) {
          $q->addWhere('i.Year = ?', $input['y']);
        }
        if (!empty($input['g'])) {
          $q->addWhere('i.GradeID = ?', $input['g']);
        }
        if (!empty($input['p'])) {
          $q->addWhere('? BETWEEN i.SalePriceMin AND i.SalePriceMax',
            $input['p']);
        }

        // выполняем запрос и добавляем результаты к представлению
         results = $q->fetchArray();
        $this->view->results = $results;
      }
    }
}
```

Сначала метод **searchAction()** формирует базовый запрос, возвращающий список всех подтвержденных элементов каталога. Затем, в зависимости от введенных пользователем данных, он добавляет к этому запросу условия **Where**, чтобы отфильтровать результирующий набор по заданным критериям. После выполнения запроса результаты присваиваются переменной представления. Если отправленные данные некорректны, переменной представления также присваивается список сообщений об ошибках валидации.

Сценарий представления отвечает за проход по списку результатов и форматирование каждой записи для отображения. Ошибки, если они есть, отображаются в верхней части страницы результатов с помощью декоратора **FormErrors**. Ниже представлен сценарий представления, который следует сохранить в файл **$APP_DIR/modules/catalog/views/scripts/item/search.phtml**:

```
<h2>Search</h2>
<?php echo $this->form; ?>

<h2>Search Results</h2>
<p><?php echo count($this->results); ?> result(s) found.</p>

<?php if (count($this->results)): ?>
  <?php $x=1; ?>
  <?php foreach ($this->results as $r): ?>

  <div>
```

```
  <?php echo $x; ?>.
    <a href="<?php echo $this->url(array('id' =>
      $this->escape($r['RecordID'])), 'catalog-display'); ?>">
      <?php echo $this->escape($r['Title']); ?>
      (<?php echo $this->escape($r['Year']); ?>)
    </a>
    <?php if (!empty($r['Description'])): ?>
    <br/><?php echo $this->escape($r['Description']); ?>
    <?php endif; ?>
    <br/>
    <strong>
    Grade:
    <?php echo $this->escape($r['Square_Model_Grade']['GradeName']);
?> |
    Country:
    <?php echo $this->escape($r['Square_Model_Country']
    ['CountryName']); ?> |
    Sale price:
    $<?php echo sprintf('%0.2f', $this->escape($r['SalePriceMin']));
?> to
    $<?php echo sprintf('%0.2f', $this->escape($r['SalePriceMax']));
?>
    </strong>
  </div>
  <br/>
    <?php $x++; ?>
  <?php endforeach; ?>
<?php endif; ?>

<div>
  <a href="<?php echo $this->url(array(
    'module' => 'catalog',
    'controller' => 'item',
    'action' => 'create'),
    null, true); ?>">Add Item</a>
</div>
```

Обновление основного макета

Единственное, что осталось сделать, — добавить к навигационным ссылкам в главном меню приложения ссылку на новую форму навигации/поиска, используя вспомогательный метод url(). Для этого внесите в основной макет, расположенный в файле $APP_DIR/application/layouts/master.phtml, изменения, выделенные жирным шрифтом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
~
    <div id="menu">
```

```
        <a href="<?php echo $this->url(array(), 'home'); ?>">HOME</a>
        <a href="<?php echo $this->url(array('page' => 'services'),
        'static-content'); ?>">SERVICES</a>
        <a href="<?php echo $this->url(array('module' => 'catalog',
        'controller' => 'item', 'action' => 'search'), 'default', true);
         ?>">CATALOG</a>
              <a href="<?php echo $this->url(array(), 'contact');
?>">CONTACT</a>
    </div>
  </div>
...
</html>
```

ПРИМЕЧАНИЕ

Обратите внимание на аргументы, передаваемые вспомогательному методу представления url() в предыдущем листинге. Второй аргумент, как вы уже знаете, определяет название маршрута, использующегося для генерации URL. Если этот аргумент имеет значение null, вспомогательный метод будет использовать маршрут, сгенерировавший текущий URL, в качестве основы для нового URL. Поскольку это может привести к трудноуловимым ошибкам при работе с маршрутами, которые сбрасывают запрос URL, в подобных случаях следует явно указать фреймворку на необходимость использования стандартного маршрута, default, соответствующего URL вида */модуль/контроллер/действие*.

Если вы сейчас откроете в веб-браузере URL http://square.localhost/catalog/item/search, то увидите форму, показанную на рис. 6.1.

Введите значения в текстовые поля и отправьте форму. Если все пойдет хорошо, вы увидите список элементов, соответствующих указанным критериям поиска (рис. 6.2).

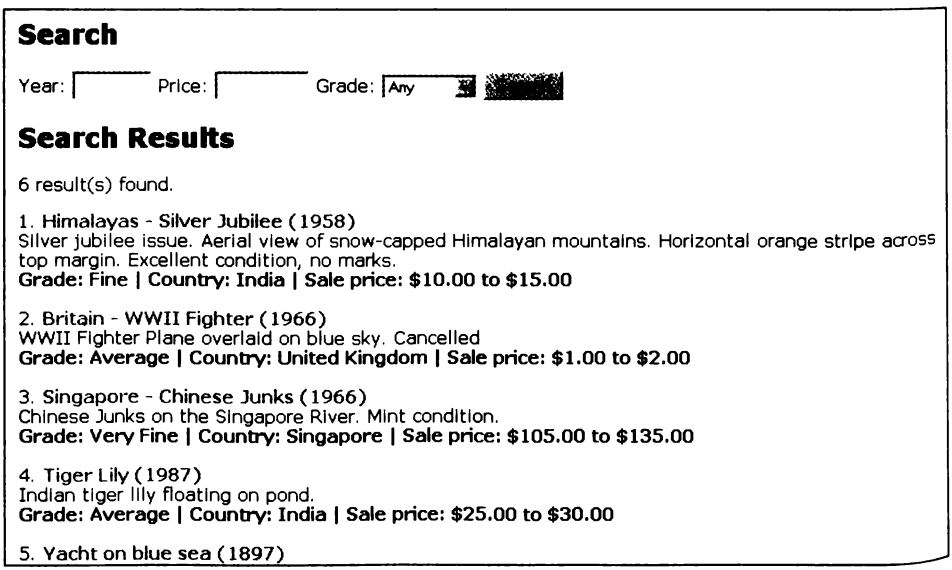


Рис. 6.1. Форма поиска по каталогу SQUARE

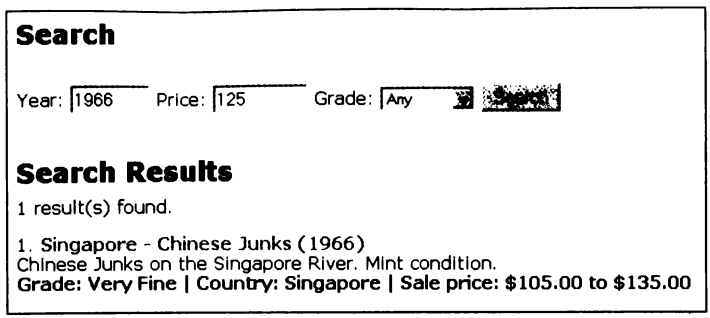


Рис. 6.2. Список элементов каталога, соответствующих критериям, указанным в форме поиска

Вы также можете попробовать ввести в форму некорректные значения. Встроенные валидаторы перехватят их и отобразят сообщения об ошибках в блоке, расположенном в верхней части страницы (рис. 6.3).

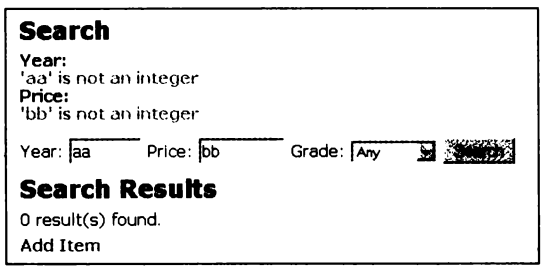


Рис. 6.3. Ошибки валидации введенных данных, перехваченные и показанные в отдельном блоке

Добавление полнотекстового поиска

При создании поискового механизма для приложения существует несколько вариантов. Можно сделать механизм, основанный на фильтрах, в котором для фильтрации результатов поиска используются определенные критерии, указанные пользователем. Или можно создать механизм полнотекстового поиска, который индексирует содержимое приложения и осуществляет поиск по ключевым словам, вводимым пользователем. В современных веб-приложениях предпочтение отдается последнему варианту, так как он работает быстрее и дает более точные результаты, нежели поиск по базе данных.

В состав Zend_Framework входит мощный компонент полнотекстового поиска, который называется Zend_Search_Lucene. Являясь PHP-реализацией Apache Lucene Project, этот компонент может использоваться как для индексации документов различных типов (включая текст, HTML и некоторые из форматов Microsoft Office 2007), так и для выполнения разнообразных поисковых запросов к индексированным данным.

Реализация полнотекстового поиска с использованием Zend_Search_Lucene включает в себя две основные операции: *индексацию данных* и *поиск данных*. В процессе индексации осуществляется анализ набора документов и создание индекса их содержимого; при поиске этот индекс используется для нахождения документов, удовлетворяющих заданным пользователем критериям. В следующих разделах дается общее представление об этих операциях.

Индексация данных

Индекс, создаваемый Zend_Search_Lucene, составляется из отдельных документов, каждый из которых в свою очередь может быть разделен на поля. При построении индекса Zend_Search_Lucene дает пользователям возможность детально указать, каким образом следует обрабатывать каждое поле документа. Двумя основными параметрами здесь являются *индексация* и *сохранение*: индексированные поля можно использовать для поиска, а сохраненные поля могут отображаться в его результатах. Существуют пять основных типов полей:

- поля Keyword не снабжаются метками, однако индексируются и сохраняются в индексе;
- поля Text снабжаются метками, индексируются и сохраняются;
- поля UnStored снабжаются метками и индексируются, но не сохраняются;
- поля UnIndexed снабжаются метками и сохраняются, но не индексируются;
- поля Binary не снабжаются метками и не индексируются, но сохраняются.

При определении того, какие типы полей следует использовать в индексе Zend_Search_Lucene, важно четко понимать, какие поля будут использоваться в качестве критериев поиска, а какие — отображаться в его результатах. Чтобы продемонстрировать это на практике, рассмотрим коллекцию XML-документов в следующем формате:

```
<?xml version='1.0'?>
<document>
  <id>5468</id>
  <from>Jim Doe <jim@example.com></from>
  <to>Jane Doe <jane@example.com></to>
  <subject>Re: Hello</subject>
  <date>Tuesday, February 27, 2008 10:45 PM</date>
  <body>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
  sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
  Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
  nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
  in reprehenderit in-voluptate velit esse cillum dolore eu fugiat
  nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt
  in culpa qui officia deserunt mollit anim id est laborum</body>
</document>
```

Далее приведен пример кода, который показывает, как можно проиндексировать коллекцию этих документов:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function indexAction()
  {
    // получаем каталог для индекса
    $index = Zend_Search_Lucene::create('/tmp/indexes');
    foreach (glob('*.xml') as $file) {
      // читаем исходный xml-файл
      $xml = simplexml_load_file($file);

      // создаем новый индексированный документ
      $doc = new Zend_Search_Lucene_Document();

      // поля для индексации и сохранения
      $doc->addField(Zend_Search_Lucene_Field::UnIndexed('id', $xml->id));
      $doc->addField(Zend_Search_Lucene_Field::Text('from', $xml->from));
      $doc->addField(Zend_Search_Lucene_Field::Text('to', $xml->to));
      $doc->addField(Zend_Search_Lucene_Field::Text(
      'date', strtotime($xml->date)));
      $doc->addField(Zend_Search_Lucene_Field::UnStored(
      'subject', $xml->subject));
      $doc->addField(Zend_Search_Lucene_Field::UnStored('body', $xml->body));

      // сохраняем результат в индексе
      $index->addDocument($doc);
    }

    // устанавливаем число документов в индексе
    $count = $index->count();
  }
}
```

Каждый документ в индексе представлен объектом **Zend_Search_Lucene_Document**, а отдельные поля каждого сообщения — объектами **Zend_Search_Lucene_Field**. Эти поля индексируются и добавляются в документ методом **Zend_Search_Lucene_Document::addField()**, а затем с помощью метода **Zend_Search_Lucene::addDocument()** весь документ помещается в индекс.

Определение полей, по которым будет осуществляться поиск, и полей, которые должны отображаться в его результатах, — основная задача при индексации данных. В предыдущем примере поиск осуществляется по полям **recipient** (получатель), **date** (дата), **subject** (тема) и **body** (текст сообщения), но в результатах поиска будут отображаться только поля **recipient** и **date**. Выбрав для полей **body** и **subject** тип **UnStored**, мы оставили возможность поиска по ним, в то же время уменьшив объем дискового пространства, занимаемого индексом, а выбор типа **Text** для полей **recipient** и **date** делает возможным как поиск по ним, так и отображение их в результатах.

Поиск данных

Следующим шагом после завершения индексации данных будет осуществление поиска по ним. В состав **Zend_Search_Lucene** входит полнофункциональный механизм запросов, который можно использовать для выполнения как простых, так и сложных запросов к индексу. Запросы можно создавать либо с помощью встроенного анализатора пользовательского ввода, либо генерируя их программно, используя методы API. Для простых запросов по ключевым словам подходит стандартный анализатор; более сложные запросы, содержащие модификаторы, ограничения близости (**proximity**), подзапросы или группы полей, должны создаваться с использованием методов API.

Ниже приведен пример кода, показывающего, как осуществлять поиск по коллекции индексированных документов и отображать подходящие результаты:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function searchAction()
  {
    // получаем строку запроса из переменной $_GET['query']
    // если строка корректна, открываем индекс и анализируем ее
    // выполняем запрос и возвращаем массив полученных объектов
    $input->setData($this->getRequest()->getParams());
    if ($input->isValid()) {
      $index = Zend_Search_Lucene::open('/tmp/indexes');
      $results = $index->find(
        Zend_Search_Lucene_Search_QueryParser::parse($input->query));
      $this->view->results = $results;
    }
  }
}
```

Как показано в этом фрагменте кода, сначала необходимо открыть дескриптор индекса, созданного ранее методом **Zend_Search_Lucene::open()**. Затем строка запроса передается методу **find()** объекта **Zend_Search_Lucene**, чтобы просмотреть индекс на предмет подходящих документов. Совпадения возвращаются в виде объектов **Zend_Search_Lucene_Search_QueryHit** и ранжируются по рейтингу. Номер документа и его рейтинг представлены в виде свойств этих объектов. Как показано в предыдущем примере, через свойства можно также получить доступ и к другим сохраненным полям документа. Значения всех полей автоматически кодируются в UTF-8.

Строка запроса, переданная методу **find()**, может содержать ключевые слова, фразы, групповые символы (wildcards), ограничения диапазона, модификаторы близости и логические операции, такие как **AND**, **OR** и **NOT**. Если строка запроса передается непосредственно пользователем, правильнее будет обработать ее методом **Zend_Search_Lucene_Search_QueryParser::parse()**, который автоматически проанализирует и разметит строку, а затем преобразует ее в набор объектов запроса. Кроме того, такой подход уменьшает объем требуемой валидации входных данных.

ВОПРОС ЭКСПЕРТУ

В: Как мне отсортировать или ограничить результаты поиска, полученные с помощью Zend_Search_Lucene?

О: По умолчанию Zend_Search_Lucene возвращает все совпадающие документы, отсортированные по рейтингу. Если вам это не подходит, вы можете определить поле для сортировки и ее порядок, передав в метод find() дополнительные аргументы. Ограничить число возвращаемых совпадений можно, указав предельное значение с помощью метода setResultSetLimit().

ПРИМЕЧАНИЕ

Ограниченный объем этой книги не позволяет полностью рассмотреть язык запросов, поддерживаемый Zend_Search_Lucene. Однако подробную информацию по этой теме вы можете найти по ссылкам, приведенным в конце главы.

Упражнение 6.2. Создание механизма полнотекстового поиска

Обладая всей необходимой информацией, давайте посмотрим, как Zend_Search_Lucene работает на практике. В следующем разделе механизм поиска по фильтрам для приложения SQUARE, созданный ранее в этой главе, будет заменен системой полнотекстового поиска, основанной на Zend_Search_Lucene.

Определение расположения индекса

Индексы полнотекстового поиска Zend_Search_Lucene хранятся в виде файлов на диске, поэтому первоочередной задачей будет определение месторасположения этих файлов. Создадим каталог $APP_DIR/data/indexes/, который в рекомендованной структуре каталогов Zend Framework по умолчанию используется для файлов индекса.

```
shell> cd /usr/local/apache/htdocs/square
shell> mkdir data
shell> mkdir data/indexes
```

Также неплохо будет добавить это расположение в конфигурационный файл приложения, $APP_DIR/application/configs/application.ini, чтобы его можно было использовать в действиях. Откройте файл и добавьте в него следующую конфигурационную директиву:

```
indexes.indexPath = APPLICATION_PATH "/../data/indexes"
```

Определение пользовательских маршрутов

Поскольку вы решили обновить конфигурационный файл, добавьте в него маршрут для действия createFulltextIndexAction(). Создание индекса относится к административным задачам, и, следовательно, это действие должно находиться в контрол-

лере Catalog_AdminItemController. Ниже приведено соответствующее определение маршрута:

```
resources.router.routes.admin-fulltext-index-create.route = /admin/catalog/
fulltext-index/create
resources.router.routes.admin-fulltext-index-create.defaults.module = catalog
resources.router.routes.admin-fulltext-index-create.defaults.controller = admin.
item
resources.router.routes.admin-fulltext-index-create.defaults.action = create.
fulltext.index
```

Определение действия и представления для индексации

Теперь необходимо создать полнотекстовый индекс содержимого базы данных приложения. Проще всего это сделать, выполнив запрос Doctrine для получения записей из базы, а затем передав их в цикле индексатору Zend_Search_Lucene. Пример кода метода createFulltextIndexAction(), решающего эту задачу:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
  // действие для создания полнотекстовых индексов
  public function createFulltextIndexAction()
  {
    // создаем и выполняем запрос
    $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Type t')
        ->where('i.DisplayStatus = 1')
        ->addWhere('i.DisplayUntil >= CURDATE()');
    $result = $q->fetchArray();

    // получаем каталог для индекса
    $config = $this->getInvokeArg('bootstrap')->getOption('indexes');
    $index = Zend_Search_Lucene::create($config['indexPath']);
    foreach ($result as $r) {
      // создаем новый индексируемый документ
      $doc = new Zend_Search_Lucene_Document();
      // поля для индексации и сохранения
      $doc->addField(Zend_Search_Lucene_Field::Text('Title',
        $r['Title']));
      $doc->addField(
        Zend_Search_Lucene_Field::Text('Country',
      $r['Square_Model_Country']['CountryName']));
        $doc->addField(
      Zend_Search_Lucene_Field::Text('Grade',
        $r['Square_Model_Grade']['GradeName']));
      $doc->addField(Zend_Search_Lucene_Field::Text('Year',
        $r['Year']));
```

```
        $doc->addField(Zend_Search_Lucene_Field::UnStored(
          'Description', $r['Description']));
        $doc->addField(Zend_Search_Lucene_Field::UnStored(
          'Denomination', $r['Denomination']));
        $doc->addField(Zend_Search_Lucene_Field::UnStored(
          'Type', $r['Square_Model_Type']['TypeName']));
        $doc->addField(Zend_Search_Lucene_Field::UnIndexed(
          'SalePriceMin', $r['SalePriceMin']));
        $doc->addField(Zend_Search_Lucene_Field::UnIndexed(
          'SalePriceMax', $r['SalePriceMax']));
        $doc->addField(Zend_Search_Lucene_Field::UnIndexed(
          'RecordID', $r['RecordID']));

        // сохраняем результат в индексе
        $index->addDocument($doc);
        }

      // устанавливаем число документов в индексе
      $count = $index->count();
      $this->_helper->getHelper('FlashMessenger')
          ->addMessage("The index was successfully created
                  with $count documents.");
        $this->_redirect('/admin/catalog/item/success');
    }
}
```

Здесь метод **createFulltextIndexAction()** сначала выполняет запрос **Doctrine** для получения списка всех подтвержденных элементов каталога. Затем он инициализирует новый индекс **Zend_Search_Lucene**, указав для него расположение, определенное на предыдущем этапе (обратите внимание на получение значения из конфигурационного файла с помощью метода **getInvokeArg()**), после чего проходит в цикле по всем результатам запроса и передает каждый из них индексатору в качестве отдельного документа. В каждом таком документе индексируются название, описание, страна, состояние, год, номинал и тип марки; при этом для отображения в результатах поиска в индексе сохраняются только название, страна, состояние и год, а также идентификатор записи и диапазон цен. После завершения индексации генерируется представление, сообщающее об успехе и количестве проиндексированных документов.

Обновление сводного представления

Обновите административную сводную страницу в файле **$APP_DIR/application/modules/catalog/views/scripts/admin-item/index.phtml**, добавив в нее ссылку на новое действие. Изменения в сценарии представления выделены жирным шрифтом.

```
<h2>List Items</h2>
<?php if (count($this->records)): ?>
<div id="records">
  <form method="post" action="<?php echo $this->url(array(),
    'admin-catalog-delete'); ?>">
```

```
    <table>

      <tr>
        <td colspan="7"><input type="submit" name="submit"
    style="width:150px" value="Delete Selected" /></td>
<td colspan="2"><a href="<?php echo $this->url(array(),
        'admin-fulltext-index-create'); ?>">Update full-text    indices</a></td>
      </tr>
    </table>
  </form>
</div>

<?php else: ?>
No records found
<?php endif; ?>
```

Обновление формы поиска

После того как мы решили вопрос с индексацией, осталось разобраться с поиском. Для начала изменим определение формы поиска в файле **$APP_DIR/library/Square/Form/Search.php**, оставив в нем только одно текстовое поле для ввода ключевых слов. Исправленное определение формы:

```
<?php
class Square_Form_Search extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/catalog/item/search')
        ->setMethod('get');

    // создаем текстовое поле для ввода ключевых слов
    $query = new Zend_Form_Element_Text('q');
    $query->setLabel('Keywords:')
        ->setOptions(array('size' => '20'))
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');
    $query->setDecorators(array(
          array('ViewHelper'),
          array('Errors'),
          array('Label', array('tag' => '<span>')),
      ));

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Search')
        ->setOptions(array('class' => 'submit'));
    $submit->setDecorators(array(
          array('ViewHelper'),
      ));
```

```
    // добавляем элементы к форме
    $this->addElement($query)
         ->addElement($submit);

    }
}
```

Обновление действия и представления для поиска

Последнее, что осталось сделать, — задействовать полнотекстовый индекс **Zend_Search_Lucene** вместо запроса **Doctrine** в методе **Catalog_ItemController::searchAction**. Исправленный код:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
{
  // действие для выполнения полнотекстового поиска
  public function searchAction()
  {
    // генерируем форму ввода
    $form = new Square_Form_Search;
    $this->view->form = $form;

    // получаем элементы, соответствующие критериям поиска
    if ($form->isValid($this->getRequest()->getParams())) {
      $input = $form->getValues();
      if (!empty($input['q'])) {
        $config = $this->getInvokeArg('bootstrap')->getOption('indexes');
    $index = Zend_Search_Lucene::open($config['indexPath']);
    $results = $index->find(
          Zend_Search_Lucene_Search_QueryParser::parse($input['q']));
        $this->view->results = $results;
      }
    }
  }
}
```

Значение, возвращаемое методом **find()** класса **Zend_Search_Lucene**, отличается от возвращаемого методом **fetchArray()** в **Doctrine**; значения полей теперь представлены как свойства объектов, а не как элементы массива. Следовательно, необходимо изменить соответствующий сценарий представления, **$APP_DIR/modules/catalog/views/scripts/item/search.phtml**, следующим образом:

```
<h2>Search</h2>
<?php echo $this->form; ?>

<h2>Search Results</h2>
<p><?php echo count($this->results); ?> result(s) found.</p>
  <?php if (count($this->results)): ?>
  <?php $x=1; ?>
  <?php foreach ($this->results as $r): ?>
  <div>
    <?php echo $x; ?>.
```

```
    <a href="<?php echo $this->url(
    array('id' => $this->escape($r->RecordID)),
    'catalog-display'); ?>">
  <?php echo $this->escape($r->Title); ?>
  (<?php echo $this->escape($r->Year); ?>)
  </a>
  <br/>
  Score: <?php printf('%1.4f', $this->escape($r->score)); ?>
  <br/>
  <strong>
  Grade: <?php echo $this->escape($r->Grade); ?> |
  Country: <?php echo $this->escape($r->Country); ?> |
  Sale price:
  $<?php echo sprintf('%0.2f', $this->escape($r->SalePriceMin)); ?>
to
  $<?php echo sprintf('%0.2f', $this->escape($r->SalePriceMax)); ?>
  </strong>
</div>
<br/>
    <?php $x++; ?>
  <?php endforeach; ?>
<?php endif; ?>

<div>
    <a href="<?php echo $this->url(array(
  'module' => 'catalog',
  'controller' => 'item',
  'action' => 'create'),
  null, true); ?>">Add Item</a>
</div>
```

Чтобы проверить работу поиска, войдите в панель администрирования SQUARE и создайте полнотекстовые индексы, перейдя по URL **http://square.localhost/admin/catalog/fulltext-index/create**. После завершения создания индекса вы должны увидеть страницу с сообщением об успехе (рис. 6.4).

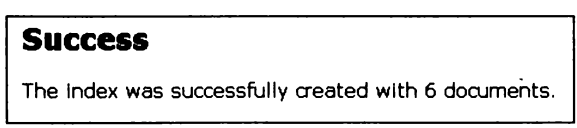


Рис. 6.4. Результат успешного создания полнотекстового индекса каталога

Теперь можно попробовать произвести поиск по индексу, открыв в браузере URL **http://square.localhost/catalog/item/search**. Вы должны увидеть измененную форму поиска (рис. 6.5).

Введите в текстовое поле одно или несколько ключевых слов и отправьте. Если все работает правильно, вы увидите список элементов, соответствующих этим ключевым словам (рис. 6.6).

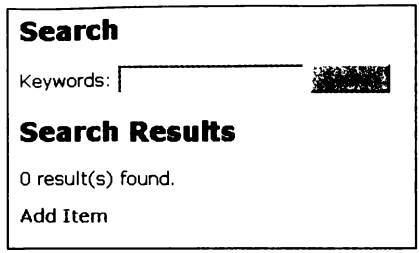


Рис. 6.5. Измененная форма поиска по каталогу SQUARE

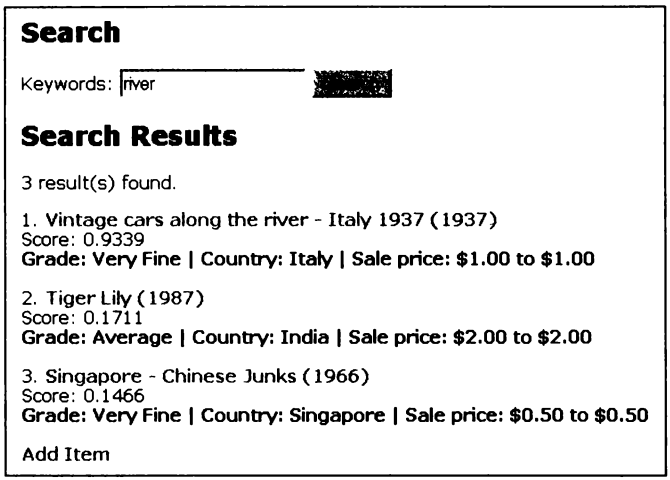


Рис. 6.6. Список элементов каталога, соответствующих критерию полнотекстового поиска, указанному в форме

ВОПРОС ЭКСПЕРТУ

В: Что лучше: полнотекстовый поиск с использованием хранящихся на диске индексов или поиск, основанный на фильтрах и использующий запросы к базе данных?

О: Обычно полнотекстовый поиск дает лучшие результаты, чем поиск по базе данных, по ряду причин. Мы не рекомендуем предоставлять пользователям возможность осуществлять поиск по *каждому* параметру записи в базе данных, к тому же возможности полнотекстового поиска, встроенные в большинство РСУБД, слишком примитивны или недостаточно эффективны по сравнению со средствами, изначально предназначенными для этих целей, такими как Lucene, Sphinx или Solr. Полнотекстовый поиск обычно быстрее поиска по базе данных, поскольку осуществляется с использованием индексов, хранящихся на диске, а не через подключение к базе. Однако полнотекстовый поиск основывается на статических индексах, созданных при анализе исходного содержимого, поэтому в случае частых обновлений последнего может возникать задержка между обновлением содержимого и появлением его в результатах поиска. При поиске по базе данных такая задержка отсутствует, так как он осуществляется непосредственно по содержимому. Поэтому, как и в случае с большинством вопросов, ответ на этот является субъективным и зависит от того, какого баланса между производительностью и точностью вы хотите достигнуть.

Обработка нескольких типов вывода

Современные веб-приложения говорят не только на языке HTML: они также могут работать с форматами RSS, JSON, XML и многими другими. Поскольку паттерн MVC отделяет данные от представлений, он идеально подходит для решения таких задач; поддержка нового типа вывода сводится к созданию нового представления, содержащего соответствующий код форматирования и/или разметку. И что более важно, эти изменения локализованы в слое представлений и могут быть сделаны без внесения каких бы то ни было модификаций в слои контроллеров и моделей.

В состав Zend Framework входит встроенный обработчик нескольких типов вывода, или *контекстов*, — вспомогательный класс **ContextSwitch**. Он решает все задачи, связанные с переключением на другой формат вывода, к которым относятся отключение стандартного макета, выбор подходящего сценария представления и отправка корректных заголовков для выбранного формата.

Чтобы показать, как это работает, рассмотрим следующий пример, создающий контекст для вывода в формате YAML:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
  {
  public function init()
{
    // инициализируем вспомогательный класс для переключения контекста
    $contextSwitch = $this->_helper->getHelper('contextSwitch');
    $contextSwitch->addContext(
      'yaml',
      array(
      'suffix' => 'yaml',
      'headers' => array('Content-Type' => 'text/yaml')
    )
  );
    $contextSwitch->addActionContext('list', 'yaml')
        ->initContext();
    }
}
```

Добавление поддержки нового типа вывода с помощью вспомогательного класса **ContextSwitch** состоит из четырех основных этапов:

1. *Определить параметры контекста.* Вспомогательный класс **ContextSwitch** содержит метод **addContext()**, который можно использовать для определения нового контекста. Этот метод принимает два аргумента: строку, содержащую название нового типа вывода, и массив с дополнительной конфигурационной информацией, такой как заголовки, отправляемые клиенту, и суффикс для имени файла, используемый при поиске сценариев представления. В предыдущем примере в методе **addContext()** использовался суффикс **.yaml** и заголовок **'Content-Type'** со значением **text/yaml**.

ПРИМЕЧАНИЕ

При работе с предопределенными контекстами XML и JSON этот шаг можно пропустить.

2. *Привязать контекст к одному или нескольким действиям.* После определения типа вывода можно использовать метод addActionContext() вспомогательного класса ContextSwitch, чтобы сопоставить этот контекст одному или нескольким действиям. В предыдущем примере с помощью метода addActionContext() было указано, что listAction() может возвращать данные в формате YAML.

СОВЕТ

Если вы работаете с несколькими различными контекстами и действиями, рассмотрите возможность использования метода addActionContexts(), который принимает массив, содержащий пары контекст-действие, и является быстрым способом сопоставления нескольких контекстов и действий с помощью единственного вызова метода.

3. *Создать сценарий представления для контекста.* Следующим шагом будет создание сценария представления для каждого действия и контекста. Сценарий содержит весь код для форматирования, разметку и бизнес-логику, необходимую для генерации вывода в заданном формате, и по умолчанию к его имени добавляется суффикс, указанный в методе addContext(). В предыдущем примере генерация представления в формате YAML на основе данных, выводимых методом listAction(), осуществлялась бы сценарием list.yaml.phtml. Если бы требовалось выводить данные в формате XLS, сценарий назывался бы list.xls.phtml и содержал код на PHP, генерирующий электронные таблицы в формате Microsoft Excel.
4. *Инициализировать контекст.* Это последний и самый простой шаг, так как он всего лишь подразумевает вызов метода initContext() вспомогательного класса ContextSwitch. Когда метод вызывается без аргументов, он инициализирует все определенные контексты и выполняет всю необходимую работу. Теперь вы можете добавить к URL действия специальный параметр ?format=имя-контекста, чтобы указать вспомогательному классу ContextSwitch требуемый формат вывода: например: ?format=yaml для YAML или ?format=xls для XLS.

Упражнение 6.3. Вывод результатов поиска в формате XML

Чтобы продемонстрировать работу вспомогательного класса ContextSwitch на практике, давайте усовершенствуем созданный ранее механизм поиска, добавив в него поддержку вывода в формате XML. В следующих разделах показан процесс модификации.

Включение контекста XML

Поскольку определения контекстов XML и JSON встроены во вспомогательный класс **ContextSwitch**, нет необходимости определять их с помощью метода **addContext()**. Вместо этого можно непосредственно перейти к сопоставлению контекста XML методу **searchAction()** с помощью метода **addActionContext()**. Добавьте вызов этого метода в метод **init()** класса **Catalog_ItemController**, как показано в примере:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
{
  public function init()
  {
    // инициализируем вспомогательный класс для переключения контекста
    $contextSwitch = $this->_helper->getHelper('contextSwitch');
    $contextSwitch->addActionContext('search', 'xml')
         ->initContext();
    }
}
```

Определение представления XML

После сопоставления и инициализации контекста XML все запросы на получение результатов поиска в формате XML будут обрабатываться соответствующим сценарием представления XML. Как обычно, это представление следует сохранить в файле **$APP_DIR/application/modules/catalog/views/scripts/item/search.xml.phtml**. Пример его содержимого:

```
<?php
// создаем документ XML
$dom = new DOMDocument('1.0', 'utf-8');

  // создаем корневой элемент
  $root = $dom->createElementNS(
    'http://square.localhost', 'square:document');
  $dom->appendChild($root);

  // преобразуем его в объект SimpleXML
  $xml = simplexml_import_dom($dom);

  // добавляем элемент со сводной информацией
  $xml->addChild('matches', count($this->results));
  $xml->addChild('searchTime', time());

  // добавляем элементы, представляющие результаты поиска
    $results = $xml->addChild('results');
    foreach ($this->results as $r) {
    $result = $results->addChild('result');
    $result->addChild('score', $this->escape($r->score));
    $result->addChild('id', $this->escape($r->RecordID));
```

```
        $result->addChild('title', $this->escape($r->Title));
        $result->addChild('year', $this->escape($r->Year));
        $result->addChild('grade', $this->escape($r->Grade));
        $result->addChild('country', $this->escape($r->Country));
        $price = $result->addChild('price');
        $price->addChild('min', $this->escape($r->SalePriceMin));
        $price->addChild('max', $this->escape($r->SalePriceMax));
    }

    // возвращаем выходные данные
    echo $xml->asXML();
?>
```

Если вы хорошо знакомы с такими расширениями PHP, как SimpleXML и DOM, вам будет нетрудно разобраться в приведенном сценарии. Он начинается с использования расширения DOM, с помощью которого генерируется новый XML-документ и его пролог и определяется пользовательское пространство имен для следующих за ним XML-данных. После этого документ преобразуется в объект SimpleXML (для удобства), и результат, возвращаемый методом **searchAction()**, обрабатывается и разбивается на последовательности XML-элементов. После обработки всех результатов поиска используется метод **asXML()**, который отправляет клиенту результирующее дерево XML-документа.

Чтобы проверить работу кода, попробуйте осуществить поиск, перейдя по URL **http://square.localhost/catalog/item/search** и указав набор ключевых слов. Вы должны увидеть результаты поиска в формате HTML (рис. 6.6).

Теперь запросите те же самые результаты поиска в формате XML, добавив к строке URL дополнительный параметр **?format=xml**. Обнаружив этот параметр, вспомогательный класс **ContextSwitch** автоматически переключит тип вывода и предоставит результаты поиска в формате XML (рис. 6.7).

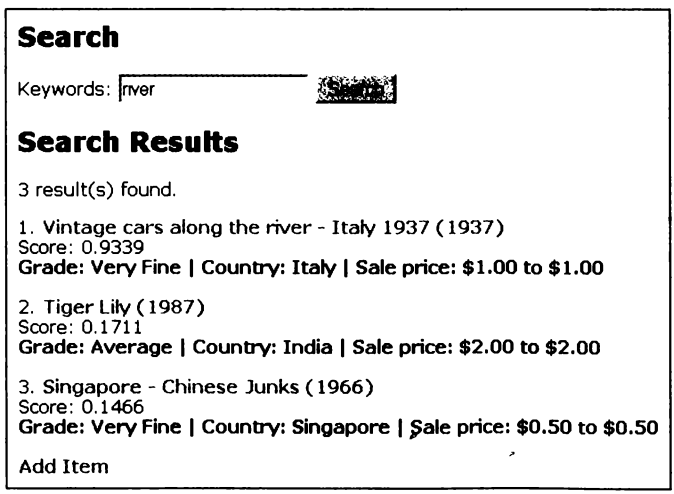


Рис. 6.7. Результаты поиска, представленные в формате XML

СОВЕТ

Параметр 'format', передаваемый вспомогательному классу ContextSwitch, необязательно должен являться параметром запроса GET. Он может передаваться из любого подходящего источника, включая само определение маршрута.

Выводы

Как только в вашем приложении появляются основные операции CRUD, пользователи и администраторы могут беспрепятственно добавлять в него данные. Это порождает новую проблему, связанную с доступом к данным, поиском по ним и представлением их в нескольких распространенных форматах. Данная глава была посвящена именно этой проблеме; в ней были рассмотрены процесс добавления в приложение функций поиска и использование компонента Zend_Search_Lucene. Также был показан вспомогательный класс ContextSwitch, упрощающий поддержку различных форматов вывода, таких как XML, RSS, JSON и других.

Чтобы узнать больше о темах, рассмотренных в этой главе, посетите следующие ссылки:

- Компонент Zend_Search_Lucene: http://framework.zend.com/manual/en/zend.search.lucene.html.
- Язык запросов Zend_Search_Lucene: http://framework.zend.com/manual/en/zend.search.lucene.query-language.html.
- Вспомогательный класс ContextSwitch: http://framework.zend.com/manual/en/zend.controller.actionhelpers.html.
- Расширение PHP SimpleXML: http://www.php.net/simplexml.
- Расширение PHP DOM: http://www.php.net/dom.
- Использование вспомогательного класса ContextSwitch для вывода электронных таблиц Microsoft Excel (Пабло Викез (Pablo Viquez)): http://www.pabloviquez.com/2009/08/export-excel-spreadsheetsusing-zend-framework/.
- Использование вспомогательного класса ContextSwitch для смены макетов на лету (Фил Браун (Phil Brown)): http://morecowbell.net.au/2009/02/changing-layouts-with-zend-contextswitch/.
- Создание REST API с помощью вспомогательного класса ContextSwitch (Крис Дэниэлсон (Chris Danielson)): http://www.chrisdanielson.com/2009/09/02/creating-a-php-rest-api-using-the-zendframework/.
- Сравнение индексации сообщений электронной почты с использованием Zend_Search_Lucene и Sphinx (Викрам Васвани (Vikram Vaswani)): http://devzone.zend.com/article/4887-Indexing-Email-Messages-with-PHP-Zend-Lucene-and-Sphinx.

7 Разбиение на страницы, сортировка и загрузка данных на сервер

Прочитав эту главу, вы:

- научитесь разбивать на страницы и сортировать результаты запросов к базе данных;
- научитесь осуществлять фильтрацию, валидацию и обработку файлов, загруженных на сервер через веб-формы;
- научитесь читать и записывать конфигурационные файлы в форматах INI и XML.

При разработке приложения одна из наиболее важных задач заключается в том, чтобы определить, какие аспекты его поведения должны настраиваться пользователем. Например, при создании системы управления содержимым вы, вероятно, захотите предоставить администраторам возможность решать, каким образом должны отображаться значения даты и времени, можно ли прикреплять к сообщениям изображения и должны ли комментарии рассматриваться и подтверждаться модератором. Однако вы, наверное, *не* захотите, чтобы администратор менял форматы, в которых хранятся данные, или способы обработки исключений, поскольку эти аспекты являются внутренними для приложения и (как правило) не предназначены для изменения на пользовательском уровне.

Определение аспектов приложения, которые должны настраиваться пользователем, — не столь простая задача, как может показаться. Чтобы правильно провести анализ, разработчик должен полностью понимать как цели приложения, так и требования конечных пользователей, которым оно должно соответствовать. Тем не менее после завершения анализа все становится намного проще. Выявленные переменные можно переместить в отдельную область хранения (обычно один или несколько конфигурационных файлов), и задача разработчика теперь состоит в том, чтобы предоставить интерфейс, посредством которого администраторы смогут изменять эти переменные во время выполнения программы.

Zend Framework содержит компонент Zend_Config, который предоставляет полнофункциональный API для чтения и записи конфигурационных файлов в различных форматах. Данная глава детально рассматривает этот компонент и его использование для управления конфигурационными параметрами в контексте демонстрационного приложения SQUARE. В ней также повторно рассмотрены два компонента из предыдущих глав, Zend_Form и Doctrine, и показано, как их можно применить, чтобы удовлетворить два основных требования: возможность загружать файлы на сервер, а также разбиение на страницы и сортировка больших наборов данных.

Упражнение 7.1. Разбиение на страницы и сортировка записей базы данных

При обработке в веб-приложении больших объемов данных обычно считается *дурным тоном* просто выводить все доступные данные на одну страницу и оставлять пользователя разбираться с ними. Разбиение больших наборов данных на более мелкие, то есть на *страницы*, — это распространенный шаблон проектирования пользовательских интерфейсов, нацеленный на улучшение читаемости и навигации. Важность разбиения на страницы обусловлена тем, что это предоставляет пользователю возможность выбирать, какая часть данных будет отображаться в данный момент, и не позволяет ему заблудиться в их бесконечном потоке. При использовании баз данных разбиение на страницы также способствует уменьшению нагрузки на сервер благодаря уменьшению количества результатов выборки.

Интересной особенностью разбиения на страницы является то, что программный код, реализующий его, стандартен и вряд ли будет изменяться между проектами. Следовательно, идеальным вариантом будет реализовать его в виде компонента для многократного использования, и фактически большинство фреймворков и слоев абстракции баз данных уже содержат готовые компоненты для разбиения на страницы. Например, в состав Zend Framework входит компонент Zend_Paginator, содержащий адаптеры для различных источников данных (в том числе и для Zend_Db). Doctrine тоже содержит компонент Doctrine_Pager, который можно использовать для непосредственной обработки и разбиения на страницы запросов Doctrine.

Между Zend_Paginator и Doctrine_Pager нет существенной разницы: оба компонента разделяют один набор данных на более мелкие и генерируют навигационные ссылки для двустороннего перемещения по этим наборам. Однако поскольку в большинстве случаев разбиение на страницы связано с результатами выборки из базы данных, используйте компонент, рекомендованный или входящий в состав слоя абстракции баз данных. В этой книге для всех операций с базой данных рекомендуется использовать Doctrine, поэтому далее будут рассматриваться преимущественно компоненты Doctrine_Pager и Doctrine_Query, а также их использование для разбиения на страницы и сортировки результатов выборки из базы данных для демонстрационного приложения SQUARE.

Добавление номеров страниц к маршрутам

Стандартный процесс работы компонента разбиения на страницы довольно прост. Сначала в компоненте указывается, сколько элементов будут отображаться на каждой странице. Затем вычисляется общее число страниц, для этого количество элементов в наборе данных делится на количество элементов на каждой странице, после чего динамически генерируется набор навигационных ссылок для перемещения между страницами. Каждая ссылка содержит дополнительный GET-параметр (номер страницы), который используется контроллером для определения запрашиваемого набора данных.

Чтобы посмотреть, как это работает, откройте конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**, и внесите следующие изменения в определение маршрута для административной сводной страницы:

```
resources.router.routes.admin-catalog-index.route = /admin/catalog/item/
index/:page
resources.router.routes.admin-catalog-index.defaults.module = catalog
resources.router.routes.admin-catalog-index.defaults.controller = admin.item
resources.router.routes.admin-catalog-index.defaults.action = index
resources.router.routes.admin-catalog-index.defaults.page = 1
```

Обновление контроллера и представления для главной страницы

Следующим шагом будет изменение **Catalog_AdminItemController::indexAction** для того, чтобы действие использовало этот параметр при формировании запроса **SELECT**. Здесь в дело вступает компонент **Doctrine_Pager**. Пример обновленного кода действия:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
// действие для отображения списка элементов каталога
public function indexAction()
{
  // устанавливаем фильтры и валидаторы для данных,
  // полученных в запросе GET
  $filters = array(
    'page' => array('HtmlEntities', 'StripTags', 'StringTrim')
  );
  $validators = array(
    'page' => array('Int')
  );
  $input = new Zend_Filter_Input($filters, $validators);
  $input->setData($this->getRequest()->getParams());

  // проверяем корректность входных данных,
  // создаем запрос и устанавливаем параметры компонента
  // для разбиения на страницы
```

```
  if ($input->isValid()) {
    $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Type t');

    $perPage = 5;
    $numPageLinks = 5;

    // инициализируем компонент для разбиения на страницы
    $pager = new Doctrine_Pager($q, $input->page, $perPage);

    // выполняем запрос, учитывающий номер страницы
    $result = $pager->execute(array(), Doctrine::HYDRATE_ARRAY);

    // инициализируем макет компонента для разбиения на страницы
    $pagerRange = new Doctrine_Pager_Range_Sliding(
    array('chunk' => $numPageLinks), $pager);
    $pagerUrlBase = $this->view->url(
      array(), 'admin-catalog-index', 1) . "/{%page}";
    $pagerLayout = new Doctrine_Pager_Layout(
      $pager, $pagerRange, $pagerUrlBase);

    // устанавливаем шаблон для отображения ссылки на страницу
   $pagerLayout->setTemplate('<a href=»{%url}»>{%page}</a>');
   $pagerLayout->setSelectedTemplate(
      '<span class="current">{%page}</span>');
    $pagerLayout->setSeparatorTemplate(' ');

    // присваиваем значения переменным представления
    $this->view->records = $result;
    $this->view->pages = $pagerLayout->display(null, true);
     } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }
}
```

Этот код отличается от первоначального тем, что запрос проходит через компонент **Doctrine_Pager**, который инициализируется этим запросом, а также номером страницы и количеством элементов на странице (в данном случае пять). Когда запрос выполняется с помощью метода **execute()** этого компонента, он автоматически ограничивается таким образом, чтобы получать только набор данных, соответствующий указанной странице.

Однако осталась еще небольшая задача генерации навигационных ссылок, позволяющих пользователям переходить к другим страницам. Она решается с помощью компонента **Doctrine_Pager_Layout**, который принимает шаблон URL, содержащий переменную-заполнитель, а затем динамически генерирует набор

навигационных ссылок на основе этого шаблона, заменяя заполнитель фактическими номерами страниц. Количество ссылок и HTML-разметка, применяемая к каждой из них, могут быть настроены с помощью методов Doctrine_Pager_Layout. Конечный результат, возвращаемый методом display() объекта Doctrine_Pager_Layout, представляет собой блок HTML-кода, который можно непосредственно добавить в представление следующим образом:

```
<h2>List Items</h2>
<?php if (count($this->records)): ?>
<div id="pager">
  Pages: <?php echo $this->pages; ?>
</div>

<div id="records">

...
</div>

<div id="pager">
  Pages: <?php echo $this->pages; ?>
</div>
<?php else: ?>
No records found
<?php endif; ?>
```

Чтобы увидеть, как все это работает, откройте сводную страницу каталога в панели администрирования SQUARE по адресу http://square.localhost/admin/catalog/item/index. Если в вашем каталоге больше пяти элементов, вы увидите набор навигационных ссылок, которые можно использовать для перемещения по результатам выборки. Пример показан на рис. 7.1.

List Items

Pages: 1 2 3

	Item ID	Title	Denomination	Country	Grade	Year		
☐	6	Yacht on blue sea	5.00	France	Poor	1897	Display	Update
☐	7	Vintage cars along the river - Italy 1937	1.00	Italy	Very Fine	1937	Display	Update
☐	8	Oranges and Lemons	1.50	United States	Very Fine	1987	Display	Update
☐	9	Indian Elephants	5.00	India	Average	1976	Display	Update
[redacted] Selected							Update full-text indices	

Pages: 1 2 3

Рис. 7.1. Сводное представление каталога с включенным разбиением на страницы

Обратите внимание, что ссылка на каждую страницу содержит в своем URL номер этой страницы в виде GET-параметра.

Более подробная информация по использованию Doctrine_Pager_Layout для определения макета для навигационных ссылок доступна по ссылкам, приведенным в конце этой главы.

Добавление критериев сортировки к маршрутам

Разбиение на страницы — это лишь один из способов фрагментации набора данных. Другим чрезвычайно распространенным требованием является возможность сортировки данных пользователями по одному из полей. В этом случае к URL также добавляются параметры, указывающие на поле и направление сортировки, а контроллер страницы настраивается таким образом, чтобы сортировать данные по этим параметрам перед тем, как передать их представлению для отображения.

Чтобы посмотреть, как это работает, вернитесь к конфигурационному файлу приложения, $APP_DIR/application/configs/application.ini, и добавьте к определению маршрута еще два параметра, показанных ниже:

```
resources.router.routes.admin-catalog-index.route = /admin/catalog/item/
index/:page/:sort/:dir
resources.router.routes.admin-catalog-index.defaults.module = catalog
resources.router.routes.admin-catalog-index.defaults.controller = admin.item
resources.router.routes.admin-catalog-index.defaults.action = index
resources.router.routes.admin-catalog-index.defaults.page = 1
resources.router.routes.admin-catalog-index.defaults.sort = RecordID
resources.router.routes.admin-catalog-index.defaults.dir = asc
```

Обновление контроллера и представления

Эти параметры должны быть добавлены к запросу Doctrine, который генерируется действием Catalog_AdminItemController::indexAction. Для проверки того, что основной код действия работает только с корректными параметрами, можно использовать валидатор InArray. Измененный код показан в следующем примере:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
// действие для отображения списка элементов каталога
public function indexAction()
{
    // устанавливаем фильтры и валидаторы для данных,
    // полученных в запросе GET
      $filters = array(
      'sort' => array('HtmlEntities', 'StripTags', 'StringTrim'),
      'dir' => array('HtmlEntities', 'StripTags', 'StringTrim'),
      'page' => array('HtmlEntities', 'StripTags', 'StringTrim')
    );
    $validators = array(
      'sort' => array(
        'Alpha',
```

```
    array('InArray', 'haystack' =>
      array('RecordID', 'Title', 'Denomination',
            'CountryID', 'GradeID', 'Year'))
  ),
  'dir' => array(
    'Alpha',
      array('InArray', 'haystack' =>
array('asc', 'desc'))
  ),
  'page' => array('Int')
);
$input = new Zend_Filter_Input($filters, $validators);
$input->setData($this->getRequest()->getParams());

// проверяем корректность входных данных,
// создаем запрос и устанавливаем параметры компонента
// для разбиения на страницы
if ($input->isValid()) {
  $q = Doctrine_Query::create()
    ->from('Square_Model_Item i')
    ->leftJoin('i.Square_Model_Grade g')
    ->leftJoin('i.Square_Model_Country c')
    ->leftJoin('i.Square_Model_Type t')
    ->orderBy(sprintf('%s %s', $input->sort, $input->dir));

  $perPage = 4;
  $numPageLinks = 5;

  // инициализируем компонент для разбиения на страницы
  $pager = new Doctrine_Pager($q, $input->page, $perPage);

  // выполняем запрос, учитывающий номер страницы
  $result = $pager->execute(array(), Doctrine::HYDRATE_ARRAY);

  // инициализируем макет  компонента для разбиения на страницы
  $pagerRange = new Doctrine_Pager_Range_Sliding(array('chunk' =>
                $numPageLinks), $pager);
  $pagerUrlBase = $this->view->url(array(), 'admin-catalog-index',1) .
                "/{%page}/{$input->sort}/{$input->dir}";
  $pagerLayout = new Doctrine_Pager_Layout($pager, $pagerRange,
                 $pagerUrlBase);

  // устанавливаем шаблон для отображения ссылки на страницу
  $pagerLayout->setTemplate('<a href="{%url}">{%page}</a>');
  $pagerLayout->setSelectedTemplate('<span
               class="current">{%page}</span>');
  $pagerLayout->setSeparatorTemplate(' ');

  // присваиваем значения переменным представления
```

```
      $this->view->records = $result;
      $this->view->pages = $pagerLayout->display(null, true);
    } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }
}
```

Обратите внимание на то, что требуется также обновить шаблон URL, используемый компонентом Doctrine_Pager_Layout для генерации ссылок на страницы, чтобы добавить необходимые параметры сортировки и избежать их «потери» при переходе между страницами.

Последним этапом будет обновление представления и добавление ссылок, с помощью которых пользователь сможет сортировать набор данных по различным полям. Измененный сценарий представления:

```
<h2>List Items</h2>
<?php if (count($this->records)): ?>
<div id="pager">
  Pages: <?php echo $this->pages; ?>
</div>

<div id="records">
  <form method="post" action="<?php echo $this->url(array(), 'admincatalog-
delete'); ?>">
    <table>
      <tr>
        <td></td>
        <td class="key">
          Item ID
          <a href="<?php echo $this->url(array('sort' => 'RecordID',
                'dir' => 'asc'), 'admin-catalog-index'); ?>">&uArr;</a>
          <a href="<?php echo $this->url(array('sort' => 'RecordID',
                'dir' => 'desc'), 'admin-catalog-index'); ?>">&dArr;</a>
        </td>
      <td class="key">
        Title
          <a href="<?php echo $this->url(array('sort' => 'Title', 'dir'
                => 'asc'), 'admin-catalog-index'); ?>">&uArr;</a>
          <a href="<?php echo $this->url(array('sort' => 'Title', 'dir'
                => 'desc'), 'admin-catalog-index'); ?>">&dArr;</a>
      </td>
        <td class="key">
          Denomination
          <a href="<?php echo $this->url(array('sort' => 'Denomination',
                'dir' => 'asc'), 'admin-catalog-index'); ?>">&uArr;</a>
          <a href="<?php echo $this->url(array('sort' => 'Denomination',
                'dir' => 'desc'), 'admin-catalog-index'); ?>">&dArr;</a>
      </td>
        <td class="key">
```

```
        Country
        <a href="<?php echo $this->url(array('sort' => 'CountryID',
              'dir' => 'asc'), 'admin-catalog-index'); ?>">&uArr;</a>
        <a href="<?php echo $this->url(array('sort' => 'CountryID',
              'dir' => 'desc'), 'admin-catalog-index'); ?>">&dArr;</a>
      </td>
        <td class="key">
        Grade
        <a href="<?php echo $this->url(array('sort' => 'GradeID',
              'dir' => 'asc'), 'admin-catalog-index'); ?>">&uArr;</a>
        <a href="<?php echo $this->url(array('sort' => 'GradeID',
              'dir' => 'desc'), 'admin-catalog-index'); ?>">&dArr;</a>
      </td>
        <td class="key">
        Year
        <a href="<?php echo $this->url(array('sort' => 'Year', 'dir'
              => 'asc'), 'admin-catalog-index'); ?>">&uArr;</a>
        <a href="<?php echo $this->url(array('sort' => 'Year', 'dir'
              => 'desc'), 'admin-catalog-index'); ?>">&dArr;</a>
        </td>
        <td></td>
        <td></td>
      </tr>
      <?php foreach ($this->records as $r):?>

        <td><?php echo $this->escape($r['RecordID']); ?></td>
        <td><?php echo $this->escape($r['Title']); ?></td>

      </tr>
        <?php endforeach; ?>
        <tr>
          <td colspan="7">
            <input type="submit" name="submit" style="width:150px"
            value="Delete Selected" />
          </td>
          <td colspan="2">
            <a href="<?php echo $this->url(array(),
            'admin-fulltext-index-create'); ?>">Update full-text indices</a>
        </td>
      </tr>
    </table>
  </form>
</div>

<div id="pager">
  Pages: <?php echo $this->pages; ?>
</div>
<?php else: ?>
No records found
<?php endif; ?>
```

Теперь если вы перейдете по URL http://square.localhost/admin/catalog/item/index, то увидите, что можно сортировать данные по различным полям в порядке возрастания или убывания. Критерий сортировки сохраняется даже при перемещении по страницам. Если он не указан, стандартные значения для маршрута приведут к автоматической сортировке по идентификатору элемента в порядке возрастания.

На рис. 7.2 показан пример вывода, отсортированного по состоянию (чтобы лучше понять, как это работает, посмотрите на URL в адресной строке браузера).

List Items

Pages: 1 2 3

	Item ID ⇑ ⇓	Title ⇑ ⇓	Denomination ⇑ ⇓	Country ⇑ ⇓	Grade ⇑ ⇓	Year ⇑ ⇓		
☐	10	Picasso	10.00	Singapore	Good	1981	Display	Update
☐	2	Britain - WWII Fighter	1.00	United Kingdom	Average	1966	Display	Update
☐	5	Tiger Lily	2.00	India	Average	1987	Display	Update
☐	9	Indian Elephants	5.00	India	Average	1976	Display	Update
Delete Selected							Update full-text indices	

Pages: 1 2 3

Рис. 7.2. Сводное представление каталога с включенными сортировкой и разбиением на страницы

Обработка загружаемых на сервер файлов

В главе 3 вы прошли быстрый курс по созданию и обработке форм с использованием компонента Zend_Form, а в последующих главах закрепили свои знания, применив описанные приемы к различным типам форм. Однако есть еще одно довольно распространенное требование, выполнение которого пока не рассматривалось: обработка файлов, загружаемых на сервер через формы.

Разумеется, PHP поддерживает загрузку файлов через формы уже несколько лет, а для безопасного и эффективного управления загруженными файлами доступна суперглобальная переменная $_FILES и несколько встроенных методов, таких как is_uploaded_file() и move_uploaded_file(). В Zend Framework аналогичная функциональность доступна через компонент Zend_File_Transfer, предоставляющий полноценный API для получения, валидации и обработки загруженных файлов. Этот компонент без проблем выполняет и обработку файлов, загружаемых через элементы Zend_Form.

Понять, как осуществляется загрузка файлов с помощью Zend_Form, проще всего на примере. Рассмотрим следующий код, создающий простую форму, которая состоит из элемента для загрузки файла и кнопки отправки:

```
<?php
class Form_Example extends Zend_Form
```

```
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/sandbox/example/form')
        ->setMethod('post');

    // создаем поле загрузки файла для фотографии
    $photo = new Zend_Form_Element_File('photo');
    $photo->setLabel('Photo:')
         ->setDestination('/tmp/upload');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Submit');

    // добавляем элементы к форме
    $this->addElement($photo)
        ->addElement($submit);
    return $this;
  }
}
```

ВНИМАНИЕ

Несмотря на то что PHP по умолчанию поддерживает загрузку файлов как с помощью метода POST, так и с помощью метода PUT, компонент Zend_File_Transfer в данный момент поддерживает только POST.

На рис. 7.3 показано, как выглядит полученная форма.

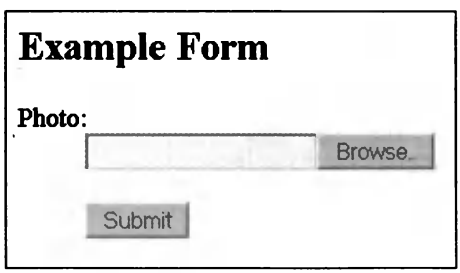


Рис. 7.3. Форма с полем загрузки файла

Метод **setDestination()** определяет целевой каталог для загружаемых файлов. Обратите внимание, что этот каталог должен существовать на момент инициализации объекта **Zend_Form** в действии; если он не существует, объект выбросит исключение. Кроме этого, во избежание неожиданных сбоев при обработке загрузки неплохо убедиться, что целевой каталог доступен веб-серверу для записи.

После отправки формы вызов ее метода getValues() внутри действия автоматически приведет к получению файла и перемещению его в указанное местоположение. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function formAction()
  {
    $form = new Form_Example;
    $this->view->form = $form;
    if ($this->getRequest()->isPost()) {
        if ($form->isValid($this->getRequest()->getPost())) {
          $values = $form->getValues();
          $this->_redirect('/form/success');
      }
    }
  }
}
```

В целях безопасности не стоит получать файлы без их предварительной проверки. Также могут существовать требования прикладного уровня, которым должны отвечать загруженные файлы, например ограничения на размер файла или его тип. Выполнение этих требований обеспечивается массивом специфичных для файлов валидаторов ввода, входящих в состав Zend (табл. 7.1).

Таблица 7.1. Валидаторы для загруженных файлов, входящие в состав Zend Framework

Название валидатора	Описание
Exists	Возвращает false, если аргумент не является правильным файлом
Count	Возвращает false, если количество загруженных файлов выходит за пределы, определенные аргументом
Size	Возвращает false, если размер загруженного файла выходит за пределы, определенные аргументом
FilesSize	Возвращает false, если общий размер загруженных файлов выходит за пределы, определенные аргументом
Extension	Возвращает false, если расширение загруженного файла не соответствует ни одному из указанных в аргументе
MimeType	Возвращает false, если MIME-тип загруженного файла не соответствует ни одному из указанных в аргументе
IsCompressed	Возвращает false, если загруженный файл не является сжатым архивом
IsImage	Возвращает false, если загруженный файл не является изображением
ImageSize	Возвращает false, если размеры загруженного изображения выходят за пределы, определенные аргументом

Название валидатора	Описание
Crc32, Md5, Sha1, Hash	Возвращает false, если содержимое загруженного файла не соответствует значению хэш-суммы, указанной в аргументе (поддерживаются алгоритмы crc32, md5 и sha1)
ExcludeExtension	Возвращает false, если расширение загруженного файла соответствует одному из указанных в аргументе
ExcludeMimeType	Возвращает false, если MIME-тип загруженного файла соответствует одному из указанных в аргументе
WordCount	Возвращает false, если количество слов в загруженном файле выходит за пределы, определенные аргументом

Кроме этого, перед сохранением загруженных файлов на диск может понадобиться выполнить над ними различные действия, например переименование, изменение содержимого или шифрование. Эти задачи решаются с помощью специфичных для файлов фильтров ввода, входящих в состав Zend Framework (табл. 7.2).

Таблица 7.2. Фильтры для загруженных файлов, входящие в состав Zend Framework

Название фильтра	Описание
Encrypt	Шифрует содержимое загруженного файла
Decrypt	Расшифровывает содержимое загруженного файла
LowerCase	Переводит содержимое загруженного файла в нижний регистр
UpperCase	Переводит содержимое загруженного файла в верхний регистр
Rename	Переименовывает загруженный файл

Чтобы продемонстрировать, как работают упомянутые фильтры и валидаторы, рассмотрим измененную версию приведенного ранее определения формы. Эта версия использует набор фильтров и валидаторов, для того чтобы ограничить множество загружаемых файлов изображений в формате JPEG размером меньше 40 Кбайт и дать им уникальные имена.

```
<?php
class Form_Example extends Zend_Form

 public function init()
 {
   // инициализируем форму
   $this->setAction('/sandbox/example/form')
       ->setMethod('post');

   // создаем поле загрузки файла для фотографии
   $photo = new Zend_Form_Element_File('photo');
   $photo->setLabel('Photo:')
       ->setDestination('/tmp/upload')
```

```
            ->addFilter('Rename',
                sprintf('p-%s.jpg', uniqid(md5(time()), true)))
            ->addValidator('Extension', false, 'jpg')
            ->addValidator('MimeType', false, 'image/jpeg')
            ->addValidator('Size', false, 40000);

        // создаем кнопку отправки
        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setLabel('Submit');

        // добавляем элементы к форме
        $this->addElement($photo)
            ->addElement($submit);
        return $this;
    }
}
```

На рис. 7.4 показан пример ошибок, выводимых при попытке загрузить файл, не соответствующий указанным ограничениям.

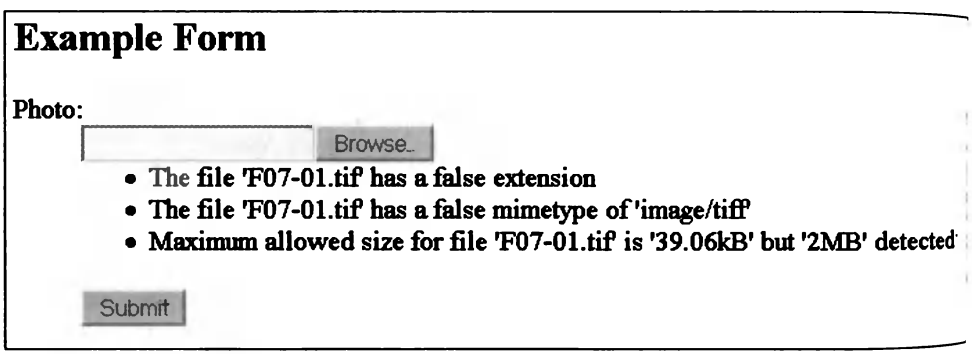


Рис. 7.4. Результат загрузки неподходящего файла

Упражнение 7.2. Добавление возможности загрузки изображений

Теперь, имея необходимые знания, давайте модифицируем демонстрационное приложение SQUARE, разрешив продавцам добавлять к каждому элементу каталога до трех изображений. В следующих разделах описано, как это сделать.

Определение целевого каталога для загруженных файлов

Первым шагом будет определение целевого каталога для загруженных изображений. Так как эти изображения будут отображаться в общедоступном списке и, следовательно, должны быть доступны через неаутентифицированные запросы URL

сохраняйте их в каталоге, находящемся внутри **$APP_DIR/public/**. Поэтому создайте каталог **$APP_DIR/public/uploads/**, как показано ниже:

```
shell> cd /usr/local/apache/htdocs/square/public
shell> mkdir uploads
```

Поскольку эта информация будет требоваться различным контроллерам и действиям, внесите ее в конфигурационный файл приложения. Откройте файл **$APP_DIR/application/configs/application.ini** и добавьте в него следующий параметр:

```
uploads.uploadPath = APPLICATION_PATH "/../public/uploads"
```

Обновление определения формы

Теперь необходимо добавить в форму создания новых записей каталога элементы для загрузки файлов. Для этого внесите в определение формы, находящееся в файле **$APP_DIR/library/Square/Form/ItemCreate.php**, изменения, выделенные жирным шрифтом:

```
<?php
class Square_Form_ItemCreate extends Zend_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/catalog/item/create')
        ->setMethod('post');

    // -------------- //
    // определения элементов удалены для экономии места //
    // за полным определением формы обратитесь //
    // к главе 4 или архиву с кодом //
    //------------- //

    // создаем поле загрузки файла для изображений элементов
    $images = new Zend_Form_Element_File('images');
    $images->setMultiFile(3)
        ->addValidator('IsImage')
        ->addValidator('Size', false, '204800')
        ->addValidator('Extension', false, 'jpg,png,gif')
        ->addValidator('ImageSize', false, array(
            'minwidth' => 150,
            'minheight' => 150,
            'maxwidth' => 150,
            'maxheight' => 150
          ))
        ->setValueDisabled(true);

    // добавляем элемент к форме
    $this->addElement($images);
```

```
        // создаем группу отображения для файловых элементов
        $this->addDisplayGroup(array('images'), 'files');
        $this->getDisplayGroup('files')
            ->setOrder(40)
            ->setLegend('Images');

        // добавляем элемент к форме
        $this->addElement($submit);
    }
}
```

В приведенном коде с помощью метода **setMultiFile()** на форму добавляются три элемента загрузки файлов. Чтобы гарантировать, что с помощью этих элементов можно загружать только изображения в форматах JPEG, GIF и PNG, имеющие размеры меньше 2 Мбайт и 150 × 150 пикселей, используется набор валидаторов. Присвойте загруженным изображениям новые имена перед сохранением на диск для их соответствия стандартному формату именования; эта задача решается с помощью фильтра **Rename** (который мы рассмотрим в следующем разделе).

ВОПРОС ЭКСПЕРТУ

В: Зачем вызывать метод **setValueDisabled()** объекта **Zend_Form_Element_File**?

О: Данный вызов предотвращает автоматическое получение файлов при вызове в действии метода **getValues()**. Куда более интересен вопрос — почему нельзя получить файлы автоматически? И здесь надо кое о чем рассказать. Как вы видите, фильтр **Rename** на самом деле не поддерживает одновременную загрузку нескольких файлов. Однако по требованиям нашего приложения загруженные файлы должны быть переименованы в соответствии с определенным форматом. Чтобы учесть оба этих факта, известный разработчик Zend Framework Томас Виднер (Thomas Wiedner) предлагает при загрузке нескольких файлов динамически устанавливать для каждого из них фильтр **Rename**, а затем получать файл вручную с помощью метода **receive()** HTTP-адаптера **Zend_File_Transfer**. Таким образом, вызов **setValueDisabled()** гарантирует, что в действии не будет производиться автоматическое получение файлов, и дает возможность вручную осуществлять их получение и переименование.

Обновление действия для создания

Следующим шагом будет добавление в действие **Catalog_ItemController::createAction** кода, отвечающего за получение загруженных изображений, переименование их в соответствии со стандартным форматом и сохранение на диск. Для этого внесите в файл **$APP_DIR/application/modules/catalog/controllers/ItemController.php** изменения, выделенные жирным шрифтом:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
{
  public function createAction()
  {
    // генерируем форму ввода
```

```
    $form = new Square_Form_ItemCreate;
    $this->view->form = $form;

    // проверяем корректность входных данных
    // если они корректны, заполняем модель,
    // присваиваем некоторым полям их значения по умолчанию
    // и сохраняем в базу данных
    if ($this->getRequest()->isPost()) {
      if ($form->isValid($this->getRequest()->getPost())) {
        $item = new Square_Model_Item;
        $item->fromArray($form->getValues());
        $item->RecordDate = date('Y-m-d', mktime());
        $item->DisplayStatus = 0;
        $item->DisplayUntil = null;
        $item->save();
        $id = $item->RecordID;
        $config = $this->getInvokeArg('bootstrap')->getOption('uploads');
        $form->images->setDestination($config['uploadPath']);
        $adapter = $form->images->getTransferAdapter();
        for($x=0; $x<$form->images->getMultiFile(); $x++) {
          $xt = @pathinfo($adapter->getFileName('images_'.$x.'_'),
                PATHINFO_EXTENSION);
          $adapter->clearFilters();
          $adapter->addFilter('Rename', array(
            'target' => sprintf('%d_%d.%s', $id, ($x+1), $xt),
            'overwrite' => true
        ));
        $adapter->receive('images_'.$x.'_');
      }
        $this->_helper->getHelper('FlashMessenger')->addMessage(
          'Your submission has been accepted as item #' . $id .
          '. A moderator will review it and, if approved, it will
          appear on the site within 48 hours.');
        $this->_redirect('/catalog/item/success');
      }
    }
  }
}
```

ВНИМАНИЕ

При сохранении загруженных файлов под их исходными именами не забудьте проверить эти имена, перед тем как получить файлы с помощью метода receive(). Это необходимо для исключения возможности внедрения файлов в файловую систему сервера путем использования имен типа '../etc/passwd' или '../index.php', которые, безусловно, могут создать серьезные проблемы на плохо настроенном сервере.

Здесь метод getTransferAdapter() объекта Zend_Form_Element_File возвращает экземпляр HTTP-адаптера Zend_File_Transfer. Этот адаптер предназначен для управления получением, валидацией и обработкой загрузки нескольких файлов. Он предостав-

ляет метод **receive()**, который можно использовать для самостоятельного получения отдельных файлов из набора. Применение фильтра **Rename** к каждому файлу гарантирует, что эти файлы будут автоматически переименованы в соответствии с конкретным форматом именования, использующим в имени файла идентификатор записи и его порядковый номер в наборе. После этого переименованные файлы сохраняются на диск в каталог, определенный методом **setDestination()** объекта **Zend_Form_Element_File**.

Обновление действия и представления для отображения

Большая часть тяжелой работы выполнена. Теперь требуется обновить действие **Catalog_ItemController::displayAction** для отображения загруженных изображений вместе с остальной информацией о записи. Ниже приведен обновленный код метода **displayAction()**.

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
{
  // действие для отображения элемента каталога
  public function displayAction()
  {
    // устанавливаем фильтры и валидаторы для входных данных,
    // полученных в запросе GET
    $filters = array(
        'id' => array('HtmlEntities', 'StripTags', 'StringTrim')
    );
    $validators = array(
        'id' => array('NotEmpty', 'Int')
    );
    $input = new Zend_Filter_Input($filters, $validators);
    $input->setData($this->getRequest()->getParams());

    // проверяем корректность входных данных,
    // получаем запрошенную запись
    // добавляем ее к представлению
    if ($input->isValid()) {
    $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Type t')
        ->where('i.RecordID = ?', $input->id)
        ->addWhere('i.DisplayStatus = 1')
        ->addWhere('i.DisplayUntil >= CURDATE()');
    $result = $q->fetchArray();
    if (count($result) == 1) {
      $this->view->item = $result[0];
      $this->view->images = array();
      $config = $this->getInvokeArg('bootstrap')->getOption('uploads');
```

```
    foreach (glob("{$config['uploadPath']}/
      {$this->view->item['RecordID']}_*") as $file)
    {
      $this->view->images[] = basename($file);
    }
        } else {
          throw new Zend_Controller_Action_Exception('Page not found', 404);
    }
        } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }
}
```

Добавленный код считывает из конфигурационного файла расположение каталога для загрузок, использует функцию PHP **glob()**, чтобы получить список всех файлов в этом каталоге, соответствующих идентификатору элемента, а затем передает полученный список файлов представлению в виде массива. После этого сценарий представления может обработать массив и отобразить изображения в галерее следующим образом:

```
<h2>View Item</h2>
<h3>
    FOR SALE:
    <?php echo $this->escape($this->item['Title']); ?> -
    <?php echo $this->escape($this->item['Year']); ?> -
    <?php echo $this->escape($this->item['Square_Model_Grade']
    ['GradeName']); ?>
    </h3>

    <div id="container">
      <div id="images">
        <?php foreach ($this->images as $image): ?>
        <img src="/uploads/<?php echo $this->escape($image); ?>"
        width="150" height="150" />
        <?php endforeach; ?>
      </div>

      <div id="record">
        <table>
        <tr>
          <td class="key">Title:</td>
          <td class="value">
            <?php echo $this->escape($this->item['Title']); ?>
          </td>
        </tr>

      <tr>
        <td class="key">Description:</td>
        <td class="value">
```

```
          <?php echo $this->escape($this->item['Description']); ?>
        </td>
      </tr>
    </table>
  </div>
</div>
```

Аналогичные изменения следует внести и в действие **Catalog_AdminItemController::displayAction** и сценарий представления для отображения изображений, соответствующих каждому элементу. Требуемый код вы найдете в архиве с дополнительными материалами к этой главе.

Обновление действия для удаления

Последним шагом будет внесение изменений в действие **Catalog_AdminItemController::deleteAction** для автоматического удаления загруженных изображений, связанных с элементом каталога, при удалении последнего. Сделать это несложно, и соответствующий код приведен ниже:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
  // действие для удаления элементов каталога
  public function deleteAction()
{
    // устанавливаем фильтры и валидаторы для входных данных,
    // полученных в запросе GET
    $filters = array(
        'ids' => array('HtmlEntities', 'StripTags', 'StringTrim')
    );
    $validators = array(
        'ids' => array('NotEmpty', 'Int')
    );
    $input = new Zend_Filter_Input($filters, $validators);
    $input->setData($this->getRequest()->getParams());

    // проверяем корректность входных данных,
    // читаем массив идентификаторов записей,
    // удаляем записи из базы данных
    if ($input->isValid()) {
    $q = Doctrine_Query::create()
        ->delete('Square_Model_Item i')
        ->whereIn('i.RecordID', $input->ids);
    $result = $q->execute();
    $config = $this->getInvokeArg('bootstrap')->getOption('uploads');
    foreach ($input->ids as $id) {
        foreach (glob("{$config['uploadPath']}/{$id}_*") as $file) {
          unlink($file);
      }
    }
```

```
        $this->_helper->getHelper('FlashMessenger')
            ->addMessage('The records were successfully deleted.');
        $this->_redirect('/admin/catalog/item/success');
      } else {
        throw new Zend_Controller_Action_Exception('Invalid input');
      }
    }
}
```

Чтобы проверить работу всех изменений, попробуйте добавить в каталог новую запись, перейдя по URL **http://square.localhost/catalog/item/create**. Как показано на рис. 7.5, форма ввода теперь отображает три дополнительных элемента для загрузки файлов.

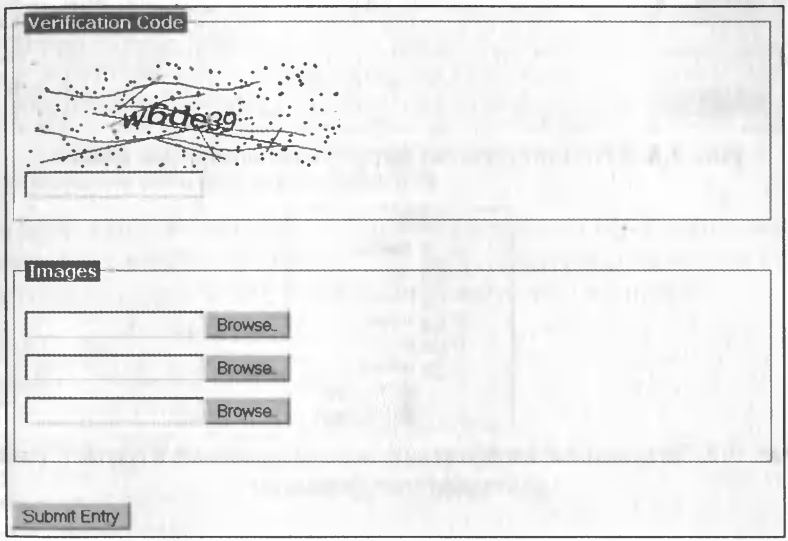


Рис. 7.5. Форма для добавления нового элемента в каталог SQUARE, с дополнительными элементами загрузки файлов

Введите информацию о новой записи, добавьте одно или несколько изображений и сохраните изменения. Если изображения не соответствуют указанным форматам и размерам, вы увидите ошибку (рис. 7.6).

После успешной отправки вы сможете найти загруженные изображения, переименованные в соответствии со стандартным форматом, в каталоге приложения (рис. 7.7). Теперь можно войти в панель администрирования, найти только что добавленный элемент и сделать его общедоступным. После этого перейдите по URL для отображения элемента, и вы увидите страницу с информацией о нем, а также набор изображений, которые вы загрузили. Пример показан на рис. 7.8.

И наконец, если вы удалите элемент через панель администрирования, а затем перейдете в каталог **$APP_DIR/public/uploads/**, то обнаружите, что файлы с изображениями также были удалены.

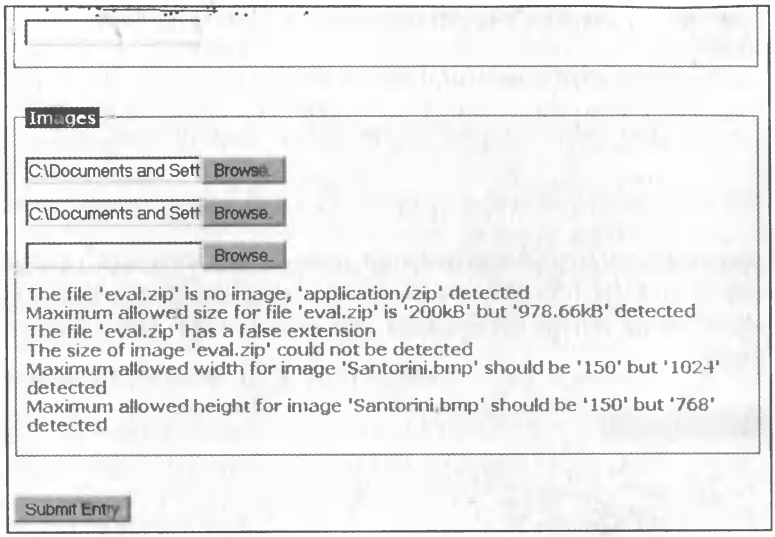


Рис. 7.6. Результат попытки загрузить неподходящие файлы

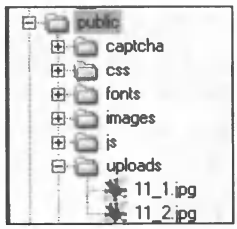


Рис. 7.7. Загруженные изображения, переименованные в соответствии со стандартным форматом

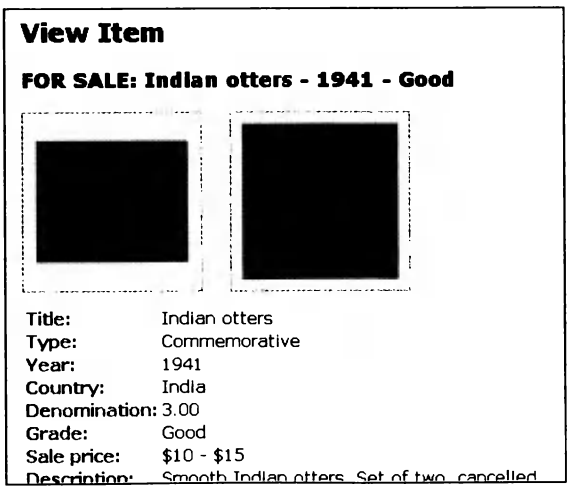


Рис. 7.8. Страница с информацией об элементе каталога, дополненной пользовательскими изображениями

Работа с конфигурационными данными

Как говорилось ранее, обычно разработчик приложения должен определить, какие аспекты поведения программы могут настраиваться пользователем. Эта информация, или *конфигурационные данные*, должна находиться в хранилище данных приложения (это может быть база данных, обычный файл или какой-либо другой вариант постоянного хранилища), и различные компоненты приложения могут обращаться к ней по мере необходимости.

Zend Framework упрощает работу с конфигурационными файлами с помощью компонента **Zend_Config**. Этот компонент предоставляет полнофункциональный API для чтения и записи конфигурационных файлов, основанный на использовании адаптеров. На момент написания этой главы компонент поддерживал адаптеры для форматов INI и XML, однако написать новые адаптеры для поддержки других форматов достаточно просто.

Работу компонента **Zend_Config** лучше всего продемонстрировать на примерах. Следующие разделы помогут приступить к его использованию.

Чтение конфигурационных файлов

Zend_Config можно использовать для чтения и анализа конфигурационных файлов в форматах XML и INI. Чтобы показать, как это делается, рассмотрим конфигурационный файл в формате INI, содержащий следующие данные:

```
[object]
shape = 'square'
size = '10'
color = 'red'
typeface.name = 'Mono'
typeface.size = 19
typeface.units = 'px'
typeface.color = 'white'
```

Эти данные могут быть считаны в объект **Zend_Config**, после чего отдельные конфигурационные переменные становятся доступны в виде свойств этого объекта, и к ним можно обратиться, используя соответствующий синтаксис. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function configAction()
  {
    // инициализируем объект конфигурации
    $config = new Zend_Config_Ini(APPLICATION_PATH . "/configs/example.ini");

    // присваиваем переменной $view->a значение 'square'
    $this->view->a = $config->object->shape;

    // присваиваем переменной $view->b значение 'white'
    $this->view->b = $config->object->typeface->color;
  }
}
```

ВНИМАНИЕ

По умолчанию конфигурационные данные, представленные объектом Zend_Config, доступны только для чтения, и любая попытка изменить их приведет к исключению. Чтобы разрешить внесение изменений, передайте объекту Zend_Config в качестве третьего аргумента массив дополнительных параметров, содержащий ключ 'allowModifications' со значением true. Более подробную информацию можно получить по ссылкам в конце этой главы.

Существует также адаптер **Zend_Config_Xml**, предназначенный для работы с конфигурационными данными в формате XML. Рассмотрим следующий XML-файл:

```
<?xml version="1.0"?>
<configuration>
  <application>
    <name>SomeApp</name>
    <version>2.3</version>
      <window>
        <height>600</height>
        <width>500</width>
        <titlebar>
          <title>Export Data</title>
          <foreColor>#ffffff</foreColor>
          <backColor>#0000ff</backColor>
        </titlebar>
      </window>
    </application>
</configuration>
```

Ниже приведен пример чтения этого файла и доступа к хранящимся в нем параметрам конфигурации с помощью адаптера **Zend_Config_Xml**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function configAction()
  {
    // инициализируем объект конфигурации
    $config = new Zend_Config_Xml(APPLICATION_PATH . "/configs/example.xml");

    // присваиваем переменной $view->a значение '2.3'
    $this->view->a = $config->application->version;
    // присваиваем переменной $view->b значение 'Export Data'
    $this->view->b = $config->application->window->titlebar->title;
  }
}
```

Запись конфигурационных файлов

С чтением файлов все понятно, но как насчет записи? **Zend_Config** содержит компонент для записи файлов, **Zend_Config_Writer**, который можно использовать для создания конфигурационных файлов в форматах PHP, XML или INI. Для удобства

чтения конфигурационные данные можно упорядочить иерархически или разделить на секции.

Ниже приведен пример, демонстрирующий использование Zend_Config_Writer для создания конфигурационного файла в формате INI:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function configAction()
  {
    // создаем объект конфигурации
    $config = new Zend_Config(array(), 1);

    // создаем секцию
    $config->blog = array();
    $config->blog->allowComments = 'yes';
    $config->blog->displayComments = 'yes';
    $config->blog->allowTrackbacks = 'yes';
    $config->blog->defaultAuthor = 'Jack Frost';
    $config->blog->numPostsOnIndexPage = 5;

    // создаем секцию
    $config->calendar = array();
    $config->calendar->weekStartsOn = 'Monday';
    $config->calendar->highlightToday = 1;

    // создаем подсекцию
    $config->calendar->events = array();
    $config->calendar->events->displayTitle = 1;
    $config->calendar->events->displayStartTime = 1;
    $config->calendar->events->displayEndTime = 0;
    $config->calendar->events->displayLocation = 1;

    // записываем данные в файл
    $writer = new Zend_Config_Writer_Ini();
    $writer->write(APPLICATION_PATH . "/configs/example.ini", $config);
  }
}
```

Сначала необходимо инициализировать объект Zend_Config, предоставляющий конфигурационные данные. Инициализация должна осуществляться массивом, содержащим эти данные (или пустым массивом, если они отсутствуют), и аргументом логического типа, указывающим на возможность их модификации. После инициализации объекта Zend_Config можно устанавливать значения отдельных конфигурационных переменных как свойств объекта, используя стандартный синтаксис. Составляя из свойств последовательности, можно создавать иерархические структуры данных; в зависимости от выбранного выходного формата иерархия будет представлена либо с помощью разделителей (INI), либо посредством вложенных элементов (XML).

Результатом выполнения приведенного кода является следующий конфигурационный файл:

```
[blog]
allowComments = "yes"
displayComments = "yes"
allowTrackbacks = "yes"
defaultAuthor = "Jack Frost"
numPostsOnIndexPage = 5

[calendar]
weekStartsOn = "Monday"
highlightToday = 1
events.displayTitle = 1
events.displayStartTime = 1
events.displayEndTime = 0
events.displayLocation = 1
```

В веб-приложениях весьма популярны конфигурационные файлы в формате XML, поскольку анализ XML поддерживается большинством языков программирования (в частности, в PHP для этого существуют несколько расширений, включая SimpleXML, DOM и XMLReader). Чтобы представить те же самые конфигурационные данные в формате XML, внесите следующие изменения в код действия для использования адаптера **Zend_Config_Writer_Xml**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function configAction()
  {
    // создаем объект конфигурации
    $config = new Zend_Config(array(), 1);

    // создаем секцию
    $config->blog = array();
    $config->blog->allowComments = 'yes';
    $config->blog->displayComments = 'yes';
    $config->blog->allowTrackbacks = 'yes';
    $config->blog->defaultAuthor = 'Jack Frost';
    $config->blog->numPostsOnIndexPage = 5;

    // создаем секцию
    $config->calendar = array();
    $config->calendar->weekStartsOn = 'Monday';
    $config->calendar->highlightToday = 1;

    // создаем подсекцию
    $config->calendar->events = array();
    $config->calendar->events->displayTitle = 1;
    $config->calendar->events->displayStartTime = 1;
    $config->calendar->events->displayEndTime = 0;
```

```
    $config->calendar->events->displayLocation = 1;

    // записываем данные в файл
    $writer = new Zend_Config_Writer_Xml();
    $writer->write(APPLICATION_PATH . "/configs/example.xml",
$config);
  }
}
```

Пример получившегося файла:

```
<?xml version="1.0"?>
<zend-config xmlns:zf="http://framework.zend.com/xml/zend-configxml/1.0/">
  <blog>
    <allowComments>yes</allowComments>
    <displayComments>yes</displayComments>
    <allowTrackbacks>yes</allowTrackbacks>
    <defaultAuthor>Jack Frost</defaultAuthor>
    <numPostsOnIndexPage>5</numPostsOnIndexPage>
  </blog>
  <calendar>
    <weekStartsOn>Monday</weekStartsOn>
    <highlightToday>1</highlightToday>
    <events>
      <displayTitle>1</displayTitle>
      <displayStartTime>1</displayStartTime>
      <displayEndTime>0</displayEndTime>
      <displayLocation>1</displayLocation>
    </events>
  </calendar>
</zend-config>
```

Случаются ситуации, в которых предпочтительнее хранить конфигурационные данные в виде обычного массива, определенного на языке PHP, который можно добавить в контекст приложения с помощью обычных операторов include() или require(). Для этого используйте в коде действия адаптер Zend_Config_Writer_Array, который создаст файл примерно следующего содержания:

```
<?php
return array (
  'blog' =>
  array (
      'allowComments' => 'yes',
      'displayComments' => 'yes',
      'allowTrackbacks' => 'yes',
      'defaultAuthor' => 'Jack Frost',
      'numPostsOnIndexPage' => 5,
  ),
  'calendar' =>
  array (
      'weekStartsOn' => 'Monday',
      'highlightToday' => 1,
```

```
        'events' =>
    array (
        'displayTitle' => 1,
        'displayStartTime' => 1,
        'displayEndTime' => 0,
        'displayLocation' => 1,
      ),
    ),
);
```

СОВЕТ

Представление конфигурационных данных в виде массива PHP обеспечивает лучшую производительность, так как они сразу же могут быть помещены в кэш механизмом кэширования кодов операций, таким как APC, без участия разработчика. Аналогичное кэширование конфигурационных файлов в форматах INI и XML невозможно без явного вмешательства разработчика.

Упражнение 7.3. Настройка параметров приложения

Разобравшись с основами Zend_Config, давайте рассмотрим его использование на практике, создав простую панель настройки приложения SQUARE. Процесс ее создания описан в следующих разделах.

Определение формы настройки

Для начала определим форму настройки. Как правило, это можно сделать только после некоторого анализа, определяющего, какие аспекты работы приложения должны быть доступны для изменения, и после принятия решения о том, в каком виде их следует представить. В данном случае предположим, что настроить можно следующие параметры:

- Адрес электронной почты для сообщений, отправляемых через форму обратной связи.
- Стандартный или запасной адрес электронной почты для всех сообщений от приложения и от пользователей.
- Количество элементов для отображения на одной странице в панели администрирования.
- Отображение имен продавцов и их адресов в публичном каталоге.
- Журналирование исключений в файл на диске.

Ниже приведен пример соответствующего определения формы, которое необходимо сохранить в файл **$APP_DIR/library/Square/Form/Configure.php**:

```
<?php
class Square_Form_Configure extends Zend_Form
```

```
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/admin/config')
         ->setMethod('post');

    // создаем текстовое поле для ввода стандартного адреса электронной почты
    $default = new Zend_Form_Element_Text('defaultEmailAddress');
    $default->setLabel('Fallback email address for all operations:')
         ->setOptions(array('size' => '40'))
         ->setRequired(true)
         ->addValidator('EmailAddress')
         ->addFilter('HtmlEntities')
         ->addFilter('StringTrim');

    // создаем текстовое поле для ввода адреса электронной почты отдела продаж
    $sales = new Zend_Form_Element_Text('salesEmailAddress');
    $sales->setLabel('Default email address for sales enquiries:')
         ->setOptions(array('size' => '40'))
         ->addValidator('EmailAddress')
         ->addFilter('HtmlEntities')
         ->addFilter('StringTrim');

    // создаем текстовое поле для ввода количества элементов,
    // отображаемых на одной странице в панели администрирования
    $items = new Zend_Form_Element_Text('itemsPerPage');
    $items->setLabel('Number of items per page in administrative
views:')
         ->setOptions(array('size' => '4'))
         ->setRequired(true)
         ->addValidator('Int')
         ->addFilter('HtmlEntities')
         ->addFilter('StringTrim');

    // создаем радиокнопку для настройки отображения
    // имени и адреса продавца
    $seller = new Zend_Form_Element_Radio('displaySellerInfo');
    $seller->setLabel('Seller name and address visible in public
catalog:')
         ->setRequired(true)
         ->setMultiOptions(array(
             '1' => 'Yes',
             '0' => 'No'
           ));

    // создаем радиокнопку для настройки журналирования исключений
    $log = new Zend_Form_Element_Radio('logExceptionsToFile');
    $log->setLabel('Exceptions logged to file:')
         ->setRequired(true)
```

```
        ->setMultiOptions(array(
            '1' => 'Yes',
            '0' => 'No'
          ));

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('Save configuration')
        ->setOptions(array('class' => 'submit'));

    // добавляем элементы к форме
    $this->addElement($sales)
         ->addElement($default)
         ->addElement($items)
         ->addElement($seller)
         ->addElement($log)
         ->addElement($submit);
  }
}
```

Определение конфигурационного файла

Следующим шагом будет определение файла для хранения пользовательских настроек. В Zend Framework для всех конфигурационных данных рекомендуется использовать каталог **$APP_DIR/application/configs/**, и мы последуем этой рекомендации. Откройте основной конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**, и добавьте в него следующий параметр, определяющий расположение пользовательского файла конфигурации:

```
configs.localConfigPath = APPLICATION_PATH "/configs/square.ini"
```

Определение пользовательских маршрутов

Теперь необходимо определить маршрут к контроллеру, отвечающему за настройку. Поскольку панель настройки не относится к какому-либо конкретному модулю, этот контроллер можно разместить в универсальном модуле **default**. Ниже приведено определение маршрута, которое следует добавить в конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**:

```
resources.router.routes.admin-config.route = /admin/config
resources.router.routes.admin-config.defaults.module = default
resources.router.routes.admin-config.defaults.controller = config
resources.router.routes.admin-config.defaults.action = index
resources.router.routes.admin-config-success.route = /admin/config/success
resources.router.routes.admin-config-success.defaults.module = default
resources.router.routes.admin-config-success.defaults.controller = config
resources.router.routes.admin-config-success.defaults.action = success
```

ВОПРОС ЭКСПЕРТУ

В: Я в замешательстве. Зачем вы сохраняете расположение одного конфигурационного файла в другом?

О: Конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**, — это глобальный файл, который читается приложением Zend Framework. Как правило, он содержит множество параметров, которые определяются проектировщиком приложения и не должны изменяться без серьезного рассмотрения. Однако в каждом приложении существуют различные настройки пользовательского уровня, которые должны быть доступны для изменения пользователями и/или администраторами. В приложении SQUARE эти настройки хранятся в отдельном конфигурационном файле пользовательского уровня, **$APP_DIR/application/configs/square.ini**.

Обычно рекомендуется использовать такое разделение системных и пользовательских настроек, поскольку оно позволяет независимо вносить изменения в переменные этих двух типов без риска навредить приложению. Тем не менее между ними должна существовать связь, чтобы действия и контроллеры, использующие пользовательские настройки, знали, где их искать. Так как конфигурационный файл приложения **application.ini** доступен всем действиям и контроллерам, путь к файлу пользовательских настроек, **square.ini**, удобно хранить именно в нем.

Если вы все еще в замешательстве, вот другое объяснение: изменения в глобальном конфигурационном файле **application.ini** могут потребовать изменений кода приложения, но изменения в пользовательском файле конфигурации никогда этого не потребуют.

Определение контроллера и представления

Следующим шагом будет определение действия и представления для интерфейса настройки. Ниже приведен код контроллера **ConfigController**, который следует сохранить в файл **$APP_DIR/application/modules/default/controllers/ConfigController.php**:

```
<?php
class ConfigController extends Zend_Controller_Action
{
  protected $localConfigPath;

  public function init()
  {
    // устанавливаем значение doctype
    $this->view->doctype('XHTML1_STRICT');

    // получаем путь к локальному конфигурационному файлу
    $config = $this->getInvokeArg('bootstrap')->getOption('configs');
    $this->localConfigPath = $config['localConfigPath'];
  }

  // действие для обработки административных URL
  public function preDispatch()
  {
    // устанавливаем административный макет,
    // проверяем, аутентифицирован ли пользователь,
```

```
    // если нет, перенаправляем его на страницу входа
    $url = $this->getRequest()->getRequestUri();
    $this->_helper->layout->setLayout('admin');
    if (!Zend_Auth::getInstance()->hasIdentity()) {
      $session = new Zend_Session_Namespace('square.auth');
      $session->requestURL = $url;
      $this->_redirect('/admin/login');
    }
  }

  // действие для сохранения конфигурационных данных
  public function indexAction()
  {
    // генерируем форму ввода
    $form = new Square_Form_Configure();;
    $this->view->form = $form;

    // если конфигурационный файл существует,
    // читаем из него конфигурационные параметры
    // и заполняем ими форму
        if (file_exists($this->localConfigPath)) {
        $config = new Zend_Config_Ini($this->localConfigPath);
        $data['defaultEmailAddress'] = $config->global->defaultEmailAddress;
        $data['salesEmailAddress'] = $config->user->salesEmailAddress;
        $data['itemsPerPage'] = $config->admin->itemsPerPage;
        $data['displaySellerInfo'] = $config->user->displaySellerInfo;
        $data['logExceptionsToFile'] = $config->global->logExceptionsToFile;
        $form->populate($data);
    }

    // проверяем корректность входных данных
    // если они корректны, создаем новый объект конфигурации
    // и секции настроек,
    // сохраняем конфигурационные параметры в файл,
    // заменяя их предыдущие версии
    if ($this->getRequest()->isPost()) {
    if ($form->isValid($this->getRequest()->getPost())) {
        $values = $form->getValues();
        $config = new Zend_Config(array(), true);
        $config->global = array();
        $config->admin = array();
        $config->user = array();
        $config->global->defaultEmailAddress = $values['defaultEmailAddress'];
        $config->user->salesEmailAddress = $values['salesEmailAddress'];
        $config->admin->itemsPerPage = $values['itemsPerPage'];
        $config->user->displaySellerInfo = $values['displaySellerInfo'];
        $config->global->logExceptionsToFile = $values['logExceptionsToFile'];
        $writer = new Zend_Config_Writer_Ini();
        $writer->write($this->localConfigPath, $config);
        $this->_helper->getHelper('FlashMessenger')->addMessage(
```

```
            'Thank you. Your configuration was successfully saved.');
        $this->_redirect('/admin/config/success');
      }
    }
  }

    // действие, выполняющееся при успешном завершении операции
    public function successAction()
    {
    if ($this->_helper->getHelper('FlashMessenger')->getMessages()) {
    $this->view->messages =
    $this->_helper->getHelper('FlashMessenger')->getMessages();
  } else {
      $this->_redirect('/');
    }
  }
}
```

Основная часть работы в этом контроллере лежит на методе **indexAction()**, ответственном за валидацию полученных в нем входных данных и запись конфигурационных данных в INI-файл с помощью компонента **Zend_Config_Writer**. При инициализации формы настройки метод **indexAction()** также читает исходный конфигурационный файл (если он доступен) с помощью компонента **Zend_Config**, а затем заполняет форму текущими настройками.

В методе **init()** расположение файла конфигурации считывается непосредственно из конфигурационного файла приложения и сохраняется в виде защищенного свойства класса. Так как этот контроллер предназначен для использования только администраторами, метод **preDispatch()** проверяет статус пользователя и перенаправляет его на страницу входа, если он не аутентифицирован.

Ниже приведен соответствующий сценарий представления, который следует сохранить в файл **$APP_DIR/application/modules/default/views/scripts/config/index.phtml**:

```
<h2>Edit Settings</h2>
<?php echo $this->form; ?>
```

Обновление основного макета

Все, что осталось сделать, — обновить административный макет в файле **$APP_DIR/application/layouts/admin.phtml**, чтобы при входе пользователя в главном меню отображался пункт **Settings**. В следующем примере изменения в макете выделены **жирным** шрифтом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

    <div id="menu">
      <?php if (Zend_Auth::getInstance()->hasIdentity()): ?>
      <a href="<?php echo $this->url(array(),
```

```
        .admin-catalog-index'); ?>">CATALOG</a>
        <a href="<?php echo $this->url(array(),
        'admin-config'); ?>">SETTINGS</a>
        <a href="<?php echo $this->url(array(), 'logout'); ?>">LOGOUT</a>
        <?php else: ?>
        <a href="<?php echo $this->url(array(), 'login'); ?>">LOGIN</a>
        <?php endif; ?>
      </div>
...
</html>
```

Для проверки работы кода войдите в панель администрирования приложения и перейдите по URL **http://square.localhost/admin/config**. На рис. 7.9 показана форма настройки, которую вы должны увидеть.

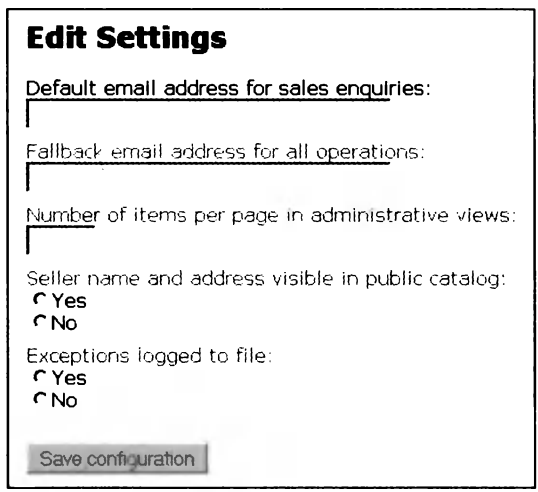


Рис. 7.9. Форма настройки приложения SQUARE

Введите значения в форму и отправьте ее. Данные должны сохраниться в файл **$APP_DIR/application/configs/square.ini**. Откройте его, и вы увидите данные в формате INI (рис. 7.10).

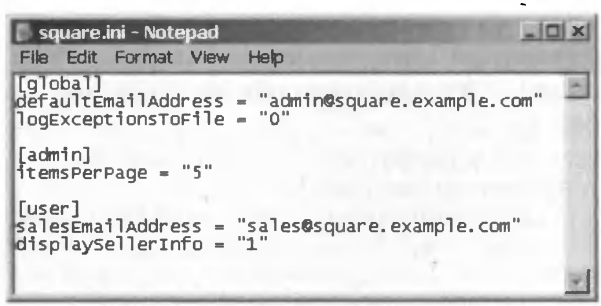


Рис. 7.10. Пример конфигурационных данных в формате INI

Если вы повторно перейдете по URL **http://square.localhost/admin/config**, то увидите, что форма настройки автоматически заполняется сохраненными ранее значениями (рис. 7.11).

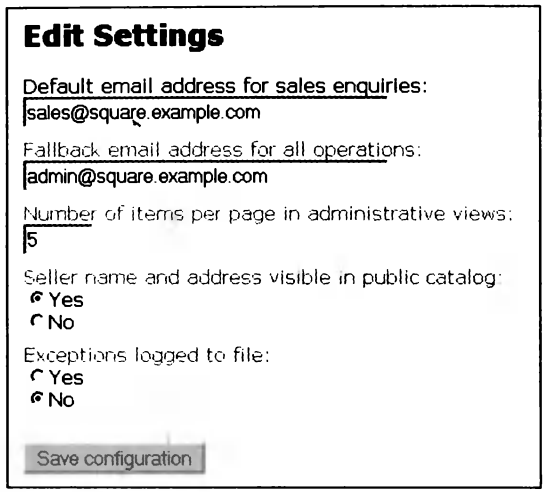


Рис. 7.11. Форма настройки приложения SQUARE, заполненная текущими значениями

Использование конфигурационных данных

Успешное чтение и запись конфигурационных данных — только половина дела: вам все еще нужно задействовать эти данные для изменения поведения определенных контроллеров и действий. В качестве примера предположим, что действие **ContactController::indexAction** теперь должно читать пользовательский файл конфигурации и отправлять сообщения электронной почты на адрес отдела продаж, указанный в этом файле. Обновленный код **ContactController::indexAction**:

```
<?php
class ContactController extends Zend_Controller_Action
{
  public function indexAction()
  {
    $form = new Square_Form_Contact();;
    $this->view->form = $form;
    if ($this->getRequest()->isPost()) {
      if ($form->isValid($this->getRequest()->getPost())) {
        $values = $form->getValues();
        $configs = $this->getInvokeArg('bootstrap')->getOption('configs');
        $localConfig = new Zend_Config_Ini($configs['localConfigPath']);
        $to = (!empty($localConfig->user->salesEmailAddress)) ?
          $localConfig->user->salesEmailAddress :
          $localConfig->global->defaultEmailAddress;
        $mail = new Zend_Mail();
```

```
        $mail->setBodyText($values['message']);
        $mail->setFrom($values['email'], $values['name']);
        $mail->addTo($to);
        $mail->setSubject('Contact form submission');
        $mail->send();
        $this->_helper->getHelper('FlashMessenger')->addMessage(
        'Thank you. Your message was successfully sent.');
        $this->_redirect('/contact/success');
      }
    }
  }
}
```

Похожие изменения следует внести и в действие **Catalog_ItemController::displayAction**, чтобы оно проверяло, должна ли информация о продавце быть доступна в публичном каталоге, и отображало ее в зависимости от результата проверки. Ниже приведен обновленный код **Catalog_ItemController::displayAction**, который после чтения пользовательского конфигурационного файла устанавливает флаг для представления:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
{
  // действие для отображения элемента каталога
  public function displayAction()
  {
    // устанавливаем фильтры и валидаторы для входных данных,
    // полученных в запросе GET
    $filters = array(
      'id' => array('HtmlEntities', 'StripTags', 'StringTrim')
    );
    $validators = array(
      'id' => array('NotEmpty', 'Int')
    );
    $input = new Zend_Filter_Input($filters, $validators);
    $input->setData($this->getRequest()->getParams());

    // проверяем корректность входных данных,
    // получаем запрошенную запись,
    // добавляем ее к представлению
    if ($input->isValid()) {
      $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Type t')
        ->where('i.RecordID = ?', $input->id)
        ->addWhere('i.DisplayStatus = 1')
        ->addWhere('i.DisplayUntil >= CURDATE()');
    $result = $q->fetchArray();
    if (count($result) == 1) {
```

```
    $this->view->item = $result[0];
    $this->view->images = array();
    $config = $this->getInvokeArg('bootstrap')->getOption('uploads');
    foreach (glob("{$config['uploadPath']}/
      {$this->view->item['RecordID']}_*") as $file)
    {
      $this->view->images[] = basename($file);
    }
    $configs = $this->getInvokeArg('bootstrap')->getOption('configs');
    $localConfig = new Zend_Config_Ini($configs['localConfigPath']);
      $this->view->seller = $localConfig->user->displaySellerInfo;
    } else {
      throw new Zend_Controller_Action_Exception('Page not found',
404);
      }
    } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }
}
```

Код соответствующего представления, использующего флаг, установленный контроллером, для определения, требуется ли добавлять на страницу информацию о продавце:

```
<h2>View Item</h2>
<h3>
  FOR SALE:
  <?php echo $this->escape($this->item['Title']); ?> -
  <?php echo $this->escape($this->item['Year']); ?> -
  <?php echo $this->escape($this->item['Square_Model_Grade']
['GradeName']); ?>
</h3>

<div id="container">
  <div id="images">
    <?php foreach ($this->images as $image): ?>
    <img src="/uploads/<?php echo $this->escape($image); ?>"
    width="150" height="150" />
    <?php endforeach; ?>
  </div>

  <div id="record">
    <table>
      <tr>
        <td class="key">Title:</td>
        <td class="value">
          <?php echo $this->escape($this->item['Title']); ?>
        </td>
      </tr>
      <!-- другие отображаемые поля -->
```

```
      <tr>
        <td class="key">Description:</td>
        <td class="value">
          <?php echo $this->escape($this->item['Description']); ?>
        </td>
      </tr>
    <?php if ($this->seller == 1): ?>
    <tr>
        <td class="key">Seller:</td>
        <td class="value">
          <?php echo $this->escape($this->item['SellerName']); ?>,
          <?php echo $this->escape($this->item['SellerAddress']); ?>
        </td>
      </tr>
      <?php endif; ?>
    </table>
  </div>
</div>
```

Также можно использовать пользовательские настройки для управления поведением компонента Doctrine_Pager в действии Catalog_AdminItemController::indexAction. Таким образом гарантируется, что количество элементов, отображаемых на каждой странице, соответствует значению, определенному пользователем, а не произвольному значению, указанному проектировщиком приложения. В целях экономии места требуемый код здесь не приводится, однако вы можете найти его в дополнительных материалах к этой главе.

СОВЕТ

Если в вашем приложении несколько различных контроллеров и действий читают данные из одного пользовательского файла конфигурации, вы можете избежать дублирования кода, автоматически считав этот файл в реестр локальных ресурсов в начальном загрузчике приложения. С помощью метода getResource() класса Zend_Application_Bootstrap_Bootstrap к этому реестру можно получить доступ из любой части приложения. Реестр представлен экземпляром компонента Zend_Registry, предоставляющим хранилище уровня приложения для глобальных объектов и переменных. Более подробную информацию о начальном загрузчике приложения и компоненте Zend_Registry можно получить, пройдя по ссылкам в конце этой главы, а в следующих главах этой книги вы встретите множество примеров использования Zend_Registry.

Выводы

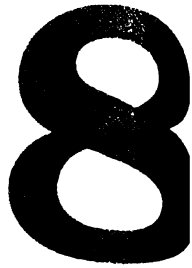
Эта глава была посвящена трем новым для вас компонентам: Doctrine_Pager, представляющему собой основу для разбиения на страницы больших результатов выборки из базы данных, Zend_File_Transfer, предоставляющему полнофункциональный API для валидации, фильтрации и получения файлов, загруженных через веб-формы, а также Zend_Config, упрощающему обработку конфигурационных данных, представленных в различных форматах. Все эти компоненты были использованы в демонстрационном приложении SQUARE, которое теперь обладает

возможностями разбиения на страницы и сортировки, панелью настройки и системой загрузки изображений.

Чтобы узнать больше о рассмотренных в этой главе темах, посетите следующие ссылки:

- Компонент Doctrine_Pager: http://www.doctrine-project.org/documentation/manual/1_1/en/utilities#pagination:working-with-pager.
- Компонент Zend_Paginator: http://framework.zend.com/manual/en/zend.paginator.html.
- Компонент Zend_File_Transfer: http://framework.zend.com/manual/en/zend.file.html.
- Фильтры и валидаторы для компонентов Zend_File_Transfer: http://framework.zend.com/manual/en/zend.file.transfer.filters.html и http://framework.zend.com/manual/en/zend.file.transfer.validators.html.
- Компонент Zend_Config: http://framework.zend.com/manual/en/zend.config.html.
- Компонент Zend_Registry: http://framework.zend.com/manual/en/zend.registry.html.
- Класс Bootstrap и реестр локальных ресурсов: http://framework.zend.com/manual/en/zend.application.theory-of-operation.html.
- Описание паттерна Registry (Мартин Фоулер (Martin Fowler)): http://martinfowler.com/eaaCatalog/registry.html.
- Создание пользовательского плагина к начальному загрузчику Zend Framework для управления ресурсами (Стефан Шмалхаус (Stefan Schmalhaus)): http://blog.log2e.com/2009/06/01/creating-a-custom-resourceplugin-in-zend-framework-18/.
- Учебник по компоненту Zend_Config (Аарон Вормус (Aaron Wormus)): http://devzone.zend.com/article/1264.

Журналирование и отладка исключений



Прочитав эту главу, вы:

- изучите модель исключений PHP 5.x;
- научитесь обрабатывать исключения в приложении Zend Framework;
- интегрируете классы пользовательских исключений в обработчик ошибок Zend Framework;
- научитесь управлять видимостью информации об ошибках в окружениях для эксплуатации программы;
- научитесь вести журнал исключений, сохраняя их в файл или базу данных;
- дополните сообщения в журнале отладочной информацией;
- сможете отслеживать исключения в реальном времени, используя отладчик для консоли Firebug.

Принято считать, что «хорошая» программа — это программа, работающая без ошибок. На самом деле это не так, и хорошая программа — это программа, в которой все возможные ошибки предусмотрены заранее и обрабатываются единообразно.

Написание «умных» программ, соответствующих этому определению, — в равной степени и искусство, и наука. Опыт и воображение играют важную роль в оценке возможных причин ошибок и определении надлежащих действий, но не менее важен и сам фреймворк, используемый для программирования, который определяет инструменты и функции, доступные для перехвата и исправления ошибок.

При разработке приложений с использованием Zend Framework обработка ошибок представляет меньшую трудность, нежели в случае с другими фреймворками. Стандартный обработчик, входящий в состав Zend Framework, по умолчанию может обрабатывать большинство распространенных ошибок, а несколько дополнительных компонентов, таких как Zend_Log или Zend_Debug, могут использоваться для предоставления дополнительной информации разработчикам и администраторам. В этой главе будет рассказано обо всех этих инструментах и показано их использование на практике.

Исключения

В PHP 5.x введена новая модель исключений, схожая с той, что используется в других языках программирования, таких как Java и Python. При подходе, основанном на использовании исключений, код программы помещается в блок try, а генерируемые им исключения «перехватываются» и обрабатываются одним или несколькими блоками catch. Так как можно использовать несколько блоков catch, разработчики могут перехватывать различные типы исключений и по-своему обрабатывать каждый из них.

Чтобы показать, как это работает, рассмотрим следующий листинг, в котором совершается попытка получить доступ к несуществующему элементу массива с помощью ArrayIterator:

```
<?php
// определяем массив
$cities = array(
  'London',
  'Washington',
  'Paris',
  'Delhi'
);

// попытка получить доступ к несуществующему элементу массива
// генерирует исключение OutOfBoundsException
try {
  $iterator = new ArrayIterator($cities);
  $iterator->seek(10);
} catch (Exception $e) {
  echo "ERROR: Something went wrong!\n";
  echo "Error message: " . $e->getMessage() . "\n";
  echo "Error code: " . $e->getCode() . „\n";
  echo "File name: " . $e->getFile() . "\n";
  echo "Line: " . $e->getLine() . "\n";
  echo "Backtrace: " . $e->getTraceAsString() . "\n";
}
?>
```

Когда интерпретатор PHP встречает код, помещенный в блок try-catch, он сначала пытается выполнить код внутри блока try. Если код выполняется без исключений, управление передается коду, следующему за блоком try-catch. Однако если при выполнении кода в блоке try генерируется исключение (как в приведенном примере), PHP останавливает выполнение блока в этой точке и начинает проверять каждый из блоков catch на предмет наличия обработчика исключения. Если обработчик найден, выполняется код внутри соответствующего блока catch, а затем код, следующий за блоком try; в противном случае генерируется критическая ошибка и выполнение сценария прекращается.

Каждый объект Exception содержит дополнительную информацию, которую можно использовать для отладки источника ошибки. Эта информация доступна

через встроенные методы объекта Exception и содержит сообщение, описывающее ошибку, код ошибки, имя файла и номер строки, где она произошла, а также трассировку (backtrace) вызовов функций, которые привели к ошибке. Соответствующие методы приведены в табл. 8.1.

Таблица 8.1. Методы объекта PHP Exception

Имя метода	Описание
getMessage()	Возвращает сообщение, описывающее, что пошло не так
getCode()	Возвращает числовой код ошибки
getFile()	Возвращает путь на диске и название сценария, сгенерировавшего исключение
getLine()	Возвращает номер строки, сгенерировавшей исключение
getTrace()	Возвращает трассировку вызовов, которые привели к ошибке, в виде массива
getTraceAsString()	Возвращает трассировку вызовов, которые привели к ошибке, в виде строки

ВОПРОС ЭКСПЕРТУ

В: Я видел, что неперехваченные исключения генерируют критическую ошибку, приводящую к непредвиденному завершению сценария. Можно ли изменить это поведение?

О: И да, и нет. В PHP есть функция set_exception_handler(), которая позволяет заменить стандартный обработчик исключений PHP вашим собственным кодом, как это делает функция set_error_handler(). Однако здесь есть один важный момент. Стандартный обработчик ошибок PHP выводит сообщение, а затем завершает выполнение сценария. При использовании собственного обработчика исключений уровень контроля над этим поведением ограничен: вы можете изменить способ отображения и внешний вид уведомления, но не можете сделать так, чтобы сценарий продолжил выполняться с позиции, в которой возникло исключение.

Более сложным подходом является наследование от базового объекта Exception и создание индивидуальных объектов Exception для каждой возможной ошибки. Такой подход полезен, если необходимо по-разному обрабатывать различные типы исключений, так как он позволяет использовать для каждого типа исключения отдельный блок catch (и отдельный код обработчика). Ниже приведен измененный вариант предыдущего примера, иллюстрирующий данный подход:

```
<?php
// создаем класс, производный от Exception
class MissingFileException extends Exception { }
class DuplicateFileException extends Exception { }
class FileIOException extends Exception { }

// устанавливаем имя файла,
// пытаемся скопировать, а затем удалить файл
$file = 'dummy.txt';
```

```
try {
  if (!file_exists($file)) {
    throw new MissingFileException($file);
  }
  if (file_exists("$file.new")) {
    throw new DuplicateFileException("$file.new");
  }
  if (!copy($file, "$file.new")) {
    throw new FileIOException("$file.new");
  }
  if (!unlink($file)) {
    throw new FileIOException($file);
  }
}
catch (MissingFileException $e) {
  echo 'ERROR: Could not find file \'' . $e->getMessage() . '\'';
  exit();
}
catch (DuplicateFileException $e) {
  echo 'ERROR: Destination file \'' . $e->getMessage() . '\' already exists';
  exit();
}
catch (FileIOException $e) {
  echo 'ERROR: Could not perform file input/output operation on file \''
    $e->getMessage() . '\'';
  exit();
}
catch (Exception $e) {
  echo ,ERROR: Something bad happened on line , . $e->getLine() . ': '
    $e->getMessage();
  exit();
}
echo 'SUCCESS: File operation successful.';
?>
```

Этот сценарий расширяет базовый класс **Exception** для создания трех новых типов **Exception**, каждый из которых представляет собой одну из возможных ошибок. Теперь отдельный блок **catch** для каждого типа **Exception** делает возможным изменение способа обработки каждого исключения. Последний блок **catch** — это универсальный «обобщенный» обработчик: он обрабатывает исключения, которые не были обработаны специализированными блоками, находящимися выше.

Подход, основанный на исключениях, имеет ряд преимуществ:

- Без обработки исключений необходимо проверять возвращаемое значение каждой вызванной функции, чтобы определить возникновение ошибки и принять меры по ее исправлению. Это приводит к созданию излишне сложного кода и блоков с большим уровнем вложенности. В модели, основанной на исключениях, единственный блок **catch** может использоваться для перехвата всех исключений, возникших в предшествующем ему блоке кода. Таким образом, исчезает необходимость во множестве каскадных проверок, и код получается более простым и удобочитаемым.

- В отсутствие исключений обработка ошибок основывается на предписаниях: разработчик должен продумать все возможные ошибки и написать код таким образом, чтобы учесть каждую из них. Основанный на исключениях подход, напротив, более гибок. Универсальный обработчик исключений выполняет роль страховки, перехватывая и обрабатывая даже те ошибки, код обработки которых не был написан. Это помогает сделать приложение более надежным и устойчивым к непредвиденным ситуациям.
- Поскольку модель исключений использует объектно-ориентированный подход, разработчики могут использовать такие концепции ООП, как наследование и расширяемость, чтобы наследоваться от базового класса **Exception** и создавать различные объекты **Exception** для различных типов исключений. Таким образом, становится возможным распознавать разные типы ошибок и обрабатывать их по-разному.
- Подход, основанный на исключениях, заставляет разработчиков принимать четкие решения относительно того, как будут обработаны разные типы ошибок. Когда исключения недоступны, разработчики могут легко пропустить (случайно или намеренно) проверки возвращаемых функциями значений. Однако игнорировать исключения не так просто. Требуя от разработчиков создания и заполнения блоков **catch**, эта модель заставляет их думать о причинах и следствиях ошибок и в конечном счете приводит к лучшему проектированию и более надежной реализации.

Все компоненты Zend Framework явным образом используют модель, основанную на исключениях, где каждый компонент по мере необходимости расширяет базовый класс **Zend_Exception**. Поэтому вне зависимости от того, используете вы компоненты Zend Framework как отдельные элементы большого приложения или в контексте MVC-приложения Zend Framework, эта согласованность в подходе может значительно уменьшить время, затраченное на разработку функциональности, связанной с обработкой ошибок прикладного уровня.

Стандартный процесс обработки ошибок

Теперь давайте поближе познакомимся с обработкой ошибок в MVC-приложении Zend Framework. По умолчанию, когда некоторым действием выбрасывается исключение, оно «всплывает» по цепочке выполнения, пока не достигнет front-контроллера. Front-контроллер содержит программный модуль для обработки ошибок (который автоматически регистрируется в процессе начальной загрузки), осуществляющий перенаправление исключения стандартному контроллеру ошибок приложения. Затем последний проверяет тип исключения и отображает соответствующее представление об ошибке запрашивающему клиенту.

Откуда берется этот контроллер ошибок? Для проектов, созданных с помощью сценария командной строки **zf**, он автоматически добавляется в стандартную структуру каталогов проекта. Для проектов, созданных вручную, его также необходимо

сгенерировать вручную. В любом случае он представлен в виде отдельных контроллера и действия, **ErrorController::errorAction**, и находится в модуле **default** приложения. Обычно он выглядит следующим образом:

```
<?php
class ErrorController extends Zend_Controller_Action
{
  public function errorAction()
  {
    $errors = $this->_getParam('error_handler');
    switch ($errors->type) {
        case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
        case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
            // 404 error -- controller or action not found
            $this->getResponse()->setHttpResponseCode(404);
            $this->view->message = 'Page not found';
            break;
    default:
            // application error
            $this->getResponse()->setHttpResponseCode(500);
            $this->view->message = 'Application error';
            break;
    }
    $this->view->exception = $errors->exception;
    $this->view->request = $errors->request;
  }
}
```

Стандартный контроллер ошибок способен обрабатывать два типа событий: некорректные HTTP-запросы и исключения, генерируемые действиями. Результатом первого события является ошибка с кодом 404, а второго — ошибка с кодом 500. Чтобы убедиться в этом, взгляните на следующий пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function failAction()
  {
    // попытка получить доступ к несуществующему элементу массива
    // генерирует исключение OutOfBoundsException
    $cities = array(
        'London',
        'Washington',
        'Paris',
        'Delhi'
    );
    $iterator = new ArrayIterator($cities);
    $iterator->seek(10);
  }
}
```

В данном случае любой запрос действия приведет к исключению, которое будет передано вверх по цепочке выполнения, пока не достигнет ErrorController::errorAction. Это действие проверит тип исключения и установит для кода ответа сервера значение 500 (внутренняя ошибка сервера), а затем сформирует представление с сообщением об ошибке. На рис. 8.1 показан пример такого представления, сгенерированного приведенным кодом.

An error occurred

Application error

Exception information:

Message: Seek position 10 is out of range

Stack trace:

```
#0 /usr/local/apache/htdocs/square/application/modules/sandbox/controllers/ExampleController.p
#1 /usr/local/apache/htdocs/square/library/Zend/Controller/Action.php(513): Sandbox_ExampleCon
#2 /usr/local/apache/htdocs/square/library/Zend/Controller/Dispatcher/Standard.php(289): Zend_
#3 /usr/local/apache/htdocs/square/library/Zend/Controller/Front.php(946): Zend_Controller_Dis
#4 /usr/local/apache/htdocs/square/library/Zend/Application/Bootstrap/Bootstrap.php(77): Zend_
#5 /usr/local/apache/htdocs/square/library/Zend/Application.php(335): Zend_Application_Bootstr
#6 /usr/local/apache/htdocs/square/public/index.php(26): Zend_Application->run()
#7 {main}
```

Request Parameters:

Рис. 8.1. Результат внутренней ошибки, сгенерированной приложением

Если вы попытаетесь получить доступ к несуществующему контроллеру или действию, то получите в ответ ошибку 404 (страница не найдена). Пример показан на рис. 8.2.

An error occurred

Page not found

Exception information:

Message: Invalid controller specified (no)

Stack trace:

```
#0 /usr/local/apache/htdocs/square/library/Zend/Controller/Front.php(946): Zend_Controller_Dis
#1 /usr/local/apache/htdocs/square/library/Zend/Application/Bootstrap/Bootstrap.php(77): Zend_
#2 /usr/local/apache/htdocs/square/library/Zend/Application.php(335): Zend_Application_Bootstr
#3 /usr/local/apache/htdocs/square/public/index.php(26): Zend_Application->run()
#4 {main}
```

Request Parameters:

```
array (
  'controller' => 'no',
```

Рис. 8.2. Результат запроса некорректного или отсутствующего ресурса приложения

Обратите внимание, что кроме сообщения об ошибке стандартное представление также содержит трассировку стека для внутренних запросов, которые привели к исключению, а также список параметров исходного HTTP-запроса. Эта информация будет полезна при выявлении источника ошибки.

СОВЕТ

Поведение стандартного контроллера ошибок удовлетворяет потребности подавляющего большинства приложений Zend Framework: в большинстве случаев ошибка 500 для исключений уровня приложения и ошибка 404 для некорректных запросов — это ожидаемое и соответствующее стандартам поведение, поэтому вам редко потребуется изменять его. Однако если ваше приложение использует собственные классы исключений, вы, возможно, захотите изменить стандартный обработчик и использовать различные процедуры для обработки различных типов исключений. Этот подход рассмотрен в следующем разделе.

Если вы хотите отключить стандартный контроллер ошибок и предпочитаете, чтобы исключения обрабатывались средствами PHP, добавьте в конфигурационный файл вашего приложения следующий параметр:

```
resources.frontController.throwExceptions = 1
```

После этого изменения на разработчика возлагается ответственность за перехват и обработку исключений локально, в области видимости сгенерировавшего их действия. Неперехваченные исключения приведут к критической ошибке и остановят выполнение сценария в точке их возникновения. На рис. 8.3 показан пример неперехваченного исключения, возникшего при отключенном стандартном обработчике.

Fatal error: Uncaught exception 'Zend_Controller_Dispatcher_Exception' with message 'Invalid controller specified (no)' in /usr/local/apache/htdocs/square/library/Zend/Controller/Dispatcher/Standard.php:242 Stack trace: #0 /usr/local/apache/htdocs/square/library/Zend/Controller/Front.php(946): Zend_Controller_Dispatcher_Standard->dispatch(Object(Zend_Controller_Request_Http), Object(Zend_Controller_Response_Http)) #1 /usr/local/apache/htdocs/square/library/Zend/Application/Bootstrap/Bootstrap.php(77): Zend_Controller_Front->dispatch() #2 /usr/local/apache/htdocs/square/library/Zend/Application.php(335): Zend_Application_Bootstrap_Bootstrap->run() #3 /usr/local/apache/htdocs/square/public/index.php(26): Zend_Application->run() #4 {main} thrown in **/usr/local/apache/htdocs/square/library/Zend/Controller/Dispatcher/Standard.php** on line **242**

Рис. 8.3. Результат отключения стандартного контроллера ошибок и передачи обработки исключений непосредственно PHP

Использование пользовательских классов исключений

Если ваше приложение использует пользовательские классы исключений, внесите изменения в стандартный контроллер ошибок, чтобы его поведение зависело от типа перехваченного исключения. Этот подход позволяет разработчику не ограничиваться двумя основными типами ошибок, обрабатываемыми стандартным контроллером, и обеспечивает более точный контроль над обработкой различных типов исключений.

Чтобы показать, как это работает, рассмотрим следующий вариант приведенного выше кода, который выбрасывает пользовательское исключение в зависимости от наличия или отсутствия требуемого файла:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function failAction()
  {
    $this->file = APPLICATION_PATH . '/configs/no.such.file.xml';
    if (!file_exists($this->file)) {
      throw new Example_Invalid_File_Exception(
        'File does not exist: ' . $this->file);
    }
    if (file_exists($this->file)) {
      throw new Example_Duplicate_File_Exception(
        'File already exists: ' . $this->file);
    }
  }
}
```

При таком применении пользовательских исключений в стандартный контроллер ошибок зачастую вносятся изменения, добавляющие специфический код обработки различных типов исключений. Пример:

```
<?php
class ErrorController extends Zend_Controller_Action
{
  public function errorAction()
  {
    $errors = $this->_getParam('error_handler');
    $this->view->exception = $errors->exception;
    $this->view->request = $errors->request;
    switch (get_class($errors->exception)) {
      // handle invalid file exceptions
      case 'Example_Invalid_File_Exception':
        $this->getResponse()->setHttpResponseCode(500);
        $this->view->title = 'File Not Found';
        $this->view->message =
          'There was an error reading a file needed by the application.';
    break;
      // handle duplicate file exceptions
      case 'Example_Duplicate_File_Exception':
        $this->getResponse()->setHttpResponseCode(500);
        $this->view->title = 'File Already Exists';
        $this->view->message =
          'There was an error writing a file needed by the application.';
    break;
      // handle 404 exceptions
      case 'Zend_Controller_Dispatcher_Exception':
        $this->getResponse()->setHttpResponseCode(404);
```

```
            $this->view->title = 'Page Not Found';
            $this->view->message = 'The requested page could not be found.';
        break;
          // handle generic action exceptions
          case 'Zend_Controller_Action_Exception':
          if ($errors->exception->getCode() == 404) {
            $this->getResponse()->setHttpResponseCode(404);
            $this->view->title = 'Page Not Found';
            $this->view->message = 'The requested page could not be found.';
          } else {
            $this->getResponse()->setHttpResponseCode(500);
            $this->view->title = 'Internal Server Error';
            $this->view->message =
              'Due to an application error, the requested page could
               not be displayed.';
        }
          break;
          // handle everything else
          default:
            $this->getResponse()->setHttpResponseCode(500);
            $this->view->title = 'Internal Server Error';
            $this->view->message =
              'Due to an application error, the requested page could
               not be displayed.';
          break;
        }
      }
}
```

В данном случае действие **errorAction()** начинается с получения параметра запроса **'error_handler'**, содержащего объект исключения, выброшенного в контроллере исходного действия. Затем анализируется класс исключения и, в зависимости от названия этого класса, выполняются различные блоки кода. В данном примере кроме стандартных случаев (запросы некорректных ресурсов и общие неперехваченные исключения) контроллер также содержит специальные блоки для пользовательских классов исключений, используемых в приложении. Разумеется, оператор **switch-case()** можно расширить, добавив поддержку любого требуемого количества пользовательских классов исключений.

Управление видимостью исключений

Вывод трассировки и другой отладочной информации в сообщениях об ошибках после сдачи приложения в эксплуатацию — не очень хорошая идея. Иногда эта информация содержит конфиденциальные сведения, такие как вывод результатов SQL-запросов, расположения файлов и/или пароли пользователей, которыми злоумышленники могут воспользоваться для определения слабых мест приложения и осуществления удаленных атак. Таким образом, сокрытие этой информации при эксплуатации приложения — важный шаг к увеличению его безопасности в целом.

Стандартное приложение Zend Framework содержит параметры конфигурации для четырех окружений: *разработка*, *тестирование*, *отладка* (staging) и *эксплуатация*. При настройках по умолчанию отображение ошибок в этих окружениях осуществляется следующим образом:

- В окружениях для разработки и тестирования конфигурационным переменным PHP 'display_errors' и 'display_startup_errors' присваиваются значения true, в результате чего сообщения об ошибках отображаются пользователям. В окружениях для отладки и эксплуатации этим переменным присваиваются значения false, поэтому сообщения об ошибках скрыты от пользователей.
- В окружении для разработки стандартное представление об ошибке содержит текст сообщения об ошибке, трассировку и список значений переменных, доступных на момент обработки запроса. В окружениях для эксплуатации, отладки и тестирования пользователю отображается только сообщение об ошибке.

Эти параметры можно настроить, изменив соответствующий блок в конфигурационном файле приложения, а также модифицировав представление об ошибке с целью отображения большего или меньшего количества информации в зависимости от текущего окружения (пример приведен в следующем разделе).

ВОПРОС ЭКСПЕРТУ

В: Как мне переключаться между окружениями для эксплуатации и для разработки?

О: Информация о текущем окружении находится в файле .htaccess приложения в виде переменной окружения сервера, носящей название APPLICATION_ENV. При каждом запросе сценарий index.php проверяет значение этой переменной и передает его компоненту Zend_Application. Затем Zend_Application читает соответствующий блок конфигурационного файла приложения и устанавливает значения переменных в соответствии с директивами в этом блоке. Поэтому, чтобы переключиться с одного окружения на другое, просто отредактируйте файл .htaccess и присвойте переменной APPLICATION_ENV имя нужного окружения.

Упражнение 8.1. Создание пользовательской страницы с сообщением об ошибке

Получив всю необходимую информацию, давайте создадим собственную страницу ошибки для демонстрационного приложения SQUARE.

Поскольку у нас нет каких-либо определенных требований касательно обработки исключений, мы можем оставить большую часть контроллера ошибок нетронутой и внести лишь одно незначительное изменение: добавить переменную с заголовком страницы для сценария представления. Вы увидите это изменение в разделе «Упражнение 8.2» и найдете его в архиве с дополнительными файлами к этой главе.

Теперь можно перейти к настройке представления об ошибке, обновив стандартный сценарий представления, $APP_DIR/application/modules/default/views/scripts/error/error.phtml, чтобы добавить дополнительную отладочную информацию:

```
<div>
  <div id="error-image">
    <img src="/images/error.gif" />
  </div>
  <div id="error-message">
    <h1><?php echo $this->title; ?></h1>
    <?php echo $this->message; ?>
  </div>
</div>

<?php if ('development' == APPLICATION_ENV): ?>
<div id="error-data">
  <h2>Exception message:</h2>
  <pre><?php echo $this->exception->getMessage(); ?></pre>

  <h2>Stack trace:</h2>
  <pre><?php echo $this->exception->getTraceAsString() ?></pre>

  <h2>Request method:</h2>
  <pre><?php echo $this->request->getMethod(); ?></pre>

  <h2>Request parameters:</h2>
  <?php Zend_Debug::dump($this->request->getParams()); ?>

  <h2>Server environment:</h2>
  <?php Zend_Debug::dump($this->request->getServer()); ?>

  <h2>System environment:</h2>
  <?php Zend_Debug::dump($this->request->getEnv()); ?>
  <?php endif ?>
</div>
```

Это представление об ошибке отображает одно из двух сообщений в зависимости от ее кода: 404 или ошибка сервера 500. Количество отображаемой информации также различается в зависимости от текущего окружения. Например, на рис. 8.4 показан вывод в окружении для эксплуатации, а на рис. 8.5 — в окружении для разработки.

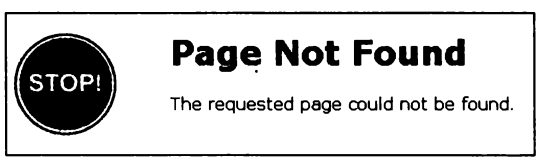


Рис. 8.4. Вывод сообщения об ошибке в окружении для эксплуатации

СОВЕТ

Объект Zend_Controller_Request_Http предоставляет различные методы для получения информации о текущем запросе, и некоторые из них использовались в предыдущем листинге. Метод getParams() возвращает массив текущих параметров запроса, метод getServer() возвращает массив переменных окружений сервера ($_SERVER), а метод getEnv() возвращает массив текущих системных переменных окружения ($_ENV).

Рис. 8.5. Вывод сообщения об ошибке в окружении для разработки

ВОПРОС ЭКСПЕРТУ

В: Что такое Zend_Debug?

О: Компонент Zend_Debug предоставляет удобный способ для исследования переменных в PHP. Он содержит единственный открытый статический метод dump(), который можно использовать для вывода содержимого переменных на стандартное устройство вывода. Этот метод часто используется для быстрого просмотра значений переменных, особенно при отладке и журналировании ошибок. По своей сути метод Zend_Debug::dump() является оберткой вокруг функции PHP var_dump(), помещая ее вывод в теги <pre>...</pre> и приводя в порядок переносы строк для улучшения форматирования.

Журналирование данных

В предыдущем разделе вы уже видели, как можно изменить отладочную информацию, выводимую пользователям в окружениях для эксплуатации. Однако несмотря на то что эта информация не должна быть видна пользователям, она представляет ценность для разработчика и администратора приложения. Поэтому Zend Frame-

work содержит компонент **Zend_Log**, предоставляющий универсальную основу для организации ведения журналов любых типов данных (в том числе исключений) на дисках или в других типах хранилищ. Как правило, эта информация анализируется и используется для отладки, аудита и формирования отчетов.

Компонент **Zend_Log** чрезвычайно гибок. Он поддерживает вывод журналов в различные приемники (стандартное устройство вывода, файл, база данных, адрес электронной почты или системный журнал) и несколько приоритетов журналирования (от «информационного» до «аварийного»). В число других функций входят поддержка определенных пользователями уровней приоритетов и форматов журнала, фильтрация журналов по приоритетам, а также поддержка вывода в консоль Firebug (отладчик для браузера Firefox).

В следующих разделах эти возможности рассмотрены более подробно.

Запись сообщений журнала

Ниже приведен простой пример использования **Zend_Log** для записи сообщения в файл в указанном формате:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function logAction()
  {
    // инициализируем механизм журналирования
    $logger = new Zend_Log();

    // добавляем к механизму журналирования класс записи
    $writer = new Zend_Log_Writer_Stream(
      APPLICATION_PATH . '/../data/logs/example.log');
    $logger->addWriter($writer);

    // добавляем к классу записи класс форматирования
    $format = '%timestamp%: %priorityName%: %message%' . PHP_EOL;
    $formatter = new Zend_Log_Formatter_Simple($format);
    $writer->setFormatter($formatter);

    // записываем сообщение в журнал
    $logger->log('Body temperature critical', Zend_Log::EMERG);
  }
}
```

На рис. 8.6 показан пример вывода.

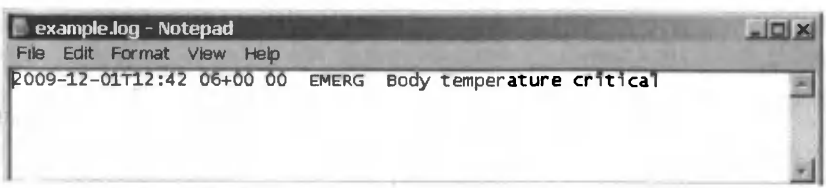


Рис. 8.6. Пример сообщения журнала, записанного в файл

Из предыдущего примера видно, что в процессе записи сообщений журнала участвуют несколько объектов:

- Объект Zend_Log служит основой для генерации сообщений журнала. Он предоставляет метод log(), который начинает операцию записи в журнал.
- Объект Zend_Log_Writer отвечает за процесс записи сообщения в указанное хранилище данных, будь то файл, база данных, адрес электронной почты или системный журнал. Этот объект добавляется к объекту Zend_Log с помощью метода addWriter() последнего. К одному объекту Zend_Log могут быть добавлены несколько классов записи; в табл. 8.2 приведены те из них, которые входят в состав Zend Framework.
- Объект Zend_Log_Formatter отвечает за преобразование формата сообщения журнала либо в XML, либо в формат, определенный пользователем. Zend_Log_Formatter всегда связан с классом записи и добавляется к нему с помощью метода setFormatter() последнего. Каждому классу записи может соответствовать только один класс форматирования. В табл. 8.3 приведены классы форматирования, входящие в состав Zend Framework.

Таблица 8.2. Классы записи журналов, входящие в состав Zend Framework

Класс записи	Описание
Zend_Log_Writer_Stream	Записывает сообщения журнала в потоки PHP (устройство вывода, устройство ошибок, URL или локальные файлы)
Zend_Log_Writer_Db	Записывает сообщения журнала в базу данных
Zend_Log_Writer_Firebug	Записывает сообщения журнала в консоль Firebug
Zend_Log_Writer_Mail	Отправляет сообщения журнала по указанному адресу электронной почты
Zend_Log_Writer_Mock	Выводит сообщения журнала на страницу в целях тестирования
Zend_Log_Writer_Null	Записывает сообщения журнала в dev/null /
Zend_Log_Writer_Syslog	Записывает сообщения журнала в системный журнал (syslogd в UNIX и Просмотр Событий в Windows)

Таблица 8.3. Классы форматирования журналов, входящие в состав Zend Framework

Объект форматирования	Описание
Zend_Log_Formatter_Simple	Форматирует сообщения журнала в соответствии с определенным пользователем форматом
Zend_Log_Formatter_Firebug	Форматирует сообщения журнала для консоли Firebug
Zend_Log_Formatter_Xml	Форматирует сообщения журнала в соответствии с форматом XML

Также обратите внимание на второй параметр, передаваемый методу log() объекта Zend_Log. Он определяет приоритет сообщения, который может принимать значение одного из восьми предопределенных уровней. Эти уровни приведены в табл. 8.4 и практически полностью соответствуют уровням, используемым службой журналирования UNIX (syslog). Разработчик может определять уровень для каждого сообщения в зависимости от степени важности.

Таблица 8.4. Уровни приоритета, поддерживаемые компонентом Zend_Log

Уровень приоритета	Описание
DEBUG	Отладочное сообщение
INFO	Информационное сообщение
NOTICE	Сообщение
WARNING	Предупреждение
ERR	Сообщение об ошибке
CRIT	Сообщение о критической ошибке
ALERT	Тревожное сообщение
EMERG	Сообщение об аварийной ситуации

Для записи сообщений не в файл, а в базу данных (что рекомендуется при высокой интенсивности журналирования) используйте класс записи Zend_Log_Writer_Db. Он инициализируется экземпляром класса Zend_Db_Adapter (представляющим используемое подключение к базе данных), таблицей для записи и массивом, указывающим, какие поля таблицы следует использовать для различных элементов сообщения. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function logAction()
  {
    // инициализируем механизм журналирования
    $logger = new Zend_Log();

    // устанавливаем соответствие между записью и таблицей
    $table = 'log';
    $fields = array(
        'LogLevel' => 'priority',
        'LogMessage' => 'message',
        'LogTime' => 'timestamp'
    );

    // добавляем к механизму журналирования класс записи
    $writer = new Zend_Log_Writer_Db($this->adapter, $table, $fields);
    $logger->addWriter($writer);
```

```
    // записываем сообщение в журнал
    $logger->log('Body temperature critical', Zend_Log::EMERG);
  }
}
```

ВНИМАНИЕ

Для объекта, производящего запись в базу данных, невозможно использовать класс форматирования. Попытка это сделать приведет к исключению.

Также можно отсылать сообщения журнала на адрес электронной почты, используя класс записи **Zend_Log_Writer_Mail**, который принимает на вход настроенный компонент **Zend_Mail**. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function logAction()
  {
    // создаем шаблон сообщения электронной почты
    $mail = new Zend_Mail();
    $mail->setFrom('error-bot@host.example.com')
        ->addTo('admin@host.example.com');
    $mail->setSubject('Email error log');

    // инициализируем механизм журналирования
    $logger = new Zend_Log();

    // добавляем к механизму журналирования класс записи
    $writer = new Zend_Log_Writer_Mail($mail);
    $logger->addWriter($writer);

    // записываем сообщение в журнал
    // примечание: для корректной работы этой функции должен
    // быть определен агент пересылки почты
    $logger->log('Body temperature critical', Zend_Log::EMERG);
  }
}
```

ВНИМАНИЕ

Отправка журнала по электронной почте может значительно снизить производительность, особенно если используемый почтовый сервер находится с приложением на разных компьютерах.

Чтобы записывать сообщения непосредственно в системный журнал (syslogd в UNIX и **Event Viewer** в Windows), используйте класс записи **Zend_Log_Writer_Syslog**, как показано в следующем примере. Обратите внимание на использование метода **setFacility()**, с помощью которого устанавливается категория (facility) сообщения для службы журналирования UNIX.

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
```

```
{
  public function logAction()
  {
    // инициализируем механизм журналирования
    $logger = new Zend_Log();

    // добавляем к механизму журналирования класс записи
    $writer = new Zend_Log_Writer_Syslog();
    $writer->setFacility(LOG_USER);
    $logger->addWriter($writer);

    // записываем сообщение в журнал
    $logger->log('Body temperature critical', Zend_Log::EMERG);
  }
}
```

Стоит отметить, что если ни одна из перечисленных методик не подходит под требования вашего приложения, вы можете создать пользовательские классы записи для использования других хранилищ данных, просто расширив базовый класс **Zend_Log_Writer_Abstract**. Производный класс должен предоставлять как минимум защищенный метод **_write()**, принимающий на вход сообщение журнала и записывающий его в указанный приемник. В следующем разделе приведен пример такого класса.

Добавление данных в сообщения журнала

По умолчанию каждое сообщение журнала содержит четыре ключа: *текст сообщения*, *приоритет* (как в виде числа, так и в виде строки) и *временную метку*, указывающую на время генерации сообщения. Однако к сообщению можно добавить и определенные пользователем ключи с помощью метода **setEventItem()** объекта **Zend_Log**. Метод принимает два параметра — ключ и значение — и добавляет эту пару к сообщению журнала. Он особенно полезен, если вы хотите включить в сообщение дополнительную отладочную информацию.

Ниже приведен пример, добавляющий к сообщению журнала IP-адрес клиента и запрошенный им URL:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function logAction()
  {
    // инициализируем механизм журналирования
    $logger = new Zend_Log();

    // добавляем к механизму журналирования класс записи
    $writer = new Zend_Log_Writer_Mock();
    $logger->addWriter($writer);

    // добавляем к сообщению IP-адрес клиента и запрошенный URL
```

```
    $logger->setEventItem('request', $this->getRequest()->getRequestUri());
    $logger->setEventItem('host', $this->getRequest()->getClientIp());

    // записываем сообщение в журнал
    $logger->log('Body temperature critical', Zend_Log::EMERG);
  }
}
```

Форматирование сообщений журнала

У добавляемых в сообщения журнала ключей есть и другое предназначение: их можно использовать для создания пользовательской строки форматирования сообщений. Она используется объектом **Zend_Log_Formatter_Simple** для настройки формата, в котором сообщения будут записываться в приемник. Спецификаторы формата помещаются между символами процента (**%**) (см. первый листинг в этом разделе).

Стандартная строка форматирования содержит временную метку, уровень приоритета (как число, так и строку) и текст сообщения. Однако можно изменить этот формат и поменять элементы для соответствия требованиям приложения. Для иллюстрации рассмотрим другой пример, в котором используются дополнительные ключи, установленный методом **setEventItem()**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function logAction()
  {
    // инициализируем механизм журналирования
    $logger = new Zend_Log();

    // добавляем к механизму журналирования класс записи
    $writer = new Zend_Log_Writer_Stream(
        APPLICATION_PATH . '/../data/logs/example.log');
    $logger->addWriter($writer);

    // добавляем к классу записи класс форматирования
    $format = '%timestamp%: %priorityName%: %request%:
         %host%: %message%' . PHP_EOL;
    $formatter = new Zend_Log_Formatter_Simple($format);
    $writer->setFormatter($formatter);

    // добавляем к сообщению IP-адрес клиента и запрошенный URL
    $logger->setEventItem('request', $this->getRequest()->getRequestUri());
    $logger->setEventItem('host', $this->getRequest()->getClientIp());

    // записываем сообщение в журнал
    $logger->log('Body temperature critical', Zend_Log::EMERG);
  }
}
```

Пример вывода, создаваемого приведенным листингом, показан на рис. 8.7.

ВНИМАНИЕ

При использовании класса форматирования Zend_Log_Formatter_Simple дополнительные ключи, указанные методом setEventItem(), по умолчанию не будут вноситься в журнал. Для записи этих элементов необходимо явно включить их в строку формата, переданную классу форматирования.

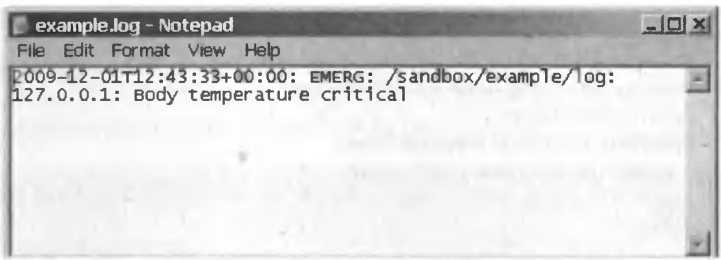


Рис. 8.7. Пример сообщения журнала, содержащего дополнительную информацию

Чтобы вести журнал в формате XML, а не в виде обычного текста, замените класс форматирования на Zend_Log_Formatter_Xml следующим образом:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function logAction()
  {
    // инициализируем механизм журналирования
    $logger = new Zend_Log();

    // добавляем к механизму журналирования класс записи
    $writer = new Zend_Log_Writer_Stream(
        APPLICATION_PATH . '/../data/logs/example.log');
    $logger->addWriter($writer);

    // добавляем к классу записи класс форматирования
    $formatter = new Zend_Log_Formatter_Xml();
    $writer->setFormatter($formatter);

    // добавляем к сообщению IP-адрес клиента и запрошенный URL
    $logger->setEventItem('request', $this->getRequest()->getRequestUri());
    $logger->setEventItem('host', $this->getRequest()->getClientIp());

    // записываем сообщение в журнал
    $logger->log('Body temperature critical', Zend_Log::EMERG);
  }
}
```

Теперь сообщения журнала представлены элементами XML, пригодными для обработки любым анализатором XML (рис. 8.8).

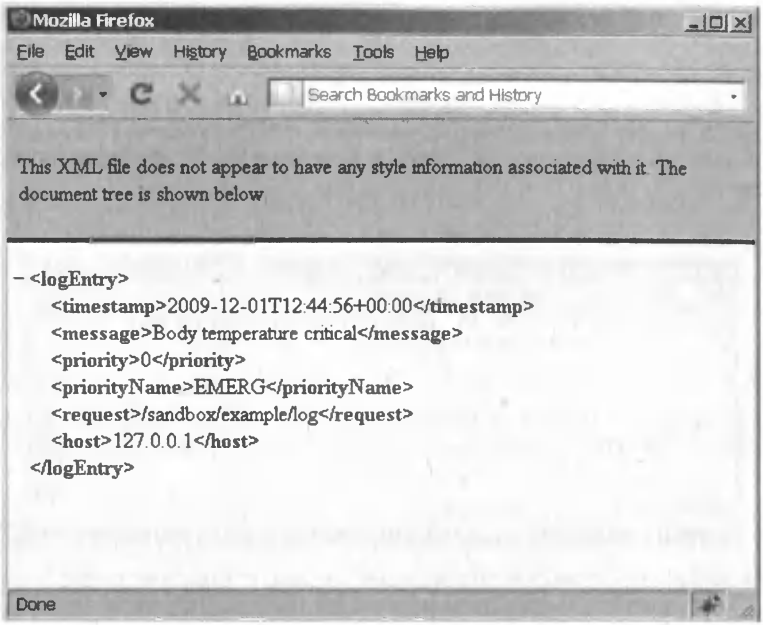


Рис. 8.8. Пример сообщения журнала, представленного в формате XML

Как и в случае с классом форматирования **Zend_Log_Formatter_Simple**, получаемый XML можно настроить по собственному усмотрению, указав другие имена элементов. Эта информация передается конструктору объекта **Zend_Log_Formatter_Simple** в виде массива. Пример, демонстрирующий данную возможность:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function logAction()
  {
    // инициализируем механизм журналирования
    $logger = new Zend_Log();

    // добавляем к механизму журналирования класс записи
    $writer = new Zend_Log_Writer_Stream(
        APPLICATION_PATH . '/../data/logs/example.log');
    $logger->addWriter($writer);

    // настраиваем параметры элементов XML
    $xml = array(
        'data' => 'message',
        'time' => 'timestamp',
        'level' => 'priorityName',
        'request' => 'request',
        'host' => 'host',
    );
```

```
        // добавляем к классу записи класс форматирования
        $formatter = new Zend_Log_Formatter_Xml('entry', $xml);
        $writer->setFormatter($formatter);

        // добавляем к сообщению IP-адрес клиента и запрошенный URL
        $logger->setEventItem('request', $this->getRequest()->getRequestUri());
        $logger->setEventItem('host', $this->getRequest()->getClientIp());

        // записываем сообщение в журнал
        $logger->log('Body temperature critical', Zend_Log::EMERG);
    }
}
```

Пример измененного вывода показан на рис. 8.9.

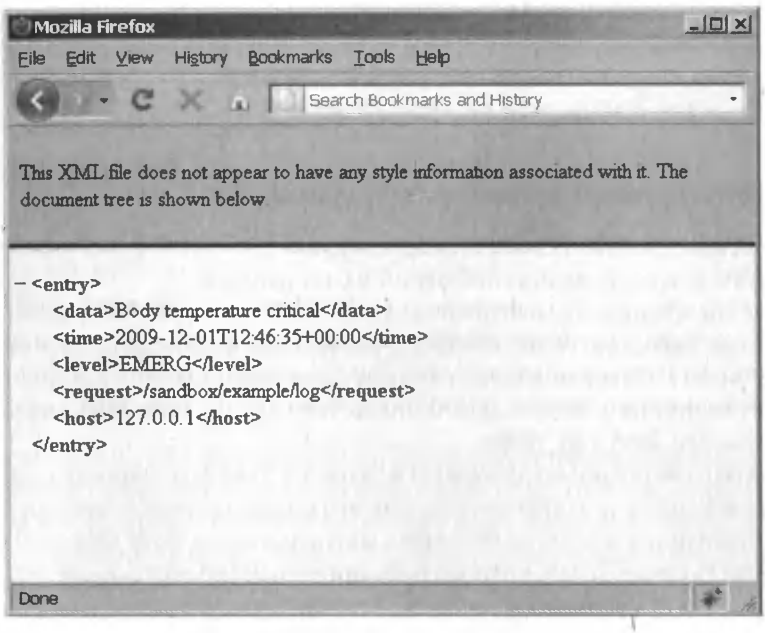


Рис. 8.9. Пример сообщения журнала, представленного в формате XML с пользовательскими именами элементов

Упражнение 8.2. Журналирование исключений приложения

Теперь давайте применим полученную информацию на практике, добавив в демонстрационное приложение SQUARE полноценную систему журналирования исключений, способную записывать данные об исключениях в файл, в базу данных и в консоль Firebug. Как вы вскоре увидите, это довольно простая задача, решение которой не составит труда. Процесс разработки описан в следующих разделах.

ПРИМЕЧАНИЕ

В следующих разделах, помимо прочего, рассматривается вывод сведений об исключениях в консоль Firebug, отладчика для браузера Firefox. Предполагается, что у вас уже установлен и работает браузер Firefox с расширениями Firebug и FirePHP. Если это не так, вы можете загрузить перечисленные компоненты из Интернета по ссылкам в конце главы.

Определение расположения журнала

Для начала определим расположение файла журнала. Создайте каталог **$APP_DIR/data/logs/**, который рекомендуется использовать для журналов в структуре каталогов Zend Framework.

```
shell> cd /usr/local/apache/htdocs/square/data
shell> mkdir logs
```

Внесите путь к этому каталогу в конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**, чтобы его можно было использовать в действиях. Откройте файл и внесите в него следующий параметр:

```
logs.logPath = APPLICATION_PATH "/../data/logs"
```

Определение класса записи в базу данных

Следующий шаг — создание класса записи журнала, который будет взаимодействовать с **Doctrine** для сохранения сообщений в базу данных.

Как уже говорилось, это несложная задача: ее можно решить, просто расширив базовый класс **Zend_Log_Writer_Abstract** и реализовав в производном классе метод **_write()**, которому известно, как осуществлять операцию записи в журнал. Однако у вас может возникнуть вопрос, зачем это вообще нужно, если **Zend_Log** уже содержит класс записи **Zend_Log_Writer**.

Причины были подробно изложены в главе 5: это эффективность и согласованность. Согласованность заключается в том, что в данный момент все запросы к базе данных выполняются через модели **Doctrine**, и использование **Zend_Db** внесет путаницу. Эффективность связана с тем, что использование одной библиотеки для доступа к базе данных вместо двух уменьшит количество компонентов, загружаемых приложением, и количество конфигурационной информации, которую придется продублировать.

С учетом этого объяснения перейдем к созданию пользовательского класса записи **Square_Log_Writer_Doctrine**, представленного следующим кодом:

```
<?php
class Square_Log_Writer_Doctrine extends Zend_Log_Writer_Abstract
{
    // конструктор
    // принимает название модели и массив, устанавливающий соответствие
    // между записью и столбцами таблицы
    public function __construct($modelName, $columnMap)
    {
      $this->_modelName = $modelName;
      $this->_columnMap = $columnMap;
    }
```

```
    // функция-заглушка, запрещающая
    // использование классов форматирования
    public function setFormatter($formatter)
    {
      require_once 'Zend/Log/Exception.php';
      throw new Zend_Log_Exception(get_class() . ' does not support formatting');
    }

    // основной метод для записи в журнал
    // сопоставляет поля таблицы полям сообщения,
    // сохраняет сообщения журнала в виде записей базы данных,
    // используя методы модели
    protected function _write($message)
    {
      $data = array();
    foreach ($this->_columnMap as $messageField => $modelField) {
      $data[$modelField] = $message[$messageField];
    }
    $model = new $this->_modelName();
    $model->fromArray($data);
    $model->save();
  }

  // статический метод-фабрика
  static public function factory($config)
  {
    return new self(self::_parseConfig($config));
  }
}
```

Сохраните код в файл **$APP_DIR/library/Square/Log/Writer/Doctrine.php**.

Приведенный класс основан на прототипе **Zend_Log_Writer_Doctrine**, созданном Мэтью Лурзом (Matthew Lurz) и добавленном на вики-сайт сообщества Zend Framework (см. ссылку в конце главы). Основную работу в классе выполняет метод **_write()**, который инициализирует модель **Doctrine** по указанному имени и добавляет в нее ключи из сообщения журнала. Затем используется метод **save()** этой модели для записи сообщения в базу данных.

Обновление контроллера ошибок

Все, что осталось сделать, — обновить действие **ErrorController::errorAction**, чтобы автоматически вносить исключения в файл и базу данных и выводить их на консоль **Firebug**. Ниже приведен исправленный контроллер из файла **$APP_DIR/application/modules/default/controllers/ErrorController.php**, а изменения выделены жирным шрифтом:

```
<?php
class ErrorController extends Zend_Controller_Action
{

  public function errorAction()
```

```
{
  $errors = $this->_getParam('error_handler');
  switch ($errors->type) {
    case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
    case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
      // 404 error -- controller or action not found
      $this->getResponse()->setHttpResponseCode(404);
      $this->view->title = 'Page Not Found';
      $this->view->message = 'The requested page could not be found.';
      break;
    default:
  // application error
  $this->getResponse()->setHttpResponseCode(500);
  $this->view->title = 'Internal Server Error';
  $this->view->message =
        'Due to an application error, the requested page could
        not be displayed.';
      break;
  }
  $this->view->exception = $errors->exception;
  $this->view->request = $errors->request;

  // инициализируем механизм журналирования
  $logger = new Zend_Log();

  // добавляем класс записи в формате XML
  $config = $this->getInvokeArg('bootstrap')->getOption('logs');
  $xmlWriter = new Zend_Log_Writer_Stream(
    $config['logPath'] . '/error.log.xml');
  $logger->addWriter($xmlWriter);
  $formatter = new Zend_Log_Formatter_Xml();
  $xmlWriter->setFormatter($formatter);

  // добавляем класс записи, использующий Doctrine
  $columnMap = array(
      'message' => 'LogMessage',
      'priorityName' => 'LogLevel',
      'timestamp' => 'LogTime',
      'stacktrace' => 'Stack',
      'request' => 'Request',
  );

  $dbWriter = new Square_Log_Writer_Doctrine('Square_Model_Log', $columnMap);
  $logger->addWriter($dbWriter);

  // добавляем класс записи для консоли Firebug
  $fbWriter = new Zend_Log_Writer_Firebug();
  $logger->addWriter($fbWriter);

  // добавляем к сообщению журнала дополнительные данные:
  // трассировку стека и параметры запроса
```

```
        $logger->setEventItem('stacktrace',
          $errors->exception->getTraceAsString());
        $logger->setEventItem('request',
          Zend_Debug::dump($errors->request->getParams()));

        // передаем сообщение классу записи для внесения в журнал
        $logger->log($errors->exception->getMessage(), Zend_Log::ERR);
    }
}
```

Этот код инициализирует экземпляр компонента **Zend_Log** и добавляет к нему три класса записи: стандартный для записи в файл, класс для записи в базу данных с использованием **Doctrine**, созданный в предыдущем разделе, и класс для вывода в консоль **Firebug**. Для добавления к сообщению журнала двух дополнительных полей — трассировки стека исключения и списка параметров запроса — используется метод **setEvent()** объекта **Zend_Log**. В результате этих изменений любое исключение, достигшее действия **ErrorController::errorAction**, будет автоматически записано в три различных приемника.

Чтобы убедиться в этом, попробуйте перейти по несуществующему URL, например **http://square.localhost/no/such/resource**. Исключение дойдет до контроллера ошибок, который отправит клиенту ответ с кодом 404. Исключение также будет записано в файл (рис. 8.10), в базу данных приложения (рис. 8.11) и в консоль Firebug (рис. 8.12).

Рис. 8.10. Возникшее в приложении исключение, записанное в файл в формате XML

Рис. 8.11. Возникшее в приложении исключение, записанное в базу данных приложения

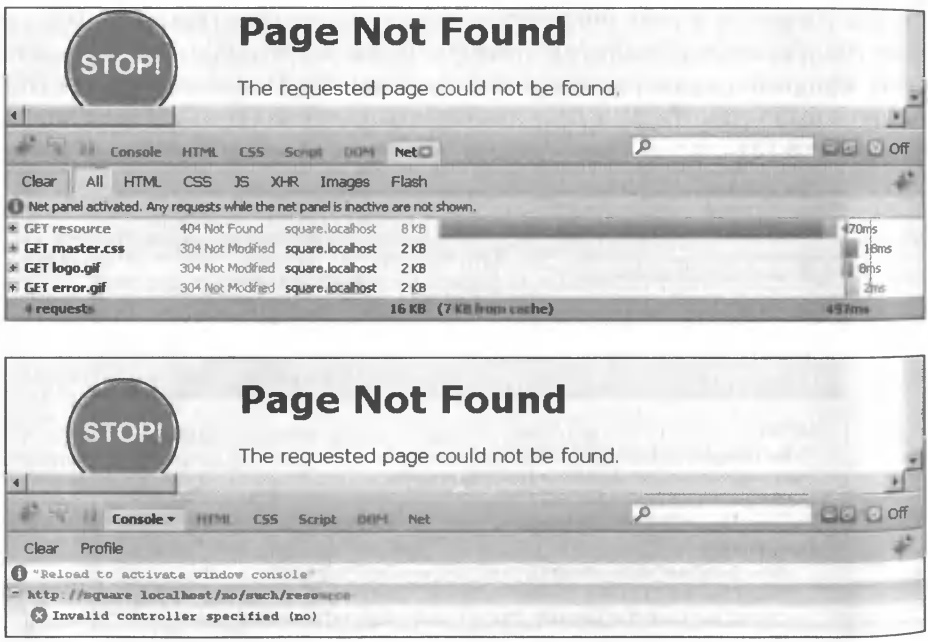


Рис. 8.12. Возникшее в приложении исключение, записанное в консоль Firebug

ВНИМАНИЕ

Следует отметить, что журналирование исключений в три различных приемника, рассмотренное в этом разделе, не рекомендуется использовать в процессе эксплуатации приложения. Запись сообщений в файл или базу данных может сказаться на производительности из-за блокировок или задержек в сети, а вывод в консоль Firebug увеличивает размер пакета с ответом, отправляемого клиенту сервером. При добавлении в приложение Zend Framework возможности ведения журналов следует тщательно анализировать эти и другие проблемы с производительностью.

Выводы

Эта глава завершает первую часть книги, которая должна была познакомить вас с основными приемами разработки приложений с использованием Zend Framework. В ней мы рассмотрели обработку исключений — важную часть процесса разработки приложений и область, в которой должен хорошо ориентироваться каждый разработчик.

Глава началась с введения в модель исключений, появившуюся в PHP 5.x, и краткого описания ее достоинств и преимуществ по сравнению со старой моделью PHP 4.x. После этого была рассмотрена стандартная система обработки исключений, входящая в состав Zend Framework, изложены функции стандартного контроллера ошибок и рассмотрены способы изменения стандартного процесса обработки ошибок в соответствии с различными ситуациями. Было кратко объяснено, каким образом можно (и нужно) изменять обработку ошибок в зависимости от того, запущено приложение в окружении для разработки или находится в процессе эксплуатации, а с помощью пользовательской страницы с сообщением об ошибке этот процесс был показан на практике.

Вторая часть главы в основном была посвящена компоненту Zend_Log, предоставляющему полнофункциональную основу для организации журналирования в приложении Zend Framework. Этот компонент можно использовать для записи сообщений в различные хранилища данных, включая файл, базу данных, адрес электронной почты, системный журнал и консоль Firebug, и его можно легко дополнить поддержкой новых приемников. Некоторые из перечисленных возможностей были применены на практике для добавления в демонстрационное приложение SQUARE кода, реализующего журналирование исключений.

Чтобы получить больше информации по темам, рассмотренным в этой главе, посетите следующие ссылки:

- Обзор исключений и способов их обработки в PHP 5.x: http://www.php.net/manual/en/language.exceptions.php.
- Обзор исключений и способов их обработки в Zend Framework: http://framework.zend.com/manual/en/zend.controller.exceptions.html.
- Программное дополнение к Zend Framework для обработки ошибок: http://framework.zend.com/manual/en/zend.controller.plugins.html.
- Компонент Zend_Log: http://framework.zend.com/manual/en/zend.log.html.
- Компонент Zend_Debug: http://framework.zend.com/manual/en/zend.debug.html.
- Веб-браузер Firefox: http://www.mozilla.com/firefox.
- Консольный отладчик Firebug: http://www.getfirebug.com/.
- Расширение FirePHP: http://www.firephp.org/.
- Использование компонентов Firebug и FirePHP с Zend Framework (Кристоф Дорн (Christoph Dorn)): http://www.christophdorn.com/Blog/2008/09/02/firephp-andzend-framework-16/.

- Обработка ошибок в приложении Zend Framework (Яни Хартикайнен (Jani Hartikainen)): http://codeutopia.net/blog/2009/03/02/handling-errors-in-zend-framework/.
- Усовершенствованный класс форматирования исключений Zend Framework (Ларри Рут (Larry Root)): http://code.google.com/p/zend-framework-exception-formatter/.
- Первоначально предложенный вариант и исходный код класса записи журналов, использующего Doctrine (Мэтью Лурз (Matthew Lurz)): http://framework.zend.com/wiki/display/ZFPROP/Zend_Log_Writer_Doctrine+-+Matthew+Lurz.

9 Локализация приложений

Прочитав эту главу, вы:

- локализуете ваше приложение для различных регионов;
- научитесь автоматически применять локальные правила для дат, валют, температур и единиц измерения;
- создадите многоязычное веб-приложение.

Одной из лучших черт Всемирной паутины является ее «всемирность»: люди со всего мира могут получать доступ к веб-приложениям и использовать их, не имея ничего, кроме браузера и подключения к Интернету. Стремительное развитие таких приложений, как Flickr, Google, Twitter, MySpace и Facebook, свидетельствуют о глобальном распространении Интернета и его популярности в качестве средства общения и совместной работы.

Если вы разрабатываете веб-приложение, то должны знать об одном важном следствии, вытекающем из этого факта. Когда ваше приложение станет доступно в Интернете, весьма вероятно, что его будут использовать люди, которые могут не говорить на вашем языке или даже жить в другой стране. Чтобы сделать приложение доступным для таких пользователей, необходимо обеспечить поддержку местных языков, символов и форматов, принятых в их странах. Этого подхода придерживается большинство популярных сегодня приложений; посмотрите на любое из перечисленных выше, и вы найдете версию, переведенную на ваш язык или диалект.

Zend Framework дает возможность без проблем осуществить локализацию приложения с помощью компонента Zend_Locale, а многие другие компоненты, такие как Zend_Date и Zend_Currency, могут автоматически изменять свое поведение в зависимости от выбранного пользователем местонахождения. Кроме того, существует компонент Zend_Translate, который предоставляет полноценный API для работы со строками перевода и их использования. В этой главе детально рассмотрены перечисленные компоненты и показано, как можно объединить их, чтобы создать локализованное многоязычное приложение.

Локализация и локали

Сторонники глобализации хотят убедить нас в том, что мир становится все более однородным, но это не исключает национальных особенностей и культурных традиций. Эти различия присутствуют как на уровне восприятия: например, считается, что итальянцы хорошо разбираются в моде, а французы — нация гурманов, — так и на уровне более прозаических культурных особенностей, таких как язык, валюта, единицы измерений и календари. Например, в США дата 4 июля 2011 г. записывается как 7/4/2011, а в Индии ту же самую дату написали бы как 4/7/2011. То же самое и с расстояниями: в Германии, Франции, Италии, Индии и многих других странах их измеряют в километрах, а в США — в милях.

Локализация — это процесс адаптации приложения под соглашения, принятые в определенном географическом регионе, стране и/или языке. Он подразумевает выполнение как простых действий, например отображение дат, времени и валюты в соответствующем выбранному региону формате, так и сложных, таких как перевод меток, меню и сообщений на язык(и) этого региона. Причина проста: чем более «локально» приложение, тем легче пользователям из различных регионов в нем разобраться и начать его использовать. Это обычная практика для большинства популярных приложений: посмотрите на рис. 9.1, где показан внешний вид поисковой системы Google.com в трех различных странах: США, Франции и Индии.

Рис. 9.1. Внешний вид поисковой системы Google для различных стран

В основе процесса локализации лежит концепция *локалей* и *идентификаторов локалей*. Идентификатор локали — это строка, описывающая язык и географиче-

ский регион пользователя. Она состоит из кода языка и кода страны, разделенных символом подчеркивания. Задача этого идентификатора — предоставить приложению точную информацию о текущем местонахождении и языке пользователя, тем самым позволяя представлять информацию в соответствии с принятыми для этого расположения и языка правилами.

Поясним на примере. Предположим, что для некоторого приложения установлена локаль **en_US** (язык: английский, страна: США). Если приложение учитывает локаль, оно использует эту информацию, чтобы обеспечить, например, представление дат в формате ММ-ДД-ГГГГ и добавление к валюте префикса в виде символа доллара ($). Если для этого же приложения изменить локаль на **de_AT** (язык: немецкий, страна: Австрия), оно автоматически произведет необходимую внутреннюю настройку, чтобы представлять даты в формате ДД-ММ-ГГГГ и добавлять к валюте символ евро (€).

Таблица 9.1. Распространенные идентификаторы локалей

Локаль	Язык	Страна
en_US	английский	США
en_GB	английский	Великобритания
fr_CA	французский	Франция
fr_FR	французский	Канада
fr_BE	французский	Бельгия
fr_CH	французский	Швейцария
de_DE	немецкий	Германия
it_IT	итальянский	Италия
es_ES	испанский	Испания
es_US	испанский	США
cs_CZ	чешский	Чехия
hi_IN	хинди	Индия
mr_IN	маратхи	Индия
en_IN	английский	Индия
pt_BR	португальский	Бразилия
zh_CN	китайский	Китай

В табл. 9.1 приведен список распространенных идентификаторов локалей с названиями языков и стран, которые они представляют.

ПРИМЕЧАНИЕ

Полный список кодов языков, используемых в идентификаторах локалей, доступен в стандарте ISO 639, а соответствующий список кодов стран — в стандарте ISO 3166. Ссылки на стандарты приведены в конце главы.

Установка локали приложения

Zend Framework содержит компонент **Zend_Locale**, который можно использовать для определения локали приложения. С его помощью можно либо устанавливать локаль вручную, либо автоматически определять ее во время выполнения. Пример ручной установки локали **fr_FR** (язык: французский, страна: Франция):

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function localeAction()
  {
    $locale = new Zend_Locale('fr_FR');
  }
}
```

Отдельные компоненты идентификатора локали можно получить с помощью методов **getLanguage()** и **getRegion()** объекта **Zend_Locale**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function localeAction()
  {
    // определяем локаль
    $locale = new Zend_Locale('fr_BE');

    // результат: 'fr'
    $this->view->message = $locale->getLanguage();

    // результат: 'BE'
    $this->view->message = $locale->getRegion();
  }
}
```

Как правило, объект **Zend_Locale** инициализируется в начальном загрузчике и сохраняет свое состояние в реестре приложения. После такой регистрации другие компоненты Zend Framework, учитывающие локаль, например **Zend_Date** и **Zend_Form**, будут автоматически считывать значение выбранной локали и отображать информацию в соответствии с принятыми для нее нормами. Ниже приведен демонстрационный пример, который вы неоднократно встретите в этой главе:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initLocale()
  {
    // определяем локаль
    $locale = new Zend_Locale('fr_FR');

    // регистрируем локаль
    $registry = Zend_Registry::getInstance();
```

```
    $registry->set('Zend_Locale', $locale);
  }
}
```

Если вы не уверены в том, что ваше приложение будет использовать только одну локаль, ее ручная установка на этапе разработки не рекомендуется. Zend_Locale обладает возможностью автоматического определения локали и может динамически устанавливать ее во время выполнения, проанализировав следующие источники информации:

- Параметры локали в клиентском окружении. Большинство современных веб-клиентов позволяют пользователям выбирать предпочтительные язык и локаль для просмотра веб-страниц. В приложении на PHP эта информация доступна в массиве $_SERVER['HTTP_ACCEPT_LANGUAGES']. Если она присутствует, Zend_Locale использует ее для автоматической установки локали приложения. Если пользователь укажет несколько предпочтительных языков и локалей, будет использоваться первая корректная локаль.
- Параметры локали в окружении сервера. Если клиентские настройки недоступны, Zend_Locale проверяет окружение сервера и устанавливает локаль приложения, основываясь на системной локали. В PHP эту информацию можно получить и изменить с помощью функции setlocale().

Для использования автоматического определения локали необходимо инициализировать объект Zend_Locale одной из специальных локалей — 'browser', 'environment' или 'auto'. Использование локали 'browser' ограничивает определение только браузером пользователя, а локали 'environment' — только сервером. Специальная локаль 'auto' (или отсутствие аргументов у конструктора объекта) сообщает Zend_Locale о том, что искать информацию о локали следует как в браузере пользователя, так и в окружении сервера.

Пример, иллюстрирующий описанные действия:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initLocale()
  {
    // определяем локаль из настроек браузера пользователя
    $locale = new Zend_Locale('browser');

    // определяем локаль из окружения сервера
    $locale = new Zend_Locale('environment');

    // автоматически определяем локаль сначала из настроек браузера,
    // затем из настроек сервера
    $locale = new Zend_Locale('auto');

    // автоматически определяем локаль сначала из настроек браузера,
    // затем из настроек сервера
    $locale = new Zend_Locale();
```

```
    // регистрируем локаль
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Locale', $locale);
  }
}
```

Если автоматически определить локаль не удалось, **Zend_Locale** выбросит соответствующее исключение. В этой ситуации принято перехватывать исключение и вручную устанавливать для локали подходящее значение, «угадывая» предпочтения основной аудитории приложения. Например, если приложение представляет собой интернет-магазин, предназначенный для индийских пользователей, следует использовать локаль **hi_IN** в случаях, когда другая информация о ней недоступна. Пример:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initLocale()
  {
    // пробуем определить локаль автоматически
    // если сделать это не удается, устанавливаем ее вручную
    try {
      $locale = new Zend_Locale('browser');
    } catch (Zend_Locale_Exception $e) {
    $locale = new Zend_Locale('hi_IN');
}
    // регистрируем локаль
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Locale', $locale);
  }
}
```

ВНИМАНИЕ

Следует отметить, что Zend_Locale *не будет* выбрасывать исключение, если идентификатор локали содержит только код языка, но не содержит кода региона. Например, в некоторых клиентских и/или серверных окружениях может быть установлена локаль вида en или fr. Такая ситуация может создавать проблемы, когда Zend_Locale используется с другими классами, учитывающими локаль и ожидающими полного идентификатора, такими как Zend_Currency. Указанная проблема не имеет другого решения, кроме ручной установки локали на основании доступных фактов и «угадывания». Новая функциональность, предложенная для добавления в Zend_Locale в будущем, позволит производить «обновление локали», при котором частично определенные идентификаторы (например, en) будут автоматически преобразовываться в полные (например, en_US).

Локализация чисел

В состав компонента **Zend_Locale** входит подкомпонент **Zend_Locale_Format**, предназначенный для помощи в локализации числовых значений. Этот компонент содержит статический метод **toNumber()**, который можно использовать для представления числа в соответствии с локальными соглашениями. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function numberAction()
  {
    // локализуем число
    // результат: '195,740.676'
    $this->view->message = Zend_Locale_Format::toNumber(195740.676,
                array('locale' => 'en_GB'));

    // результат: '1,95,740.676'
    $this->view->message = Zend_Locale_Format::toNumber(
                          195740.676, array('locale' => 'en_GB'));
  }
}
```

Стоит отметить, что, в отличие от **Zend_Date** и **Zend_Currency**, компонент **Zend_Locale_Format** не может непосредственно получать локаль приложения из реестра. Поэтому передача идентификатора локали статическому методу **toNumber()** в качестве аргумента обязательна.

Можно дополнительно настроить выводимое число, указав собственный формат и точность. Эти параметры передаются методу **toNumber()** в виде элементов массива. Примеры:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function numberAction()
  {
    // используем собственный формат числа
    // результат: '1,95,740.67'
    $this->view->message = Zend_Locale_Format::toNumber(195740.676, array(
                    'locale' => 'en_GB',
                    'number_format' => '##,##,##0.00'
                  ));
  }
}
```

Другим примечательным методом компонента **Zend_Locale_Format** является статический метод **convertNumerals()**, который дает возможность конвертировать числа между арабской и римской нотацией. Учтите, что для этого расширение PCRE для PHP должно быть скомпилировано с поддержкой UTF-8 (она часто отсутствует в версиях PHP для Windows). Пример использования метода:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function numberAction()
  {
    // осуществляем преобразования между представлениями чисел
    $this->view->message =
```

```
      Zend_Locale_Format::convertNumerals('1956', 'Latn', 'Thai');
    $this->view->message =
      Zend_Locale_Format::convertNumerals('6482093.6498', 'Latn', 'Deva');
  }
}
```

СОВЕТ

Вы можете получить полный список представлений чисел, поддерживаемых Zend_Locale_Format::convertNumberals, вызвав Zend_Locale::getTranslationList('script').

Локализация дат и времени

Zend Framework содержит компонент, предназначенный для управления отображением времени и его изменения. Этот компонент, **Zend_Date**, предоставляет простой API для создания и форматирования значений даты и времени; он также поддерживает значительно больший диапазон временных меток, чем встроенная в PHP функция **mktime()**, и может автоматически определять локаль приложения (если она установлена с помощью **Zend_Locale**) и представлять значения дат и времени в соответствии с ней.

Простой пример использования **Zend_Date**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function dateAction()
  {
    // устанавливаем значение даты на текущее
    $date = new Zend_Date();

    // результат: 'Dec 18, 2009 5:58:38 AM'
    $this->view->message = $date;
  }
}
```

В этом примере объект **Zend_Date** автоматически инициализируется текущими датой и временем. Его также можно инициализировать указанными датой и временем, передав в конструктор удобочитаемую строку с датой, временную метку UNIX или массив с компонентами даты и времени. Примеры:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function dateAction()
  {
    // устанавливаем дату, используя временную метку UNIX
    $date = new Zend_Date(1261125725);

    // результат: '2009-12-18T08:42:05+00:00'
    $this->view->message = $date->get(Zend_Date::ISO_8601);
```

```
    // устанавливаем дату, используя строку
    // результат: '2010-12-25T15:56:00+00:00'
    $date->set('25/12/2010 15:56');
    $this->view->message = $date->get(Zend_Date::ISO_8601);

    // устанавливаем дату, используя массив
    // возвращает '2010-06-24T16:35:00+00:00'
    $date->set(
      array(
        'year' => 2010,
        'month' => 6,
        'day' => 24,
        'hour' => 16,
        'minute' => 35,
        'second' => 0
      ));
    $this->view->message = $date->get(Zend_Date::ISO_8601);
  }
}
```

ВНИМАНИЕ

Избегайте создания нескольких экземпляров Zend_Date, поскольку каждый из них значительно увеличивает время запуска приложения. Поэтому когда требуется несколько экземпляров, эффективнее повторно использовать существующий экземпляр, а не создавать новые.

Как показано в предыдущем примере, **Zend_Date** содержит набор предопределенных констант для распространенных форматов дат, таких как ISO 8601 и временные метки UNIX. Эти константы можно передать методу **get()** объекта **Zend_Date**, чтобы получить дату и время в правильном формате. В следующем примере показано использование некоторых из констант и получившийся в результате вывод:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function dateAction()
  {
    // устанавливаем дату, используя строку
    $date = new Zend_Date('25/12/2010 15:56');

    // результат: '2010-12-25T15:56:00+00:00'
    $this->view->message = $date->get(Zend_Date::ISO_8601);

    // результат: 'Sat, 25 Dec 2010 15:56:00 +0000'
    $this->view->message = $date->get(Zend_Date::RFC_2822);

    // результат: 'Saturday, 25-Dec-10 15:56:00 UTC'
    $this->view->message = $date->get(Zend_Date::RFC_850);

    // результат: '2010-12-25T15:56:00+00:00'
    $this->view->message = $date->get(Zend_Date::ATOM);
```

```
    // возвращает '2010-06-24T16:35:00+00:00'
    $this->view->message = $date->get(Zend_Date::ISO_8601);

    // результат: 'Saturday, December 25, 2010'
    $this->view->message = $date->get(Zend_Date::DATE_FULL);

    // результат: 'December 25, 2010 3:56:00 PM UTC'
    $this->view->message = $date->get(Zend_Date::DATETIME_LONG);
  }
}
```

СОВЕТ

Если встроенных констант недостаточно для ваших нужд, вы можете создать собственные форматы, используя спецификаторы формата Zend_Date либо встроенной в PHP функции date(). Более подробно это описано в руководстве по Zend Framework, ссылка на соответствующий раздел которого приведена в конце главы.

Zend_Date тоже в полной мере учитывает локаль: компонент автоматически осуществит поиск зарегистрированного объекта Zend_Locale и, если он доступен, будет использовать указанную локаль при форматировании значений даты и времени. Кроме того, можно явно передать идентификатор локали в качестве аргумента конструктора объекта Zend_Date, и в этом случае Zend_Date будет использовать указанное значение. В следующем примере рассмотрены оба варианта:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function dateAction()
  {
    // определяем и регистрируем локаль
    $locale = new Zend_Locale('fr_FR');
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Locale', $locale);

    // создаем объект даты
    // локаль определится автоматически
    $date = new Zend_Date();

    // результат: 'vendredi 18 décembre 2009'
    $this->view->message = $date->get(Zend_Date::DATE_FULL);

    // создаем объект даты
    // передаем ему локаль или объект локали
    $locale = new Zend_Locale('de_DE');
    $date = new Zend_Date($locale);

    // результат: 'Freitag, 18. Dezember 2009 '
    $this->view->message = $date->get(Zend_Date::DATE_FULL);
  }
}
```

Zend_Date обладает встроенной поддержкой часовых поясов, представленной методами getTimezone() и setTimezone(), и автоматически изменяет даты и время в соответствии с выбранным поясом. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function dateAction()
  {
    // устанавливаем дату, используя строку
    $date = new Zend_Date('25/12/2010 15:56');

    // присваиваем часовому поясу значение UTC
    $date->setTimezone('UTC');

    // результат: '2010-12-25T15:56:00+00:00'
    $this->view->message = $date->get(Zend_Date::ISO_8601);

    // присваиваем часовому поясу значение IST
    $date->setTimezone('Asia/Calcutta');

    // результат: '2010-12-25T21:26:00+05:30'
    $this->view->message = $date->get(Zend_Date::ISO_8601);
  }
}
```

Локализация валют

В каждой стране (за исключением стран еврозоны) есть своя собственная валюта, что делает локализацию валют ключевым аспектом разработки приложений. Как и в случае с датами и временем, Zend Framework содержит компонент Zend_Currency, разработанный специально для решения этой задачи. Zend_Currency предоставляет методы для создания строк со значениями валюты, которые соответствуют местным соглашениям и содержат символ валюты, а также ее краткое или полное название. Кроме того, в нем представлены методы для получения информации о регионах, где указанная валюта находится в обращении, и списка валют, находящихся в обращении в указанном регионе.

По умолчанию после создания объект Zend_Currency проверяет присутствие в приложении зарегистрированного объекта Zend_Locale, и если он есть, Zend_Currency будет использовать зарегистрированную локаль для локализации значений валюты. Можно изменить это поведение и явно передать идентификатор локали в качестве аргумента конструктора объекта Zend_Currency. В любом случае Zend_Currency будет автоматически выбирать подходящую валюту для указанной локали. Локализованные значения валюты можно получить с помощью метода toCurrency() объекта Zend_Currency, который в качестве аргумента принимает значение валюты.

Рассмотрим следующий пример, иллюстрирующий использование объекта:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function currencyAction()
  {
    // инициализируем объект валюты
    $currency = new Zend_Currency('fr_FR');

    // результат: '990,65 €'
    $this->view->message = $currency->toCurrency(990.65);
  }
}
```

ВНИМАНИЕ

При передаче объекту Zend_Currency идентификатора локали требуется включать в этот идентификатор как код языка, так и код региона. Отсутствие одного из кодов приведет к исключению, поскольку для правильного определения валюты Zend_Currency требуется полный идентификатор.

В случае с регионами, использующими несколько валют, или просто чтобы переопределить стандартный выбор валюты, конструктор объекта **Zend_Currency** принимает код валюты в качестве необязательного аргумента. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function currencyAction()
  {
    // инициализируем объект валюты
    $currency = new Zend_Currency('fr_FR', 'FRF');

    // результат: '990,65 F'
    $this->view->message = $currency->toCurrency(990.65);
  }
}
```

ПРИМЕЧАНИЕ

Полный список кодов валют приведен в стандарте ISO 4217, ссылка на который дана в конце главы.

С помощью метода **setFormat()** объекта **Zend_Currency** можно настроить строку со значением валюты, возвращаемую этим объектом, изменив позицию символа валюты, ее название, точность представления числа и наличие в выводе трехбуквенного кода валюты. Метод принимает массив пар ключ-значение и соответствующим образом изменяет строку валюты. Пример использования метода:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
```

```
{
  public function currencyAction()
  {
    // инициализируем объект валюты
    $currency = new Zend_Currency('hi_IN');

    // настраиваем формат валюты
    $currency->setFormat(
      array(
        'position' => Zend_Currency::RIGHT,
        'display' => Zend_Currency::USE_NAME,
        'precision' => 2,
      )
    );

    // результат: '990.65 भारतीय रूपया ' '
    $this->view->message = $currency->toCurrency(990.65);

    // создаем собственный формат валюты
    $currency->setFormat(
      array(
        'position' => Zend_Currency::LEFT,
        'display' => Zend_Currency::USE_SYMBOL,
        'precision' => 4,
        'name' => 'Driit',
        'symbol' => 'DRT'
      )
    );

    // результат: 'DRT 990.6500'
    $this->view->message = $currency->toCurrency(990.65);
  }
}
```

СОВЕТ

Zend_Currency дает возможность осуществлять преобразования курсов валют перед локализацией их значений. Другим вариантом является использование пакета Services_ExchangeRates из репозитория PEAR, который предоставляет API для выполнения преобразований валюты в реальном времени с учетом текущих курсов. Более подробную информацию вы сможете найти по ссылкам в конце главы.

Локализация единиц измерения

Единицы измерений — еще одна область, где часто требуется локализация, и на этот случай в Zend Framework тоже есть решение: **Zend_Measure** — компонент, помогающий осуществлять преобразования между различными единицами измерения и представлять результат в соответствии с местными соглашениями. Компонент поддерживает более 30 различных единиц измерения, начиная от распростра-

ненных, таких как длина, вес, объем и температура, и заканчивая более редкими, такими как ускорение, освещенность, мощность, частота и энергия.

Чтобы инициализировать объект **Zend_Measure**, необходимо передать в его конструктор два аргумента: числовое значение и константу, указывающую на меру, представленную этим значением. Рассмотрим следующий пример, демонстрирующий создание объекта **Zend_Measure_Temperature**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function measureAction()
  {
    // инициализируем объект измерения
    $tempF = new Zend_Measure_Temperature(
                98.6, Zend_Measure_Temperature::FAHRENHEIT);

    // результат: '98.6 °F'
    $this->view->message = $tempF;
  }
}
```

Другой пример, показывающий создание объектов **Zend_Measure_Length** и **Zend_Measure_Weight**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function measureAction()
  {
    // инициализируем объект измерения
    $lengthCm = new Zend_Measure_Length(2540, Zend_Measure_Length::CENTIMETER);

    // результат: '2,540 cm'
    $this->view->message = $lengthCm;

    // инициализируем объект измерения
    $weightKg = new Zend_Measure_Weight(6500, Zend_Measure_Weight::KILOGRAM);

    // результат: '6,500 kg'
    //$this->view->message = $weightKg;
  }
}
```

Каждый объект **Zend_Measure** предоставляет метод **convertTo()**, который можно использовать для преобразования из одной единицы измерения в другую. Целевая единица измерения должна быть передана методу в качестве аргумента. Несколько примеров использования метода:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function measureAction()
```

```
  {
    // преобразуем температуру
    // результат: '37.0 °C'
    $tempF = new Zend_Measure_Temperature(
                98.6, Zend_Measure_Temperature::FAHRENHEIT);
    $this->view->message = $tempF->convertTo(
    Zend_Measure_Temperature::CELSIUS);

    // преобразуем длину
    // результат: '1,000.0 in'
    $lengthCm = new Zend_Measure_Length(2540, Zend_Measure_Length::CENTIMETER);
    $this->view->message = $lengthCm->convertTo(Zend_Measure_Length::INCH);

    // преобразуем вес
    // результат: '6,500,000.0 g'
    $weightKg = new Zend_Measure_Weight(6500, Zend_Measure_Weight::KILOGRAM);
    $this->view->message = $weightKg->convertTo(Zend_Measure_Weight::GRAM);
  }
}
```

ВНИМАНИЕ

Zend_Measure не позволяет преобразовывать один тип измерений в другой. Например, попытка преобразовать длину в вес или скорость в частоту приведет к исключению.

Так же как **Zend_Date** и **Zend_Currency**, **Zend_Measure** учитывает локаль. Идентификатор локали можно передать в качестве необязательного третьего аргумента конструктора объекта **Zend_Measure**, а если этот аргумент отсутствует, **Zend_Measure** может использовать значение, указанное в зарегистрированном экземпляре **Zend_Locale**. В следующем примере рассмотрены оба варианта:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function measureAction()
  {
    // определяем и регистрируем локаль
    $locale = new Zend_Locale('pt_BR');
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Locale', $locale);

    // преобразуем температуру, используя локаль по умолчанию
    // результат: '37,0 °C'
    $tempF = new Zend_Measure_Temperature(
                98.6, Zend_Measure_Temperature::FAHRENHEIT);
    $this->view->message = $tempF->convertTo(Zend_Measure_Temperature::CELSIUS);

    // преобразуем длину, используя собственную локаль
    // результат: '1'000.0 in'
```

```
    $lengthCm = new Zend_Measure_Length(
                2540, Zend_Measure_Length::CENTIMETER, 'fr_CH');
    $this->view->message = $lengthCm->convertTo(Zend_Measure_Length::INCH);
  }
}
```

Локализация строк

Хотя в Интернете вы чаще всего будете встречать сайты на английском языке, важно помнить, что существует большой процент пользователей, для которых английский не является родным языком. Этот факт имеет большое значение при локализации веб-приложений, так как основная работа в любом проекте по локализации, как правило, заключается в переводе текстового содержимого приложения на несколько языков.

Компонент **Zend_Translate** из состава Zend Framework предоставляет всеобъемлющий API для работы со строками перевода, представленными на различных языках. Этот компонент поддерживает большое число источников данных для перевода, включая CSV-файлы, массивы PHP, а также файлы в формате GNU gettext, и может быть с легкостью дополнен поддержкой других источников (например, строк перевода, полученных из базы данных). Как и другие компоненты Zend Framework, рассмотренные в этой главе, он в полной мере учитывает локаль и может динамически выбирать подходящий язык для приложения, основываясь на зарегистрированном объекте **Zend_Locale** или клиентских параметрах локали.

Работу с **Zend_Translate** лучше всего показать на примере. Предположим, что существуют следующие исходные файлы переводов для нескольких языков:

```
<?php
// languages/messages.
fr_FR.php
return array(
  'welcome' =>
'Bienvenue',
  'dog' => 'chien',
  'cat' => 'chat',
  'one' => 'un',
  'two' => 'deux',
  'three' => 'trois'
);
```

```
<?php
// languages/messages.
de_DE.php
return array(
  'welcome' =>
'Wilkommen',
  'dog' => 'hund',
  'cat' => 'katz',
  'one' => 'eins',
  'two' => 'zwei',
  'three' => 'drei'
);
```

```
<?php
// languages/messages.
es_ES.php
return array(
  'welcome' =>
'Bienvenido',
  'dog' => 'gato',
  'cat' => 'perro',
  'one' => 'un',
  'two' => 'dos',
  'three' => 'tres'
);
```

Это простейший формат файла перевода, использующий ассоциативный массив PHP, в котором соответствующие ключам значения представляют собой строки, переведенные на местный язык. Теперь можно настроить **Zend_Translate** таким образом, чтобы он читал эти файлы и динамически переводил строки во время выполнения программы в зависимости от выбранной локали. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function translateAction()
  {
    // создаем объект перевода
    // указываем адаптер перевода
    // указываем исходные файлы для каждой локали
    $translate = new Zend_Translate('array', APPLICATION_PATH .
                 '/../languages/messages.es_ES.php', 'es_ES');
    $translate->addTranslation(APPLICATION_PATH .
                 '/../languages/messages.de.php', 'de_DE');
    $translate->addTranslation(APPLICATION_PATH .
                 '/../languages/messages.fr_FR.php', 'fr_FR');

    // устанавливаем текущую локаль
    $translate->setLocale('fr_FR');

    // переводим строку
    // результат: 'un, deux, trois'
    $this->view->message = sprintf('%s, %s, %s',
                           $translate->_('one'),
                           $translate->_('two'),
                           $translate->_('three')
                         );

    // устанавливаем текущую локаль
    $translate->setLocale('de_DE');

    // переводим строку
    // результат: 'eins, zwei, drei'
    $this->view->message = sprintf('%s, %s, %s',
                           $translate->_('one'),
                           $translate->_('two'),
                           $translate->_('three')
                         );
  }
}
```

Объект **Zend_Translate** инициализируется тремя аргументами: типом используемого адаптера перевода, местоположением исходных файлов перевода и локалью для файла. Затем используется метод **setLocale()** для установки текущей локали. Имея эту информацию, **Zend_Translate** может найти подходящий файл перевода и вернуть правильный перевод для каждой строки.

СОВЕТ

Если Zend_Translate не сможет найти источник перевода для указанной локали, он вернет исходный ключ перевода и сгенерирует уведомление об ошибке PHP. Чтобы отключить это уведомление, передайте конструктору объекта Zend_Translate специальный параметр 'disableNotices' в качестве четвертого аргумента.

Работа с адаптерами и источниками данных

Zend_Translate поддерживает несколько различных адаптеров для исходных файлов перевода. В предыдущем примере использовался адаптер Array, подходящий в большинстве случаев. Однако существуют и другие варианты, перечисленные в табл. 9.2.

Таблица 9.2. Адаптеры перевода, входящие в состав Zend Framework

Адаптер	Описание
Zend_Translate_Adapter_Array	Читает исходные файлы перевода, представленные в формате массива PHP
Zend_Translate_Adapter_Csv	Читает исходные файлы перевода, представленные в формате CSV
Zend_Translate_Adapter_Ini	Читает исходные файлы перевода, представленные в формате INI
Zend_Translate_Adapter_Gettext	Читает исходные файлы перевода, представленные в двоичном формате GNU gettext
Zend_Translate_Adapter_Tbx	Читает исходные файлы перевода, представленные в формате TermBase eXchange (TBX)
Zend_Translate_Adapter_Tmx	Читает исходные файлы перевода, представленные в формате Translation Memory eXchange (TMX)
Zend_Translate_Adapter_Qt	Читает исходные файлы перевода, представленные в формате QtLinguist
Zend_Translate_Adapter_Xliff	Читает исходные файлы перевода, представленные в формате XML Localization Interchange File Format (XLIFF)
Zend_Translate_Adapter_XmlTm	Читает исходные файлы перевода, представленные в формате XML Text Memory (XML:TM)

Zend_Translate может рекурсивно просматривать дерево каталогов, автоматически находить источники перевода и добавлять их в объект Zend_Translate. Локаль для каждого файла перевода определяется автоматически на основании имен соответствующих файлов и/или каталогов. Это избавляет от необходимости добавлять источники вручную, используя метод addTranslation().

На рис. 9.2 приведен пример структуры каталогов с исходными файлами перевода, в которой каталоги названы в соответствии с локалями, а в следующем листинге показан код для настройки Zend_Translate на автоматический просмотр этих каталогов и добавление всех найденных источников перевода:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function translateAction()
  {
    // создаем объект перевода
    // указываем адаптер перевода
```

```
    // автоматически находим исходные файлы для каждой локали
    $translate = new Zend_Translate('array', APPLICATION_PATH .
                    '/../languages', null,
                    array('scan' => Zend_Translate:: LOCALE_DIRECTORY));
  }
}
```

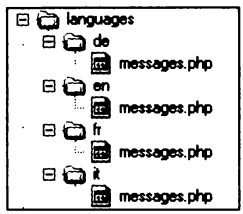


Рис. 9.2. Размещение исходных файлов перевода в отдельных каталогах

Обратите внимание, что в данном случае вторым аргументом конструктора объекта **Zend_Translate** является корень дерева просматриваемых каталогов, а дополнительный четвертый аргумент указывает на то, что информация о локали содержится в их именах.

Альтернативный подход заключается в использовании структуры, основанной на файлах, где идентификатор локали для источника перевода содержится в именах файлов. Пример структуры показан на рис. 9.3, а следующий код демонстрирует использование **Zend_Translate** для ее анализа:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function translateAction()
  {
    // создаем объект перевода
    // указываем адаптер перевода
    // автоматически находим исходные файлы для каждой локали
    $translate = new Zend_Translate('array', APPLICATION_PATH .
                    '/../languages', null,
                    array('scan' => Zend_Translate::LOCALE_FILENAME));
  }
}
```

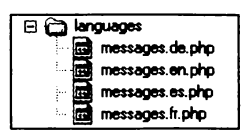


Рис. 9.3. Размещение источников перевода в отдельных файлах

Разумеется, кроме рассмотренных подходов существуют и другие. **Zend_Translate** поддерживает несколько стандартных структур каталогов для размещения исходных файлов перевода, и следует выбирать ту из них, которая лучше всего подходит

под размер и масштабы вашего проекта. По ссылкам в конце главы вы найдете более полную информацию об этих структурах, а также примеры.

Использование локали приложения

Как и многие другие компоненты Zend Framework, **Zend_Translate** учитывает локаль: если доступен зарегистрированный экземпляр **Zend_Locale**, он автоматически будет использоваться по умолчанию для всех действий по переводу. Это удобно, так как позволяет установить локаль только один раз (в начальном загрузчике приложения) и применять ее ко всем операциям, использующим **Zend_Translate**, во всем приложении.

ВОПРОС ЭКСПЕРТУ

В: Что такое GNU gettext и как его использовать?

О: GNU gettext является частью проекта GNU Translation Project и предоставляет стандартный способ создания и использования таблиц переводов для приложений. В этой системе строки перевода для различных языков хранятся в удобочитаемой форме в файлах шаблонов сообщений (*.po), по одному на каждый язык. Затем эти файлы преобразуются в двоичные файлы каталога (*.mo), распространяемые с приложением.

Для файлов перевода *.mo в GNU gettext принято использовать структуру, основанную на каталогах, где файлы перевода для различных языков хранятся в отдельных каталогах, названных в соответствии с локалью или языком (рис. 9.4). Стоит отметить, что механизм обнаружения источников **Zend_Translate** может автоматически распознавать и использовать эту структуру каталогов, тем самым позволяя разработчикам использовать в приложении на PHP существующие источники перевода в формате GNU gettext, не внося дополнительных изменений.

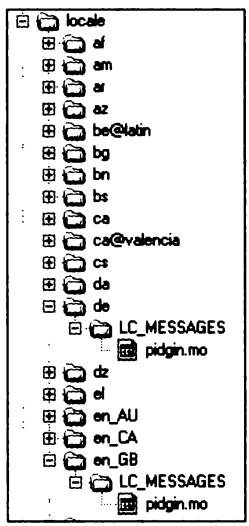


Рис. 9.4. Общепринятая структура каталогов с источниками перевода для GNU gettext

В следующем примере локаль de_DE (язык: немецкий, страна: Германия) регистрируется в реестре, а затем автоматически распознается и используется компонентом Zend_Translate:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function localeAction()
  {
    // определяем и регистрируем локаль
    $locale = new Zend_Locale('de_DE');
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Locale', $locale);

    // создаем объект перевода
    // определяем адаптер перевода
    // автоматически находим исходные файлы для каждой локали
    $translate = new Zend_Translate('array', APPLICATION_PATH .
                  '/../languages', null,
                  array('scan' => Zend_Translate::LOCALE_FILENAME));

    // переводим строку
    // результат: 'eins, zwei, drei'
    $this->view->message = sprintf('%s, %s, %s',
                      $translate->_('one'),
                      $translate->_('two'),
                      $translate->_('three')
                    );
  }
}
```

Следует отметить, что если Zend_Translate не может найти источник перевода для полного идентификатора локали, он автоматически «ухудшит» локаль и будет использовать тот источник перевода, который лучше всего подходит для этой ухудшенной версии. Например, если для приложения установлена локаль fr_LU (язык: французский, страна: Люксембург), но источник перевода для этой локали недоступен, Zend_Translate будет автоматически использовать вместо него источник для локали fr, если он есть.

Использование вспомогательного класса представления для перевода

В примерах из предыдущего раздела было показано выполнение действий по переводу внутри контроллера. Однако на самом деле правильнее будет выполнять эти действия в сценарии представления, а не в контроллере. Zend_Translate содержит вспомогательный класс представления для перевода, который можно использовать в этих целях.

Чтобы использовать вспомогательный класс представления для перевода, требуется сначала зарегистрировать объект **Zend_Translate** в реестре приложения, сделав его доступным из всех частей приложения. Пример:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initTranslate()
  {
    // инициализируем и регистрируем объект перевода
    $translate = new Zend_Translate('array',
                    APPLICATION_PATH . '/../languages/',
                    null,
                    array('scan' => Zend_Translate::LOCALE_FILENAME));
    $registry->set('Zend_Translate', $translate)
  }
}
```

После этого перевод в сценарии представления можно осуществлять, просто вызывая метод **translate()** объекта представления, как показано в следующем коде. При таком способе вызова вспомогательный класс получит соответствующую строку на нужном языке и добавит ее в сценарий перед формированием представления.

```
<div id="header">
  <?php echo $this->translate('welcome'); ?>
</div>
```

После регистрации объекта **Zend_Translate** в реестре приложения другие компоненты Zend Framework тоже смогут использовать его для локализации строк. Отличный пример — **Zend_Form**, который может автоматически использовать зарегистрированный объект **Zend_Translate** для подготовки локализованных версий полей формы, надписей на кнопках, элементов списков выбора, легенд для наборов полей и сообщений об ошибках валидации. Пример приведен в следующем разделе.

Упражнение 9.1. Локализация демонстрационного приложения

Предыдущие разделы показали, что Zend Framework предлагает всеобъемлющий API для локализации приложений. Теперь, когда вы знакомы с теорией, давайте применим ее на практике, проведя локализацию демонстрационного приложения SQUARE для поддержки трех различных языков: английского (как британского (UK), так и американского (US)), французского и немецкого.

Установка локали приложения

Первым шагом на пути к локализации приложения будет определение его локали. Впоследствии эта информация будет использоваться другими компонентами, такими как **Zend_Date** и **Zend_Translate**. Рекомендуется сначала попробовать авто-

матически определить локаль по данным клиентского приложения пользователя, а в случае неудачи вручную присвоить локали разумное значение.

Начнем с редактирования начального загрузчика приложения, **$APP_DIR/application/Bootstrap.php**, и добавим в него метод для осуществления автоматического определения локали:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initLocale()
  {
    try {
      $locale = new Zend_Locale('browser');
    } catch (Zend_Locale_Exception $e) {
      $locale = new Zend_Locale('en_GB');
    }
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Locale', $locale);
  }
}
```

В данном случае **Zend_Locale** попытается автоматически определить текущую локаль пользователя, а при невозможности это сделать установит для нее значение **en_GB** (язык: английский, страна: Великобритания). С помощью **Zend_Registry** эта локаль будет зарегистрирована в реестре приложения, чтобы ее могли использовать другие компоненты.

Локализация чисел и дат

Следующим шагом после определения локали будет просмотр приложения и начало процесса локализации чисел и дат. Взглянув на представления для каталога марок, вы увидите, что каждая запись о марке содержит определенные числовые значения: номинал, а также минимальную и максимальную цену продажи. В идеальном случае эти значения должны быть отформатированы в соответствии с правилами для текущей локали пользователя.

Еще один элемент, пока отсутствующий на странице с информацией о марке, — дата его выставления на продажу. Эта информация полезна для потенциальных покупателей, так как позволяет им судить о «свежести» списка. Лучше всего, если это значение также будет представлено в соответствии с языком и локальными стандартами пользователя.

Компонент **Zend_Locale_Format** предоставляет средства, необходимые для локализации чисел, в виде статического метода **toNumber()**, а компонент **Zend_Date** решает задачу локализации и форматирования дат. Для использования этих средств внесите следующие изменения в действие **Catalog_ItemController::displayAction**, находящееся в файле **$APP_DIR/application/modules/catalog/controllers/ItemController.php**:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
```

```
{
  // действие для отображения элемента каталога
  public function displayAction()
  {
    // устанавливаем фильтры и валидаторы для входных данных,
    // полученных в запросе GET
    $filters = array(
      'id' => array('HtmlEntities', 'StripTags', 'StringTrim')
    );
    $validators = array(
      'id' => array('NotEmpty', 'Int')
    );
    $input = new Zend_Filter_Input($filters, $validators);
    $input->setData($this->getRequest()->getParams());

    // проверяем корректность входных данных
    // если они корректны, получаем запрошенную запись
    // и добавляем ее к представлению
    if ($input->isValid()) {
      $q = Doctrine_Query::create()
            ->from('Square_Model_Item i')
            ->leftJoin('i.Square_Model_Country c')
            ->leftJoin('i.Square_Model_Grade g')
            ->leftJoin('i.Square_Model_Type t')
            ->where('i.RecordID = ?', $input->id)
            ->addWhere('i.DisplayStatus = 1')
            ->addWhere('i.DisplayUntil >= CURDATE()');
    $result = $q->fetchArray();
    if (count($result) == 1) {
      $this->view->item = $result[0];
      $this->view->images = array();
      $config = $this->getInvokeArg('bootstrap')->getOption('uploads');
      foreach (glob("{$config['uploadPath']}/
        {$this->view->item['RecordID']}_*") as $file) {
        $this->view->images[] = basename($file);
      }
        $configs = $this->getInvokeArg('bootstrap')->getOption('configs');
        $localConfig = new Zend_Config_Ini($configs['localConfigPath']);
        $this->view->seller = $localConfig->user->displaySellerInfo;
        $registry = Zend_Registry::getInstance();
        $this->view->locale = $registry->get('Zend_Locale');
        $this->view->recordDate = new Zend_Date($result[0]['RecordDate']);
      } else {
        throw new Zend_Controller_Action_Exception('Page not found', 404);
      }
        } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }
}
```

Здесь метод displayAction() получает из реестра приложения текущую локаль и присваивает ее переменной представления. Он также получает дату сохранения записи и присваивает ее объекту Zend_Date. Теперь можно локализовать эти даты и числа в соответствующем сценарии представления, $APP_DIR/application/modules/catalog/views/scripts/item/display.phtml:

```
<h2>View Item</h2>
<h3>
  FOR SALE:
  <?php echo $this->escape($this->item['Title']); ?> -
  <?php echo $this->escape($this->item['Year']); ?> -
  <?php echo $this->escape($this->item['Square_Model_Grade']['GradeName']); ?>
</h3>

<div id="container">
  <div id="images">
    <?php foreach ($this->images as $image): ?>
    <img src="/uploads/<?php echo $this->escape($image); ?>"
    width="150" height="150" />
    <?php endforeach; ?>
</div>
<div id="record">
  <table>

  <tr>
    <td class="key">Denomination:</td>
    <td class="value">
      <?php echo $this->escape(
        Zend_Locale_Format::toNumber($this->item['Denomination'],
          array(
            'locale' => $this->locale,
            'precision' => 2
        )
      )); ?>
    </td>
    </tr>
    <tr>
      <td class="key">Grade:</td>
      <td class="value">
        <?php echo $this->escape(
          $this->item['Square_Model_Grade']['GradeName']); ?>
      </td>
    </tr>
    <tr>
      <td class=»key»>Sale price:</td>
      <td class="value">
        $<?php echo $this->escape(
          Zend_Locale_Format::toNumber($this->item['SalePriceMin'],
            array(
                'locale' => $this->locale,
```

```
                'precision' => 2
            )
          )); ?> -
        $<?php echo $this->escape(
          Zend_Locale_Format::toNumber($this->item['SalePriceMax'],
            array(
                'locale' => $this->locale,
                'precision' => 2
            )
          )); ?>
        </td>
    </tr>
    <tr>
      <td class="key">Description:</td>
      <td class="value">
        <?php echo $this->escape($this->item['Description']); ?>
      </td>
    </tr>
    <tr>
      <td class="key">Date posted:</td>
      <td class="value">
        <?php echo $this->escape($this->recordDate->get(
        Zend_Date::DATE_FULL)); ?>
      </td>
    </tr>
    <?php if ($this->seller == 1): ?>
     <tr>
       <td class="key">Seller:</td>
       <td class="value">
          <?php echo $this->escape($this->item['SellerName']); ?>,
          <?php echo $this->escape($this->item['SellerAddress']); ?>
        </td>
       </tr>
      <?php endif; ?>
    </table>
  </div>
</div>
```

Определение строк, требующих перевода

Следующим шагом после решения проблем с датами и числами будет локализация строк. Из-за ограниченного объема книги невозможно локализовать каждую строку приложения, поэтому в примере будет рассматриваться локализация строк только в следующих его элементах: главное меню приложения, нижний колонтитул страницы и форма обратной связи.

Для выполнения локализации строк необходимо отредактировать каждый макет и представление приложения, а также воспользоваться вспомогательным классом представления для перевода, чтобы динамически добавить в них строки перевода. Начнем с основного макета приложения, содержащего главное меню.

Внесите в файл $APP_DIR/application/layouts/master.phtml изменения, выделенные жирным шрифтом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

    <div id=»menu»>
      <a href=»<?php echo $this->url(array(), 'home'); ?>">
        <?php echo $this->translate('menu-home'); ?>
      </a>
      <a href="<?php echo $this->url(array('page' => 'services'),
      'static-content'); ?>">
        <?php echo $this->translate('menu-services'); ?>
      </a>
      <a href="<?php echo $this->url(array('module' => 'catalog',
      'controller' => 'item', 'action' => 'search'), 'default', true); ?>">
      <?php echo $this->translate('menu-catalog'); ?>
        </a>
        <a href=»<?php echo $this->url(array(), 'contact'); ?>">
      <?php echo $this->translate('menu-contact'); ?>
    </a>
  </div>
</div>

<div id="content">
    <?php echo $this->layout()->content ?>
  </div>

  <div id="footer">
    <p>
      <?php echo $this->translate('created-with'); ?>
      <a href="http://framework.zend.com/">Zend Framework</a>.
      <?php echo $this->translate('licensed-under'); ?>
      <a href="http://www.creativecommons.org/">Creative Commons</a>.
    </p>
  </div>
</body>
</html>
```

Теперь перейдем к форме обратной связи. Форма определена с помощью компонента Zend_Form, который может автоматически обнаруживать наличие объекта Zend_Translate и использовать его для локализации строк во время выполнения программы. Все, что на самом деле нужно сделать, — заменить метку для каждого элемента идентификатором строки, который во время выполнения будет автоматически заменен переведенным значением. Внесите изменения, выделенные жирным шрифтом, в определение формы, которое находится в файле $APP_DIR/library/Square/Form/Contact.php:

```
<?php
class Square_Form_Contact extends Zend_Form
```

```
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/contact/index')
         ->setMethod('post');

    // создаем текстовое поле для ввода имени
    $name = new Zend_Form_Element_Text('name');
    $name->setLabel('contact-name')
        ->setOptions(array('size' => '35'))
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
         ->addValidator('Alpha', true)
         ->addFilter('HTMLEntities')
         ->addFilter('StringTrim');

    // создаем текстовое поле для ввода адреса электронной почты
    $email = new Zend_Form_Element_Text('email');
    $email->setLabel('contact-email-address');
    $email->setOptions(array('size' => '50'))
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addValidator('EmailAddress', true)
        ->addFilter('HTMLEntities')
        ->addFilter('StringToLower')
        ->addFilter('StringTrim');

    // создаем текстовое поле для сообщения
    $message = new Zend_Form_Element_Textarea('message');
    $message->setLabel('contact-message')
        ->setOptions(array('rows' => '8','cols' => '40'))
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addFilter('HTMLEntities')
        ->addFilter('StringTrim');

    // создаем CAPTCHA
      $captcha = new Zend_Form_Element_Captcha('captcha', array(
        'captcha' => array(
        'captcha' => 'Image',
        'wordLen' => 6,
        'timeout' => 300,
        'width' => 300,
        'height' => 100,
        'imgUrl' => '/captcha',
        'imgDir' => APPLICATION_PATH . '/../public/captcha',
        'font' => APPLICATION_PATH .
                '/../public/fonts/LiberationSansRegular.ttf',
      )
```

```
    ));
    $captcha->setLabel('contact-verification');

    // создаем кнопку отправки
    $submit = new Zend_Form_Element_Submit('submit');
    $submit->setLabel('contact-send-message')
        ->setOptions(array('class' => 'submit'));

    // добавляем элементы к форме
    $this->addElement($name)
        ->addElement($email)
        ->addElement($message)
        ->addElement($captcha)
        ->addElement($submit);
  }
}
```

Создание источников перевода

Следующим шагом после подготовки различных элементов представления к локализации будет создание исходных файлов перевода для каждого языка. Для простоты будем использовать массивы PHP с указанием локали в именах файлов.

Начнем с создания каталога для источников перевода. Перейдите в каталог **$APP_DIR** и выполните следующую команду:

```
shell> mkdir languages
```

Теперь необходимо создать исходные файлы перевода для каждого поддерживаемого приложением языка. Ниже приведен пример такого файла для французского языка, его следует сохранить под именем **$APP_DIR/languages/messages.fr.php**:

```
<?php
return array(
    'menu-home' => 'ACCUEIL',
    'menu-services' => 'SERVICES',
    'menu-catalog' => 'BROCHURE',
    'menu-contact' => 'CONTACTEZ-NOUS',
    'welcome' => 'Bienvenue',
    'created-with' => 'Créé avec',
    'licensed-under' => 'Sous license',
    'contact-name' => 'Nom:',
    'contact-email-address' => 'Adresse email:',
    'contact-message' => 'Message:',
    'contact-verification' => 'Vérification:',
    'contact-send-message' => 'Envoyer Message',
    'contact-title' => 'Contactez-Nous',
);
```

После того как вы закончите, структура каталогов должна иметь вид, представленный на рис. 9.5.

Рис. 9.5. Источники перевода для приложения, расположенные в отдельных файлах

СОВЕТ

Чтобы избежать повреждения данных, при создании файлов, содержащих отличные от латинских символы, используйте кодировку UTF-8. Поддержка UTF-8 есть в некоторых бесплатных и коммерческих текстовых редакторах, таких как gedit в UNIX и Notepad2 в Microsoft Windows. Ссылки для их загрузки можно найти в конце этой главы.

ВОПРОС ЭКСПЕРТУ

В: Почему кроме файлов для локалей en_US и en_GB присутствует еще и отдельный файл перевода для локали en?

О: Как было объяснено ранее в этой главе, если Zend_Translate не сможет найти исходный файл перевода для идентификатора локали, он автоматически ухудшит идентификатор локали, оставив только код языка, и будет использовать соответствующий этому языку исходный файл перевода, если таковой имеется. Поэтому, помимо файлов, в точности соответствующих локали, рекомендуется всегда добавлять обобщенный файл перевода для каждого языка, который вы планируете поддерживать в приложении. Именно по этой причине в дополнение к языковым файлам en_US и en_GB в каталоге находится файл en.

Регистрация объекта перевода

Последний этап — обновление начального загрузчика приложения, а также настройка и регистрация экземпляра объекта Zend_Translate. Этот экземпляр будет доступен во всех контроллерах и представлениях приложения, позволяя использовать вспомогательный класс представления для перевода и возможности автоматического перевода, поддерживаемые Zend_Form.

Чтобы выполнить указанные действия, добавьте в файл начальной загрузки, $APP_DIR/application/Bootstrap.php, следующий метод:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initTranslate()
  {
    $translate = new Zend_Translate('array',
        APPLICATION_PATH . '/../languages/',
        null,
       array('scan' => Zend_Translate::LOCALE_FILENAME,
              'disableNotices' => 1));
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Translate', $translate);
  }
}
```

СОВЕТ

Вместо того чтобы получать экземпляр Zend_Registry, вы можете непосредственно обращаться к значениям реестра с помощью методов-сокращений Zend_Registry::get($index) и Zend_Registry::set($index, $value).

Все готово! Чтобы проверить, как это работает, измените настройки вашего браузера и добавьте в список предпочтительных языков французский или немецкий. На рис. 9.6 показано, как сделать это в Mozilla Firefox, а на рис. 9.7 — в Microsoft Internet Explorer.

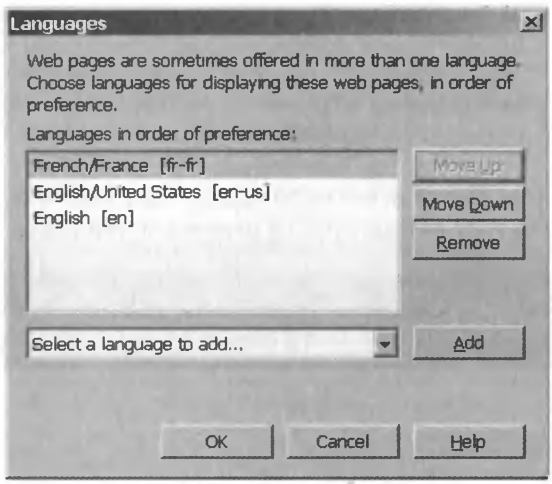


Рис. 9.6. Настройка языков страниц в Mozilla Firefox

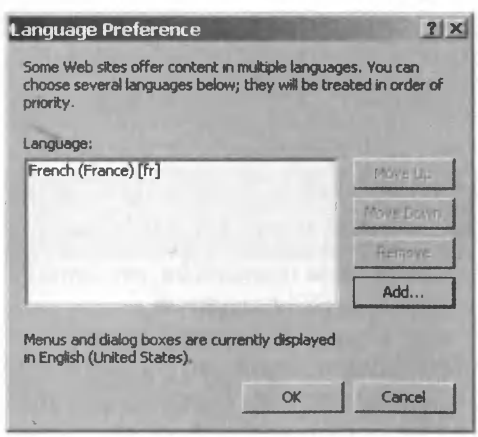


Рис. 9.7. Настройка языков страниц в Microsoft Internet Explorer

Теперь перейдите на главную страницу приложения по адресу http://square.localhost, и вы увидите страницу, похожую на рис. 9.8. Обратите внимание, что главное меню и нижний колонтитул представлены на выбранном языке.

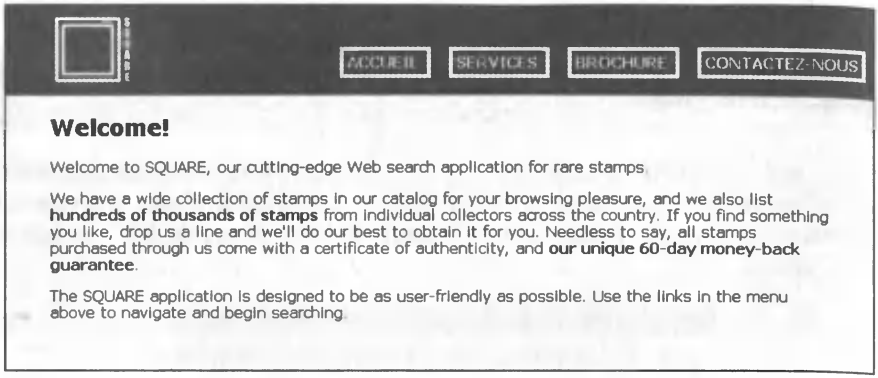


Рис. 9.8. Главная страница приложения, автоматически переведенная на французский

Перейдите к форме обратной связи по адресу http://square.localhost/contact, и вы обнаружите, что Zend_Form автоматически перевел метки полей (рис. 9.9).

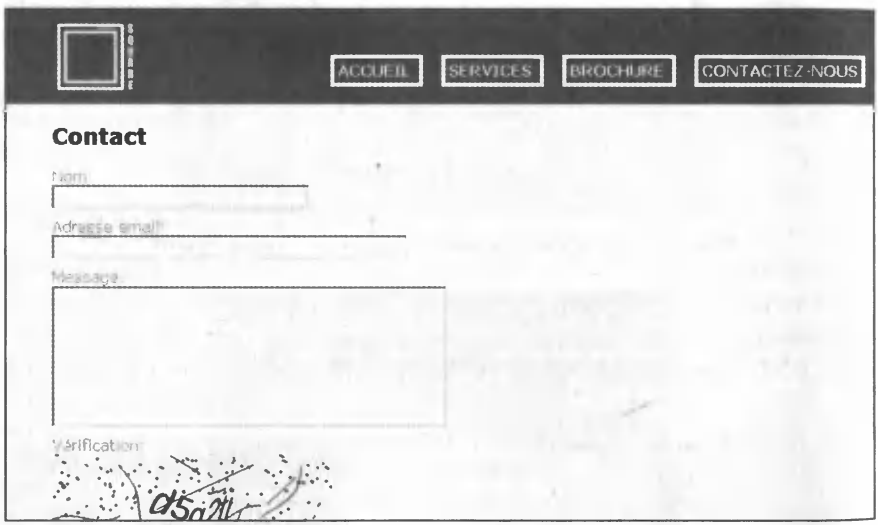


Рис. 9.9. Форма обратной связи приложения, автоматически переведенная на французский

Если вы измените предпочтительный язык в настройках браузера на другой и Zend_Translate сможет найти источник перевода для этого языка, содержимое будет автоматически переведено на выбранный язык (рис. 9.10). Если же Zend_Translate не сможет найти источник перевода, перевод будет невозможен и вместо него будут показаны исходные строки (рис. 9.11). И наконец, если в браузере не выбран какой-то конкретный язык, автоматическое определение локали завершится неудачей, в результате чего компоненты Zend_Locale и Zend_Translate будут использовать запасную локаль en_GB (рис. 9.12).

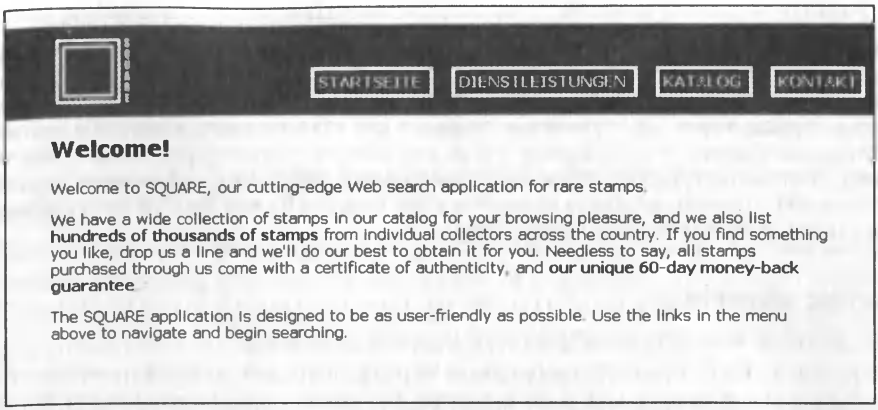


Рис. 9.10. Главная страница приложения, автоматически переведенная на немецкий

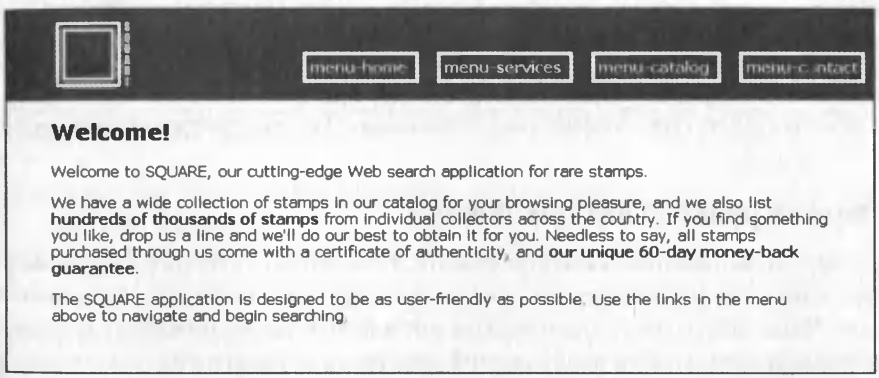


Рис. 9.11. Главная страница приложения в случае недоступности перевода

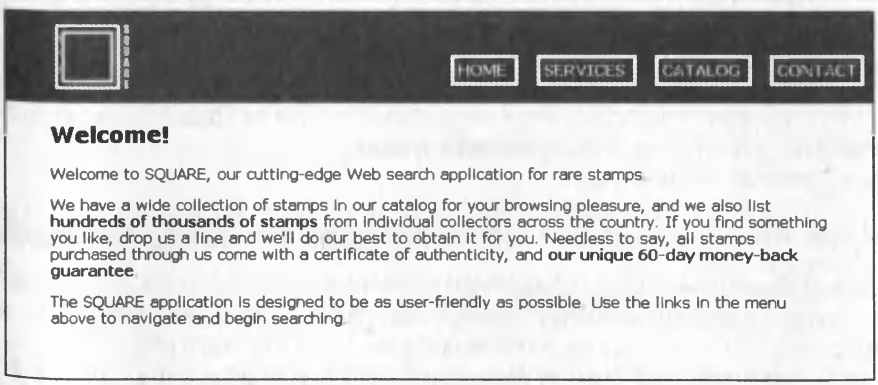


Рис. 9.12. В случае, когда автоматическое определение локали невозможно, язык UK English устанавливается для индексной страницы приложения вручную

ПРИМЕЧАНИЕ

Вы, вероятно, заметили, что, несмотря на автоматическое определение локали, статическое содержимое страницы все еще представлено на английском языке. Данный аспект локализации невозможно уместить в ограниченное пространство этой главы, но вы легко можете решить задачу, создав отдельные статические страницы для каждого языка и настроив контроллер статических страниц на отображение той из них, которая соответствует текущей локали или языку. Этот метод предпочтительнее создания ключей перевода с содержимым для каждой статической страницы, поскольку поддержка столь больших блоков локализованного содержимого может быстро превратиться в проблему.

ВОПРОС ЭКСПЕРТУ

В: Требуется ли менять регистр ключей перевода на нижний?

О: Не существует правила, требующего использовать для ключей перевода только нижний регистр букв. Такой подход означал бы, что вам придется помещать все вызовы translate() в представлениях внутрь вызовов strtoupper() и/или ucfirst(), что довольно утомительно. Кроме того, разные языки могут требовать различных способов использования прописных букв в одном и том же слове, а применение преобразований регистра в представлениях сделает это невозможным. И наконец, если Zend_Translate не может найти источник перевода для выбранного языка, он отображает сами ключи. Следовательно, ограничение значений ключей нижним регистром нежелательно. Тем не менее вы должны убедиться, что ключи перевода имеют корректные названия, поскольку они используются Zend_Translate в качестве запасного варианта.

Поддержка ручного выбора локали

Теперь, когда с автоматическим определением локали все понятно, у вас может возникнуть идея дать пользователям возможность вручную выбирать желаемые язык и локаль. Чаще всего в пользовательском интерфейсе эта возможность реализуется в виде последовательности изображений флагов; при нажатии на одно из них приложение переходит на соответствующий язык.

Чтобы добавить такую возможность, определите в модуле default приложения новый контроллер LocalController, который будет отвечать за ручную смену локали на выбранную пользователем, и добавьте в него следующий код:

```
<?php
class LocaleController extends Zend_Controller_Action
{
  // действие для ручного переопределения локали
  public function indexAction()
  {
    // если локаль поддерживается, сохраняем ее в сессии
    if (Zend_Validate::is(
      $this->getRequest()->getParam('locale'), 'InArray',
        array('haystack' => array('en_US', 'en_GB', 'de_DE', 'fr_FR'))
    )) {
      $session = new Zend_Session_Namespace('square.l10n');
      $session->locale = $this->getRequest()->getParam('locale');
    }
```

```
    // перенаправляем на запрошенный URL
    $url = $this->getRequest()->getServer('HTTP_REFERER');
    $this->_redirect($url);
  }
}
```

Сохраните код в файл **$APP_DIR/application/modules/default/controllers/LocaleController.php**.

При вызове действия **LocaleController::indexAction** оно вначале получает URL исходного сценария, ссылающегося на него, и сохраняет этот URL в локальной переменной. Затем оно ищет в запросе URL параметр **$_GET['locale']**, проверяет, поддерживается ли запрашиваемая локаль, и если да, сохраняет ее в сессии. После этого клиент перенаправляется на исходный сценарий.

Однако это лишь часть задачи. Необходимо также обновить начальный загрузчик приложения и изменить метод **_initLocale()** таким образом, чтобы он сначала проверял наличие в сессии идентификатора локали, а затем, если он есть, давал ему более высокий приоритет относительно локали, определенной автоматически. Измененный метод **_initLocale()**:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initLocale()
  {
    $session = new Zend_Session_Namespace('square.l10n');
    if ($session->locale) {
      $locale = new Zend_Locale($session->locale);
    }

    if ($locale === null) {
      try {
        $locale = new Zend_Locale('browser');
      } catch (Zend_Locale_Exception $e) {
        $locale = new Zend_Locale('en_GB');
      }
    }
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Locale', $locale);
  }
}
```

Обновление основного макета

Последним шагом будет добавление в главное меню приложения последовательности флагов и назначение им ссылок на **LocaleController::indexAction**. Чтобы сделать это, внесите в файл основного макета, **$APP_DIR/application/layouts/master.phtml**, изменения, выделенные жирным шрифтом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

      <div id="menu-locale-container">
        <div id="locale">
          <a href="<?php echo $this->url(array('module' => 'default',
            'controller' => 'locale', 'action' => 'index',
            'locale' => 'en_GB'), 'default', true); ?>">
            <img src="/images/locale/en_gb.gif" />
          </a>
          <a href="<?php echo $this->url(array('module' => 'default',
            'controller' => 'locale', 'action' => 'index',
            'locale' => 'en_US'), 'default', true); ?>">
            <img src="/images/locale/en_us.gif" />
          </a>
          <a href="<?php echo $this->url(array('module' => 'default',
            'controller' => 'locale', 'action' => 'index',
            'locale' => 'fr_FR'), 'default', true); ?>">
            <img src="/images/locale/fr_fr.gif" />
          </a>
          <a href="<?php echo $this->url(array('module' => 'default',
            'controller' => 'locale', 'action' => 'index',
            'locale' => 'de_DE'), 'default', true); ?>">
            <img src="/images/locale/de_de.gif" />
          </a>
          </div>
...
</html>
```

Вы, наверное, заметили, что основной макет использует различные изображения из каталога **$APP_DIR/public/images/locale/**. Эти изображения вместе с соответствующими правилами CSS вы найдете в архиве с дополнительными материалами к этой главе.

Рис. 9.13. Главная страница приложения с кнопками для ручного выбора языка

Если вы сейчас перейдете на главную страницу приложения, **http://square.localhost/**, то увидите, что на каждой странице появилась возможность вручную вы-

бирать язык из строки меню. Выбор языка имеет больший приоритет, чем локаль, определенная по настройкам браузера, и сохраняется в течение всего времени вашего нахождения на сайте. Пример страницы показан на рис. 9.13.

Выводы

Как показано в этой главе, Zend Framework содержит все необходимое, чтобы полностью локализовать ваше приложение для различных языков и стран. Компонент Zend_Locale предоставляет основу для определения локали приложения как вручную, во время разработки, так и автоматически, во время выполнения. Вне зависимости от способа определения другие компоненты Zend Framework, такие как Zend_Date, Zend_Currency и Zend_Measure, могут автоматически обнаруживать эту локаль и изменять свое поведение в соответствии с местными нормами для дат, валюты и единиц измерения.

Самым сложным аспектом любого проекта по локализации является перевод строк в приложении на разные языки, и для этой задачи Zend Framework тоже предлагает решение в виде компонента Zend_Translate. Он предоставляет гибкий API для работы с источниками перевода, представленными множеством различных форматов, и может автоматически интегрироваться с Zend_View через вспомогательный класс представления для перевода, чтобы облегчить задачу локализации строк во время выполнения.

Все эти аспекты локализации были показаны на примере демонстрационного приложения SQUARE, которое к концу главы поддерживает четыре различные локали и может представлять даты, номера и (некоторое) содержимое на текущем или выбранном пользователе языке. Хотя в рамках этой книги невозможно осуществить полную локализацию, приведенных примеров должно быть достаточно, чтобы дать вам определенное понимание процесса и сформировать основу для дальнейшего изучения.

Чтобы получить больше информации по изложенным в этой главе темам, посетите следующие ссылки:

- Компонент Zend_Locale: http://framework.zend.com/manual/en/zend.locale.html.
- Компонент Zend_Date: http://framework.zend.com/manual/en/zend.date.html.
- Спецификаторы формата Zend_Date: http://framework.zend.com/manual/en/zend.date.constants.html.
- Компонент Zend_Currency: http://framework.zend.com/manual/en/zend.currency.html
- Преобразование валюты с помощью Zend_Currency: http://framework.zend.com/manual/en/zend.currency.exchange.html.
- Компонент Zend_Measure: http://framework.zend.com/manual/en/zend.measure.html.
- Компонент Zend_Translate: http://framework.zend.com/manual/en/zend.translate.html.
- Структуры каталогов Zend_Translate для исходных файлов перевода: http://framework.zend.com/manual/en/zend.translate.using.html.

- Проект GNU gettext: http://www.gnu.org/software/gettext/.
- Стандарт ISO 639 (коды языков): http://www.iso.org/iso/support/faqs/faqs_widely_used_standards/widely_used_standards_other/language_codes.htm.
- Стандарт ISO 3166 (коды государств): http://www.iso.org/iso/support/faqs/faqs_widely_used_standards/widely_used_standards_other/country_name_codes.htm.
- Стандарт ISO 4217 (коды валют): http://www.iso.org/iso/support/faqs/faqs_widely_used_standards/widely_used_standards_other/currency_codes.htm.
- Пакет Services_ExchangeRates в репозитории PEAR: http://pear.php.net/package/Services_ExchangeRates/.
- Текстовый редактор gedit: http://www.gnome.org/projects/gedit/.
- Текстовый редактор Notepad2: http://www.flos-freeware.ch/notepad2.html.
- Создание многоязычных сайтов с использованием Zend Framework (Джейсон Гилмор (Jason Gilmore)): http://www.developer.com/design/article.php/3683571/Build-Multi-lingual-Websites-With-the-Zend-Framework.htm.
- Локализация строк с помощью gettext и Zend Framework (Питер Торнстранд (Peter Törnstrand)): http://www.tornstrand.com/2008/03/29/string-localization-with-gettext-and-zendframework/.

Работа с новостными лентами и веб-сервисами

10

Прочитав эту главу, вы:

- научитесь генерировать и читать новостные ленты в форматах Atom и RSS;
- узнаете о различных типах архитектур веб-приложений;
- интегрируете результаты поиска и данные, полученные от Google, Amazon и Twitter;
- получите доступ к сторонним веб-сервисам, используя SOAP и REST;
- реализуете простой REST API.

Если вы когда-либо пробовали заставить два (или более) совершенно разных веб-приложения согласованно работать вместе, вам, вероятно, уже известно, насколько сложным и полным разочарований может быть этот процесс. В многоплатформенном многоязычном мире Всемирной паутины обмен данными между приложениями, написанными на различных языках программирования, — зачастую самый большой камень преткновения на пути к настоящей совместимости приложений.

За последние годы появилось несколько технологий и протоколов, пытающихся решить эту проблему. Форматы Web Distributed Data eXchange (WDDX, обмен данными, распределенными во Всемирной паутине), Simple Object Access Protocol (SOAP, простой протокол доступа к объектам), Really Simple Syndication (RSS, очень простой сбор сводной информации), Atom Syndication Format (ASF, формат сбора сводной информации Atom) и другие предоставляют независимую от языка и платформы основу для обмена данными между веб-приложениями. Как правило, этот обмен осуществляется по протоколу HTTP, а запросы и ответы передаются с использованием разновидности XML. На сегодняшний день эти технологии повсеместно используются для сбора данных новостных лент и реализации веб-сервисов.

Zend Framework содержит несколько компонентов, которые могут оказаться полезными при работе с веб-сервисами и новостными лентами. Компоненты Zend_Feed и Zend_Feed_Reader упрощают создание, изменение, анализ и обработку новостных лент в форматах RSS и Atom. Компонент Zend_Rest_Client позволяет получить доступ к существующим веб-сервисам, основанным на архитектуре REST, а Zend_Rest_Controller упрощает задачу реализации новых веб-сервисов, основанных на REST. И наконец, обширный набор реализаций Zend_Service упрощает интеграцию с популярными веб-приложениями, такими как Twitter, Technorati, Google и Flickr. В этой главе подробно рассматриваются все перечисленные компоненты.

Работа с новостными лентами

Вы, вероятно, уже знаете, что такое новостная лента. Это способ распространения информации о том, что нового и интересного появилось на конкретном сайте в данное время. Эта информация, обычно представляющая собой список заголовков и фрагментов новостей, расположенных в хронологическом порядке, публикуется в виде машиночитаемого XML, который может быть проанализирован сторонними веб-сайтами и приложениями для чтения новостных лент.

В состав Zend Framework входят два компонента, помогающие в работе с такими новостными лентами. Компонент Zend_Feed_Reader предоставляет API для чтения и анализа новостных ленты в двух наиболее распространенных форматах, а Zend_Feed делает возможным генерацию новых лент или модификацию существующих, используя встроенные структуры данных PHP. В следующих разделах эти компоненты рассмотрены более подробно.

Форматы новостных лент

Новостные ленты обычно публикуются либо в формате Really Simple Syndication (RSS), либо в формате Atom Syndication Format (ASF, чаще называется «Atom»). Оба формата основаны на XML и содержат размеченный список ресурсов, каждый из которых снабжен описательной информацией, такой как заголовок, описание, URL и дата. Оба формата также могут ссылаться на разнообразные типы ресурсов, включая документы (обычно записи в блогах или новостные статьи), изображения, а также аудио- и видеопотоки.

По аналогии с лентами RSS, ленты Atom содержат краткую информацию об описываемых ими ресурсах; однако они немного сложнее лент RSS, а формат сбора данных Atom был принят специальной комиссией интернет-разработок (IETF, Internet Engineering Task Force) в качестве стандарта, чего пока нельзя сказать об RSS. Различия между форматами не случайны: формат сбора данных Atom был придуман в 2003 году в качестве альтернативы формату RSS, первоначально разработанному Netscape Communications в начале 1997-го. С момента своего по-

явления RSS прошел через множество этапов развития и альтернативных версий, а формат Atom задумывался как свежий подход, преимуществом которого было отсутствие необходимости поддерживать обратную совместимость с существующими лентами.

На рис. 10.1 показаны сходства (и различия) между форматами лент RSS и Atom.

RSS Example

```
<rss version="2.0">
 <channel>
  <title>Recent Posts</title>
  <link>http://www.melonfire.com</link>
  <language>en</language>
  <pubDate>Tue, 26 Jan 2010 10:47:47 GMT+00:00</pubDate>
  <item>
   <title>Building A PHP-Based Mail Client (part 1)</title>
   <link>http://www.melonfire.com/community/columns/trog/article.php?id=100</link>
   <category>PHP</category>
   <pubDate>Tue, 26 Jan 2010 09:17:54 GMT+00:00</pubDate>
   <description>Ever wondered how Web-based mail clients work? Find out here.</description>
  </item>
  <item>
   <title>Access Granted</title>
   <link>http://www.melonfire.com/community/columns/trog/article.php?id=62</link>
   <category>MySQL</category>
   <pubDate>Mon, 25 Jan 2010 16:17:54 GMT+00:00</pubDate>
   <description>Precisely control access to information with the MySQL grant tables.</description>
  </item>
 </channel>
</rss>
```

Atom Syndication Format Example

```
<feed xmlns="http://www.w3.org/2005/Atom">
 <id>1264503315014</id>
 <title>Recent Posts</title>
 <link rel="self" type="application/atom+xml" href="http://www.melonfire.com/feed.php?format=atom"></link>
 <author>
  <name>Melonfire</name>
  <email>example@example.com</email>
 </author>
 <updated>2010-01-26T10:55:15Z</updated>
 <entry>
  <id>tag:melonfire.com,2010-01-26:/community/columns/trog/article.php#100</id>
  <title>Building A PHP-Based Mail Client (part 1)</title>
  <category>PHP</category>
  <updated>2010-01-26T09:17:54Z</updated>
  <link rel="alternate" href="http://www.melonfire.com/community/columns/trog/article.php?id=100" />
  <content>Ever wondered how Web-based mail clients work? Find out here.</content>
 </entry>
 <entry>
  <id>tag:melonfire.com,2010-01-25:/community/columns/trog/article.php#62</id>
  <title>Access Granted</title>
  <category>PHP</category>
  <updated>2010-01-25T16:17:54Z</updated>
  <link rel="alternate" href="http://www.melonfire.com/community/columns/trog/article.php?id=62" />
  <content>Precisely control access to information with the MySQL grant tables.</content>
 </entry>
</feed>
```

Рис. 10.1. Пример лент в форматах RSS и Atom

Получение новостных лент

Самым простым способом интеграции новостных лент в приложение Zend Framework является компонент **Zend_Feed_Reader**, предоставляющий простую и расширяемую основу для анализа данных лент и содержащихся в них записей. Компонент может работать с лентами Atom и RSS и поддерживает все существующие версии обоих форматов. Это сильно экономит время, учитывая значительное количество используемых на сегодняшний день версий RSS. Лучше всего подход к проектированию этого компонента отражен в руководстве к Zend Framework: «если вы можете извлекать из ленты нужную вам информацию, вас не должно беспокоить, представлена эта лента в формате RSS или Atom».

Рассмотрим следующий пример, иллюстрирующий использование **Zend_Feed_Reader**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function feedAction()
  {
    // импортируем ленту, расположенную по указанному URL
    $feed = Zend_Feed_Reader::import(
        'http://news.google.com/news?hl=en&topic=w&output=rss');

    // получаем элементы со сводной информацией о ленте
    $this->view->feed = array(
        'title' => $feed->getTitle(),
        'description' => $feed->getDescription(),
        'url' =>$ feed->getLink(),
        'generator' => $feed->getGenerator(),
        'numEntries' => $feed->count()
    );

    // получаем первую запись ленты
    $feed->rewind();
    $entry = $feed->current();
    $this->view->entries = array(
      array(
        'title' => $entry->getTitle(),
        'description' => $entry->getDescription(),
        'body' => $entry->getContent(),
        'url' => $entry->getLink(),
        'date' => $entry->getDateModified()
      )
    );
  }
}
```

Как показано в примере, для получения новостной ленты с помощью **Zend_Feed_Reader** необходимо сначала импортировать ее, воспользовавшись методом **import()**. Этот метод принимает URL ленты и, в зависимости от ее типа, возвращает *либо*

объект Zend_Feed_Reader_Feed_Rss, либо объект Zend_Feed_Reader_Feed_Atom. Этот объект является основной точкой входа для ленты и ее записей и предоставляет различные методы, обеспечивающие доступ к сводной информации о ленте, такие как getTitle(), getLink() и getDescription().

Записи в ленте могут быть получены путем итерации по основному объекту ленты. Каждый отдельный объект записи содержит практически те же самые методы, что и основной объект ленты. В частности, внимания заслуживают методы getTitle(), getDescription(), getLink(), getDateModified(), getDateCreated() и getContent(), которые возвращают заголовок записи, описание, URL, дату последнего изменения, дату создания и содержимое записи соответственно. В большинстве случаев этих методов будет достаточно.

Также можно получить доступ к лежащему в основе записи XML-представлению в виде объекта DOMDocument, DOMXPath или DOMElement. Такое представление может быть полезно, когда вам требуется работать непосредственно с XML-описанием ленты, и его можно получить методами getDomDocument(), getElement() и getXpath() объекта ленты. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function feedAction()
  {
    // импортируем ленту, расположенную по указанному URL
    $feed = Zend_Feed_Reader::import(
        'http://news.google.com/news?hl=en&topic=w&output=rss');

    // получаем лежащий в основе объект XPath
    $xpath = $feed->getXpath();

    // запрашиваем список заголовков всех элементов
    $titles = $xpath->query('//item/title');

    // возвращаем третий по счету элемент списка
    $this->view->title = $titles->item(2)->nodeValue;
  }
}
```

ПРИМЕЧАНИЕ

С помощью методов Zend_Feed_Reader::importFile() и Zend_Feed_Reader::importString() можно импортировать новостную ленту из файла или строки соответственно.

Помимо Zend_Feed_Reader в состав Zend Framework входит компонент Zend_Feed, предоставляющий всеобъемлющий API для генерации и изменения новостных лент. Хотя Zend_Feed можно использовать и для их анализа, Zend_Feed_Reader больше подходит для этих целей, так как имеет более простой API и может автоматически определять тип ленты. Тем не менее для полноты картины ниже приведен пример получения ленты RSS с помощью Zend_Feed:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function feedAction()
  {
    // импортируем ленту, расположенную по указанному URL
    $feed = Zend_Feed::import(
        'http://news.google.com/news?hl=en&topic=w&output=rss');

    // получаем элементы со сводной информацией о ленте
    $this->view->feed = array(
        'title' => $feed->title(),
        'description' => $feed->description(),
        'url' => $feed->link(),
        'generator' => $feed->generator(),
        'numEntries' => $feed->count()
    );

    // получаем первую запись ленты
    $feed->rewind();
    $entry = $feed->current();
    $this->view->entries = array(
      array(
        'title' => $entry->title(),
        'description' => $entry->description(),
        'url' => $entry->link(),
        'date' =>$entry->pubDate()
      )
    );
  }
}
```

Обратите внимание, что при использовании **Zend_Feed** содержимое отдельных элементов уровня ленты и уровня записи можно получить, используя синтаксис **$object->method ($объект->метод)**, где имя метода соответствует имени элемента. Например, **$feed->title()** получает содержимое элемента **<title>**, а **$entry->description()** — содержимое элемента **<description>**.

Создание новостных лент

Итак, с чтением существующих лент все понятно. А как насчет создания новых?

Создание новостной ленты — тривиальная задача, которая легче всего решается с помощью **Zend_Feed**. Для этого передайте методу **Zend_Feed::importArray** представление ленты в виде массива и требуемый выходной формат (RSS или Atom), и метод сформирует правильную ленту в указанном формате, пригодную для использования с любой программой чтения новостных лент, соответствующей стандартам. Ниже приведен пример, иллюстрирующий создание ленты Atom на основе результатов выборки из базы данных:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function feedAction()
  {
    // запрос для получения 10 последних статей
    $q = Doctrine_Query::create()
        ->from('App_Model_Posts p')
        ->where('p.status = 1')
        ->orderBy('p.date desc')
        ->limit(10);
    $result = $q->fetchArray();

    // генерируем выходной массив
    // устанавливаем значения элементов со сводной информацией о ленте
    $output = array(
         'title' => 'Newest posts on melonfire.com/trog',
         'link' => 'http://www.melonfire.com/community/columns/trog/',
         'author' => 'Melonfire Feed Generator/1.0',
         'charset' => 'UTF-8',
         'entries' => array()
    );

    // проходим по списку результатов выборки
    // и добавляем значения элементов с информацией о записях
    // в виде вложенных массивов
    foreach ($result as $r) {
      $entry = array(
        'title' => $r['title'],
        'link' => 'http://www.melonfire.com/community/columns/trog/'.$r['id'],
        'description' => $r['abstract'],
        'lastUpdate' => strtotime($r['date']),
      );
      $output['entries'][] = $entry;
    }

    // импортируем массив в объект Zend_Feed и преобразуем его в ленту Atom
    $feed = Zend_Feed::importArray($output, 'atom');

    // отключаем макет и формирование представления
    $this->_helper->layout->disableLayout();
    $this->getHelper('viewRenderer')->setNoRender(true);

    // отправляем ленту клиенту
    $feed->send();
    exit();
  }
}
```

СОВЕТ

Метод send() объекта Zend_Feed установит собственные заголовки; если они вам не нужны или вы хотите использовать свои, можно установить их вручную с помощью метода header(), а затем вызвать метод saveXml() для отправки XML-ленты клиенту.

В приведенном листинге создается вложенный массив, содержащий ленту и содержимое записей, который затем передается методу Zend_Feed::importArray для преобразования в ленту Atom. Метод возвращает объект Zend_Feed_Atom, предоставляющий два полезных метода: send() и saveXml(). send() отправляет ленту клиенту вместе с корректными заголовками HTTP, а saveXml() возвращает XML-представление ленты для дальнейшей обработки.

Пример ленты Atom, созданной предыдущим примером, показан на рис. 10.2.

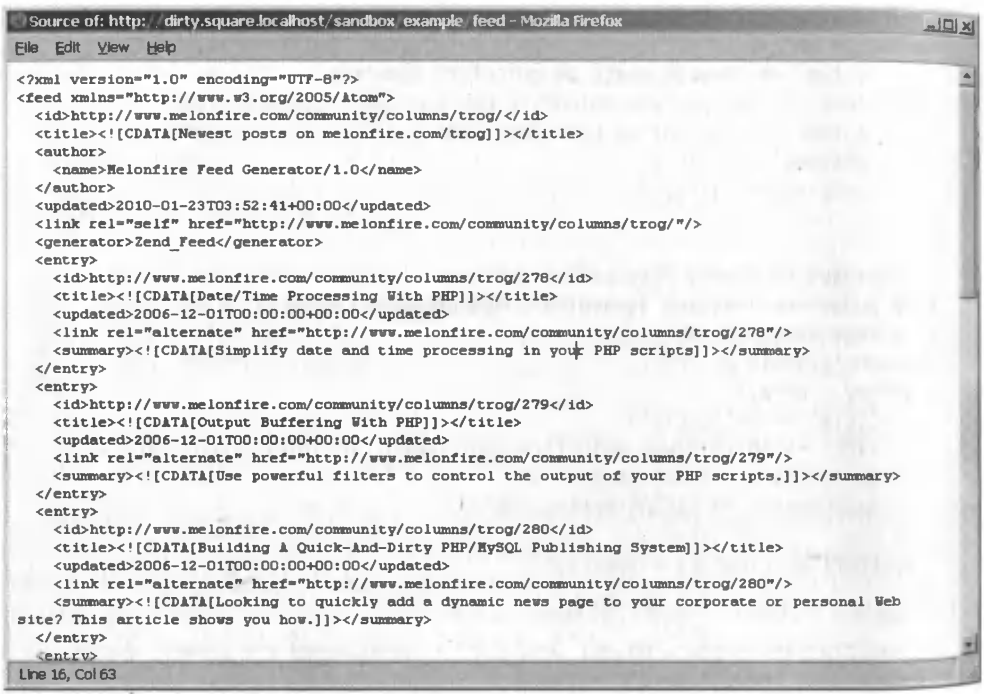


Рис. 10.2. Динамически сгенерированная лента Atom

ПРИМЕЧАНИЕ

Новые версии Zend Framework также содержат компонент Zend_Feed_Writer, предоставляющий альтернативный API для создания и модификации лент.

Доступ к веб-сервисам

За последние годы для потребительских веб-приложений стало практически обязательным предоставление некоторых или всех своих функций и/или данных посредством веб-сервисов для использования третьими лицами. Google, Facebook, Twitter, Technorati и Flickr — лишь некоторые из многих сотен и тысяч веб-приложений, которые обеспечивают подобный доступ к своим внутренним данным. Вы, как разработчик приложений, скорее всего, будете либо интегрировать эти веб-сервисы в свои приложения, либо создавать собственные веб-сервисы для использования другими.

Zend Framework включает в себя реализацию множества распространенных веб-сервисов (включая упомянутые в предыдущем параграфе, а также несколько других). Кроме того, он содержит универсальные клиенты SOAP и REST для доступа к веб-сервисам, основанным на этих архитектурах. Данные компоненты подробно рассматриваются в следующих разделах.

Веб-сервисы

Веб-сервис можно считать «удаленным API», позволяющим клиенту запрашивать выполнение какого-либо действия или запроса на удаленном сервере. Тип запроса может быть разным; как правило, это поиск и возврат записи базы данных, соответствующей определенному идентификатору, проверка номера кредитной карты или получение широты и долготы, соответствующих указанному почтовому индексу. Сервер получает запрос, выполняет необходимый запрос или действие и возвращает результат клиенту в структурированном формате.

Веб-сервисы могут быть основаны либо на архитектуре Simple Object Access Protocol (SOAP, простой протокол доступа к объектам), либо на архитектуре Representational State Transfer (REST, передача репрезентативного состояния). В обоих случаях клиент передает сервисные запросы и получает сервисные ответы по протоколу HTTP. Однако ключевое различие заключается в способах кодирования и передачи этих запросов. В сервисах, основанных на SOAP, сообщения-запросы обычно кодируются в XML-конверты и передаются серверу с помощью метода HTTP POST, а ответы возвращаются в похожем XML-конверте. В сервисах, основанных на REST, запросы и ответы не требуют кодирования в XML, а в определении действия и ответа сервера важную роль играет метод HTTP, используемый для передачи запроса от клиента серверу (это может быть GET, POST, PUT или DELETE).

Поскольку для обмена информацией модель REST использует существующие методы HTTP, такие как GET (получить данные), POST (создать данные), PUT (обновить данные) и DELETE (удалить данные), она считается более простой в использовании, нежели модель SOAP. Чтобы лучше понять это, взгляните на рис. 10.3, демонстрирующий разницу между пакетами запроса и ответа SOAP и REST.

Example SOAP Request and Response

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:ns1="http://rpc.geocoder.us/Geo/Coder/US/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <ns1:geocode>
   <location xsi:type="xsd:string">1600 Pennsylvania Av, Washington, DC</location>
  </ns1:geocode>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <geocodeResponse xmlns="http://rpc.geocoder.us/Geo/Coder/US/">
   <geo:results soapenc:arrayType="geo:GeocoderAddressResult[1]"
    xsi:type="soapenc:Array"
    xmlns:geo="http://rpc.geocoder.us/Geo/Coder/US/">
    <geo:item xsi:type="geo:GeocoderAddressResult" xmlns:geo="http://rpc.geocoder.us/Geo/Coder/US/">
     <geo:number xsi:type="xsd:int">1600</geo:number>
     <geo:lat xsi:type="xsd:float">38.898748</geo:lat>
     <geo:street xsi:type="xsd:string">Pennsylvania</geo:street>
     <geo:state xsi:type="xsd:string">DC</geo:state>
     <geo:city xsi:type="xsd:string">Washington</geo:city>
     <geo:zip xsi:type="xsd:int">20502</geo:zip>
     <geo:suffix xsi:type="xsd:string">NW</geo:suffix>
     <geo:long xsi:type="xsd:float">-77.037684</geo:long>
     <geo:type xsi:type="xsd:string">Ave</geo:type>
     <geo:prefix xsi:type="xsd:string" />
    </geo:item>
   </geo:results>
  </geocodeResponse>
 </soap:Body>
</soap:Envelope>
```

Example REST Request and Response

```
GET http://geocoder.us/service/rest/geocode?
    address=1600+Pennsylvania+Ave,+Washington+DC

<?xml version="1.0"?>
<rdf:RDF
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 <geo:Point rdf:nodeID="ald47091944">
  <dc:description>1600 Pennsylvania Ave NW, Washington DC 20502</dc:description>
  <geo:long>-77.037684</geo:long>
  <geo:lat>38.898748</geo:lat>
 </geo:Point>
</rdf:RDF>
```

Рис. 10.3. Примеры взаимодействия в SOAP и REST

Если на какой-нибудь вечеринке вам зададут вопрос об основных различиях между подходами SOAP и REST, а неподалеку будет находиться привлекательный представитель противоположного пола, вот ответ на этот вопрос:

- SOAP активно использует XML для кодирования запросов и ответов, а также строгую типизацию данных, гарантирующую их целостность при передаче между клиентом и сервером. С другой стороны, запросы и ответы в REST могут передаваться в ASCII, XML, JSON или любых других форматах, распознаваемых одновременно и клиентом, и сервером. Кроме того, в модели REST отсутствуют встроенные требования к типизации данных. В результате пакеты запросов и ответов в REST имеют намного меньшие размеры, чем соответствующие им пакеты SOAP.
- В модели SOAP уровень передачи данных протокола HTTP является «пассивным наблюдателем», и его роль ограничивается передачей запросов SOAP от клиента серверу с использованием метода **POST**. Детали сервисного запроса, такие как имя удаленной процедуры и входные аргументы, кодируются в теле запроса. Архитектура REST, напротив, рассматривает уровень передачи данных HTTP как активного участника взаимодействия, используя существующие методы HTTP, такие как **GET**, **POST**, **PUT** и **DELETE**, для обозначения типа запрашиваемого сервиса. Следовательно, с точки зрения разработчика, запросы REST в общем случае более просты для формулирования и понимания, так как они используют существующие и хорошо понятные интерфейсы HTTP.
- Модель SOAP поддерживает определенную степень интроспекции, позволяя разработчикам сервиса описывать его API в файле формата Web Service Description Language (WSDL, язык описания веб-сервисов). Создавать эти файлы довольно сложно, однако это стоит затраченных усилий, поскольку клиенты SOAP могут автоматически получать из этих файлов подробную информацию об именах и сигнатурах методов, типах входных и выходных данных и возвращаемых значениях. С другой стороны, модель REST избегает сложностей WSDL в угоду более интуитивному интерфейсу, основанному на стандартных методах HTTP, описанных выше.
- В основе REST лежит концепция *ресурсов*, в то время как SOAP использует интерфейсы, основанные на *объектах* и *методах*. Интерфейс SOAP может содержать практически неограниченное количество методов; интерфейс REST, напротив, ограничен четырьмя возможными операциями, соответствующими четырем методам HTTP.

Использование веб-сервисов

Zend Framework содержит универсальные клиенты для веб-сервисов, основанных на REST и SOAP, а также несколько специализированных реализаций клиентов, которые могут использоваться для взаимодействия с популярными веб-сервисами. Они подробно рассматриваются в следующих разделах.

Использование реализаций клиентов для конкретных сервисов

В состав Zend Framework входят несколько предварительно настроенных клиентов для популярных веб-сервисов, предоставляемых Google, Amazon, Flickr, Technorati и другими. Использование этих реализаций может снизить затраты времени и уси-

лий по интеграции в веб-приложение данных из перечисленных сервисов. В большинстве случаев эти реализации предоставляют объектно-ориентированные API, где ответы сервисов возвращаются в виде встроенных объектов и массивов PHP.

В табл. 10.1 перечислены реализации для конкретных сервисов, входящие в текущую версию Zend Framework.

ВОПРОС ЭКСПЕРТУ

В: Если я создаю веб-сервис, что я должен использовать: SOAP или REST?

О: Как и в случае со всеми подобными вопросами, ответ на самом деле зависит от ваших требований и целей. Для сервисов, связанных с банковскими операциями или электронной торговлей, или в случаях, когда сервис является частью большой распределенной платформы, строгая типизация данных и формализованное описание сервиса делают SOAP лучшим выбором, чем REST. С другой стороны, если вы хотите «на скорую руку» обеспечить доступ к внутренним данным вашего приложения и делаете ставку на простоту использования и гибкость форматов, то, возможно, лучше выбрать REST. Несмотря на это, стоит отметить, что подход REST постепенно обгоняет по популярности подход SOAP, так как он грамотно реализует существующие технологии и при этом более прост для использования и понимания.

Таблица 10.1. Реализации клиентов веб-сервисов, входящие в состав Zend Framework

Класс клиента	Описание
Zend_Service_Akismet	Клиент для приложения Akismet, блокирующего спам
Zend_Service_Amazon	Клиент для API веб-сервисов Amazon, с отдельными реализациями для Amazon EC2, Amazon SQS и Amazon S3
Zend_Service_AudioScrobbler	Клиент для системы поиска музыки Last.fm
Zend_Service_Delicious	Клиент для приложения del.icio.us, предлагающего услугу социальных закладок
Zend_Service_Flickr	Клиент для приложения Flickr, позволяющего обмениваться фотографиями
Zend_Service_Nirvanix	Клиент для приложения Nirvanix, предоставляющего услуги по хранению данных
Zend_Service_ReCaptcha	Клиент для генератора CAPTCHA ReCaptcha
Zend_Service_Simply	Клиент для приложения Simpy, предлагающего услугу социальных закладок
Zend_Service_SlideShare	Клиент для приложения SlideShare, предназначенного для обмена презентациями
Zend_Service_StrikeIron	Клиент для коммерческого агрегатора баз данных StrikeIron
Zend_Service_Technorati	Клиент для системы поиска по блогам Technorati
Zend_Service_Twitter	Клиент для сервиса микроблогов Twitter
Zend_Service_Yahoo	Клиент для Yahoo! Search API
Zend_Gdata	Клиент для Google Data API, с отдельными реализациями для таких приложений Google, как Notebook (Блокнот), Calendar (Календарь), YouTube, Base, Spreadsheets (Таблицы) и Documents (Документы)

В рамках данной главы невозможно дать полное описание каждой реализации, но мы рассмотрим несколько наглядных примеров. Следующий листинг использует компонент **Zend_Service_Delicious** для получения списка тегов и закладок пользователя на сервисе del.icio.us:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function deliciousAction()
  {
    // инициализируем объект сервиса
    $client = new Zend_Service_Delicious('username', 'password');

    // получаем список всех тегов, созданных пользователем,
    // а также частоту их использования
    $this->view->tags = $client->getTags();

    // получаем все сообщения, отмеченные тегом PHP
    $this->view->posts = $client->getPosts('php');
  }
}
```

Ниже приведен другой пример, который использует компонент **Zend_Gdata** для запроса списка из пяти наиболее популярных в данный момент видео на сервисе YouTube:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function youtubeAction()
  {
    // инициализируем объект сервиса
    $client = new Zend_Gdata_YouTube();

    // получаем ленту с пятью наиболее популярными видео
    $feed = $client->getVideoFeed(
        'http://gdata.youtube.com/feeds/api/standardfeeds/most_popular?
        max-results=5');

    // обрабатываем ленту и получаем отдельные записи
    // получаем название, категорию, рейтинг,
    // количество просмотров и url каждой записи
    $this->view->mostViewed = array();
    foreach ($feed as $entry) {
    $this->view->mostViewed[] = array(
        'title' => $entry->getVideoTitle(),
        'rating' => $entry->getVideoRatingInfo(),
        'category' => $entry->getVideoCategory(),
        'views' => $entry->getVideoViewCount(),
        'watch' => $entry->getVideoWatchPageUrl()
```

В этом примере метод **getVideoFeed()** возвращает ленту Atom, содержащую результаты поиска видео; она автоматически анализируется и преобразуется в массив объектов **Zend_Gdata_YouTube_VideoEntry**, каждый из которых представляет собой одну запись в ленте. Теперь достаточно лишь пройти по массиву, получить информацию о каждой записи, используя свойства объекта, и представить ее в виде веб-страницы.

И наконец, пример использования реализации **Zend_Service_Amazon** для поиска книг по определенному автору на сервисе Amazon.com:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function amazonAction()
  {
    // инициализируем объект сервиса
    $client = new Zend_Service_Amazon('ACCESS-KEY', 'US', 'SECRET-KEY');

    // осуществляем поиск элементов на Amazon.com по их атрибутам
    $items = $client->itemSearch(array(
        'SearchIndex' => 'Books',
        'Author' => 'Dan Brown',
        'ResponseGroup' => 'Large'
    ));

    // обрабатываем результаты поиска
    $this->view->results = array();
    foreach ($items as $i) {
    $this->view->results[] = array (
        'asin' => $i->ASIN,
        'title' => $i->Title,
        'author' => $i->Author,
        'url' => $i->DetailPageURL,
        'rating' => $i->AverageRating,
        'salesRank' => $i->SalesRank,
      );
    }
  }
}
```

Здесь метод **itemSearch()** объекта сервиса запрашивает у веб-сервиса Amazon список товаров, соответствующих заданным параметрам, и возвращает массив объектов **Zend_Service_Amazon_Item**, каждый из которых содержит информацию об определенном товаре. Теперь достаточно просто пройтись по этому массиву и извлечь нужную информацию для отображения.

Использование универсальных реализаций клиентов

Если вам нужно воспользоваться услугами веб-сервиса, для которого не существует встроенной реализации, не отчаивайтесь: для доступа к таким сервисам Zend Framework содержит универсальные клиенты SOAP и REST. Они реализованы в виде компонентов **Zend_Soap_Client** и **Zend_Rest_Client** соответственно.

Чтобы продемонстрировать работу с ними, рассмотрим следующий пример, использующий универсальный клиент SOAP для доступа к веб-сервису GeoCoder и возвращающий широту и долготу, соответствующие определенному адресу:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function soapAction()
  {
    // инициализируем клиент SOAP
    $soap = new Zend_Soap_Client(
        'http://geocoder.us/dist/eg/clients/GeoCoderPHP.wsdl', array(
        'soap_version' => SOAP_1_1,
        'compression' => SOAP_COMPRESSION_ACCEPT,
    ));

    // получаем широту и долготу указанного адреса
    $response = $soap->geocode('E Capitol St NE & 1st St NE, Washington, DC');
    $this->view->results = array();
    foreach ($response as $r) {
    $this->view->results[] = array(
        'zip' => $r->zip,
        'latitude' => $r->lat,
        'longitude' => $r->long
      );
    }

    // получаем запрос и ответ SOAP для отладки
    $this->view->request = $soap->getLastRequest();
    $this->view->response = $soap->getLastResponse();
  }
}
```

В этом примере создается новый объект **Zend_Soap_Client**, который инициализируется URL, ссылающимся на WSDL-файл веб-сервиса. Как говорилось ранее, этот файл позволяет клиенту автоматически получать информацию о доступных методах, ожидаемых типах данных и содержимом входных и выходных аргументов. Второй аргумент конструктора объекта представляет собой массив конфигурационных параметров, который можно использовать для определения различных аспектов поведения клиента.

ПРИМЕЧАНИЕ

Zend_Soap_Client можно также использовать и без WSDL, убрав соответствующий URL и непосредственно указав URL сервиса SOAP (endpoint (конечная точка)) в качестве аргумента в массиве параметров.

После создания экземпляра объекта **Zend_Soap_Client** можно получать доступ к методам удаленного сервиса «по доверенности», вызывая соответствующие методы объекта. Объект **Zend_Soap_Client** автоматически позаботится о создании пакета с сообщением SOAP, передаче его серверу посредством POST, получении

пакета с ответом и преобразовании его в собственный объект PHP Zend_Soap_Client_Response. Это видно из предыдущего примера, который вызывает метод geocode() сервиса, передает ему строку адреса (в данном случае адрес Капитолия в Вашингтоне) и получает в ответ почтовый индекс, широту и долготу.

ВНИМАНИЕ

Компонент Zend_Soap_Client использует расширение PHP ext/soap и не будет работать, если оно отсутствует в вашей установке PHP.

Универсальный и единообразный способ доступа к веб-сервисам, основанным на REST, предоставлен реализацией Zend_Rest_Client. Рассмотрим следующий пример, иллюстрирующий попытку доступа к REST-сервису GeoNames для получения информации о стране:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function restAction()
  {
    // инициализируем клиент REST
    $client = new Zend_Rest_Client('http://ws.geonames.org/countryInfo');

    // устанавливаем значения аргументов
    $client->country('IN');

    // выполняем запрос GET для получения информации о стране,
    // анализируем результаты
    $result = $client->get();
    $this->view->countryName = $result->countryName();
    $this->view->countryCode = $result->countryCode();
    $this->view->capital = $result->capital();
    $this->view->areaKM = $result->areaInSqKm();
    $this->view->population = $result->population();
    $this->view->languages = $result->languages();
    $this->view->mapCoordinates = array(
        $result->bBoxWest(),
        $result->bBoxEast(),
        $result->bBoxNorth(),
        $result->bBoxSouth()
    );

    // получаем запрос и ответ REST для отладки
    $this->view->request = $client->getLastRequest();
    $this->view->response = $client->getLastResponse();
  }
}
```

Как и в предыдущем примере, сначала необходимо инициализировать клиент REST, создав экземпляр объекта Zend_Rest_Client и передав в его конструктор URL REST-сервиса. Аргументы указываются с помощью «текучего интерфейса», где

названиям аргументов соответствуют методы объекта, а фактическая передача запроса осуществляется посредством метода get(), post(), put() или delete(), который внутри объекта преобразуется в соответствующий метод HTTP. Это ясно видно из приведенного примера, который формирует запрос GET к сервису GeoNames, предоставляющему информацию о странах, и передает в качестве аргумента код Индии: http://ws.geonames.org/countryInfo?country=IN.

Ответ на запрос Zend_Rest_Client является экземпляром объекта Zend_Rest_Client_Response и обычно представлен в виде объекта SimpleXML или массива таких объектов. Доступ к отдельным элементам ответа можно получить, либо используя нотацию SimpleXML, либо непосредственно обращаясь к методу, соответствующему имени элемента. В последнем случае для возврата всех подходящих элементов, вне зависимости от их иерархической позиции в дереве документа XML, будет использован запрос XPath.

Как видно из двух приведенных примеров, Zend Framework упрощает доступ к любому веб-сервису, основанному на SOAP или REST, даже при отсутствии специализированной реализации клиента, и не требует обширных знаний о внутреннем устройстве сервиса. Эти факторы делают Zend Framework ценным инструментом для быстрой и эффективной интеграции сторонних веб-сервисов в приложение на PHP.

СОВЕТ

Обратите внимание на методы getLastRequest() и getLastResponse() из двух последних листингов. Эти методы очень удобны для отладки транзакций SOAP и REST, а также для проверки содержимого пакетов запроса и ответа.

Упражнение 10.1. Интеграция с Twitter и добавление результатов поиска по блогам

Обладая всей исходной информацией, давайте посмотрим, как использовать ее в реальных условиях. В следующих разделах вы увидите, как можно объединить компоненты, рассмотренные в предыдущих разделах, для отображения в демонстрационном приложении SQUARE свежих новостей и сообщений в блогах и Twitter, относящихся к теме филателии.

Определение пользовательских маршрутов

Как обычно, первый шаг — определение пользовательского маршрута к странице новостей. Отредактируйте конфигурационный файл приложения, $APP_DIR/application/configs/application.ini, и добавьте в него следующее определение маршрута:

```
resources.router.routes.news.route = /news
resources.router.routes.news.defaults.module = default
resources.router.routes.news.defaults.controller = news
resources.router.routes.news.defaults.action = index
```

Определение контроллера и представления

Следующим шагом будет определение действия и представления для интерфейса новостей. Ниже приведен код контроллера **NewsController**, который следует сохранить в файл **$APP_DIR/application/modules/default/controllers/NewsController.php**:

```
<?php
class NewsController extends Zend_Controller_Action
{
  public function indexAction()
  {
    // получаем ленту с результатами поиска по Twitter
    $q = 'philately';
    $this->view->q = $q;
    $twitter = new Zend_Service_Twitter_Search();
    $this->view->tweets = $twitter->search($q,
      array('lang' => 'en', 'rpp' => 8, 'show_user' => true));

    // получаем ленту Google News, представленную в формате Atom
    $this->view->feeds = array();
    $gnewsFeed = "http://news.google.com/news?hl=en&q=$q&output=atom";
    $this->view->feeds[0] = Zend_Feed_Reader::import($gnewsFeed);

    // получаем ленту BPMA, представленную в формате RSS
    $bpmaFeed = "http://www.postalheritage.org.uk/news/RSS";
    $this->view->feeds[1] = Zend_Feed_Reader::import($bpmaFeed);
  }
}
```

Метод **indexAction()** отвечает за доступ к различным новостным лентам и лентам веб-сервисов, а также за их импорт.

- Реализация **Zend_Service_Twitter** позволяет выполнять поиск в Twitter на предмет недавних обновлений статусов, содержащих слово **'philately'** (филателия). Метод **search()** объекта принимает критерий запроса, а также массив параметров, определяющих фильтрацию по языку, количество возвращаемых элементов и необходимость включения имени автора сообщения в каждый элемент списка результатов поиска. Возвращаемым значением является вложенный массив результатов, соответствующих критерию поиска, который затем добавляется в представление для дальнейшей обработки.
- Для импорта двух новостных лент используется реализация **Zend_Feed_Reader**. Первая из этих лент имеет формат Atom и содержит заголовки последних новостей, относящихся к филателии, на сервисе Google News, а вторая содержит список недавних сообщений из официального блога Британского почтового музея и архива (BPMA, British Postal Museum and Archive). Полученные в результате объекты ленты затем присваиваются переменным сценария представления.

Ниже приведен соответствующий сценарий представления, который следует сохранить в файл **$APP_DIR/application/modules/default/views/scripts/news/index.phtml**:

```
2>News</h2>
iv id="newsfeeds">
<div id="posts">
<strong>Recent news about
  <a href="http://news.google.com/news?hl=en&q=<?php echo $this->q; ?>">
    '<?php echo $this->q; ?>'
  </a>:
</strong>
<p id="hdiv"></p>
<?php $count = 0; ?>
<?php foreach ($this->feeds[0] as $entry):?>
  <?php if ($count >= 5) break; ?>
  <p class="post">
  <span class="text">
  <a href="<?php echo $entry->getLink(); ?>">
    <?php echo $entry->getTitle(); ?>
  </a>
</span>
  <span class="time">
    <?php echo $entry->getDateModified(); ?>
  </span>
  </p>
<?php $count++; ?>
<?php endforeach; ?>
<p style="padding-top: 4px"/>
<strong>Recent posts from
  <a href="http://www.postalheritage.org.uk/">
    the British Postal Museum and Archive
  </a> blog:
</strong>
  <p id="hdiv"></p>
  <?php $count = 0; ?>
  <?php foreach ($this->feeds[1] as $entry):?>
  <?php if ($count >= 5) break; ?>
  <p class="post">
   <span class="text">
  <a href="<?php echo $entry->getLink(); ?>">
  <?php echo $entry->getTitle(); ?>
  </a>
</span>
  <span class="time">
    <?php echo $entry->getDateModified(); ?>
  </span>
</p>
  <?php $count++; ?>
  <?php endforeach; ?>
</div>

<div id="tweets">
  <strong>Recent tweets about
```

```
    <a href="http://search.twitter.com/search?q=<?php echo $this->q; ?>">
      '<?php echo $this->q; ?>'
    </a>:
  </strong>
  <?php foreach ($this->tweets['results'] as $tweet):?>
    <p class="tweet">
    <span class="image">
      <img src="<?php echo $tweet['profile_image_url']; ?>" />
    </span>
    <span class="user">
      <?php echo $tweet['from_user'] . ': '; ?>
    </span>
    <span class="text">
      <?php echo $tweet['text']; ?>
    </span>
    <span class="time">
      <?php echo $tweet['created_at']; ?>
    </span>
      </p>
    <?php endforeach; ?>
  </div>
</div>
```

Здесь нет ничего сложного. Сценарий представления всего лишь проходит по результирующему списку объектов, созданному в контроллере, извлекает из каждого объекта необходимую информацию и отображает ее в в виде двух аккуратно отформатированных колонок.

Обновление основного макета

Все, что осталось сделать, — обновить основной макет в файле **$APP_DIR/application/layouts/master.phtml**, чтобы добавить в главное меню ссылку на страницу новостей. В следующем листинге изменения в макете выделены жирным шрифтом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

    <div id="menu-locale-container">
      <div id="menu">


      <a href="<?php echo $this->url(array(), 'news'); ?>">
        <?php echo $this->translate('menu-news'); ?>
      </a>
      </div>
  </div>
...
</html>
```

ПРИМЕЧАНИЕ

Не забудьте добавить в каждый из исходных файлов перевод нового элемента меню для всех поддерживаемых языков, как это делалось в главе 9. Архив с материалами к этой главе содержит все необходимые дополнения.

Чтобы посмотреть, как работает написанный код, откройте в вашем браузере URL http://square.localhost/news, и вы увидите примерно то, что показано на рис. 10.4.

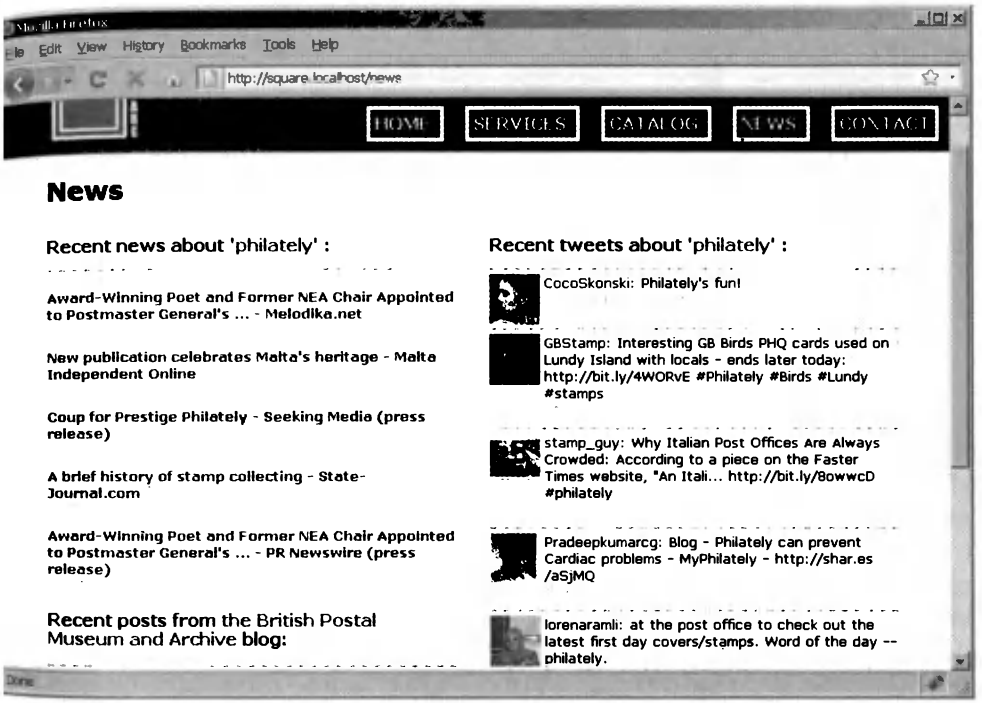


Рис. 10.4. Динамическое получение и интеграция результатов поиска по блогам, новостям и Twitter

Создание веб-сервисов, основанных на REST

Zend Framework не только предоставляет набор средств для доступа к новостным лентам и веб-сервисам, а также их интеграции в приложение на PHP, но и упрощает предоставление доступа к функциям приложения путем создания собственных веб-сервисов, основанных на REST. Как правило, это делается с помощью компонентов Zend_Rest_Route и Zend_Rest_Controller, которые рассматриваются в следующем

Маршруты REST

В основу архитектуры REST положены два ключевых принципа:

- Приложение состоит из нескольких ресурсов, каждый из которых доступен по уникальному URL.
- Метод, используемый для доступа к ресурсу, определяет действие, выполняемое с этим ресурсом.

В табл. 10.2 перечислены типовые маршруты URL для сервисов, основанных на REST.

Таблица 10.2. Стандартные маршруты REST

URL запроса	Метод запроса	Описание	Пример
/api/items	GET	Получить все элементы	GET /api/items
/api/items	POST	Добавить новый элемент	POST /api/items
/api/items/:id	GET	Получить запись об элементе	GET /api/items/25
/api/items/:id	PUT	Обновить запись об элементе	PUT /api/items/15
/api/items/:id	DELETE	Удалить запись об элементе	DELETE /api/items/13

Из табл. 10.2 должно быть понятно, что в модели REST для обозначения действия, выполняемого с указанным ресурсом, используются существующие методы HTTP, такие как GET (получить данные), POST (создать данные), PUT (обновить данные) и DELETE (удалить данные). Если требуется, к URL запроса добавляется уникальный идентификатор ресурса. Любые дополнительные аргументы, особенно при создании новых ресурсов или обновлении существующих, передаются в теле запроса POST или PUT.

В состав Zend Framework входит компонент Zend_Rest_Route, который может автоматически активировать стандартные маршруты REST для всего приложения, отдельного модуля или отдельного контроллера. Для соответствующих запросов URL этот компонент будет автоматически определять метод запроса, анализировать его аргументы и вызывать одно из подходящих стандартных действий: indexAction(), getAction(), postAction(), putAction() и deleteAction().

Пять перечисленных стандартных действий определены в классе Zend_Rest_Controller в виде абстрактных методов. Этот контроллер, который расширяет класс Zend_Controller_Action, служит в Zend Framework основой для всех контроллеров, использующих REST. Ниже приведен его каркас, поясняющий эту идею:

```
<?php
class Sandbox_RestController extends Zend_Rest_Controller
{
  public function indexAction()
  {
    // обработка запросов GET
  }
```

```
  public function getAction()
  {
    // обработка запросов GET
  }

  public function postAction()
  {
    // обработка запросов POST
  }
  public function putAction()
  {
    // обработка запросов PUT
  }
  public function deleteAction()
  {
    // обработка запросов DELETE
  }
}
```

Упражнение 10.2. Реализация веб-сервисов, основанных на REST

Из материалов предыдущего раздела становится ясно, что применение к приложению архитектуры REST, в особенности по отношению к стандартным операциям CRUD, подразумевает использование двух важных компонентов:

1. Контроллера, который расширяет **Zend_Rest_Controller** и реализует действия, соответствующие стандартным методам HTTP.
2. Маршрута, который использует **Zend_Rest_Route** для сопоставления входящих запросов URL действиям вышеупомянутого контроллера.

Понять работу этих компонентов проще всего на примере. Он будет приведен в следующих разделах, где для демонстрационного приложения SQUARE мы создадим простой API, основанный на REST. Этот API будет поддерживать методы **GET** и **POST** и позволит пользователям получать информацию о существующих элементах каталога, а также добавлять в него новые элементы.

Создание нового модуля

Для начала создадим новый модуль (назовем его **api**), в котором будет размещаться контроллер REST. Для этого перейдите в каталог **$APP_DIR/application/** и выполните следующие команды:

```
shell> mkdir modules/api
shell> mkdir modules/api/controllers
shell> mkdir modules/api/views
shell> mkdir modules/api/views/scripts
```

Определение контроллера

Следующим шагом будет определение контроллера API, который займется обработкой всех запросов REST. Как говорилось ранее, этот контроллер должен расширять абстрактный класс **Zend_Rest_Controller** и реализовывать все его абстрактные методы. Пример:

```
<?php
class Api_CatalogController extends Zend_Rest_Controller
{
  // отключаем макеты и формирование страницы
  public function init()
  {
    $this->apiBaseUrl = 'http://square.localhost/api/catalog';
    $this->_helper->layout->disableLayout();
    $this->getHelper('viewRenderer')->setNoRender(true);
  }

  public function indexAction()
  {
    // обработка запросов GET
  }

  public function getAction()
  {
    // обработка запросов GET
  }

  public function postAction()
  {
    // обработка запросов POST
  }

  public function putAction()
  {
    // обработка запросов PUT
  }

  public function deleteAction()
  {
    // обработка запросов DELETE
  }
}
```

Обратите внимание на метод **init()** контроллера, отключающий все макеты и формирование страницы сценарием представления. Это делается потому, что REST API обычно возвращает ответы в виде XML, JSON или (в некоторых случаях) текста в кодировке ASCII. Этот вывод можно сгенерировать в самом контроллере и непосредственно отправить клиенту.

Сохраните приведенный контроллер в файл **$APP_DIR/application/modules/api/controllers/CatalogController.php**.

Определение действий GET

Теперь давайте определим действие **Api_CatalogController::indexAction**. Как вы помните, это действие автоматически вызывается, когда маршрутизатор обнаруживает запрос GET без переменной запроса **id**, и должно возвращать список всех подходящих ресурсов. Код действия:

```
<?php
class Api_CatalogController extends Zend_Rest_Controller
{
  public function indexAction()
  {
    // получаем записи из базы данных
    $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Type t')
        ->addWhere('i.DisplayStatus = 1');
    $result = $q->fetchArray();

    // устанавливаем значения элементов с информацией о ленте
    $output = array(
      'title' => 'Catalog records',
      'link' => $this->apiBaseUrl,
      'author' => 'Square API/1.0',
      'charset' => 'UTF-8',
      'entries' => array()
    );

    // устанавливаем значения элементов с информацией о записях
    foreach ($result as $r) {
      $output['entries'][] = array(
        'title' => $r['Title'] . ' - ' . $r['Year'],
        'link' => $this->apiBaseUrl . '/' . $r['RecordID'],
        'description' => $r['Description'],
        'lastUpdate' => strtotime($r['RecordDate']),
        'square:title' => $r['Title']
      );
    }

    // импортируем массив в ленту Atom
    // и отправляем клиенту
    $feed = Zend_Feed::importArray($output, 'atom');
    $feed->send();
    exit;
    }

    // перенаправляем на indexAction
    public function listAction() {
     return $this->_forward('index');
  }
}
```

Действие Api_CatalogController::indexAction начинается с выполнения запроса Doctrine для получения списка всех активных элементов каталога. Затем оно проходит по списку результатов, преобразуя их во вложенный массив, подходящий для преобразования в ленту Atom с помощью метода Zend_Feed::importArray. После этого получившаяся лента возвращается клиенту посредством метода send() объекта ленты.

ПРИМЕЧАНИЕ

Версии Zend Framework 1.9.2 и более ранние содержали ошибку, приводящую к тому, что маршрутизатор пытался передать запросы REST, предназначенные для метода indexAction(), методу listAction(). В предыдущем примере показан простой способ обхода этой ошибки: перенаправьте запросы к listAction(), если они есть, методу indexAction(), используя метод контроллера _forward().

Действие Api_CatalogController::getAction автоматически вызывается, когда маршрутизатор обнаруживает запрос GET, содержащий переменную запроса id, обозначающую запрос определенного ресурса. В этом случае действие должно получить из каталога указанную запись и вернуть запросившему клиенту подробную информацию о ней. Код действия:

```
<?php
class Api_CatalogController extends Zend_Rest_Controller
{
  public function getAction()
  {
    // получаем из базы строку с информацией об элементе
    $id = $this->_getParam('id');
    $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Type t')
        ->where('i.RecordID = ?', $id)
        ->addWhere('i.DisplayStatus = 1');
    $result = $q->fetchArray();

    // если строка доступна,
    // устанавливаем значения элементов с информацией о записи
    if (count($result) == 1) {
      // устанавливаем значения элементов с информацией о ленте
      $output = array(
        'title' => 'Catalog record for item ID: ' . $id,
        'link' => $this->apiBaseUrl . '/' . $id,
        'author' => 'Square App/1.0',
        'charset' => 'UTF-8',
        'entries' => array()
    );
```

```
        $output['entries'][0] = array(
          'title' => $result[0]['Title'] . ' - ' . $result[0]['Year'],
          'link' => $this->apiBaseUrl . '/' . $id,
          'description' => $result[0]['Description'],
          'lastUpdate' => strtotime($result[0]['RecordDate'])
        );

        // импортируем массив в ленту Atom
        $feed = Zend_Feed::importArray($output, 'atom');
        Zend_Feed::registerNamespace('square', 'http://square.localhost');

        // устанавливаем значения элементов в пользовательском
        // пространстве имен
        $feed->rewind();
        $entry = $feed->current();
        if ($entry) {
          $entry->{'square:id'} = $result[0]['RecordID'];
          $entry->{'square:title'} = $result[0]['Title'];
          $entry->{'square:year'} = $result[0]['Year'];
          $entry->{'square:grade'} =
            $result[0]['Square_Model_Grade']['GradeName'];
          $entry->{'square:description'} = $result[0]['Description'];
          $entry->{'square:country'} =
            $result[0]['Square_Model_Country']['CountryName'];
          $entry->{'square:price'} = null;
          $entry->{'square:price'}->{'square:min'}= $result[0]['SalePriceMin'];
          $entry->{'square:price'}->{'square:max'} = $result[0]['SalePriceMax'];
        }

        // отправляем клиенту
          $feed->send();
          exit;
        } else {
          $this->getResponse()->setHttpResponseCode(404);
          echo 'Invalid record identifier';
          exit;
        }
      }
    }
```

Этот листинг очень похож на предыдущий, за исключением того, что в этом случае лента Atom содержит дополнительные элементы (в пользовательском пространстве имен), содержащие информацию о типе марки, ее состоянии, цене и местоположении, а также описание. Обратите внимание, что в базе данных отсутствует запись, соответствующая переданному идентификатору ресурса, процесс создания ленты пропускается и клиенту сразу отправляется ошибка 404 (страница не найдена). Это пример грамотного использования архитектурой REST существующих стандартов — в данном случае кодов ошибок — для предоставления информации о статусе запроса.

Определение действия POST

Действие **Api_CatalogController::postAction** автоматически вызывается, когда маршрутизатор обнаруживает запрос **POST**. В соответствии с принятыми в REST соглашениями такой запрос должен приводить к созданию нового ресурса, а тело запроса **POST** должно содержать необработанные данные, необходимые для его создания. Пример кода:

```
<?php
class Api_CatalogController extends Zend_Rest_Controller
{
  public function postAction()
  {
    // читаем параметры запроса POST и сохраняем их в базу данных
    $item = new Square_Model_Item;
    $item->fromArray($this->getRequest()->getPost());
    $item->RecordDate = date('Y-m-d', mktime());
    $item->DisplayStatus = 0;
    $item->DisplayUntil = null;
    $item->save();
    $id = $item->RecordID;

    // присваиваем коду ответа значение 201
    // отправляем идентификатор только что созданной записи
    $this->getResponse()->setHttpResponseCode(201);
    $this->getResponse()->setHeader('Location', $this->apiBaseUrl.'/'.$id);
    echo $this->apiBaseUrl.'/'.$id;
    exit;
  }
}
```

Обратите внимание, что после создания записи о ресурсе и сохранения ее в базу данных клиенту отправляется код ответа 201 (создано), а также URL только что созданного ресурса.

ВОПРОС ЭКСПЕРТУ

В: Почему ваш REST API возвращает результаты в виде лент Atom?

О: Хотя API, основанные на REST, могут возвращать данные в любом формате, часто для этих целей используется формат лент Atom или RSS. Это связано с тем, что большинство современных HTTP-клиентов имеют встроенную поддержку данных лент, а в большинстве языков программирования есть встроенные или легкодоступные средства для анализа этих форматов.

Инициализация маршрутов REST

Последний этап — инициализация маршрута к REST API. Для этого откройте начальный загрузчик приложения и добавьте в него следующий метод:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
```

```
protected function _initRoutes()
{
  $front = Zend_Controller_Front::getInstance();
  $router = $front->getRouter();
  $restRoute = new Zend_Rest_Route($front, array(), array('api'));
  $router->addRoute('api', $restRoute);
}
```

Метод **_initRoutes()** отвечает за инициализацию нового экземпляра **Zend_Rest_Route** и его сопоставление новому модулю **api**, созданному на первом этапе. В качестве третьего аргумента конструктору **Zend_Rest_Route** передается массив имен модулей и/или контроллеров, для которых требуется активировать поддержку REST.

К этому моменту ваш REST API настроен и работает. Вы можете убедиться этом, запросив в веб-браузере URL **http://square/localhost/api/catalog**, в результате его вы должны получить ленту Atom, содержащую список доступных для отображения марок из каталога (рис. 10.5).

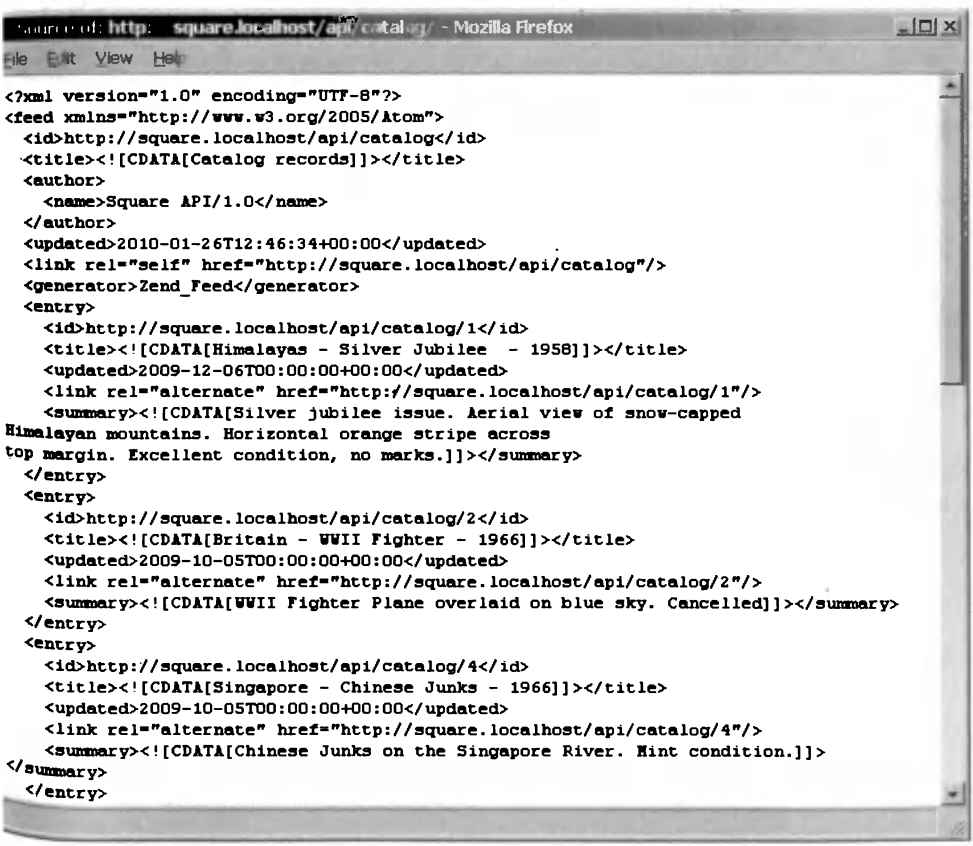


Рис. 10.5. Ответ API на запрос GET для получения всех записей

Аналогичным образом обращение к тому же URL с указанием идентификатора элемента, например http://square.localhost/api/catalog/1, генерирует ленту Atom с подробной информацией о нем (рис. 10.6).

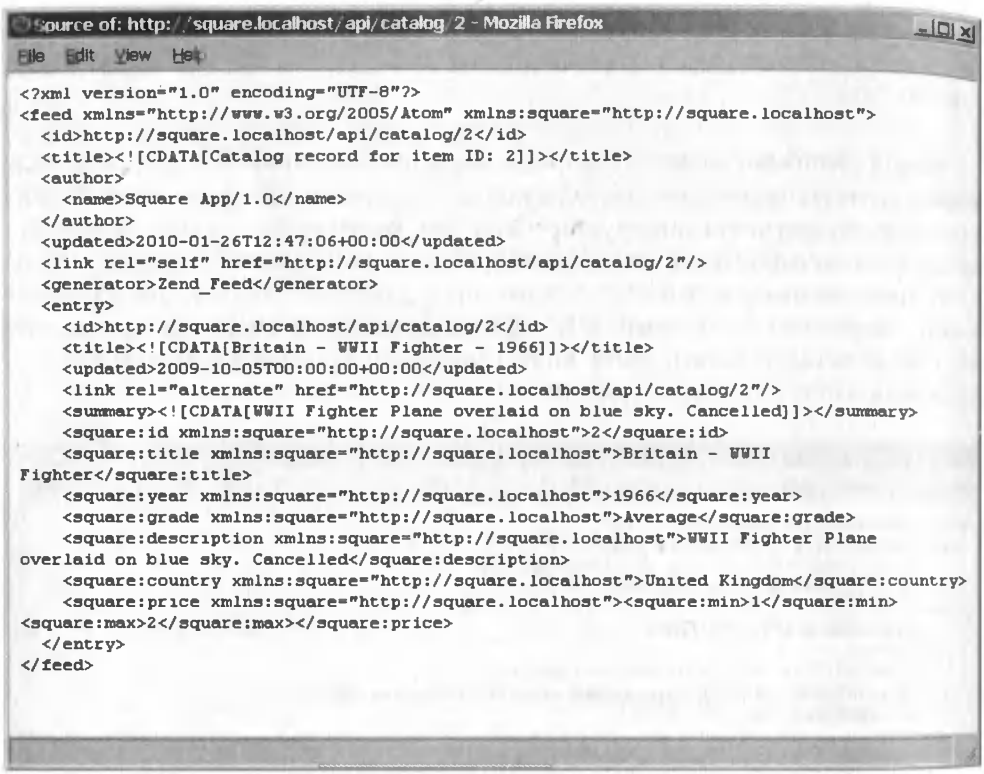


Рис. 10.6. Ответ API на запрос GET для получения конкретной записи

И наконец, передача массива данных в запросе POST по URL http://square/localhost/api/catalog приведет к созданию в базе данных новой записи об элементе, а клиенту будет возвращен URL этого элемента (рис. 10.7).

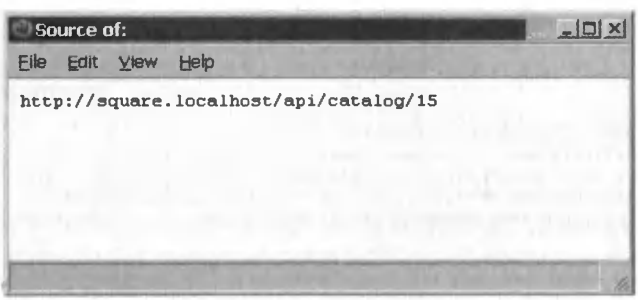


Рис. 10.7. Ответ API на запрос POST

ыводы

Большинство блогов и новостных сайтов содержат ленты в формате RSS или
:om, которые можно использовать для сбора нового содержимого в сторонних
иложениях. Написание анализаторов для получения этих новостных лент — рас-
юстраненная задача при разработке приложений, и Zend Framework упрощает ее
шение, предоставляя набор встроенных компонентов для генерации, изменения
анализа новостных лент. В данной главе были подробно рассмотрены эти компо-
нты и показано, как можно использовать их для интеграции в демонстрационное
иложение SQUARE лент из источников новостей в Интернете.

Давая третьим лицам доступ к функциям приложения, веб-сервисы позволяют
здавать «мэшапы» и сопутствующие продукты, которые интегрируют данные
нескольких совершенно разных источников, способствуя техническим инно-
циям. Zend Framework позволяет с легкостью следовать повальному увлечению
б-сервисами: он содержит универсальные клиенты для доступа к сервисам REST
SOAP, а также готовые реализации клиентов для веб-сервисов, предлагаемых
пулярными приложениями, такими как Google, Amazon и Twitter. В этой главе
ыли продемонстрированы некоторые из таких реализаций, а также на примере
монстрационного приложения SQUARE объяснены базовые принципы создания
б-сервиса, основанного на REST.

Чтобы получить больше информации по темам, рассмотренным в этой главе,
ысетите следующие ссылки:

Компонент Zend_Feed_Reader: http://framework.zend.com/manual/en/zend.feed.reader.html.

Компонент Zend_Feed: http://framework.zend.com/manual/en/zend.feed.html.

Компонент Zend_Rest_Client: http://framework.zend.com/manual/en/zend.rest.client.html.

Компонент Zend_Soap_Client: http://framework.zend.com/manual/en/zend.soap.client.html.

Компонент Zend_Service: http://framework.zend.com/manual/en/zend.service.html.

Компонент Zend_Gdata: http://framework.zend.com/manual/en/zend.gdata.html.

Спецификация формата Atom Syndication Format 1.0: http://tools.ietf.org/html/rfc4287.

Информация о формате RSS в Wikipedia: http://en.wikipedia.org/wiki/RSS.

Информация об архитектурах SOAP and REST в Wikipedia: http://en.wikipedia.org/wiki/Web_service.

Сравнение веб-сервисов REST и SOAP (Амит Асаравала (Amit Asaravala)): http://www.devx.com/DevX/Article/8155.

Создание интерфейсов на основе REST в Zend Framework (Мэтью Вайер О'Финни (Matthew Weier O'Phinney)): http://weierophinney.net/matthew/archives/228-Building-RESTful-Services-with-Zend-Framework.html.

- Скринкаст о создании веб-сервиса и клиента на основе REST (Йон Лебенсолд (Jon Lebensold)): http://www.zendcasts.com/writing-a-restful-web-service-and-client-with-zend_controller-and-zend_httpclient/2009/04/.
- Подход к созданию REST API (без использования фреймворка) (Ян Селби (Ian Selby)): http://www.gen-x-design.com/archives/create-a-rest-api-with-php/.
- Несколько примеров кодов ответа HTTP и их использования в приложении Zend Framework, основанном на REST (Судхир Сатьянарайна (Sudheer Satyanarayana)): http://techchorus.net/create-restful-applications-using-zend-framework-part-ii-usinghttp-response-code.

Работа с элементами пользовательского интерфейса 11

Прочитав эту главу, вы:

- улучшите навигацию по сайту с помощью меню, карт сайта и навигационных цепочек;
- узнаете об интеграции Zend Framework с Dojo;
- создадите форму ввода с автоматическим дополнением, использующую AJAX;
- используете в Zend Framework элементы управления из библиотеки YUI;
- поймете, как создавать и использовать вспомогательные классы действий.

Выделение сути веб-приложения для создания понятной и единообразной навигации по нему — одна из тех задач, которые кажутся легкими в теории, но правильное решение которых на практике оказывается весьма непростым. Создание удобной в использовании и единообразной схемы навигации для веб-приложения зачастую занимает много часов раздумий и экспериментов. Однако затраченные усилия, безусловно, того стоят: навигация — один из ключевых элементов, определяющих удобство пользования сайтом, и правильная ее реализация помогает пользователям эффективно находить информацию, делая их счастливыми и побуждая к повторному посещению сайта.

Кроме хорошей навигации удобство использования приложения можно также повысить путем разумного использования приемов программирования на стороне клиента. Богатые клиентские API, представленные в большинстве современных браузеров, наряду с легкодоступным инструментарием для программирования наподобие jQuery, mooTools и Dojo позволяют быстро добавлять в веб-приложение новое поведение и функции, улучшающие его отзывчивость, сокращающие время ожидания и (опять же) делающие пользователей счастливыми.

«Какое отношение все это имеет к Zend Framework?» — спросите вы. Дело в том, что Zend Framework содержит пару компонентов, имеющих непосредственное от-

ношение к повышению удобства пользования приложениями. В частности, в нем присутствует компонент **Zend_Navigation**, который предоставляет гибкий и развитый API для реализации различных типов структур для навигации по сайту и управления ими. А компонент **Zend_Dojo** делает возможной непосредственную интеграцию в веб-приложение элементов управления из состава Dojo Toolkit, использующих DHTML и AJAX. В этой главе рассмотрены оба компонента и показано, насколько просто интегрировать в приложение Zend Framework сторонний инструментарий на JavaScript — библиотеку Yahoo! User Interface (YUI).

Работа с навигационными структурами

Zend_Navigation обеспечивает объектно-ориентированный подход к управлению навигационными ссылками на веб-сайте или в веб-приложении. Он предоставляет механизм, позволяющий описывать связи между различными страницами веб-приложения и отображать эти связи в виде меню, карт сайта или навигационных цепочек. Ссылки и связи между ними могут быть представлены вложенными массивами PHP, XML-документами или INI-файлами.

Страницы и контейнеры

Основной единицей навигации в используемом **Zend_Navigation** подходе является *страница*, которая обычно представлена экземпляром класса **Zend_Navigation_Page**. Каждой страница имеет метку и либо URL, либо комбинацию **модуль/контроллер/действие**; кроме этого, она может содержать другую необязательную информацию, такую как видимость страницы, порядок сортировки, права доступа, а также прямые и обратные связи. Страницы организуются в *навигационные контейнеры*, представляющие собой иерархические коллекции страниц.

Чтобы лучше разобраться в этом, взгляните на следующий PHP-листинг, иллюстрирующий связь между страницами и контейнерами:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  function navAction()
  {
    // инициализируем страницы
    $config = array(
      Zend_Navigation_Page::factory(array(
        'label' => 'Foreword',
        'uri' => '/foreword',
      )),
      Zend_Navigation_Page::factory(array(
        'label' => 'Chapter 1: Introducing the Zend Framework',
        'uri' => '/chapter-01',
        'pages' => array(
```

```
      Zend_Navigation_Page::factory(array(
        'label' => 'Overview',
        'uri' => '/chapter-01/overview',
      )),
      Zend_Navigation_Page::factory(array(
        'label' => 'Features',
        'uri' => '/chapter-01/features',
      ))
    )
      )),
      Zend_Navigation_Page::factory(array(
        'label' => 'Index',
        'uri' => '/index',
      ))
    );

    // инициализируем навигационный контейнер
    $container = new Zend_Navigation($config);
  }
}
```

В приведенном листинге создается пример навигационного контейнера для книги. Сам контейнер представлен объектом **Zend_Navigation**, и ему передается массив объектов **Zend_Navigation_Page**. Каждый объект **Zend_Navigation_Page** верхнего уровня в контейнере представляет собой страницу, находящуюся на верхнем уровне в навигационной иерархии, а каждая страница содержит метку и URL. Страница может содержать дочерние страницы; они также представлены объектами **Zend_Navigation_Page** и добавляются к своему предку в виде вложенных массивов. Иерархическую структуру можно легко расширить для покрытия всего веб-сайта.

Сами объекты страниц создаются в виде экземпляров класса **Zend_Navigation_Page_Mvc** либо **Zend_Navigation_Page_Uri**, каждый из которых расширяет абстрактный класс **Zend_Navigation_Page**. Различие между этими классами заключается в способе определения ссылки на страницу: первый класс использует маршрутизатор Zend Framework, указывая для каждой страницы ее модуль, действие и контроллер, а второй задает ссылку на страницу непосредственно в виде URL. Данное различие показано в следующем примере:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  function pageAction()
  {
    // определяем страницу, используя MVC
    $page = Zend_Navigation_Page::factory(array(
        'label' => 'Contact Us',
        'module' => 'default',
        'controller' => 'contact',
        'action' => 'index',
    ));
```

```
        // определяем страницу, используя URI
        $page = Zend_Navigation_Page::factory(array(
            'label' => 'Contact Us',
            'uri' => '/contact',
        ));
    }
}
```

Каждый объект страницы предоставляет несколько методов получения и установки значений, которые можно использовать для задания свойств страницы. Например, метка, прямые связи, обратные связи и видимость могут быть изменены с помощью методов **setLabel()**, **setRel()**, **setRev()** и **setVisibility()** соответственно и получены с помощью методов **getLabel()**, **getRel()**, **getRev()** и **getVisibility()**. Существует также метод **isActive()**, возвращающий логическое значение, которое показывает, соответствует ли страница текущему запросу.

СОВЕТ

В большинстве случаев предпочтительно определять объекты страниц как экземпляры Zend_Navigation_Page_Uri. Соответствующие URL в некоторой степени проще читать и понимать, к тому же гибкая природа этих объектов позволяет использовать их для создания ссылок на внешние или сторонние ресурсы. Объекты Zend_Navigation_Page_Mvc, напротив, должны анализироваться маршрутизатором Zend Framework, и поэтому могут использоваться только для создания ссылок на внутренние ресурсы приложения.

Как показано в предыдущем примере, контейнер Zend Navigation можно инициализировать массивом объектов **Zend_Navigation_Page**. Однако для длинных или сложных навигационных структур этот способ может быть весьма неудобен, поэтому контейнер **Zend_Navigation** можно также инициализировать объектом **Zend_Config**, в котором параметры навигации описаны либо XML-документом, либо INI-файлом. Рассмотрим следующий пример, демонстрирующий один из таких XML-документов:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <home>
    <label>Home</label>
    <uri>/home</uri>
  </home>

  <products>
    <label>Products</label>
    <uri>/products</uri>
    <pages>
      <men>
        <label>Men</label>
        <uri>/products/men</uri>
        <pages>
          <item_1>
            <label>Dress Shirts</label>
        <uri>/products/men/16339</uri>
```

```
            </item_1>
            <item_2>
        <label>Trousers</label>
        <uri>/products/men/85940</uri>
            </item_2>
            <item_3>
        <label>Shoes</label>
        <uri>/products/men/75393</uri>
        </item_3>
      </pages>
    </men>
    <women>
      <label>Women</label>
      <uri>/products/women</uri>
      <pages>
        <item_1>
            <label>Skirts and Dresses</label>
            <uri>/products/women/75849</uri>
          </item_1>
          <item_2>
            <label>Bags</label>
            <uri>/products/women/64830</uri>
          </item_2>
          <item_3>
            <label>Shoes</label>
            <uri>/products/women/58303</uri>
          </item_3>
        </pages>
      </women>
    </pages>
  </products>

  <about>
    <label>About Us</label>
    <uri>/about</uri>
    <pages>
      <history>
        <label>Company History</label>
        <uri>/about/history</uri>
      </history>
      <team>
        <label>Management Team</label>
        <uri>/about/team</uri>
      </team>
      <awards>
        <label>Awards</label>
        <uri>/about/awards</uri>
      </awards>
    </pages>
  </about>
```

```
  <feedback>
    <label>Feedback</label>
    <uri>/feedback</uri>
  </feedback>
</config>
```

С помощью **Zend_Config** этот XML-документ может быть считан в контейнер **Zend_Navigation** следующим образом:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  function navAction()
  {
    // инициализируем навигационный контейнер содержимым XML-файла
    $config = new Zend_Config_Xml(APPLICATION_PATH . '/configs/site.xml');
    $container = new Zend_Navigation($config);
  }
}
```

СОВЕТ

Zend_Navigation в полной мере учитывает локаль и, при наличии правильно настроенного объекта Zend_Translate, будет автоматически переводить метки страниц на их местные эквиваленты. Более подробно о Zend_Translate вы можете прочитать в главе 9.

Используя **RecursiveIteratorIterator**, можно довольно просто пройти по контейнеру **Zend_Navigation** и получить содержимое отдельных объектов **Zend_Navigation_Page**. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  function navAction()
  {
    // инициализируем навигационный контейнер
    $config = new Zend_Config_Xml(APPLICATION_PATH . '/configs/site.xml');
    $container = new Zend_Navigation($config);

    // проходим по контейнеру
    // и отображаем информацию о страницах
    foreach (new RecursiveIteratorIterator(
      $container, RecursiveIteratorIterator::CHILD_FIRST) as $page) {
        echo $page->getLabel();
    }
  }
}
```

Контейнер **Zend_Navigation** тоже предоставляет различные методы для взаимодействия с хранящимися в нем страницами. В частности, для получения всех страниц, подходящих под указанные критерии, может быть использован метод поиска **findBy()**. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  function navAction()
  {
    // инициализируем навигационный контейнер
    $config = new Zend_Config_Xml(APPLICATION_PATH . '/configs/site.xml');
    $container = new Zend_Navigation($config);

    // находим все страницы с подходящими URL
    $container->findBy('uri', '/about');
  }
}
```

Формирование навигационных элементов

Разумеется, определение навигационной структуры сайта — лишь часть задачи; эту структуру нужно как-то использовать. Обычно контейнер Zend_Navigation инициализируется в начальном загрузчике приложения и помещается в его реестр на постоянное хранение. Ниже приведен пример настройки контейнера:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initNavigation()
  {
    $config = new Zend_Config_Xml(APPLICATION_PATH . '/configs/site.xml');
    $container = new Zend_Navigation($config);
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Navigation', $container);
  }
}
```

После выполнения этих действий вспомогательные классы представлений Zend_Navigation могут автоматически обращаться к зарегистрированному навигационному контейнеру из представлений, преобразуя его в меню, карты сайта или навигационные цепочки, подходящие для использования на веб-странице. В следующих разделах эти аспекты рассмотрены более подробно.

Меню

Вспомогательный класс Zend_Navigation_Menu способен автоматически преобразовывать навигационные данные в иерархическое множество элементов ненумерованного списка. Пример его использования в сценарии представления:

```
<?php echo $this->navigation()->menu(); ?>
```

Этот код выводит ненумерованный список ссылок с несколькими уровнями вложенности (рис. 11.1).

Рис. 11.1. Автоматически сгенерированное меню сайта

Стандартный вывод можно настроить, указав только одну ветвь меню, которую требуется вывести, либо задав глубину выводимого дерева меню. Эти параметры передаются методу **renderMenu()** вспомогательного класса представления в виде ассоциативного массива. Измененная версия предыдущего примера:

```
<?php
echo $this->navigation()
        ->menu()
        ->renderMenu(null, array(
          'minDepth' => null,
          'maxDepth' => 1,
          'ulClass' => 'nav',
          'onlyActiveBranch' => true
  ));
?>
```

Получившийся в результате вывод показан на рис. 11.2.

СОВЕТ

Если вы хотите сформировать меню, используя собственную разметку вместо стандартного ненумерованного списка, вы можете указать для вспомогательного класса представления Menu собственный сценарий представления с помощью метода setPartial().

Рис. 11.2. Автоматически сгенерированное меню сайта с одной отображаемой ветвью

Навигационные цепочки

Еще одним вспомогательным классом для навигации является класс Breadcrumbs, который автоматически создает навигационные цепочки, основываясь на запрашиваемом в данный момент URL. Их внешний вид тоже можно настроить, изменив разделитель и отступы, а также указав способ представления последнего звена цепочки: в виде ссылки или в виде текста. Пример использования вспомогательного класса в сценарии представления:

```
<?php
echo $this->navigation()
        ->breadcrumbs()
        ->setLinkLast(true)
        ->setSeparator(' / ');
?>
```

Вывод показан на рис. 11.3.

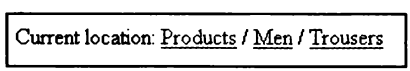


Рис. 11.3. Автоматически сгенерированная навигационная цепочка

Элементы Link

Вспомогательный класс **Links** генерирует набор элементов **<link>**, которые описывают взаимосвязь определенной страницы с другими страницами в наборе документов, а также предоставляют дополнительную информацию для поисковых систем. Эти элементы могут описывать как прямые, так и обратные связи и в HTML-документе всегда располагаются в теге **<head>**.

Пример использования вспомогательного класса в сценарии представления:

```
<?php echo $this->navigation()->links(); ?>
```

Пример вывода показан на рис. 11.4.

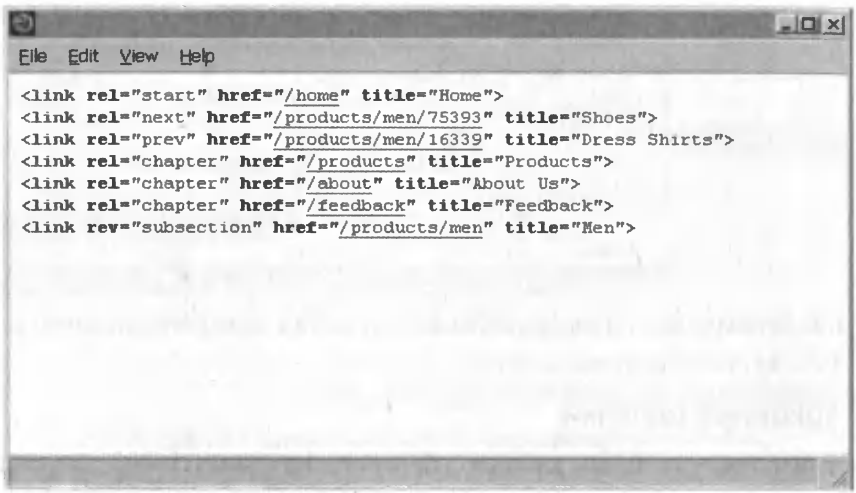


Рис. 11.4. Автоматически сгенерированные элементы Link в заголовке HTML-документа

Карты сайтов

Вспомогательный класс **Sitemap** позволяет автоматически генерировать карту сайта в формате XML на основе данных навигационного контейнера. Карта сайта соответствует протоколу **Sitemap**, принятому в Google, Yahoo! и Microsoft. В конце этой главы приведены ссылки на более подробную информацию об этом протоколе.

Пример использования вспомогательного класса в сценарии представления:

```
<?php
echo $this->navigation()
        ->sitemap()
        ->setFormatOutput(true)
        ->setMaxDepth(1);
?>
```

Пример вывода показан на рис. 11.5.

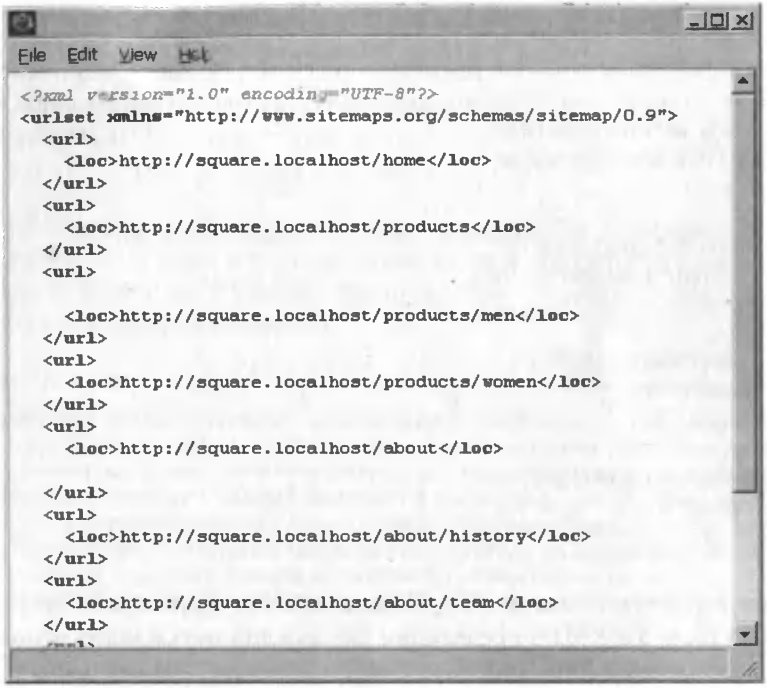


Рис. 11.5. Автоматически сгенерированная карта сайта в формате XML

Упражнение 11.1. Добавление навигационного меню

Теперь, когда вы обладаете некоторыми знаниями о работе компонента Zend_Navigation, давайте используем его в контексте демонстрационного приложения SQUARE. С учетом того, что в приложении используется одноуровневое меню, решение этой задачи не составит труда; тем не менее она хорошо иллюстрирует типовое использование Zend_Navigation на практике.

Определение навигационных страниц и контейнеров

Первым шагом будет определение навигационного контейнера и добавление в него страниц. Как говорилось в предыдущих разделах, самым простым способом сделать это является представление нужной информации в виде XML-файла. Поэтому создайте в текстовом редакторе новый файл и добавьте в него следующие данные:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <home>
```

```
    <label>menu-home</label>
    <uri>/home</uri>
  </home>
  <services>
    <label>menu-services</label>
    <uri>/content/services</uri>
  </services>
  <catalog>
    <label>menu-catalog</label>
    <uri>/catalog/item/search</uri>
  </catalog>
  <news>
    <label>menu-news</label>
    <uri>/news</uri>
  </news>
  <contact>
    <label>menu-contact</label>
    <uri>/contact</uri>
  </contact>
</config>
```

Сохраните файл под именем **$APP_DIR/application/configs/navigation.xml**. Обратите внимание на то, что в XML-определении для каждой метки меню используются строковые идентификаторы; они будут автоматически локализованы в соответствии с выбранным языком благодаря встроенной совместимости **Zend_Navigation** с **Zend_Locale**.

Регистрация объекта навигации

Следующим шагом будет внесение изменений в начальный загрузчик приложения для настройки и регистрации экземпляра объекта **Zend_Navigation**. После этого данный экземпляр будет доступен всем контроллерам и представлениям приложения, позволяя использовать различные вспомогательные классы представления для навигации.

Обновите начальный загрузчик приложения в файле **$APP_DIR/application/Bootstrap.php**, добавив в него следующий метод:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initNavigation()
  {
    // читаем XML с навигационной информацией и инициализируем контейнер
    $config = new Zend_Config_Xml(
        APPLICATION_PATH . '/configs/navigation.xml');
    $container = new Zend_Navigation($config);

    // регистрируем навигационный контейнер
    $registry = Zend_Registry::getInstance();
    $registry->set('Zend_Navigation', $container);
```

```
    // добавляем вспомогательный класс действия
    Zend_Controller_Action_HelperBroker::addHelper(
      new Square_Controller_Action_Helper_Navigation()
    );
  }
}
```

Обратите внимание на то, что последняя строка метода _initNavigation() использует вспомогательный класс действия Navigation. Он вызывается при каждом запросе и отвечает за установку текущей страницы в главном меню. Более подробно об этом рассказано в следующем разделе.

ВНИМАНИЕ

Чтение внешнего файла для загрузки навигационного контейнера при каждом запросе, как это сделано в предыдущем разделе, может значительно снизить производительность высоконагруженных приложений. В таких случаях существует альтернатива, особенно удобная для простых навигационных контейнеров, которая заключается в использовании программного расширения для работы с навигационными ресурсами и определении контейнера непосредственно в конфигурационном файле приложения. Такой подход избавляет от необходимости чтения дополнительного файла и помогает сохранить высокую производительность.

Создание вспомогательного класса представления для навигации

Вспомогательные классы действий в Zend Framework обычно считаются сложными для понимания. Однако на самом деле это утверждение далеко от истины. *Вспомогательные классы действий* предназначены для замены основных контроллеров; они предоставляют разработчикам способ размещения общей функциональности программы во «вспомогательных объектах», которые затем могут быть загружены и использованы во время выполнения в контроллере любого действия.

Вспомогательные классы действий регистрируются с помощью *брокера вспомогательных классов*, который предоставляет методы addHelper() и removeHelper() для моментального добавления вспомогательных классов в список и удаления их оттуда. Это особенно важно для вспомогательных классов, использующихся в методах preDispatch() и postDispatch(). Альтернативным подходом является использование метода addPath() или addPrefix() для определения пути к вспомогательному классу; впоследствии эту информацию можно использовать для загрузки вспомогательного класса по требованию.

После загрузки вспомогательных классов или определения путей к ним они могут быть получены с помощью метода getHelper(), который возвращает объект вспомогательного класса. Брокер вспомогательных классов доступен во всех контроллерах действий через свойство _helper объекта контроллера, что делает возможным загрузку вспомогательных классов по требованию либо в начальном загрузчике приложения, либо когда они понадобятся в определенных контроллерах.

СОВЕТ

Возможно, вы не знаете, но в предыдущих главах вы уже видели и использовали некоторые из встроенных в Zend Framework вспомогательных классов действий. Вспомогательные классы ContextSwitch из главы 6 и FlashMessenger из главы 3 являются примерами встроенных вспомогательных классов действий, а доступ к ним из контроллеров осуществляется с помощью метода getHelper() брокера.

Полный список вспомогательных классов действий, входящих в состав Zend Framework, приведен в табл. 11.1.

Таблица 11.1. Вспомогательные классы действий, входящие в состав Zend Framework

Вспомогательный класс действия	Описание
ActionStack	Ставит несколько действий в очередь для последующего выполнения
ContextSwitch	Позволяет осуществлять вывод в различных форматах
FlashMessenger	Сохраняет сообщения для их получения при следующем запросе
Autocomplete	Форматирует и отправляет массивы данных в форматах JSON/HTML для элементов ввода с автоматическим дополнением
JSON	Позволяет осуществлять вывод в формате JSON
Redirector	Управляет перенаправлением клиента
ViewRenderer	Регистрирует сценарии представления и управляет формированием страниц сценариями представления

Получив всю необходимую информацию, давайте вернемся к текущей задаче: создать пользовательский вспомогательный класс действия, который будет автоматически проверять URL текущего запроса, сопоставлять его с навигационным контейнером и отмечать активную в данный момент страницу. Для этого добавьте в файл $APP_DIR/library/Square/Controller/Action/Helper/Navigation.php следующий код:

```
<?php
// автор кода – Райан Могер (Ryan Mauger), технический редактор
class Square_Controller_Action_Helper_Navigation extends
Zend_Controller_Action_Helper_Abstract
{
  protected $_container;

  // конструктор, устанавливает навигационный контейнер
  public function __construct(Zend_Navigation $container = null)
  {
    if (null !== $container) {
        $this->_container = $container;
    }
}

  // проверяем текущий запрос и устанавливаем активную страницу
  public function preDispatch()
```

```
    {
      $this->getContainer()
          ->findBy('uri', $this->getRequest()->getRequestUri())
          ->active = true;
    }

    // получение навигационного контейнера
    public function getContainer()
    {
      if (null === $this->_container) {
          $this->_container = Zend_Registry::get('Zend_Navigation');
    }

      if (null === $this->_container) {
          throw new RuntimeException ('Navigation container unavailable');
    }

      return $this->_container;
    }
}
```

Этот код определяет новый вспомогательный класс действия, который получает из реестра приложения контейнер **Zend_Navigation** и использует его метод **findBy()** для сравнения URL текущего запроса со ссылками на страницы, хранящиеся в контейнере. Если совпадение найдено, соответствующая страница помечается как активная. Благодаря размещению соответствующего кода в методе **preDispatch()** это действие выполняется при каждом запросе.

Использование вспомогательного класса представления Menu

Для чего вообще нужно определять активную страницу? Эта информация полезна для вспомогательного класса представления **Menu**, который на этапе формирования страницы добавляет к соответствующей ветви главного меню CSS-класс **active**. Это позволяет визуально (и автоматически) выделить элемент меню, который активен в данный момент времени.

Чтобы проверить, как это работает, обновите основной макет, чтобы использовать вспомогательный класс представления **Menu** и динамически генерировать главное меню. Внесите в файл основного макета, **$APP_DIR/application/layouts/master.phtml**, изменения, выделенные жирным шрифтом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
...
        <div id="menu">
          <?php echo $this->navigation()->menu(); ?>
        </div>
...
</html>
```

Обратите внимание, что вам также потребуется обновить таблицу стилей основного макета и добавить в нее правила для динамически сгенерированного меню и его активного элемента. Соответствующие правила вы найдете в архиве с дополнительными материалами к этой главе.

Теперь, если перейти на главную страницу приложения и выбрать некоторый элемент главного меню — скажем, форму обратной связи, расположенную по адресу http://square.localhost/contact, — можно увидеть, что соответствующий элемент меню автоматически подсвечивается. Теперь загляните в код страницы и обратите внимание на то, что вспомогательный класс представления Menu из состава Zend_Navigation автоматически сгенерировал для вас главное меню.

Сформированная страница показана на рис. 11.6.

Рис. 11.6. Автоматически сгенерированное навигационное меню с подсветкой текущего элемента

Работа с Dojo Toolkit

Dojo Toolkit — это кроссбраузерная библиотека на JavaScript, содержащая слой доступа к данным для осуществления AJAX-запросов и получения ответов (Dojo Data), а также набор встроенных компонентов пользовательского интерфейса (Dijit). Эти компоненты включают в себя элементы размещения и меню, таблицы данных, диалоговые окна, элементы управления для форм, такие как поля ввода с автоматическим дополнением, а также элементы для выбора дат и тем и различные анимированные эффекты для веб-страниц. Этот проект поддерживается несколькими крупнейшими интернет-компаниями, такими как Google, IBM, AOL и Sun Microsystems.

Zend Framework упрощает знакомство с Dojo Toolkit. Полная версия Zend Framework содержит комплект компонентов интерфейса Dojo и Dijit, а также компонент Zend_Dojo, облегчающий внедрение и использование Dojo Toolkit в веб-приложении. Zend_Dojo содержит реализацию слоя данных, вспомогательный класс представления, помогающий настраивать окружение Dojo, и набор расширений

форм и макетов, которые можно использовать для добавления в сценарии представления функциональности Dojo/Dijit. В следующих разделах эти вопросы рассматриваются более подробно.

Работа с Dojo Data

Dojo Toolkit определяет единообразный формат для обмена данными между клиентом и сервером и предоставляет два основных хранилища для взаимодействия с этими данными: **dojo.data.ItemFileReadStore** и **dojo.data.ItemFileWriteStore**. В простейшем случае данные представлены в виде структурированного объекта JSON и выглядят следующим образом:

```
{
  "identifier":"name",
  "items": [
    {"name":"Agatha Christie"},
    {"name":"J. K. Rowling"},
    {"name":"Dan Brown"},
    {"name":"William Shakespeare"},
    {"name":"Dennis Lehane"}
  ]
}
```

Компонент **Zend_Dojo_Data** предоставляет объектно-ориентированный API для создания представления в формате JSON, добавления к нему элементов, прохода по нему и осуществления запросов для поиска элементов. Для иллюстрации рассмотрим следующий листинг, создающий вывод, аналогичный приведенному выше:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function dojoAction()
  {
    // определяем элементы данных
    $authors = array(
        'Agatha Christie', 'J. K. Rowling', 'Dan Brown',
        'William Shakespeare', 'Dennis Lehane'
    );
    foreach ($authors as $a) {
        $items[] = array('name' => $a);
    }

    // инициализируем объект и передаем ему элементы данных
        $data = new Zend_Dojo_Data('name', $items);

    // выводим данные в структурированном формате JSON
    echo $data->toJson();
  }
}
```

Как показано в примере, объект инициализируется простой передачей в его конструктор названия поля с уникальным идентификатором (в данном случае 'name') и массива с элементами данных. Эти элементы должны быть представлены в форме ассоциативных массивов (или объектов, реализующих метод **toArray()**) и содержать указанный идентификатор в списке ключей массива. После инициализации объект можно экспортировать либо в виде массива PHP, либо в виде строки JSON, используя метод **toArray()** или **toJson()**.

СОВЕТ

Компонент Zend_Dojo_Data используется для реализации конечных точек AJAX-запросов Dojo/Dijit. Пример вы увидите далее в этой главе.

Использование вспомогательных классов представлений Dojo

Как правило, для добавления к веб-странице компонентов Dojo и Dijit требуется загрузить файлы библиотеки Dojo, инициализировать анализатор Dojo и включить требуемые компоненты, при необходимости передав им конфигурационные параметры. Несмотря на то что это можно сделать и вручную в сценариях представления, Zend Framework предоставляет более простой способ: вспомогательный класс представления **Dojo**, который может автоматически сгенерировать в представлении весь необходимый клиентский код инициализации.

Ниже приведен пример использования вспомогательного класса **Dojo** в сценарии представления:

```
<?php
Zend_Dojo::enableView($this);
$this->dojo()->setLocalPath('/js/dojo/dojo.js')
        ->addStyleSheetModule('dijit.themes.tundra')
        ->setDjConfigOption('parseOnLoad', true)
        ->setDjConfigOption('locale', 'en-US')
        ->setDjConfigOption('isDebug', true);
echo $this->dojo();
?>
```

На рис. 11.7 показана разметка, сгенерированная вспомогательным классом из предыдущего листинга.

ВНИМАНИЕ

Обратите внимание на вызов Zend_Dojo::enableView. Этот метод добавляет в объект Zend_View путь к вспомогательному классу представления Dojo; отсутствие данного вызова обычно приводит к исключению, так как в этом случае Zend_View не сможет найти и загрузить необходимый(е) вспомогательный(е) класс(ы).

Если вы предпочитаете загружать библиотеки Dojo Toolkit из внешнего источника, например из сети доставки содержимого (CDN, Content Delivery Network), используйте метод **setCdnBase()**, как показано в следующем примере:

```
<?php
Zend_Dojo::enableView($this);
$this->dojo()->setCdnBase(Zend_Dojo::CDN_BASE_AOL)
        ->addStyleSheetModule('dijit.themes.tundra')
        ->setDjConfigOption('parseOnLoad', true)
        ->setDjConfigOption('locale', 'en-US')
        ->setDjConfigOption('isDebug', true);
echo $this->dojo();
?>
```

```
<style type="text/css">
<!--
    @import "/js/dijit/themes/tundra/tundra.css";
-->
</style>
<script type="text/javascript">
//<![CDATA[
    var djConfig = {"parseOnLoad":true,"locale":"en-US","isDebug":true};
//]]>
</script>
<script type="text/javascript" src="/js/dojo/dojo.js"></script>
```

Рис. 11.7. Разметка, автоматически сгенерированная вспомогательным классом представления Dojo

ВНИМАНИЕ

Если вы не укажете локальный путь к библиотекам Dojo с помощью setLocalPath(), вспомогательный класс представления Dojo будет по умолчанию считать, что вы хотите использовать Google CDN.

В дополнение к вспомогательному классу представления Dojo Zend Framework также содержит несколько других вспомогательных классов представления, которые можно использовать для непосредственного формирования контейнеров размещения Dojo и Dijit и элементов формы внутри сценария представления. Краткий список этих вспомогательных классов приведен в табл. 11.2.

Стоит отметить, что вспомогательный класс представления Dojo автоматически обнаруживает использование этих вспомогательных классов в сценарии представления и генерирует клиентский код, требующийся для их инициализации. Пример:

```
<html>
  <head>
  <?php
  Zend_Dojo::enableView($this);
    $element = $this->NumberSpinner('temp', 25, array(
        'min' => 12,
        'max' => 41,
        'places' => 2
    )
  );
```

```
    $this->dojo()->setCdnBase(Zend_Dojo::CDN_BASE_AOL)
        ->addStyleSheetModule('dijit.themes.tundra');
    echo $this->dojo();
  ?>
  </head>
  <body class="tundra">
    <?php echo $element; ?>
  </body>
</html>
```

В этом примере с помощью вспомогательного класса представления NumberSpinner генерируется элемент формы Dijit, представляющий собой изменяемое числовое поле. На рис. 11.8 показана итоговая разметка, сгенерированная вспомогательным классом представления Dojo.

Таблица 11.2. Вспомогательные классы представления Dojo, входящие в состав Zend Framework (неполный список)

Вспомогательный класс Dojo	Описание
Button	Кнопка
CheckBox	Флажок
ComboBox	Комбинированный элемент выбора/ввода текста
ContentPane	Контейнер для содержимого
CurrencyTextBox	Поле для ввода валюты
DateTextBox	Элемент выбора даты
Editor	Редактор Rich Text
FilteringSelect	Элемент для выбора с поддержкой фильтрации
HorizontalSlider	Горизонтальный ползунок
NumberSpinner	Числовое поле с кнопками для изменения значения
StackContainer	Контейнер для размещения элементов стопкой
TabContainer	Контейнер для размещения элементов во вкладках
Textarea	Поле для ввода текста
TextBox	Поле для ввода текста
TimeTextBox	Элемент выбора времени
ValidationTextBox	Поле для ввода текста с валидатором
VerticalSlider	Вертикальный ползунок

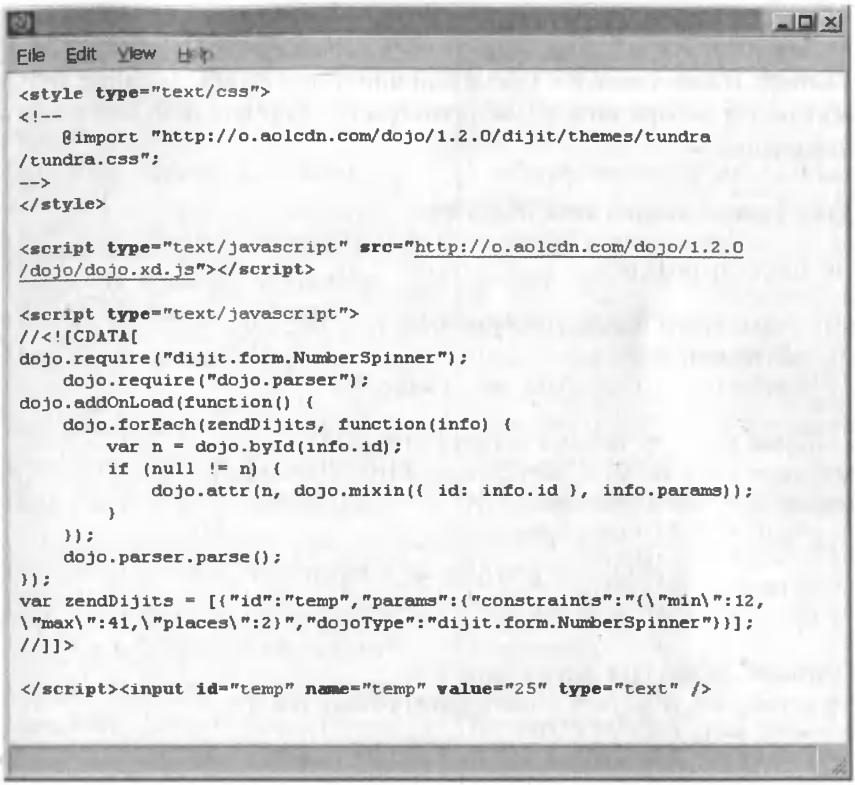


Рис. 11.8. Разметка, автоматически сгенерированная вспомогательным классом представления Dojo

Использование элементов формы Dojo

Dojo Toolkit содержит готовые элементы формы, которые можно использовать для быстрого добавления в веб-формы выбора цвета, текстового редактора WYSIWYG (What You See Is What You Get, «что видишь, то и получаешь»), изменяемого числового поля, поля для ввода текста с автоматическим дополнением, горизонтального ползунка и так далее. Zend_Dojo дополняет Zend_Form поддержкой этих дополнительных элементов, позволяя использовать их точно так же, как и экземпляры обычных классов Zend_Form_Element.

В табл. 11.3 приведен список элементов формы из состава Dojo, поставляющихся вместе с Zend Framework.

Чтобы использовать эти элементы в форме, измените ее базовый класс с Zend_Form на Zend_Dojo_Form, а затем, как обычно, добавьте к ней элементы Zend_Dojo_Form_Element. Ниже приведен пример, демонстрирующий создание веб-формы с элементом для выбора даты, редактором текста в формате Rich Text и вертикальным ползунком:

```
<?php
class Form_Example extends Zend_Dojo_Form
{
  public function init()
  {
    $this->setAction('/sandbox/example/form')
        ->setMethod('post')
        ->setOptions(array('class' => 'tundra'));

    // создаем редактор текста в формате Rich Text
    $message = new Zend_Dojo_Form_Element_Editor('message');
    $message->setLabel('Message:')
        ->setOptions(array(
          'width' => '150px',
          'height' => '100px',
        ));

    // создаем элемент для выбора даты
    $dob = new Zend_Dojo_Form_Element_DateTextBox('dob');
    $dob->setLabel('Date of birth:');

    // создаем ползунок
    $volume = new Zend_Dojo_Form_Element_VerticalSlider('volume');
    $volume->setLabel('Volume level:')
        ->setOptions(array(
          'minimum' => '100',
          'maximum' => '0',
          'discreteValues' => '10',
          'style' => 'height: 100px'
        ));

    // создаем кнопку отправки
    $submit = new Zend_Dojo_Form_Element_SubmitButton('submit');
    $submit->setLabel('Submit');
    $this->addElement($message)
        ->addElement($dob)
        ->addElement($volume)
        ->addElement($submit);
  }
}
```

На рис. 11.9 показан пример получившейся формы.

Таблица 11.3. Классы элементов Dojo Form, входящие в состав Zend Framework

Класс элемента формы	Описание
Zend_Dojo_Form_Element_Button	Кнопка
Zend_Dojo_Form_Element_CheckBox	Переключатель
Zend_Dojo_Form_Element_ComboBox	Комбинированный элемент для выбора/ввода
Zend_Dojo_Form_Element_CurrencyTextBox	Элемент для ввода валюты
Zend_Dojo_Form_Element_DateTextBox	Элемент для ввода даты
Zend_Dojo_Form_Element_DijitMulti	Группа переключателей
Zend_Dojo_Form_Element_Editor	Поле для ввода текста в формате Rich Text
Zend_Dojo_Form_Element_FilteringSelect	Элемент для выбора с фильтрацией
Zend_Dojo_Form_Element_HorizontalSlider	Горизонтальный ползунок
Zend_Dojo_Form_Element_NumberSpinner	Числовое поле с кнопками для изменения значения
Zend_Dojo_Form_Element_NumberTextBox	Поле для ввода числа
Zend_Dojo_Form_Element_PasswordTextBox	Поле для ввода пароля
Zend_Dojo_Form_Element_RadioButton	Радиокнопка
Zend_Dojo_Form_Element_SimpleTextarea	Область для ввода текста
Zend_Dojo_Form_Element_SubmitButton	Кнопка отправки
Zend_Dojo_Form_Element_Textarea	Поле для ввода текста
Zend_Dojo_Form_Element_TextBox	Поле для ввода текста
Zend_Dojo_Form_Element_TimeTextBox	Поле для ввода времени
Zend_Dojo_Form_Element_ValidationTextBox	Поле для ввода текста с валидатором
Zend_Dojo_Form_Element_VerticalSlider	Вертикальный ползунок

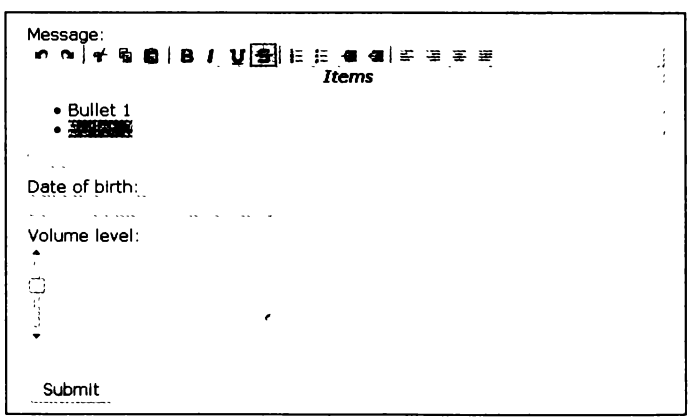


Рис. 11.9. Форма с элементами Dojo/Dijit

Упражнение 11.2. Добавление элемента Dojo с автоматическим дополнением

Теперь вы знаете о возможностях компонента Zend_Dojo, и можно попробовать воспользоваться им в практических целях — для добавления к экземпляру Zend_Form поля ввода с автоматическим дополнением. Это требование встречается довольно часто, и в следующих разделах будет показан процесс его реализации.

Обновление формы обратной связи

Для примера предположим, что поле ввода с автоматическим дополнением будет добавляться к форме обратной связи приложения и «предлагать» список стран, основываясь на введенном пользователем значении. Поэтому, как было сказано в предыдущем разделе, для начала следует обновить объект Square_Form_Contact, чтобы он расширял класс Zend_Dojo_Form вместо класса Zend_Form, а также добавить к нему элемент Dijit ComboBox. Пример кода:

```
<?php
class Square_Form_Contact extends Zend_Dojo_Form
{
  public function init()
  {
    // инициализируем форму
    $this->setAction('/contact')
        ->setMethod('post');

    // создаем текстовое поле для ввода имени

    // создаем текстовое поле для ввода адреса электронной почты

    // создаем поле с автоматическим дополнением для ввода названия страны
    $country = new Zend_Dojo_Form_Element_ComboBox('country');
    $country->setLabel('contact-country');
    $country->setOptions(array(
          'autocomplete' => false,
          'storeId' => 'countryStore',
          'storeType' => 'dojo.data.ItemFileReadStore',
          'storeParams' => array('url' => "/default/contact/autocomplete"),
          'dijitParams' => array('searchAttr' => 'name')))
        ->setRequired(true)
        ->addValidator('NotEmpty', true)
        ->addFilter('HTMLEntities')
        ->addFilter('StringToLower')
        ->addFilter('StringTrim');

    // создаем текстовое поле для ввода текста сообщения
```

```
        // создаем поле CAPTCHA

        // создаем кнопку отправки

        // добавляем элементы к форме
        $this->addElement($name)
             ->addElement($email)
            ->addElement($country)
             ->addElement($message)
             ->addElement($captcha)
             ->addElement($submit);
    }
}
```

Это обновленное определение добавляет к форме обратной связи новый элемент Dijit **ComboBox** и указывает на использование удаленного хранилища данных Dojo для вариантов автоматического дополнения. Также указывается конечная точка для хранилища — **/default/contact/autocomplete**. Сохраните данное определение объекта в файл **$APP_DIR/library/Square/Form/Contact.php** и продолжайте чтение, чтобы узнать, как определяется эта конечная точка.

Инициализация вспомогательного класса представления Dojo

Следующим шагом будет инициализация вспомогательного класса представления **Dojo**. Если в вашем приложении не используются макеты, ее можно выполнять непосредственно в соответствующем сценарии представления. Но так как демонстрационное приложение SQUARE использует их, инициализацию необходимо выполнять в макете.

И здесь возникает интересная ситуация. Как правило, важно уменьшать общее число удаленных HTTP-запросов, генерируемых веб-страницей, поскольку каждый запрос добавляет накладные расходы и увеличивает время ожидания пользователем полной загрузки страницы. В связи с этим очевидно, что проводить инициализацию вспомогательного класса представления **Dojo** в сценарии макета не очень эффективно, так как в результате библиотеки Dojo будут запрашиваться и загружаться даже в тех случаях, когда они не нужны.

Выходом из подобной ситуации является настройка вспомогательного класса **Dojo** в начальном загрузчике приложения и явное отключение его, с тем чтобы он не загружался по умолчанию в каждом сценарии представления. Впоследствии его можно будет динамически включать по мере необходимости в тех сценариях, где это требуется.

Чтобы применить такой подход, сначала измените конфигурационный файл приложения, добавив в него следующую строчку, включающую программное расширение для управления ресурсами представления:

Затем добавьте следующий метод в начальный загрузчик приложения, расположенный в файле **$APP_DIR/application/Bootstrap.php**:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initDojo()
  {
    // получаем ресурс представления
    $this->bootstrap('view');
    $view = $this->getResource('view');

    // добавляем в представление путь к вспомогательному классу
    Zend_Dojo::enableView($view);

    // настраиваем вспомогательный класс представления Dojo и отключаем его
    $view->dojo()->setCdnBase(Zend_Dojo::CDN_BASE_AOL)
        ->addStyleSheetModule('dijit.themes.tundra')
        ->disable();
  }
}
```

Этот метод получает экземпляр **Zend_View** из программного расширения для управления ресурсами представления, а затем использует метод **Zend_Dojo::enableView**, чтобы задействовать в представлении вспомогательный класс **Dojo**. После этого он настраивает окружение **Dojo**, устанавливая URL CDN и таблицу стилей, и, наконец, явно отключает вспомогательный класс, используя его метод **disable()**. В результате всех перечисленных действий вспомогательный класс представления **Dojo** будет настроен для использования, однако библиотеки Dojo Toolkit не будут загружаться в сценариях представления, пока это не будет запрошено явно вызовом метода **enable()**.

Обновление основного макета

Следующим шагом будет обновление основного макета для проверки включения вспомогательного класса представления **Dojo** и последующей загрузки окружения Dojo. Как видно из приведенных ниже дополнений к содержимому тега **<head>** в макете **$APP_DIR/application/layouts/master.phtml**, сделать это несложно:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <base href="/" />
    <link rel="stylesheet" type="text/css" href="/css/master.css" />
    <?php if ($this->dojo()->isEnabled()):
          echo $this->dojo();
        endif;
```

```
    ?>
    </head>
    <body class="tundra">

  </body>
</html>
```

Обновление контроллера

Последним шагом будет обновление **ContactController** и определение конечной точки для поля ввода с автоматическим дополнением. Ниже приведен код, который следует добавить в файл **$APP_DIR/application/modules/default/controllers/ContactController.php**:

```
<?php
class ContactController extends Zend_Controller_Action
{
  public function autocompleteAction()
  {
    // отключаем макет и формирование страницы
    $this->_helper->layout->disableLayout();
    $this->getHelper('viewRenderer')->setNoRender(true);

    // получаем список стран, воспользовавшись Zend_Locale
    $territories = Zend_Locale::getTranslationList('territory', null, 2);
    $items = array();
    foreach ($territories as $t) {
      $items[] = array('name' => $t);
    }

    // генерируем строку в формате JSON, совместимую
    // со структурой dojo.data, и возвращаем ее
    $data = new Zend_Dojo_Data('name', $items);
    header('Content-Type: application/json');
    echo $data->toJson();
  }
}
```

С помощью рассмотренного ранее компонента **Zend_Dojo_Data** этот метод преобразует массив, содержащий список стран, в формат JSON, ожидаемый Dojo, а затем возвращает список запрашивающему клиенту.

Чтобы увидеть, как это работает, перейдите к форме обратной связи приложения SQUARE по адресу **http://square.localhost/contact**, и вы увидите новое текстовое поле для ввода названия страны. Введите в него несколько символов, чтобы посмотреть, как Dojo запрашивает и выводит список соответствующих вариантов, предоставленный действием **ContactController::autocompleteAction**, определенным в предыдущем разделе (рис. 11.10).

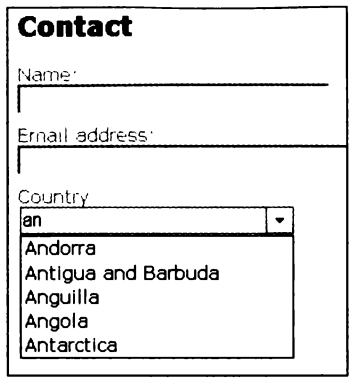


Рис. 11.10. Использование Dojo ComboBox

Упражнение 11.3. Добавление календаря YUI

В предыдущем разделе вы видели, насколько легко добавлять в приложение Zend Framework элементы управления Dojo. Однако Dojo не единственный вариант, и вы можете интегрировать элементы управления и библиотеки из других инструментариев программирования на стороне клиента. В следующем разделе описан процесс добавления к экземпляру Zend_Form всплывающего календаря из библиотеки Yahoo! User Interface Library (YUI, библиотека пользовательского интерфейса Yahoo!).

ВОПРОС ЭКСПЕРТУ

В: Что такое Yahoo! User Interface Library?

О: Как сказано на официальном веб-сайте, Yahoo! User Interface Library, также известная как YUI, — это «набор средств и элементов управления, написанных с использованием JavaScript и CSS и предназначенных для создания полностью интерактивных веб-приложений». Библиотека работает во всех современных браузерах и предоставляет огромное количество готовых элементов управления, которые разработчики могут с минимальными усилиями интегрировать в веб-приложения. В числе прочих к этим элементам управления относятся карусель изображений, элемент для выбора цвета, интерфейс drag-and-drop, элемент слежения за процессом загрузки файла на сервер, система меню, представленных в виде вкладок или дерева, календарь и механизм автоматического дополнения для полей ввода.

Среди этих элементов присутствует элемент календаря, который отображает календарь на месяц и предоставляет пользователю элементы управления для смены месяца и года. Пользователь может выбрать определенную дату, щелкнув на ней мышкой; при этом элемент автоматически преобразует выбранную дату в стандартную строку и добавит ее в указанный элемент формы.

Теперь модификации подвергнется форма ItemUpdate, определенная в главе 5 и позволяющая администраторам приложения изменять свойства элементов каталога и устанавливать статус их отображения. В этом примере мы изменим форму,

чтобы использовать для ввода даты элемент графического календаря YUI, а не отображать отдельные элементы для указания дня, месяца и года.

В примере предполагается, что все необходимые файлы YUI находятся на удаленном сервере и загружаются из сети доставки содержимого Yahoo!.

Обновление формы

Для начала обновим объект формы, создав новый элемент, который будет содержать календарь, и удалив предыдущие списки выбора даты. Ниже приведено исправленное определение формы, которое следует сохранить в файл **$APP_DIR/library/Square/Form/ItemUpdate.php**:

```
<?php
class Square_Form_ItemUpdate extends Square_Form_ItemCreate
{
  public function init()
  {
    // получаем родительскую форму
    parent::init();

    // устанавливаем действие для формы (укажите значение false для использования
    текущего URL)
    $this->setAction('/admin/catalog/item/update');

    // удаляем нежелательные элементы
    $this->removeElement('Captcha');
    $this->removeDisplayGroup('verification');
    $this->removeElement('images');
    $this->removeDisplayGroup('files');

    // создаем скрытое поле для идентификатора элемента
    $id = new Zend_Form_Element_Hidden('RecordID');
    $id->addValidator('Int')
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');

    // создаем список выбора для статуса отображения элемента
    $display = new Zend_Form_Element_Select('DisplayStatus',
      array('onChange' =>
       "javascript:handleInputDisplayOnSelect('DisplayStatus',
         'divDisplayUntil', new Array('1')); cal.hide();"));
    $display->setLabel('Display status:')
        ->setRequired(true)
        ->addValidator('Int')
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim');
    $display->addMultiOptions(array(
      0 => 'Hidden',
      1 => 'Visible'
    ));
```

```
    // создаем текстовое поле для ввода даты отображения элемента
    $displayUntil = new Zend_Form_Element_Text('DisplayUntil');
    $displayUntil->setLabel('Display until (yyyy-mm-dd):')
        ->addValidator('Date', false, array('format' =>
'yyyy-MM-dd'))
        ->addFilter('HtmlEntities')
        ->addFilter('StringTrim')
         ->addDecorators(array(
                array('HTMLTag', array('tag' => 'div', 'id' =>
    'divDisplayUntil')),
    ));

    // создаем контейнер для календаря YUI
    $calendar = new Zend_Form_Element_Text('Calendar');
    $calendar->setDecorators(array(
                array('Label', array('tag' => 'dt')),
                array('HTMLTag', array('tag' => 'div', 'id' =>
          'divCalendar', 'class' => 'yui-skin-sam yui-calcontainer', 'style' =>
          'display:none;')),
    ));

    // добавляем элементы к форме
    $this->addElement($id)
        ->addElement($display)
        ->addElement($calendar)
        ->addElement($displayUntil);

    // создаем группу отображения для статуса
    $this->addDisplayGroup(
                array('DisplayStatus', 'DisplayUntil', 'Calendar'),
          'display');
    $this->getDisplayGroup('display')
        ->setOrder(25)
        ->setLegend('Display Information');
      }
    }
?>
```

Обновление основного макета

Вспомогательные классы Zend Framework **HeadScript** и **HeadLink** обеспечивают простой способ указания внешних ресурсов для каждого действия. Они позволяют разработчикам указывать файлы JavaScript и CSS для каждого сценария действия и представления на этапе разработки, а во время выполнения автоматически генерируют необходимую разметку в виде тегов <link> и <script>.

Чтобы использовать эти вспомогательные классы, внесите в административный макет в файле **$APP_DIR/application/layouts/admin.phtml** изменения, выделенные жирным шрифтом:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <base href="/" />
    <link rel="stylesheet" type="text/css" href="/css/master.css" />
    <link rel="stylesheet" type="text/css" href="/css/admin.css" />
    <?php echo $this->headLink(); ?>
    <?php echo $this->headScript(); ?>
    </head>
    <body>
...
</body>
</html>
```

Обновление контроллера

Теперь необходимо внести два изменения в действие Catalog_AdminItemController::updateAction:

- используя вспомогательные классы представления HeadLink и HeadScript, указать источник файлов JavaScript и CSS для календаря YUI;
- модифицировать код для использования строки с датой, предоставляемой элементом календаря, вместо трех отдельных значений (месяц, дата и год), предоставляемых текущими списками выбора.

Измененный код:

```
<?php
class Catalog_AdminItemController extends Zend_Controller_Action
{
  // действие для изменения отдельного элемента каталога
  public function updateAction()
  {
    // загружаем файлы JavaScript и CSS
    $this->view->headLink()->appendStylesheet(
      'http://yui.yahooapis.com/combo?2.8.0r4/build/calendar/
      assets/skins/sam/calendar.css'
    );
    $this->view->headScript()->appendFile('/js/form.js');
    $this->view->headScript()->appendFile(
      'http://yui.yahooapis.com/combo?2.8.0r4/build/yahoo-dom-event/
      yahoo-dom-event.js&2.8.0r4/build/calendar/calendar-min.js'
    );

    // генерируем форму ввода
    $form = new Square_Form_ItemUpdate;
    $this->view->form = $form;
```

```
if ($this->getRequest()->isPost()) {
  // если получен запрос POST,
  // проверяем корректность входных данных,
  // получаем текущую запись,
  // обновляем значения и заменяем хранящиеся в базе данных
  $postData = $this->getRequest()->getPost();

  // закомментируем код настройки даты
  //$postData['DisplayUntil'] = sprintf('%04d-%02d-%02d',
  // $this->getRequest()->getPost('DisplayUntil_year'),
  // $this->getRequest()->getPost('DisplayUntil_month'),
  // $this->getRequest()->getPost('DisplayUntil_day')
  );

  if ($form->isValid($postData)) {
    $input = $form->getValues();
    $item = Doctrine::getTable('Square_Model_Item')
            ->find($input['RecordID']);
    $item->fromArray($input);
    $item->DisplayUntil =
      ($item->DisplayStatus == 0) ? null : $item->DisplayUntil;
    $item->save();
    $this->_helper->getHelper('FlashMessenger')
      ->addMessage('The record was successfully updated.');
    $this->_redirect('/admin/catalog/item/success');
  }
} else {
  // если получен запрос GET,
  // устанавливаем фильтры и валидаторы для входных данных из этого запроса,
  // проверяем корректность последних,
  // получаем запрошенную запись,
  // заполняем форму
  $filters = array(
  'id' => array('HtmlEntities', 'StripTags', 'StringTrim')
  );
  $validators = array(
    'id' => array('NotEmpty', 'Int')
  );
  $input = new Zend_Filter_Input($filters, $validators);
  $input->setData($this->getRequest()->getParams());
  if ($input->isValid()) {
    $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Type t')
        ->where('i.RecordID = ?', $input->id);
  $result = $q->fetchArray();
  if (count($result) == 1) {
```

```
        // закомментируем настройку даты
        //$date = $result[0]['DisplayUntil'];
        //$result[0]['DisplayUntil_day'] = date('d', strtotime($date));
        //$result[0]['DisplayUntil_month'] = date('m', strtotime($date));
        //$result[0]['DisplayUntil_year'] = date('Y', strtotime($date));
        $this->view->form->populate($result[0]);
      } else {
        throw new Zend_Controller_Action_Exception('Page not found', 404);
}
      } else {
        throw new Zend_Controller_Action_Exception('Invalid input');
      }
    }
  }
   }
```

Обновление представления

И наконец, для инициализации элемента управления в сценарии представления, расположенном в файле **$APP_DIR/application/modules/admin/views/scripts/admin-item/update.phtml**, требуется добавить немного кода на JavaScript:

```
<h2>Update Item</h2>
<?php echo $this->form; ?>

<script type="text/javascript">
handleInputDisplayOnSelect('DisplayStatus', 'divDisplayUntil', new Array('1'));

cal = new YAHOO.widget.Calendar('cal', 'divCalendar', {close:true});
cal.render();
YAHOO.util.Event.addListener('DisplayUntil', 'click', cal.show, cal, true);
cal.selectEvent.subscribe(handleSelect, cal, true);

function handleSelect(type,args,obj) {
    var dates = args[0];
    var date = dates[0];
    var year = date[0];
    var month = date[1];
    var day = date[2];
    var input = document.getElementById('DisplayUntil');
    input.value = year + '-' + padZero(month, 2) + '-' + padZero(day, 2);
}
</script>
```

Сохраните изменения, перейдите в панель администрирования и выберите элемент каталога для обновления. Вы увидите форму с полями для изменения различных свойств элемента. Выберите поле **Display until**, и появится всплывающий календарь для выбора даты (рис. 11.11).

Рис. 11.11. Использование календаря YUI

ВОПРОС ЭКСПЕРТУ

В: Почему бы просто не использовать в основном макете «жесткие» ссылки на файлы JavaScript и CSS для YUI вместо вспомогательных методов представления headScript() и headLink()?

О: Файлы YUI нужны только в некоторых сценариях представления. Так как они загружаются с удаленного сервера по HTTP, не стоит создавать «жесткие» ссылки на них в основном макете, поскольку в этом случае файлы будут запрашиваться при загрузке каждой страницы. В результате появятся лишние накладные расходы и увеличится время, в течение которого пользователь будет ждать полной загрузки страницы. Использование вспомогательных классов представления делает возможным загрузку этих файлов только при необходимости, что оптимизирует время загрузки и повышает производительность.

Выводы

Эта глава была посвящена элементам пользовательского интерфейса и компонентам Zend Framework, которые позволяют пользователям получить лучшее впечатление от работы с приложением. В начале главы был рассмотрен компонент Zend_Navigation и показано, как организовывать страницы в контейнеры, а затем генерировать на их основе карты сайта, меню и цепочки навигации. Затем была рассмотрена интеграция Dojo с Zend Framework и показано, как можно использовать компонент Zend_Dojo для быстрого добавления к экземпляру Zend_Form поля ввода с автоматическим дополнением. И наконец, было уделено внимание процессу использования в приложении Zend Framework сторонних элементов управления, написанных на JavaScript, а в качестве примера рассматривалось добавление к экземпляру Zend_Form элемента для выбора даты из библиотеки YUI.

Чтобы получить больше информации по рассмотренным в данной главе темам, посетите следующие ссылки:

- Компонент Zend_Navigation: http://framework.zend.com/manual/en/zend.navigation.html.
- Компонент Zend_Dojo: http://framework.zend.com/manual/en/zend.dojo.html.
- Вспомогательные классы действий Zend Framework: http://framework.zend.com/manual/en/zend.controller.actionhelpers.html.
- Программные расширения для управления ресурсами в Zend Framework: http://framework.zend.com/manual/en/zend.application.available-resources.html.
- Библиотека Yahoo! User Interface Library: http://developer.yahoo.com/yui/.
- Инструментарий Dojo Toolkit: http://www.dojotoolkit.org/.
- Протокол Sitemaps: http://www.sitemaps.org/.
- Создание пользовательских программных расширений для управления ресурсами (Стефан Шмалхаус (Stefan Schmalhaus)): http://blog.log2e.com/2009/06/01/creating-a-custom-resource-plugin-in-zendframework-18/.

12 Оптимизация производительности

Прочитав эту главу, вы:

- познакомитесь со средствами оценки производительности и профилирования приложения;
- изучите преимущества и типы кэширования;
- узнаете, как кэшировать ответы веб-сервисов и ленты RSS;
- изучите различные техники оптимизации производительности запросов к базе данных.

В последние несколько лет веб-приложения постоянно усложнялись, обрастая множеством украшательств — пользовательскими интерфейсами, использующими AJAX, потоковым мультимедиа, динамически генерируемым содержимым — в попытках привлечь и удержать посетителей. Но от усложнения приложений, увеличения их зависимости от источников динамических данных и роста количества запросов, которые требуется обслуживать за секунду, в первую очередь страдает производительность.

Однако существуют несколько распространенных приемов, которые можно использовать для определения и устранения узких мест в коде приложения. Такие приемы, как профилирование кода, кэширование, рефакторинг и отложенная загрузка, могут уменьшить нагрузку на сервер и время отклика приложений. Zend Framework содержит несколько инструментов для решения этой задачи, которые будут рассмотрены в данной главе.

Оценка производительности

Перед тем как приступить к тонкой настройке веб-приложения, соберите детальную информацию о том, какие из его частей «страдают» от низкой производительности. Существуют несколько средств для оценки производительности и профилирования, и в следующих разделах мы рассмотрим их подробнее.

Оценка производительности

ApacheBench, или *ab*, — это средство для оценки времени отклика веб-сервера. Она производится путем одновременной отправки серверу нескольких запросов определенного URL, измерения времени, прошедшего до получения ответа, и генерации подробной статистики значений времени при различных условиях загруженности. ApacheBench входит в состав дистрибутива веб-сервера Apache и доступен как для платформы Windows, так и для платформ *NIX. Несмотря на то что ApacheBench является частью дистрибутива Apache, его можно использовать для оценки производительности любого другого веб-сервера, совместимого с протоколом HTTP.

Ниже приведен пример использования ApacheBench для оценки производительности веб-приложения, симулирующего поступление 1000 запросов одного и того же ресурса, отправляемых по десять за один раз:

```
shell> ab -n 1000 -c 10 http://server/request/url
```

Пример отчета, сгенерированного ApacheBench, представлен на рис. 12.1.

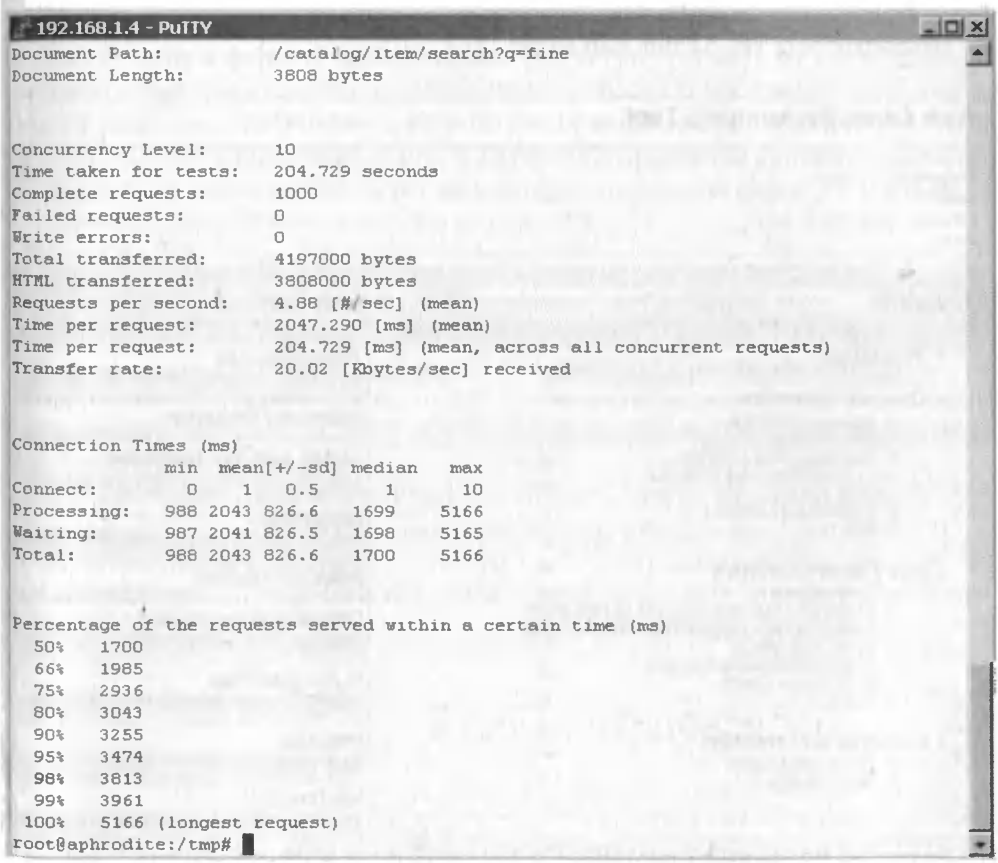


Рис. 12.1. Сводная статистика ApacheBench

Как видно из рис. 12.1, этот отчет предоставляет полезную информацию о среднем количестве секунд, необходимом серверу для ответа на запрос определенного URL. Запуск такого же теста для других маршрутов приложения создает точную картину того, какие из его частей работают «медленно» и могут быть улучшены с помощью оптимизации.

Альтернативой ApacheBench является Microsoft Web Capacity Analysis Tool (WCAT, инструмент анализа производительности веб-приложений). Как и Apache Bench, эта программа симулирует нагрузку на веб-сервер, возвращая результаты измерения пропускной способности и времени отклика. Хотя WCAT доступна только для платформы Windows, она обладает парой интересных возможностей: ее можно одновременно использовать с нескольких клиентских компьютеров и она способна оценивать производительность «транзакций», состоящих из нескольких запросов, выполняемых в определенном порядке.

Пример использования WCAT для оценки производительности веб-приложения:

```
shell> wcat.wsf -terminate -run -clients localhost -t scenario.ubr -f
settings.ubr -s server -singleip -extended
```

Пример отчета WCAT показан на рис. 12.2.

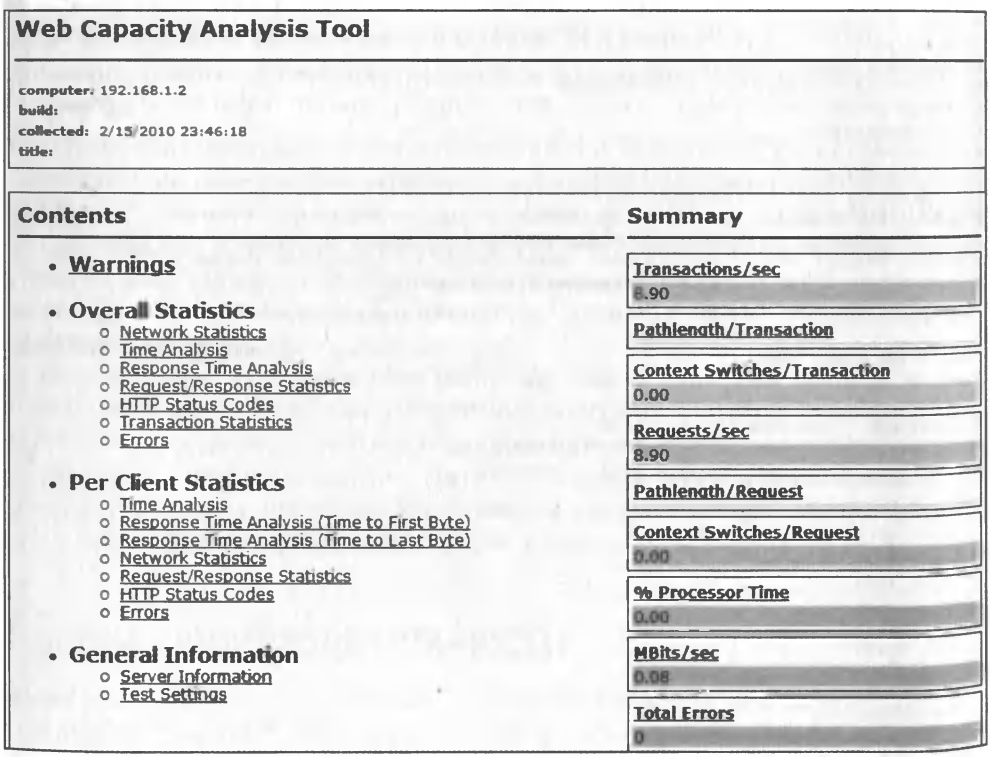


Рис. 12.2. Отчет о производительности, сгенерированный WCAT

Профилирование кода

После получения основных показателей производительности и определения областей приложения, которые требуют улучшения, следует произвести профилирование соответствующих контроллеров и действий и понять, где именно находятся узкие места. Одним из наиболее популярных средств для решения этой задачи является расширение PHP Xdebug, доступное для установок PHP как в Windows, так и в *NIX.

Профайлер Xdebug генерирует подробную статистику по количеству времени, затраченного на вызов каждой функции в сценарии, а также общего времени компиляции, обработки и выполнения сценария. С помощью этой статистики можно увидеть, вызовы каких функций приводят к максимальным накладным расходам, и, таким образом, определить области для возможной оптимизации. Кроме этого, расширение Xdebug также переопределяет стандартные процедуры обработки исключений в PHP, предоставляя более подробный отладочный вывод и трассировку стека как для критических, так и для некритических ошибок.

После настройки Xdebug будет автоматически профилировать каждый сценарий, выполняемый интерпретатором PHP. Отчеты о профилировании создаются в виде файлов в формате **cachegrind**, которые можно посмотреть с помощью нескольких средств с открытым исходным кодом, таких как WinCacheGrind (локальное приложение для Windows), KCacheGrind (локальное приложение для *NIX) и Webgrind (веб-приложение на PHP). Профили сохраняются в каталог, указанный в переменной **xdebug.profiler_output_dir** в конфигурационном файле PHP **php.ini**.

Пример такого профиля показан на рис. 12.3.

webgrind v1.0
profiling in the browser

Show 90% of /usr/local/apache/htdocs/squa in percent update
Hide PHP functions

/usr/local/apache/htdocs/square/public/index.php
cachegrind.out.1265964180.3084 @ 2010-02-12 09:47:53
1327 different functions called in 1217 milliseconds (1 runs, 65 shown)

Function	Invocation Count	Total Self Cost	Total Inclusive Cost
Zend_Search_Lucene_Storage_File->readVint	4530	12.04	23.34
Zend_Search_Lucene_Storage_File_Filesystem->_fread	5470	11.05	11.81
Zend_Search_Lucene_Index_SegmentInfo->getTermInfo	17	10.88	48.38
require_once::C:\Program	154	9.36	16.49
Zend_Search_Lucene_Storage_File->readString	783	7.23	12.34
include::C:\Program	27	3.96	11.89
Zend_Search_Lucene_Index_Term::getPrefix	743	3.63	3.99
Zend_Loader::loadClass	29	3.22	16.13
Zend_Config_Ini->_processKey	433	2.73	8.60
Zend_Loader_PluginLoader->load	34	2.37	5.86
Zend_Search_Lucene->hasTerm	14	1.18	49.81
Doctrine::autoload	11	1.12	3.32
include_once::C:\Program	27	0.84	1.36
Zend_Search_Lucene_Storage_File->readInt	114	0.65	0.89

Рис. 12.3. Просмотр профиля Xdebug, сгенерированного для запроса к приложению Zend Framework, в интерфейсе Webgrind

Этот тип информации о профилировании очень полезен: например, он помогает понять, вызовы каких методов или компонентов фреймворка занимают большую часть времени выполнения сценария и, следовательно, являются подходящими кандидатами для оптимизации или замены другими, более легковесными библиотеками.

ПРИМЕЧАНИЕ

Расширение PHP Xdebug устанавливается обычным способом в каталог ext/ установки PHP и включается добавлением в конфигурационный файл php.ini директивы zend_extension_ts = /путь/к/xdebug/ext. Помимо этого, вы должны задать в конфигурационном файле значения для переменных xdebug.profiler_enable, xdebug.profiler_output_dir и xdebug.trace_output_dir. За более подробными инструкциями по установке обратитесь к веб-сайту Xdebug, ссылка на который дана в конце этой главы.

В качестве альтернативы Xdebug выступает класс PEAR Benchmark, предоставляющий API для оценки производительности вызова функций PHP. В данном случае средство оценки производительности встраивается в определенные действия, при этом их код помещается между вызовами методов **start()** и **stop()**. Эти методы, как правило, вызываются в начале и в конце действий, хотя их можно использовать и для оценки производительности конкретных участков действий.

Установите пользовательские «метки» для секций, находящихся между вызовами двух указанных методов, чтобы обозначить действия или транзакции, выполняемые сценарием; поскольку эти метки включаются в итоговый отчет, они могут помочь определить транзакции, ответственные за накладные расходы, и процент накладных расходов, вносимых ими. Пример использования класса:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function queryAction()
  {
    // загружаем файл с профайлером
    include_once('Benchmark/Profiler.php');
    $profiler = new Benchmark_Profiler();

    // запускаем профайлер
    $profiler->start();

    // профилируем настройку фильтров и валидаторов
    $profiler->enterSection("Setup");
    $filters = array(
        'q' => array('HtmlEntities', 'StripTags', 'StringTrim'),
    );
    $validators = array(
        'q' => array('Alpha'),
    );
    $input = new Zend_Filter_Input($filters, $validators);
    $input->setData($this->getRequest()->getParams());
    $profiler->leaveSection("Setup");
```

```
    // профилируем валидацию
    $profiler->enterSection("Validate");
    $check = $input->isValid();
    $profiler->leaveSection("Validate");

    // профилируем запрос к базе данных
    if ($check) {
      $profiler->enterSection("Query");
      $db = $this->getInvokeArg('bootstrap')->getResource('database');

      $sql = "SELECT * FROM city AS ci,
        country AS co,
        countrylanguage AS cl
        WHERE
          ci.CountryCode = co.Code AND
          cl.CountryCode = co.Code AND
          cl.Language = 'English' AND
          cl.IsOfficial = 'T' AND
          co.Continent LIKE '%$input->q%'";
      $result = $db->fetchAll($sql);
      $this->view->records = $result;
      $profiler->leaveSection("Query");
    }
    $profiler->stop();

    $this->view->profile = $profiler->_getOutput('html');
  }
}
```

Пример данных профилирования, сгенерированных PEAR Benchmark, показан на рис. 12.4.

	total ex. time	netto ex. time	#calls	%	calls	callers
Setup	0.00324702262878	0.00324702262878	1	7.01%		Global (1)
Validate	0.00511384010315	0.00511384010315	1	11.03%		Global (1)
Query	0.0377938747406	0.0377938747406	1	81.54%		Global (1)
Global	0.0463509559631	0.000196218490601	1	100.00%	Setup (1), Validate (1), Query (1)	

Рис. 12.4. Профиль PEAR Benchmark, сгенерированный для запроса к приложению Zend Framework

СОВЕТ

Класс PEAR Benchmark, как и другие классы из репозитория PEAR, по умолчанию не работает с автозагрузчиком Zend Framework. Чтобы исправить это, установите данный класс с помощью PEAR Installer, убедитесь в том, что каталог PEAR входит в список путей поиска PHP, а затем добавьте в файл application.ini строку autoloaderNamespaces[] = 'Benchmark'.

Профилирование запросов

Xdebug и PEAR Benchmark чрезвычайно полезны для отладки кода на PHP, однако они не предоставляют никакой информации касательно одного из узких мест, которое зачастую оказывает существенное влияние на производительность веб-приложений, — взаимодействия между приложением и сервером баз данных. Выполняя любое профилирование, необходимо также профилировать запросы приложения к базе данных и выяснить возможности их оптимизации. Если для запросов к базе данных вы используете **Zend_Db**, сделать это можно с помощью компонента **Zend_Db_Profiler**, который позволяет исследовать запросы приложения и получить отчет о времени выполнения каждого из них.

Самый простой способ использования **Zend_Db_Profiler** — добавление его к экземпляру **Zend_Db** в начальном загрузчике приложения. Пример:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initDatabase()
  {
    $db = new Zend_Db_Adapter_Pdo_Mysql(array(
          'host' => '127.0.0.1',
          'username' => 'user',
          'password' => 'pass',
          'dbname' => 'world'
      ));
    $profiler = new Zend_Db_Profiler();
    $profiler->setEnabled(true);
    $db->setProfiler($profiler);
    return $db;
  }
}
```

Профили запросов, полученные профайлером, можно получить с помощью метода **getQueryProfiles()** объекта **Zend_Db_Profiler**. Этот метод возвращает массив объектов **Zend_Db_Profiler_Query**, каждый из которых предоставляет методы **getQuery()** и **getElapsedSecs()** для получения строки запроса и времени его выполнения соответственно. Ниже приведен пример получения данных профилирования с использованием этих методов:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function queryAction()
  {
    // получаем адаптер базы данных
    $db = $this->getInvokeArg('bootstrap')->getResource('database');

    // выполняем запрос
    $sql = "SELECT * FROM city AS ci,
```

```
            country AS co,
            countrylanguage AS cl
            WHERE
              ci.CountryCode = co.Code AND
              cl.CountryCode = co.Code AND
              cl.Language = 'English' AND
              cl.IsOfficial = 'T' AND
              co.Continent LIKE '%$input->q%'";
        $result = $db->fetchAll($sql);
        $this->view->records = $result;
         $this->view->profiles = array();

    // получаем данные профиля этого запроса
    foreach ($db->getProfiler()->getQueryProfiles() as $profile) {
    $this->view->profiles[] = array(
        'sql' => $profile->getQuery(),
       'time' => $profile->getElapsedSecs(),
      );
    }
  }
}
```

На рис. 12.5 показан пример данных профилирования, сгенерированных предыдущим листингом.

```
Array
(
    [0] => Array
        (
            [sql] => connect
            [time] => 0.010577917099
        )

    [1] => Array
        (
            [sql] => SELECT * FROM city AS ci,
                country AS co,
                countrylanguage AS cl
                WHERE
                  ci.CountryCode = co.Code AND
                  cl.CountryCode = co.Code AND
                  cl.Language = 'English' AND
                  cl.IsOfficial = 'T' AND
                  co.Continent LIKE '%amer%'
            [time] => 0.010577917099
        )

)
```

Рис. 12.5. Профиль запроса, сгенерированный Zend_Db_Profiler

Существует также специализированное расширение Zend_Db_Profiler_Firebug, которое может отправлять данные профилирования запросов непосредственно в консоль Firebug. Чтобы использовать это расширение, включите его в начальном загрузчике приложения следующим образом:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initDatabase()
  {
    $db = new Zend_Db_Adapter_Pdo_Mysql(array(
          'host' => '127.0.0.1',
          'username' => 'user',
          'password' => 'pass',
          'dbname' => 'world'
    ));
    $profiler = new Zend_Db_Profiler_Firebug('Query log');
    $profiler->setEnabled(true);
    $db->setProfiler($profiler);
    return $db;
  }
}
```

На рис. 12.6 показан пример вывода в консоль Firebug профиля, сгенерированного профайлером **Zend_Db**.

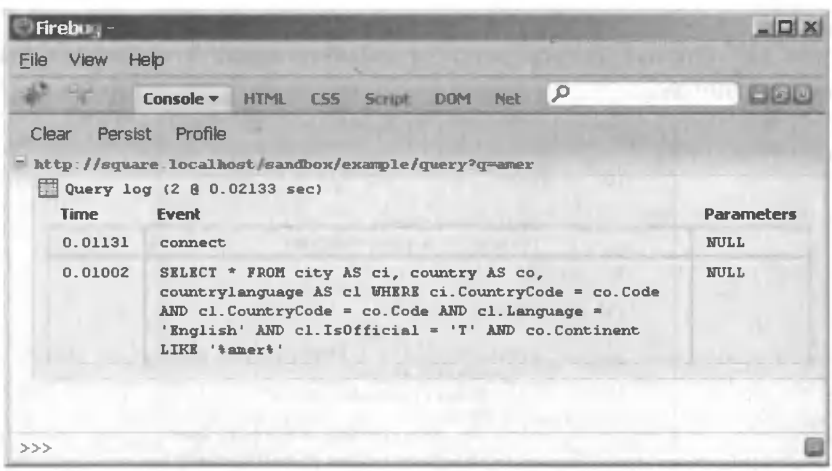


Рис. 12.6. Просмотр профиля, сгенерированного Zend_Db_Profiler, в консоли Firebug

В состав Zend Framework не входит встроенный профайлер для запросов к базе данных, использующих **Doctrine**, однако его легко добавить, воспользовавшись сторонним компонентом из проекта **Imind**. В рамках этого проекта разрабатываются несколько расширений для Zend Framework, ориентированных на работу с **Doctrine**, одним из которых является компонент **Iming_Profiler_Doctrine_Firebug**. Он записывает запросы **Doctrine** в консоль Firebug (отладчик для браузера Firefox), позволяя измерять их производительность в реальном времени.

Для профилирования запросов **Doctrine** с помощью профайлера Imind Doctrine добавьте его к менеджеру подключений **Doctrine** в начальном загрузчике приложения. Пример:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initDoctrine()
  {
    require_once 'Doctrine/Doctrine.php';
    $this->getApplication()->getAutoloader()
        ->pushAutoloader(array('Doctrine', 'autoload'), 'Doctrine');
    $manager = Doctrine_Manager::getInstance();
    $manager->setAttribute(
      Doctrine::ATTR_MODEL_LOADING,
      Doctrine::MODEL_LOADING_CONSERVATIVE
    );

    $config = $this->getOption('doctrine');
    $conn = Doctrine_Manager::connection($config['dsn'], 'doctrine');

    $profiler = new Imind_Profiler_Doctrine_Firebug();
    $conn->setListener($profiler);
    return $conn;
  }
}
```

В результате этих действий все запросы **Doctrine** будут автоматически профилироваться, а результаты — отображаться в консоли Firebug. Пример вывода в консоль, сгенерированного профайлером Imind Doctrine, показан на рис. 12.7.

Рис. 12.7. Просмотр профиля, сгенерированного профайлером Imind Doctrine, в консоли Firebug

ПРИМЕЧАНИЕ

После получения списка запросов и данных о длительности их выполнения они обычно анализируются и предпринимается попытка заставить эти запросы выполняться быстрее. Если вы используете MySQL, добавление перед каждым запросом SELECT ключевого слова EXPLAIN будет возвращать таблицу с описанием того, как будет обрабатываться этот запрос. В эту таблицу включена информация о том, к каким таблицам будут происходить обращения, а также о количестве строк, которые ожидается получить в результате запроса. Эта информация помогает увидеть, какие таблицы следует проиндексировать для увеличения производительности, и помогает найти узкие места. В разделе «Настройка запросов», расположенного далее в этой главе, вы найдете несколько наиболее распространенных приемов для ускорения выполнения запросов.

Кэширование данных

Одним из самых быстрых способов повысить производительность узких мест в веб-приложении является использование кэша.

«Что это такое?» — спросите вы. Все очень просто: кэш — это промежуточное хранилище, содержащее копии часто запрашиваемых фрагментов информации. При запросе одного из этих фрагментов система извлечет и вернет кэшированную копию, что всегда быстрее, чем повторная генерация информации из исходного источника данных.

В зависимости от требований пользователя кэш можно поддерживать на различных уровнях. Многие не знают, что наиболее часто используемый механизм кэширования — сам веб-браузер. Перед тем как отобразить загруженное содержимое пользователю, современные веб-браузеры сохраняют его во временное хранилище на жестком диске. И если вы снова посетите ту же самую страницу, браузер просто выберет ее из локального кэша (если, конечно, вы не настроили его по-другому).

Весьма вероятно, что на работе вы делите подключение к Интернету с большим числом пользователей через прокси-сервер. Толковые сетевые администраторы часто используют кэш прокси-сервера для сохранения копий часто запрашиваемых страниц. Результаты последующих запросов этих страниц извлекаются непосредственно из кэша прокси-сервера. Эта система обычно дублируется на разных уровнях инфраструктуры; зачастую провайдеры кэшируют содержимое, чтобы уменьшить трафик, который в противном случае мог бы занимать драгоценную пропускную способность магистральной сети.

И наконец, веб-сайты часто используют системы кэширования, чтобы быстрее обрабатывать собственное содержимое. В простейшей форме такая система заключается в передаче клиентам статических «снимков» динамических страниц вместо их повторного создания при каждом запросе. Это уменьшает нагрузку на сервер и освобождает ресурсы для других задач. Чтобы гарантировать относительную новизну снимков, они периодически обновляются.

В дополнение к такому кэшированию на уровне страниц можно также реализовать более низкоуровневое кэширование результатов запросов к базе данных, определений функций и классов и скомпилированного кода на PHP (последнее

также известно как кэширование кодов операций). Все эти стратегии кэширования повышают производительность вашего веб-приложения, уменьшая количество работы, которую необходимо проделать системе для выполнения определенного запроса.

Операции с кэшем

Zend Framework содержит компонент **Zend_Cache**, который предоставляет всеобъемлющий API для хранения данных в кэше и их получения. Эти данные могут включать в себя результаты выборок из базы данных, возвращаемые значения функций или методов, объекты, статические файлы или сформированные страницы. Сам кэш может быть реализован либо в виде дискового хранилища, либо в виде хранилища в памяти; **Zend_Cache** поддерживает несколько различных типов хранилищ, включая обычный файл, базу данных SQLite, сервер memcached, или кэш операций APC.

Каждая запись кэша обладает двумя основными атрибутами: *временем жизни* и *уникальным идентификатором*.

Значение времени жизни, обычно указываемое в секундах, определяет период, в течение которого запись в кэше должна считаться «корректной» или «свежей». По истечении этого периода запись становится «просроченной» и должна быть восстановлена из исходного источника данных.

Так как в одном кэше может храниться множество записей, для каждой из них должен быть определен уникальный идентификатор. Он служит отличительным признаком записи кэша и используется для получения кэшированных данных.

Чтобы лучше объяснить, как это работает, рассмотрим следующий пример:

```
php
ass Sandbox_ExampleController extends Zend_Controller_Action

ublic function cacheAction()
{
  // определяем параметры кэша
  $front = array(
        'lifetime' => 600,
        'automatic_serialization' => true
  );
  $back = array(
        'cache_dir' => APPLICATION_PATH . '/../tmp/cache'
  );

  // инициализируем кэш
  $cache = Zend_Cache::factory('Core', 'File', $front, $back);

  // используем кэш, если он доступен
  if(!($this->view->records = $cache->load('cities'))) {
    // формируем запрос
    $sql = "SELECT * FROM city AS ci,
        country AS co,
```

```
            countrylanguage AS cl
            WHERE
              ci.CountryCode = co.Code AND
              cl.CountryCode = co.Code AND
              cl.Language = 'English' AND
              cl.IsOfficial = 'T'";

        // получаем адаптер базы данных
        $db = $this->getInvokeArg('bootstrap')->getResource('database');

        // выполняем запрос и сохраняем результат в кэш
        $result = $db->fetchAll($sql);
        $cache->save($result, 'cities');
        $this->view->records = $result;
    }
  }
}
```

Листинг начинается с инициализации нового экземпляра **Zend_Cache** с помощью метода **Zend_Cache::factory()**. Этот метод получает названия *frontend-* и *backend*-компонентов кэша, а также массив конфигурационных параметров для каждого из них. В предыдущем примере использовался frontend-компонент **Core**, являющийся простейшим из доступных в **Zend_Cache** frontend-компонентов, а также backend-компонент **File**, который хранит записи кэша в виде файлов на диске. Конфигурационные параметры для каждого из этих компонентов, такие как количество секунд, в течение которых запись кэша считается корректной, и каталог для файлов кэша, указываются в виде ассоциативных массивов. Метод **Zend_Cache::factory()** возвращает должным образом настроенный экземпляр **Zend_Cache**.

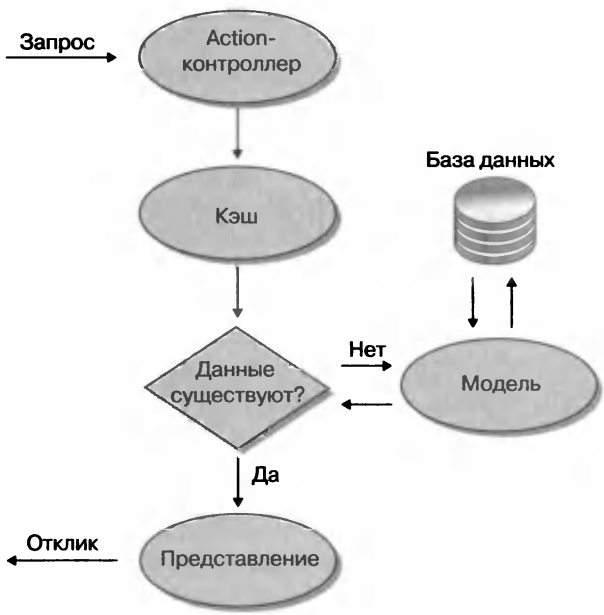


Рис. 12.8. Блок-схема функционирования кэша

Основная логика использования кэша, изображенная в виде блок-схемы на рис. 12.8, довольно проста. Эта бизнес-логика показана в предыдущем листинге, где кэшируется результат запроса к базе данных. Сначала код проверяет наличие требуемого результата в кэше. Если результат существует, он непосредственно присваивается представлению, и часть схемы, связанная с сервером баз данных, не задействуется. Если результат отсутствует, выполняется запрос к серверу баз данных для получения требуемых данных, и копия результата сохраняется в кэш. Эта копия используется для последующих запросов, пока не истечет время ее жизни и не потребуется обновление.

Проверка существования данных в кэше осуществляется с помощью метода load(), а запись данных в кэш — с помощью метода save(). Обратите внимание, что оба метода требуют уникального идентификатора записи кэша, которым в предыдущем листинге является строка 'cities'. Метод save() позволяет в качестве необязательного третьего аргумента указать время жизни записи в секундах; если этот аргумент присутствует, он заменяет значение по умолчанию, указанное в конфигурационных параметрах frontend-компонента.

СОВЕТ

Многим компонентам Zend Framework, включая Zend_Translate, Zend_Db и Zend_Feed_Reader, можно передать настроенный экземпляр Zend_Cache, вследствие чего они будут автоматически использовать его для хранения нужных данных или результатов поиска. Пример вы увидите далее в этой главе.

Frontend- и backend-компоненты кэша

Как говорилось в предыдущем разделе, Zend_Cache предлагает на выбор несколько различных frontend- и backend-компонентов, каждый из которых предназначен для разных целей. Списки компонентов приведены в табл. 12.1 и 12.2.

Таблица 12.1. Frontend-компоненты кэша, входящие в состав Zend Framework

Frontend-компонент кэша	Описание
Core	Универсальная базовая реализация кэша
Class	Кэширует возвращаемые значения методов
Function	Кэширует возвращаемые значения функций
File	Кэширует файлы на диске
Capture	Кэширует сформированные страницы или блоки
Page	Кэширует сформированные страницы

Таблица 12.2. Backend-компоненты кэша, входящие в состав Zend Framework

Backend-компонент кэша	Описание
Apc	Сохраняет записи кэша в кэше кодов операций APC
File	Сохраняет записи кэша в виде файлов на диске

продолжение

Таблица 12.2 (продолжение)

Backend-компонент кэша	Описание
Memcached	Сохраняет записи кэша на сервер memcached
Sqlite	Сохраняет записи кэша в базе данных SQLite
Static	Сохраняет записи кэша в виде статических HTML-файлов
Two_Levels	Сохраняет записи кэша, используя два хранилища
XCache	Сохраняет записи кэша, используя кэш XCache
ZendServer	Сохраняет записи кэша, используя кэш Zend Server
Blackhole	«Заглушка», которая кэширует записи в системную «корзину» (полезна только для тестирования)
Test	Тестовая реализация с предопределенными записями кэша (полезна только для отладки)

Чтобы показать, как использовать эти компоненты в приложении MVC, приведем несколько примеров. Рассмотрим, например, frontend-компоненты **Class** и **Function**, которые позволяют кэшировать процедуры с высокой вычислительной сложностью, чтобы снизить нагрузку на сервер. Пример использования frontend-компонента **Class**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function cacheAction()
  {
    // определяем параметры кэша
    $front = array(
          'lifetime' => 600,
          'automatic_serialization' => true,
          'cached_entity' => new Order
    );
    $back = array(
          'servers' => array(
              array(
            'host' => 'localhost',
            'port' => 11211,
            'timeout' => 5,
            'retry_interval' => 15,
          )
        )
      );

    // инициализируем кэш
    $cache = Zend_Cache::factory('Class', 'Memcached', $front, $back);
```

```
    // используем кэшированный объект для вызовов метода
    $this->view->result =
      $cache->findCheapestShippingProvider(100, 33.6, 1, 'NY');
    }
}
```

В этом примере метод **findCheapestShippingProvider()** принимает набор входных данных, таких как общий вес, количество предметов, скорость доставки и пункт назначения, а затем рассматривает и оценивает различные варианты, чтобы определить самую дешевую службу доставки по заданным параметрам. В большинстве случаев этот процесс будет занимать много времени, осуществляя несколько запросов к базе данных и вызовов конечных точек веб-сервисов. Следовательно, имеет смысл кэшировать результат вызова этого метода, с тем чтобы повторные запросы с таким же набором входных данных можно было получать непосредственно из кэша. Для большей скорости извлечения кэшированные данные хранятся на сервере memcached, а не в файле на диске.

Другой пример, на этот раз использующий frontend-компонент **File** и backend-компонент **APC**:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function cacheAction()
  {
    // определяем основной файл
    $localConfigFile = APPLICATION_PATH . '/configs/my.ini';

    // определяем параметры кэша
      $front = array(
          'lifetime' => 600,
          'automatic_serialization' => true,
          'master_files' => array($localConfigFile)
    );
    $back = array();
    // инициализируем кэш
    // загружаем из кэша основной файл
    $cache = Zend_Cache::factory('File', 'Apc', $front, $back);
    if (!($config = $cache->load('localconfig'))) {
      if (file_exists($localConfigFile)) {
        $config = new Zend_Config_Ini($localConfigFile);
        $cache->save($config, 'localconfig');
      }
    }
  }
}
```

Этот код автоматически кэширует указанный основной файл в кэше APC, использующем разделяемую память, и задействует этот кэш для последующих запросов. Ключевой момент здесь заключается в том, что frontend-компонент **File**

будет автоматически отслеживать любые изменения в основном файле и при их появлении будет помечать кэш как недействительный и пересоздавать его при следующем запросе.

ПРИМЕЧАНИЕ

Особый интерес представляет backend-компонент Static, поскольку он сохраняет сформированные страницы на диск в виде статических HTML-файлов. Эти файлы впоследствии могут возвращаться в ответ на запросы, не задействуя PHP вообще и тем самым обеспечивая резкое увеличение производительности. Использование этого backend-компонента требует внесения определенных изменений в правила переписывания URL, используемые сервером, однако результат зачастую стоит этих усилий, особенно для тех URL, к которым часто обращаются. В руководстве Zend Framework приведен подробный пример использования этого backend-компонента, а также пример правил переписывания, которые вы можете расширить в соответствии с различными сценариями использования.

Использование менеджера кэша

В большинстве случае кэш будет настраиваться и инициализироваться в начальном загрузчике приложения, а затем «экспортироваться» в приложение через Zend_Registry, чтобы его можно было использовать в различных действиях. Однако во многих случаях, в зависимости от типа кэшируемых данных, требуется использовать разные параметры кэша. Например, вы, возможно, захотите кэшировать ленты RSS и веб-сервисов в файлы на диске, а результаты часто выполняемых запросов хранить в оперативной памяти, так как получение данных с диска обычно занимает больше времени, нежели извлечение их из памяти.

В связи с этим нахождение правильного баланса между различными типами кэша имеет огромную важность для достижения оптимального сочетания скорости и эффективности. Эта задача несколько упрощается при использовании компонента Zend_Cache_Manager из состава Zend Framework, а также программного расширения для управления ресурсами, которое позволяет настраивать несколько типов кэша в конфигурационном файле приложения. Это уменьшает количество действий по настройке, которые требуется выполнить в начальном загрузчике, и делает возможным использование различных методик кэширования для различных действий (или даже в пределах одного действия).

Пример такой конфигурации кэша:

```
resources.cachemanager.mycache.frontend.name = Core
resources.cachemanager.mycache.frontend.options.lifetime = 600
resources.cachemanager.mycache.frontend.options.automatic_serialization = true
resources.cachemanager.mycache.backend.name = Apc
```

Теперь можно непосредственно использовать этот экземпляр кэша в действии:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function cacheAction()
  {
    // получаем кэш
```

```
    $cache = $this->getInvokeArg('bootstrap')
        ->getResource('cachemanager')
        ->getCache('mycache');

    // используем кэш, если он доступен
    if(!($this->view->records = $cache->load('cities'))) {
    // выполняем запрос
      $sql = "SELECT * FROM city AS ci,
         country AS co,
         countrylanguage AS cl
         WHERE
           ci.CountryCode = co.Code AND
           cl.CountryCode = co.Code AND
           cl.Language = 'English' AND
           cl.IsOfficial = 'T'";

      // получаем и используем адаптер базы данных
      $db = $this->getInvokeArg('bootstrap')->getResource('database');
      $result = $db->fetchAll($sql);
      $cache->save($result, 'cities');
      $this->view->records = $result;
    }
  }
}
```

ПРИМЕЧАНИЕ

Важно отметить, что настройка кэша в конфигурационном файле приложения не инициализирует его автоматически. В интересах производительности Zend_Cache_Manager будет инициализировать экземпляр Zend_Cache только при обращении к нему по имени через метод getCache(), но не раньше.

Кэширование запросов Doctrine

Хотя для кэширования результатов запросов к базе данных, безусловно, можно использовать Zend_Cache, для этих целей можно приспособить и собственный механизм кэширования Doctrine, что показано в первом примере этого раздела. В состав Doctrine входят две различные системы кэширования: для запросов и для результатов их выполнения; включить и использовать обе системы несложно, и это потребует минимального рефакторинга кода.

Кэш запросов Doctrine хранит проанализированные представления объектов Doctrine_Query, позволяя отправлять повторяющиеся запросы непосредственно серверу, без необходимости их предварительного анализа. Кэш результатов Doctrine делает то же самое, что и кэш запросов, но дополнительно хранит сериализованные представления результатов запросов и по возможности возвращает их, исключая при этом взаимодействие с базой данных и существенно повышая производительность.

Кэшированные данные можно хранить на сервере memcached, в кэше APC, использующем разделяемую память, или в любой базе данных, поддерживаемой Doctrine.

ВОПРОС ЭКСПЕРТУ

В: Всегда ли кэш в памяти предпочтительнее, чем кэш на диске?

О: Дисковый кэш обычно считается более «медленным», чем кэш в памяти, поскольку получение данных из файлов на диске, как правило, занимает больше времени, нежели их извлечение из кэша, расположенного в памяти, такого как memcached или APC. С учетом этого имеет смысл кэшировать часто запрашиваемые данные в памяти, а не в дисковом кэше, чтобы их можно было получать и обрабатывать быстрее.

Однако на основании вышесказанного будет неправильным полагать, что производительность приложения можно сделать максимальной, просто сохраняя все данные в кэше, использующем память, поскольку требуется также рассматривать и размер кэшируемых данных. В большинстве случаев память — куда более дефицитный ресурс, нежели дисковое пространство, поэтому если кэшируемые данные имеют большой объем, обычно разумнее использовать дисковый кэш. Хранение всего хлама в памяти обычно заканчивается падением производительности вместо ее прироста, так как система быстро исчерпывает память, необходимую для выполнения других важных задач.

Активировать кэш запросов Doctrine просто: инициализируйте драйвер Doctrine для соответствующего backend-компонента кэша и передайте его экземпляру Doctrine_Manager, используя атрибут Doctrine::ATTR_QUERY_CACHE. Пример:

```
<?php
// настраиваем драйвер кэша
$servers = array(
  'host' => 'localhost',
  'port' => 11211,
  'persistent' => false
);
$cache = new Doctrine_Cache_Memcache(array(
    'servers' => $servers,
    'compression' => false
  )
);

// добавляем кэш к менеджеру Doctrine
$manager = Doctrine_Manager::getInstance();
$manager->setAttribute(Doctrine::ATTR_QUERY_CACHE, $cache);
?>
```

После такой инициализации кэш запросов Doctrine автоматически используется для всех запросов, выполняемых через Doctrine.

Аналогичный подход можно использовать и для кэша результатов Doctrine:

```
<?php
// настраиваем драйвер кэша
$servers = array(
  'host' => 'localhost',
  'port' => 11211,
  'persistent' => false
);
```

```
$cache = new Doctrine_Cache_Memcache(array(
    'servers' => $servers,
    'compression' => false
  )
);

// добавляем кэш к менеджеру Doctrine
$manager = Doctrine_Manager::getInstance();
$manager->setAttribute(Doctrine::ATTR_RESULT_CACHE, $cache);
?>
```

Важно отметить, что кэш результатов **Doctrine** не используется автоматически для всех запросов, выполняемых через **Doctrine**; его необходимо явно задействовать для каждого запроса, добавив к соответствующему объекту **Doctrine_Query** вызов метода **useResultCache()**. Пример:

```
<?php
class Sandbox_ExampleController extends Zend_Controller_Action
{
  public function cacheAction()
  {
    $id = $this->_getParam('id');
    $q = Doctrine_Query::create()
        ->from('Square_Model_Item i')
        ->leftJoin('i.Square_Model_Country c')
        ->leftJoin('i.Square_Model_Grade g')
        ->leftJoin('i.Square_Model_Type t')
        ->where('i.RecordID = ?', $id)
        ->addWhere('i.DisplayStatus = 1')
        ->useResultCache(true);
    $this->view->result = $q->fetchArray();
  }
}
```

ПРИМЕЧАНИЕ

Используя MySQL в качестве базы данных, вы должны знать, что MySQL имеет собственный кэш запросов, включенный по умолчанию. Этот кэш сохраняет результаты запросов SELECT и при следующем поступлении того же запроса извлекает результаты из кэша вместо его повторного выполнения. Однако следует помнить, что запросы будут соответствовать кэшу только в случае точного совпадения их текста; любое различие будет рассматриваться как новый запрос. Например, запрос 'SELECT * FROM airport' не вернет из кэша результат запроса 'select * FROM airport'.

Оптимизация кода приложения

Хотя кэширование, безусловно, может значительно улучшить производительность веб-приложения, это не единственное средство, доступное умелым разработчикам. В следующих разделах рассматриваются некоторые другие распространенные стратегии оптимизации.

Настройка запросов

Существуют несколько приемов, которые можно использовать, чтобы оптимизировать производительность запросов к базе данных и гарантировать, что они будут работать с максимально возможной эффективностью. В следующих разделах рассматриваются некоторые из этих приемов, при этом особое внимание уделяется MySQL как самому распространенному серверу баз данных в сфере разработки веб-приложений.

Использование объединений вместо подзапросов

MySQL оптимизирует объединения лучше, чем подзапросы, поэтому если вы обнаружите, что средние значения загруженности сервера MySQL непозволительно велики, проверьте код вашего приложения и попытайтесь заменить подзапросы на объединения или последовательности объединений. Например, следующий подзапрос, безусловно, корректен:

```
SELECT id, name FROM movie WHERE directorid IN
  (SELECT id FROM director
  WHERE name = 'Alfred Hitchcock');
```

Однако эквивалентное ему объединение будет выполняться быстрее вследствие алгоритмов оптимизации, используемых в MySQL:

```
SELECT m.id, m.name FROM movie AS m, director AS d
  WHERE d.id = m.directorid AND d.name = 'Alfred Hitchcock';
```

Вы также можете преобразовать неэффективные запросы в более эффективные путем творческого использования таких утверждений MySQL, как ORDER BY и LIMIT. Рассмотрим следующий подзапрос:

```
SELECT id, duration FROM movie
  WHERE duration =
  (SELECT MAX(duration) FROM movie);
```

Вместо него лучше использовать следующий запрос, к тому же его проще читать и он выполняется быстрее:

```
SELECT id, duration FROM movie
  ORDER BY duration DESC
  LIMIT 0,1;
```

Использование временных таблиц для временных данных или вычислений

MySQL также позволяет создавать временные таблицы с помощью команды CREATE TEMPORARY TABLE. Они названы так, поскольку существуют только в течение одной сессии MySQL и автоматически удаляются, когда создавший их клиент закрывает подключение к серверу MySQL. Эти таблицы удобны для данных и вычислений, использующихся в течение одной сессии, или для временного хранения данных. И так как они зависят от сессий, две различные сессии могут использовать одно и то же название таблицы, не создавая при этом конфликт.

Поскольку временные таблицы хранятся в памяти, работа с ними происходит существенно быстрее, чем с таблицами, хранящимися на диске. Следовательно,

их можно эффективно использовать в качестве промежуточных областей для хранения, чтобы ускорить выполнение запросов, разбивая сложные запросы на более простые, или в качестве замены подзапросов и объединений.

Используемый в MySQL синтаксис INSERT...SELECT, а также ключевое слово IGNORE и поддержка временных таблиц предоставляют многочисленные возможности для креативного переписывания запросов SELECT с целью ускорения их выполнения. Предположим, к примеру, что у вас есть сложный запрос, в числе прочего выбирающий множество несовпадающих значений определенного поля, но механизм работы с данными MySQL не может оптимизировать этот запрос ввиду его сложности. При творческом подходе программисты на SQL могут увеличить производительность, разбив сложный запрос на несколько простых (которые лучше подходят для оптимизации) и воспользовавшись командой INSERT IGNORE...SELECT, чтобы сохранить полученные результаты во временную таблицу, предварительно создав ее с ключом UNIQUE для соответствующего поля. В результате получится множество неповторяющихся значений этого поля и, как результат, более быстрое выполнение запроса.

Явное указание требуемых полей

Довольно часто можно встретить запросы вроде этих:

```
SELECT (*) FROM airport;
SELECT COUNT(*) FROM airport;
```

Данные запросы для удобства используют групповой символ «звездочка» (*). Однако это удобство имеет свою цену: групповой символ * заставляет MySQL читать каждое поле или запись таблицы, увеличивая общее время обработки запроса. Чтобы избежать этого, явно перечислите поля, которые вы хотите получить в результатах выборки, как показано в следующем примере:

```
SELECT id, name FROM airport;
SELECT COUNT(id) FROM airport;
```

Индексирование полей, используемых для объединения

Поля, к которым часто обращаются, должны быть проиндексированы. Как правило, если у вас есть поле, использующееся для поиска, группировки или сортировки, его индексация, скорее всего, увеличит производительность. Процесс индексации должен затрагивать поля, участвующие в операциях объединения, а также поля, встречающиеся в утверждениях WHERE, GROUP BY или ORDER BY. Кроме того, лучшую производительность обеспечит объединение таблиц по целочисленным, а не по символьным, полям.

Использование маленьких транзакций

Хотя принцип KISS (Keep It Simple, Stupid! — не усложняй, дурак!) и может показаться избитым, он все еще работает в сложном мире транзакций. Это связано с тем, что MySQL использует механизм блокировки строк, чтобы предотвратить редактирование одной и той же записи одновременными транзакциями и возможное повреждение базы данных. Механизм блокировки строк не позволяет одно-

временно получать доступ к строке более чем одной транзакции — это защищает данные, но недостаток такого подхода состоит в том, что остальным транзакциям приходится ждать, пока транзакция, установившая блокировки, завершит свою работу. В случае с маленькими транзакциями это время ожидания не так заметно. Однако при работе с большой базой данных и множеством сложных транзакций длительное время ожидания снятия блокировок различными транзакциями может существенно повлиять на производительность.

По этой причине придерживайтесь небольших размеров транзакций и обеспечивайте для них быстрое внесение изменений и завершение, с тем чтобы остальные транзакции не задерживались в очереди. На уровне приложений для решения этой задачи существуют две распространенные стратегии:

- Убедиться, что перед выполнением команды START TRANSACTION доступны все пользовательские данные, которые требует транзакция. Зачастую начинающие разработчики инициируют транзакцию до того, как будет доступен весь набор требующихся для нее значений. В результате другие транзакции, инициированные в это же время, вынуждены ждать, пока пользователь введет необходимые данные, а приложение обработает их, затем запросит новые данные и так далее. В однопользовательской среде эти задержки не имеют значения, поскольку другие транзакции не будут пытаться получить доступ к базе данных. Однако в случае с множеством пользователей задержка в одной транзакции отразится на всех остальных транзакциях, помещенных в очередь системой, что приведет к значительному падению производительности.
- Попробовать разбивать большие транзакции на более мелкие подтранзакции и выполнять их независимо друг от друга. Это гарантирует, что каждая подтранзакция выполнится быстро и освободит ценные ресурсы системы, которые в противном случае использовались бы для сохранения ее состояния.

Оптимизация схемы таблиц и конфигурации сервера

Все приемы, описанные в этом разделе, предлагают проводить оптимизацию на уровне приложения. Однако для дальнейшего увеличения производительности вам также следует рассмотреть различные техники оптимизации производительности на уровне базы данных, такие как:

- использование индексов для ускорения поиска, группировки и сортировки;
- выбор подходящего механизма хранения таблиц;
- указание одинаковых типов данных и размеров для полей, использующихся в объединениях;
- увеличение памяти, доступной для различных серверных буферов, таких как буфер чтения, буфер сортировки и кэш потоков.

Более подробную информацию по описанным приемам вы найдете в руководстве к серверу баз данных.

Отложенная загрузка

И Zend Framework, и Doctrine содержат автозагрузчики, которые могут автоматически находить и загружать файлы с определениями классов «по требованию». Этот прием, известный как *отложенная загрузка*, может существенно увеличить производительность, снизив количество операций чтения файлов, выполняемых сценарием PHP.

Чтобы включить автозагрузчик Zend Framework, руководство по фреймворку предлагает добавить в сценарий index.php приложения следующий код:

```
<?php
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();
?>
```

Чтобы использовать все преимущества автозагрузчика, необходимо также убрать все вызовы функций require_once() и include_once(), использующиеся в вашей установке Zend Framework. В руководстве приведен пример простого сценария командной строки, способного сделать это автоматически. Ссылку на соответствующую страницу руководства вы найдете в конце этой главы.

Также хорошей идеей будет изменение порядка записей в списке путей поиска PHP и помещение каталога Zend Framework ближе к началу списка. Это связано с тем, что при попытке вызвать для файла функцию require() или include() PHP будет производить поиск в каждом каталоге в списке путей поиска, и если требуемый каталог находится в конце списка, эта операция может занять много времени. Помещение каталога Zend Framework перед другими помогает уменьшить время, необходимое для поиска и загрузки включаемых файлов.

Упражнение 12.1. Увеличение производительности приложения

Теперь давайте попробуем применить некоторые из описанных в предыдущих разделах приемов для увеличения производительности демонстрационного приложения SQUARE.

ПРИМЕЧАНИЕ

В следующем разделе рассматривается кэширование данных приложения с использованием memcached — распределенной системы кэширования данных в оперативной памяти. Предполагается, что у вас уже есть работающая установка сервера memcached, а также одноименное расширение PHP. Если вы еще не установили эти компоненты, их можно загрузить из Интернета по ссылкам, приведенным в конце главы. Если memcached недоступен для используемой платформы или вы не можете настроить его работу с PHP, воспользуетесь кодом из следующего примера, просто задействовав для второго кэша файловый backend-компонент вместо компонента, использующего оперативную память.

Настройка кэша приложения

Этот пример будет использовать два кэша: дисковый кэш и кэш в оперативной памяти, использующий memcached. Для простоты мы будем полагать, что сервер memcached запущен на локальном компьютере, использует параметры по умолчанию и не требует специальной настройки. Итак, первым шагом будет определение расположения файлового кэша и настройка его параметров с помощью компонента **Zend_Cache_Manager**.

Для начала создайте каталог для файлового кэша:

```
shell> cd /usr/local/apache/htdocs/square/data
shell> mkdir cache
```

Затем добавьте в конфигурационный файл приложения, **$APP_DIR/application/configs/application.ini**, следующие параметры:

```
resources.cachemanager.news.frontend.name = Core
resources.cachemanager.news.frontend.options.lifetime = 600
resources.cachemanager.news.frontend.options.automatic_serialization = true
resources.cachemanager.news.backend.name = File
resources.cachemanager.news.backend.options.cache_dir =
APPLICATION_PATH "/../data/cache"
resources.cachemanager.memory.frontend.name = Core
resources.cachemanager.memory.frontend.options.lifetime = 300
resources.cachemanager.memory.frontend.options.automatic_serialization = true
resources.cachemanager.memory.backend.name = Memcached
resources.cachemanager.memory.backend.options.servers.host = localhost
resources.cachemanager.memory.backend.options.servers.port = 11211
resources.cachemanager.memory.backend.options.servers.timeout = 5
resources.cachemanager.memory.backend.options.servers.retry_interval = 10
```

Разобравшись с настройкой, приступим к кэшированию!

Кэширование строк перевода

Компоненты **Zend_Translate** и **Zend_Locale** могут использовать предварительно настроенный экземпляр **Zend_Cache** для хранения данных о локали и строк перевода для ускорения их поиска. Настроить кэширование для этих компонентов несложно: передайте их статическим методам **setCache()** должным образом настроенный экземпляр **Zend_Cache**, обо всем остальном они позаботятся сами.

Чтобы включить кэширование для двух указанных компонентов, добавьте в начальный загрузчик приложения новый метод **_initCache()**:

```
<?php
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
  protected function _initCache()
  {
    $this->bootstrap('cachemanager');
    $manager = $this->getResource('cachemanager');
```

```
    $memoryCache = $manager->getCache('memory');
    Zend_Locale::setCache($memoryCache);
    Zend_Translate::setCache($memoryCache);
}
```

Кэширование результатов запросов

Если вы посмотрите на действия **Catalog_ItemController::displayAction** и **Catalog_AdminItemController::displayAction**, то увидите, что оба они выполняют один и тот же запрос Doctrine с единственным различием: запрос в первом действии содержит дополнительную проверку того, что элемент каталога был разрешен к публичному показу. Поскольку это, по сути, одинаковый код, который повторяется дважды, он является идеальным кандидатом для рефакторинга с преобразованием его в метод модели.

Для этого добавьте в класс модели **Square_Model_Item**, находящийся в файле **APP_DIR/library/Square/Model/Item.php**, следующий метод:

```
<?php
class Square_Model_Item extends Square_Model_BaseItem

{
  public function getItem($id, $active = true)
  {
    $q = Doctrine_Query::create()
         ->from('Square_Model_Item i')
         ->leftJoin('i.Square_Model_Country c')
         ->leftJoin('i.Square_Model_Grade g')
         ->leftJoin('i.Square_Model_Type t')
         ->where('i.RecordID = ?', $id);
      if ($active) {
         $q->addWhere('i.DisplayStatus = 1')
         ->addWhere('i.DisplayUntil >= CURDATE()');
      }
    return $q->fetchArray();
  }
}
```

Теперь обновите **Catalog_ItemController::displayAction**, чтобы использовать метод **getItem()**, определенный в модели, вместо непосредственного формирования запроса **Doctrine**. Одновременно с этим добавьте кэширование результатов метода с использованием кэша, расположенного в памяти и определенного ранее. Ниже приведен измененный метод **Catalog_ItemController::displayAction**:

```
<?php
class Catalog_ItemController extends Zend_Controller_Action
{
  // действие для отображения элемента каталога
  public function displayAction()
  {
    // устанавливаем фильтры и валидаторы для входных данных,
```

```
    // полученных в запросе GET
    $filters = array(
      'id' => array('HtmlEntities', 'StripTags', 'StringTrim')
    );
    $validators = array(
      'id' => array('NotEmpty', 'Int')
    );
    $input = new Zend_Filter_Input($filters, $validators);
    $input->setData($this->getRequest()->getParams());

    // проверяем корректность входных данных
    // если они корректны, получаем запрошенную запись
    // и добавляем ее к представлению
    if ($input->isValid()) {
      $memoryCache = $this->getInvokeArg('bootstrap')
                ->getResource('cachemanager')
                ->getCache('memory');
      if (!($result = $memoryCache->load('public_item_'.$input->id))) {
        $item = new Square_Model_Item;
        $result = $item->getItem($input->id, true);
        $memoryCache->save($result, 'public_item_'.$input->id);
      }
        if (count($result) == 1) {
        $this->view->item = $result[0];
        $this->view->images = array();
        $config = $this->getInvokeArg('bootstrap')->getOption('uploads');
        foreach (glob("{$config['uploadPath']}/
          {$this->view->item['RecordID']}_*") as $file) {
          $this->view->images[] = basename($file);
        }
        $configs = $this->getInvokeArg('bootstrap')->getOption('configs');
        $localConfig = new Zend_Config_Ini($configs['localConfigPath']);
        $this->view->seller = $localConfig->user->displaySellerInfo;
        $registry = Zend_Registry::getInstance();
        $this->view->locale = $registry->get('Zend_Locale');
        $this->view->recordDate = new Zend_Date($result[0]['RecordDate']);
      } else {
        throw new Zend_Controller_Action_Exception('Page not found', 404);
      }
    } else {
      throw new Zend_Controller_Action_Exception('Invalid input');
    }
  }
}
```

То же самое можно проделать с методом **Catalog_AdminItemController::displayAction**. Исправленный код вы найдете в архиве с дополнительными материалами к этой главе.

ВОПРОС ЭКСПЕРТУ

В: Каким образом рефакторинг методов модели способствует увеличению производительности?

О: В главе 4 обсуждался подход «толстая модель, тонкий контроллер», который предлагал размещать бизнес-логику в моделях, а не в контроллерах. Помимо прочих преимуществ этот подход при использовании вместе с кэшированием может увеличить производительность. Как было показано ранее в разделе «Кэширование результатов запросов», вы можете использовать кэширование для предотвращения вызовов методов модели в случае существования кэшированных данных, тем самым уменьшив число строк кода, которые потребуется выполнить. А меньшее количество строк автоматически означает более быстрый анализ и выполнение сценария.

Кэширование лент Twitter и блогов

В своем текущем варианте действие NewsController::indexAction при каждом запросе получает свежий набор результатов поиска в Twitter и самые последние ленты новостей и блогов. Получение этих данных осуществляется по HTTP и является довольно затратным; в то же время они представляют собой некритическую информацию, не оказывающую существенного влияния на приложение. Это делает их идеальным кандидатом для кэширования.

Рассмотрим следующий вариант NewsController::indexAction, использующий frontend-компонент File для кэширования результатов поиска в Twitter и лент из блогов и новостей с целью уменьшения времени отклика:

```
<?php
class NewsController extends Zend_Controller_Action
{
  public function indexAction()
  {
    // получаем ленту с результатами поиска по Twitter
    $q = 'philately';
    $this->view->q = $q;

    // получаем кэш
    $fileCache = $this->getInvokeArg('bootstrap')
            ->getResource('cachemanager')
            ->getCache('news');

    $id = 'twitter';
    if(!($this->view->tweets = $fileCache->load($id))) {
      $twitter = new Zend_Service_Twitter_Search();
      $this->view->tweets = $twitter->search($q,
        array('lang' => 'en', 'rpp' => 8, 'show_user' => true));
      $fileCache->save($this->view->tweets, $id);
    }
```

```
        Zend_Feed_Reader::setCache($fileCache);
        $this->view->feeds = array();
        // Получаем ленту Google News, представленную в формате Atom
        $gnewsFeed = "http://news.google.com/news?hl=en&q=$q&output=atom";
        $this->view->feeds[0] = Zend_Feed_Reader::import($gnewsFeed);

        // получаем ленту BPMA, представленную в формате RSS
        $bpmaFeed = "http://www.postalheritage.org.uk/news/RSS";
        $this->view->feeds[1] = Zend_Feed_Reader::import($bpmaFeed);
    }
}
```

Обратите внимание на то, что в предыдущем листинге кэширование для Zend_Feed_Reader включается простым вызовом статического метода Zend_Feed_Reader::setCache(), которому передается настроенный экземпляр объекта Zend_Cache.

Если бы вы измерили производительность до и после внесения вышеописанных изменений, то заметили бы ее явный прирост. Для иллюстрации рассмотрим табл. 12.3, где указан пример статистических данных «до» и «после», полученных с помощью ApacheBench на тестовой установке приложения SQUARE. Отчет был сгенерирован на основании 100 запросов с уровнем параллелизма, равным 10.

Таблица 12.3. Оценка производительности демонстрационного приложения, выполненная с помощью ApacheBench до и после добавления кэширования

URL	Без кэширования		С кэшированием		Улучшения от кэширования, %
	Число запросов, обрабатываемых за секунду	Среднее время обработки запроса, мс	Число запросов, обрабатываемых за секунду	Среднее время обработки запроса, мс	
http://square.localhost/news	0,45	2242,00	2,07	483,91	≈360
http://square.localhost/catalog/item/1	3,32	301,563	9,52	105,00	≈187

ВОПРОС ЭКСПЕРТУ

В: Почему вы не используете кэш результатов Doctrine для кэширования результатов запросов?

О: Хотя, безусловно, можно использовать кэш результатов Doctrine, это неизбежно потребует загрузки модели(ей) Doctrine. Использованный в приведенном примере подход не делает этого, в результате чего никакие модели Doctrine не вызываются. Это еще больше уменьшает количество выполняемого кода и способствует увеличению общей производительности.

Выводы

В последней главе книги был изучен вопрос оптимизации производительности Zend Framework, рассмотрены некоторые приемы и возможности, которые могут помочь выжать еще немного из вашего приложения. Кроме краткого обзора средств анализа производительности и профилирования кода в этой главе были исследованы различные стратегии оптимизации производительности, такие как кэширование запросов и выходных данных, отложенная загрузка, настройка SQL-запросов и рефакторинг кода.

Оптимизация приложений сама по себе является практически наукой, и в рамках этой главы невозможно рассказать о ней все. Однако применение описанных техник поможет вам создавать более эффективные приложения, и постоянное стремление к оптимизации должно стать частью вашего обычного процесса разработки.

Чтобы узнать больше об оптимизации производительности и загрузить некоторые средства, упомянутые в этой главе, посетите следующие ссылки:

- Компонент Zend_Cache: http://framework.zend.com/manual/en/zend.cache.html.
- Компонент Zend_Db_Profiler: http://framework.zend.com/manual/en/zend.db.profiler.html.
- Информация о кэшировании в Doctrine: http://www.doctrine-project.org/documentation/manual/1_1/en/caching.
- Информация об оптимизации производительности в MySQL: http://dev.mysql.com/doc/refman/5.1/en/optimization.html.
- Стратегии оптимизации производительности в Zend Framework: http://framework.zend.com/manual/en/performance.html.
- Документация к ApacheBench documentation: http://httpd.apache.org/docs/2.0/programs/ab.html.
- WCAT: http://www.iis.net/downloads/default.aspx?tabid=34&i=1466&g=6.
- Xdebug: http://xdebug.org/.
- Webgrind: http://code.google.com/p/webgrind/.
- Wincachegrind: http://sourceforge.net/projects/wincachegrind/.
- Kcachegrind: http://kcachegrind.sourceforge.net/.
- Пакет PEAR Benchmark: http://pear.php.net/package/Benchmark.
- Memcached: http://memcached.org/.
- Расширение PHP memcached: http://pecl.php.net/package/memcache.
- Профайлер для Doctrine Imind: http://code.google.com/p/imind-php/wiki/Imind_Profiler_Doctrine_Firebug.
- Профилирование сценариев PHP с помощью Xdebug и Webgrind (Энант Гарг (Anant Garg)): http://anantgarg.com/2009/03/10/php-xdebug-webgrind-installation/.

Хотя вы и достигли конца этой книги, очевидно, что ваши приключения с Zend Framework еще далеки от завершения... пожалуй, это лишь начало! Предыдущие главы дали вам необходимую подготовку, чтобы вы могли начать создавать собственные высококачественные приложения Zend Framework. Все остальное зависит только от вас и вашего воображения. Удачи и приятного программирования!

A

Приложение Установка и настройка необходимого программного обеспечения

Прочитав приложение, вы:

- узнаете, как загрузить из Интернета и установить MySQL, PHP и Apache;
- выполните простую проверку, чтобы убедиться в правильной работе всех приложений;
- выясните, как автоматически запускать все требуемые компоненты при загрузке системы;
- предпримете основные действия по обеспечению безопасности установки MySQL.

В этой книге вы узнали о Zend Framework и о том, как можно использовать его для создания сложных веб-приложений. В приведенных в книге примерах предполагается, что у вас уже есть правильно настроенная среда разработки в виде Apache/MySQL/PHP. Если это не так, данное приложение покажет вам, как установить и настроить упомянутые компоненты и создать окружение разработки, которое можно использовать для запуска примеров кода из этой книги.

ПРИМЕЧАНИЕ

Это приложение представляет собой краткий обзор и общее руководство по процессу установки и настройки MySQL, PHP и Apache в UNIX и Windows. Оно не предназначено для замены документации по установке, входящей в состав каждого программного пакета. Если вы столкнетесь с трудностями в процессе установки различных программ, описанных здесь, обратитесь на соответствующий веб-сайт программы или поищите подробную информацию по решению проблемы или совет в Интернете (несколько ссылок приведены в конце приложения).

Получение программного обеспечения

Существуют два способа получения работающего окружения разработки Apache/MySQL/PHP в системе Windows или *NIX:

1. Вы можете загрузить единый интегрированный комплект, содержащий все необходимые элементы и зависимости, заранее настроенные для совместной работы.
2. Вы можете загрузить разные элементы по отдельности и настроить их самостоятельно.

Первый подход рекомендуется в том случае, если вы новичок в PHP или в разработке открытого исходного кода в целом. Он позволяет установить и запустить все компоненты с минимумом проблем. Существуют два популярных комплекта, которые стоит рассмотреть:

- Zend Server Community Edition — это надежный интегрированный комплект, предоставляемый компанией Zend Technologies. В дополнение к стандартным компонентам Apache/MySQL/PHP он содержит несколько средств отладки и оптимизации, а также (в качестве бонуса) последнюю версию Zend Framework. Комплект доступен для операционных систем Microsoft Windows, Mac OS X и Linux и может быть свободно загружен с сайта http://www.zend.com/.
- XAMPP — это поддерживаемый сообществом дистрибутив, содержащий последние версии Apache, PHP, MySQL и Perl. Он полностью настроен для использования сразу после установки и чрезвычайно популярен для быстрого развертывания окружения разработки на PHP. Дистрибутив также содержит дополнительные средства: FTP-сервер и средство администрирования MySQL с веб-интерфейсом. Он доступен для Microsoft Windows, Mac OS X, Linux и Solaris и может быть свободно загружен с сайта http://www.apachefriends.org/en/index.html.

Недостатком использования интегрированного комплекта является то, что вы узнаете значительно меньше, чем при использовании второго подхода «сделай сам», поскольку все компоненты уже настроены за вас. Если вы хотите самостоятельно настроить все элементы по отдельности, — а это можно считать отличным способом обучения, — он подробно рассматривается в оставшейся части данного приложения.

Для начала убедитесь, что у вас есть все необходимое программное обеспечение. Ниже приведен его список:

- **PHP.** PHP предоставляет основу для разработки как веб-приложений, так и консольных программ. Его можно загрузить по адресу http://www.php.net/. Доступны исходные коды и двоичные версии для платформ Windows, UNIX и Mac OS X. Пользователи UNIX должны загрузить последний архив с исходным кодом, а пользователи Windows — последний двоичный релиз. На момент написания этой книги последней версией PHP была версия 5.3.1.

- **Apache.** Apache — это многофункциональный веб-сервер, который хорошо работает с PHP. Его можно бесплатно загрузить с сайта http://httpd.apache.org как в виде исходных кодов, так и в виде двоичных сборок для множества платформ. Пользователям UNIX следует загрузить последний архив с исходным кодом, а пользователям Windows — двоичный установщик, соответствующий их версии Windows. На момент написания книги последней версией сервера Apache была версия 2.2.14.
- **MySQL.** Сервер баз данных MySQL обеспечивает надежное и масштабируемое хранение и получение данных. Он доступен в виде исходных кодов и двоичных версий на сайте http://www.mysql.com/. Существуют двоичные дистрибутивы для Linux, Solaris, FreeBSD, Mac OS X, Windows, HP-UX, IBM AIX, SCO OpenUNIX и SGI Irix, а исходный код доступен для платформ Windows и UNIX. Есть две причины, по которым рекомендуется использовать двоичную версию: ее проще установить и она оптимизирована командой разработчиков MySQL для использования на различных платформах. На момент написания книги последней версией сервера баз данных MySQL была версия 5.1.43.

В дополнение к этим четырем основным компонентам пользователям UNIX также могут понадобиться некоторые вспомогательные библиотеки:

- Библиотека **libxml2**, доступная по адресу http://www.xmlsoft.org/.
- Библиотека **zlib**, доступная по адресу http://www.gzip.org/zlib/.
- Библиотека **gd**, доступная по адресу http://www.boutell.com/gd/.

И наконец, пользователям обеих платформ потребуется средство для распаковки архивов, способное работать с файлами TAR (Tape Archive) и GZ (GNU Zip). В UNIX утилиты tar и gzip обычно входят в состав операционной системы. В Windows хорошим инструментом для распаковки является WinZip, доступный по адресу http://www.winzip.com/.

ПРИМЕЧАНИЕ

Примеры из этой книги разрабатывались и тестировались на MySQL v5.1.30, Apache v2.2.14, PHP v5.3.1 и Zend Framework v1.9 и 1.10.

Установка и настройка программного обеспечения

После получения требуемых программ можно приступать к их установке и настройке взаимодействия между ними. В следующих разделах отмечены основные шаги, которые требуется предпринять на платформах Windows и UNIX.

ПРИМЕЧАНИЕ

Если вы используете рабочую станцию от Apple, инструкции по установке PHP в Mac OS X можно найти в руководстве к PHP по адресу http://www.php.net/manual/en/install.macosx.php.

Установка в UNIX

Процесс установки в UNIX состоит из нескольких независимых шагов: установка MySQL из двоичного дистрибутива; компиляция и установка PHP из дистрибутива с исходными файлами; компиляция и настройка Apache для корректной обработки запросов веб-страниц, использующих PHP. В следующих подразделах эти шаги описаны более подробно.

Установка MySQL

Для установки MySQL из двоичного дистрибутива выполните следующие действия:

1. Убедитесь, что вы вошли в систему как пользователь **root**.

   ```
   [user@host]# su - root
   ```

2. Извлеките содержимое двоичного архива MySQL в подходящий каталог — например, в **/usr/local/**.

   ```
   [root@host]# cd /usr/local
   [root@host]# tar -xzvf /tmp/mysql-5.1.30-linux-i686-glibc23.tar.gz
   ```

   Файлы MySQL должны распаковаться в каталог, имя которого соответствует формату mysql-версия-ОС-архитектура, — например, **mysql-5.1.30-linux-i686-glibc23**.

3. Для простоты использования задайте для каталога, созданного на шаге 2, более короткое имя, создав на том же уровне символьную ссылку с именем **mysql**, указывающую на этот каталог.

   ```
   [root@host]# ln -s mysql-5.1.30-linux-i686-glibc23 mysql
   ```

4. В целях безопасности никогда не запускайте сервер баз данных MySQL от имени суперпользователя. Для этого необходимо создать отдельных пользователя и группу **mysql**. Сделайте это с помощью команд **groupadd** и **useradd**, а затем измените владельца и группу каталога установки MySQL следующим образом:

   ```
   [root@host]# groupadd mysql
   [root@host]# useradd -g mysql mysql
   [root@host]# chown -R mysql /usr/local/mysql
   [root@host]# chgrp -R mysql /usr/local/mysql
   ```

5. Инициализируйте таблицы MySQL с помощью сценария инициализации **mysql_install_db**, входящего в состав дистрибутива.

   ```
   [root@host]# /usr/local/mysql/scripts/mysql_install_db --user=mysql
   ```

   На рис. А.1 показан пример вывода, который вы должны увидеть после выполнения приведенной команды.

   Как подсказывает представленный вывод, этот сценарий инициализации подготавливает и устанавливает различные базовые таблицы MySQL, а также устанавливает для MySQL стандартные права доступа.

6. Измените владельца двоичных файлов MySQL на **root**.

   ```
   [root@host]# chown -R root /usr/local/mysql
   ```

Убедитесь в том, что пользователь mysql, созданный в шаге 4, обладает правами на чтение и запись в каталог данных MySQL.

```
[root@host]# chown -R mysql /usr/local/mysql/data
```

7. Запустите сервер MySQL, вручную выполнив сценарий mysqld_safe.

```
[root@host]# /usr/local/mysql/bin/mysqld_safe --user=mysql &
```

MySQL должен без проблем запуститься.

```
192.168.1.5 - PuTTY
#
# mysql_install_db
Installing all prepared tables
Fill help tables

To start mysqld at boot time you have to copy support-files/mysql.server
to the right place for your system

PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
To do so, start the server, then issue the following commands:
/usr/bin/mysqladmin -u root password 'new-password'
/usr/bin/mysqladmin -u root -h production.securities.com password 'new-password'
See the manual for more instructions.

NOTE:  If you are upgrading from a MySQL <= 3.22.10 you should run
the /usr/bin/mysql_fix_privilege_tables. Otherwise you will not be
able to use the new GRANT command!

You can start the MySQL daemon with:
cd / ; /usr/bin/mysqld_safe &

You can test the MySQL daemon with the benchmarks in the 'sql-bench' directory:
cd sql-bench ; perl run-all-tests
```

Рис. А.1. Вывод сценария mysql_install_db

После успешного завершения установки и запуска сервера перейдите к разделу «Проверка MySQL», чтобы убедиться, что сервер работает должным образом.

Установка Apache и PHP

PHP может интегрироваться с веб-сервером Apache одним из двух способов: либо как динамический модуль, загружаемый веб-сервером во время выполнения, либо как статический модуль, который интегрируется в иерархию исходного кода Apache на этапе компиляции. У каждого варианта есть свои достоинства и недостатки:

- Установка PHP в виде динамического модуля облегчает последующие обновления PHP, поскольку вам требуется лишь перекомпилировать модуль PHP, не затрагивая остальную часть веб-сервера Apache. С другой стороны, производительность с динамически загружаемым модулем обычно ниже, чем со статическим, который более тесно интегрируется с сервером.
- Установка PHP в виде статического модуля повышает производительность, поскольку модуль добавляется непосредственно в иерархию исходных кодов

Apache. Однако столь тесная интеграция обладает существенным недостатком: если вы захотите обновить сборку PHP, вам придется повторно интегрировать новый модуль в исходный код Apache и перекомпилировать веб-сервер.

В этом разделе показано, как скомпилировать PHP в виде динамического модуля, загружаемого веб-сервером Apache во время выполнения.

1. Убедитесь, что вы вошли в систему как пользователь root.

```
[user@host]# su - root
```

2. Извлеките содержимое архива с исходным кодом Apache во временный каталог.

```
[root@host]# cd /tmp
[root@host]# tar -xzvf /tmp/httpd-2.2.14.tar.gz
```

3. Чтобы обеспечить динамическую загрузку PHP, сервер Apache должен быть скомпилирован с поддержкой Dynamic Shared Object (DSO, динамически загружаемый объект). Эта поддержка включается путем передачи сценарию configure параметра --enable-module=so:

```
[root@host]# cd /tmp/httpd-2.2.14
[root@host]# ./configure --prefix=/usr/local/apache
             --enable-module=so --enable-rewrite
```

4. Теперь скомпилируйте сервер, используя команду make, и установите его в систему с помощью команды make install.

```
[root@host]# make
[root@host]# make install
```

На рис. А.2 показан приблизительный вывод в процессе компиляции.
После этих действий Apache должен установиться в каталог /usr/local/apache/.

1. Теперь перейдите к компиляции и установке PHP. Начните с извлечения содержимого архива с исходным кодом во временный каталог.

```
[root@host]# cd /tmp
[root@host]# tar -xzvf /tmp/php-5.3.1.tar.gz
```

2. В процессе установки PHP этот шаг наиболее важен. На данном этапе осуществляется передача аргументов сценарию configure для настройки модуля PHP. Эти параметры командной строки определяют, какие из расширений PHP следует активировать, а также сообщают PHP местонахождение вспомогательных библиотек, необходимых для указанных расширений.

```
[root@host]# cd /tmp/php-5.3.1
[root@host]# ./configure --prefix=/usr/local/php
         --with-apxs2=/usr/local/apache/bin/apxs
        --with-zlib --with-gd --with-mysqli=mysqlnd
        --with-pdo-mysql=mysqlnd
```

Ниже приведено краткое описание значения каждого из аргументов:

- Аргумент --with-apxs2 сообщает PHP местонахождение сценария APXS (APache eXtenSion, расширение Apache) из состава Apache. Этот сценарий упрощает задачу сборки и установки загружаемых модулей Apache.
- Аргумент --with-zlib сообщает PHP о необходимости активации возможностей сжатия (zip), которые используются различными сервисами PHP.

- Аргумент **--with-gd** сообщает PHP о необходимости активации возможностей обработки изображений.
- Аргумент **--with-mysqli** активирует расширение PHP MySQLi и сообщает PHP о необходимости использования нового встроенного драйвера MySQL (mysqlnd) в качестве клиентской библиотеки MySQL.
- Аргумент **--with-pdo-mysql** активирует драйвер PHP MySQL PDO и сообщает PHP о необходимости использовать встроенный драйвер MySQL в качестве клиентской библиотеки MySQL.

3. Теперь скомпилируйте и установите PHP с помощью команд **make** и **make install**.

```
[root@host]# make
[root@host]# make install
```

На этом этапе также можно запустить **make test**, чтобы выполнить модульные тесты PHP и просмотреть возможные ошибки. Если все пойдет как надо, PHP будет установлен в каталог **/usr/local/php**.

4. Последним шагом в процессе установки будет настройка Apache для корректного распознавания запросов страниц PHP. Это можно сделать, открыв конфигурационный файл Apache, **httpd.conf** (его можно найти в подкаталоге **conf/** в каталоге установки Apache), в текстовом редакторе и добавив в него следующую строку:

```
AddType application/x-httpd-php .php
```

Сохраните изменения в файле. Также убедитесь, что в файле присутствует следующая строка:

```
LoadModule php5_module modules/libphp5.so
```

5. Запустите сервер Apache, вручную выполнив сценарий **apachectl**.

```
[root@host]# /usr/local/apache/bin/apachectl start
```

Apache должен без проблем запуститься.

Рис. А.2. Компиляция Apache

Как только установка будет успешно завершена, а сервер запущен, перейдите к разделу «Проверка PHP», чтобы удостовериться, что все работает должным образом.

Установка в Windows

Компиляция приложений в Windows — сложный процесс, особенно для начинающих разработчиков. С учетом этого пользователям Windows рекомендуется устанавливать и настраивать предварительно скомпилированные двоичные релизы MySQL, SQLite, PHP и Apache, вместо того чтобы пытаться скомпилировать их самостоятельно из исходного кода. Эти релизы можно загрузить с веб-сайтов, перечисленных в разделе «Получение программного обеспечения», и установить один за другим, как описано в следующих подразделах.

Установка MySQL

MySQL доступен как в форме исходных кодов, так и в двоичном варианте для 32-битных и 64-битных версий Microsoft Windows. Чаще всего вам потребуются дистрибутивы *Essentials* или *Complete*, которые содержат автоматический установщик для запуска MySQL за несколько минут.

Чтобы установить MySQL из двоичного дистрибутива, выполните следующие шаги:

1. Войдите в систему как администратор (если вы используете Windows XP/Vista/7).
2. Чтобы начать процесс установки, дважды щелкните на файле **mysql-*.msi**. Должно появиться окно приветствия.
3. Выберите требуемый тип установки.
4. Чаще всего будет достаточно варианта Typical (обычный); однако если вы любите изменять стандартные настройки или вам просто не хватает свободного места на диске, выберите вариант установки Custom (пользовательский) и определите, какие компоненты пакета следует установить.
5. Теперь должна начаться установка MySQL в вашу систему.
6. После ее завершения вы должны увидеть сообщение об успешной установке. На этом этапе у вас будет возможность запустить MySQL Server Instance Configuration Wizard (мастер настройки сервера MySQL), чтобы завершить настройку сервера. Выберите эту возможность, и вы увидите соответствующий экран приветствия.
7. Выберите тип конфигурации. В большинстве случаев будет достаточно варианта Standard Configuration (стандартная конфигурация).
8. Чтобы MySQL автоматически запускался и останавливался вместе с Windows, установите его как службу Windows (рис. А.3).
9. Введите пароль для учетной записи администратора MySQL (**root**) (рис. А.4).

Рис. А.3. Установка службы MySQL

Рис. А.4. Установка пароля администратора

Сервер будет настроен в соответствии с указанными параметрами и автоматически запущен. После выполнения всех необходимых действий вы увидите сообщение об успехе.

Теперь можно перейти к тестированию сервера, описанному в разделе «Проверка MySQL», чтобы убедиться, что все работает должным образом.

Установка Apache

Следующим шагом после установки MySQL будет установка веб-сервера Apache. В Windows это можно сделать с помощью нескольких щелчков мыши, так же как и при установке MySQL.

1. Чтобы начать процесс установки, дважды щелкните на установщике Apache. Вы должны увидеть окно приветствия (рис. А.5).
2. Прочитайте лицензионное соглашение и согласитесь с его условиями, чтобы продолжить установку.
3. Прочитайте описание и переходите к вводу основной информации о сервере, а также адреса электронной почты, отображаемого на страницах с ошибками.
4. Выберите требуемый тип установки. Если вы хотите указать, какие компоненты пакета следует устанавливать, можете выбрать вариант Custom (пользовательский).
5. Выберите каталог для установки Apache — например, **C:\Program Files\Apache**.
6. Apache должен приступить к установке в указанный каталог. Процесс установки занимает несколько минут, поэтому самое время выпить чашечку кофе.
7. После завершения установки установщик Apache отобразит сообщение об успехе и запустит веб-сервер.

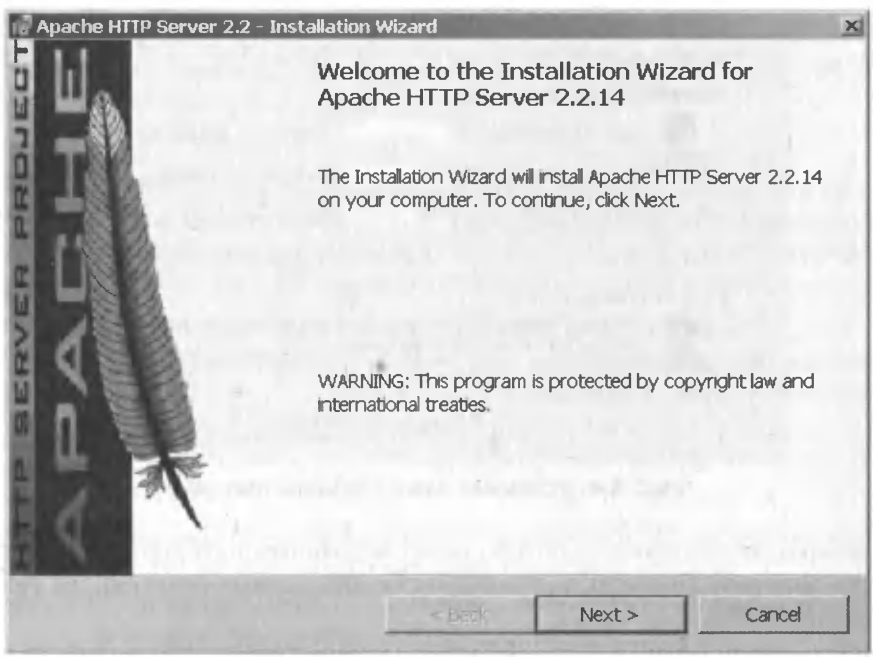


Рис. А.5. Начало установки Apache в Windows

Установка PHP

Существуют две версии двоичного релиза PHP для Windows: ZIP-архив, который содержит все стандартные расширения PHP и требует ручной установки, и автоматизированная версия в формате Windows Installer, содержащая только основные двоичные файлы PHP без дополнительных расширений. В этом разделе описан процесс установки из ZIP-архива.

1. Войдите в систему как администратор (если вы используете Windows XP/Vista/7) и извлеките содержимое архива в какой-либо каталог в вашей системе — например, **C:\php**. После извлечения содержимое этого каталога должно быть похоже на представленное на рис. А.6.
2. Затем переименуйте файл **php.ini-recommended** в каталоге установки PHP в **php.ini**. Этот файл содержит конфигурационную информацию для PHP, которую можно использовать для изменения параметров его работы. Чтобы узнать больше о доступных настройках, прочитайте комментарии в этом файле.
3. Найдите в файле **php.ini** следующую строку:

   ```
   extension_dir = "./"
   ```

   Измените ее на

   ```
   extension_dir = "c:\php\ext\"
   ```

   Это изменение сообщит PHP, где следует искать расширения, поставляемые в составе пакета. Не забудьте заменить путь «**c:\php**» фактическим местоположением вашей установки PHP.
4. Теперь найдите следующие строки и удалите точки с запятыми в их начале (если они есть), приведя их к следующему виду:

   ```
   extension=php_mysqli.dll
   extension=php_pdo_sqlite.dll
   extension=php_pdo_mysql.dll
   extension=php_zip.dll
   extension=php_gd2.dll
   ```

   Это приведет к активации расширений MySQL, GD, ip и PDO.
5. Откройте конфигурационный файл Apache, **httpd.conf** (который находится в подкаталоге **conf/** каталога установки PHP), в текстовом редакторе и добавьте в него следующие строки:

   ```
   AddType application/x-httpd-php .php
   LoadModule php5_module modules/libphp5.so
   ```

   Эти строки сообщают Apache, каким образом следует обрабатывать сценарии PHP и где искать конфигурационный файл **php.ini**.
6. После установки сервера Apache он добавляет себя в меню **Пуск**. Используйте эту группу пунктов меню для остановки и перезапуска сервера (рис. А.7).

Теперь PHP установлен и настроен для работы с Apache. Для его проверки перейдите к разделу «Проверка PHP».

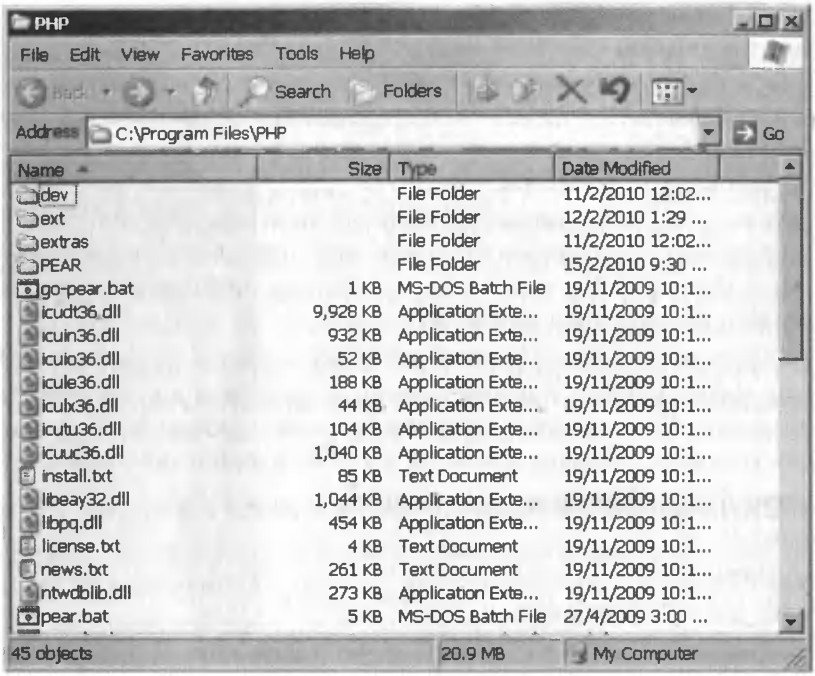


Рис. А.6. Структура каталогов, получившаяся в результате распаковки двоичного дистрибутива PHP для Windows

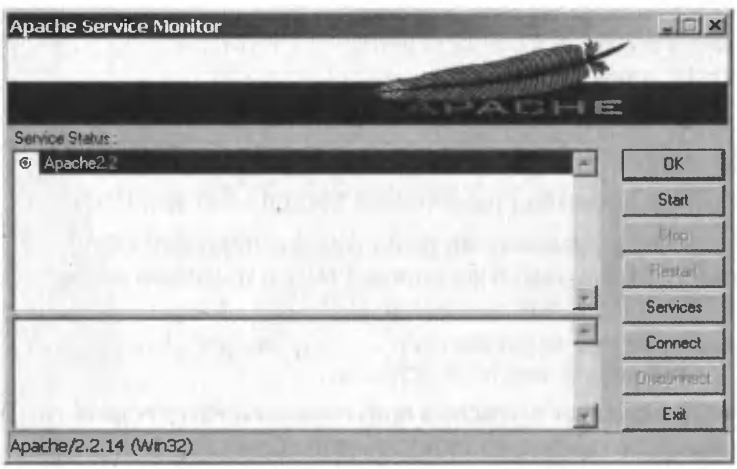


Рис. А.7. Управление веб-сервером Apache в Windows

Добавление в PHP поддержки memcached. В главе 12 этой книги используется механизм кэширования в оперативной памяти memcached. Чтобы добавить в PHP

поддержку этого механизма, требуется выполнить несколько дополнительных шагов. Вкратце они заключаются в следующем:

1. Загрузите, скомпилируйте и установите библиотеки memcached и libevent с сайтов http://tangent.org/552/libmemcached.html и http://www.monkey.org/~provos/libevent/ (только для *NIX).
2. Установите сервер memcached с сайта http://memcached.org/ (исходные файлы для *NIX) или http://code.jellycan.com/memcached/ (двоичный файл для Windows).
3. Установите расширение PHP memcache с сайта PECL по адресу http://pecl.php.net/package/memcache (исходные файлы для *NIX) или http://downloads.php.net/pierre/ (двоичный файл для Windows).
4. Включите расширение PHP memcache в конфигурационном файле PHP и перезапустите веб-сервер, чтобы применить изменения.

Проверка программного обеспечения

После установки всех программных компонентов используйте следующие разделы, чтобы убедиться, что они правильно настроены и включены.

Проверка MySQL

После успешной установки MySQL, инициализации базовых таблиц и запуска сервера вы можете убедиться, что все работает правильно, проведя несколько простых тестов.

Для начала запустите клиент командной строки MySQL, перейдя в подкаталог bin/ в каталоге установки MySQL и выполнив следующую команду:

```
prompt# mysql -u root
```

Должно появиться следующее приглашение:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 26288
Server version: 5.1.30-community MySQL Community Edition (GPL)
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

В данный момент вы подключены к серверу MySQL и можете начать выполнение команд или запросов SQL, чтобы убедиться, что сервер работает должным образом. Ниже приведены несколько примеров, а также их вывод:

```
mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| mysql |
| test |
```

```
+----------+
2 rows in set (0.13 sec)
mysql> SELECT COUNT(*) FROM mysql.user;
+----------+
| count(*) |
+----------+
| 1 |
+----------+
1 row in set (0.00 sec)
```

Если вы видите похожий вывод, ваша установка MySQL работает должным образом. Чтобы вернуться к командному интерпретатору, выйдите из клиента командной строки MySQL, выполнив команду **exit**.

Проверка PHP

После того как вы успешно установили PHP в виде модуля к Apache, следует проверить его, чтобы убедиться, что веб-сервер может распознавать и корректно обрабатывать сценарии PHP.

Для выполнения этого теста создайте в любом текстовом редакторе сценарий PHP следующего содержания:

```
<?php
phpinfo();
?>
```

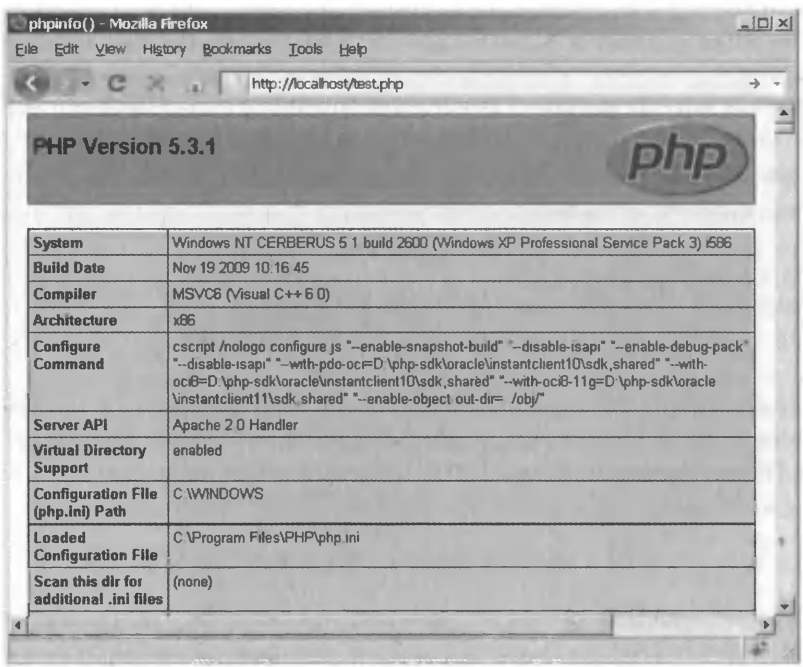


Рис. А.8. Просмотр вывода команды phpinfo()

Сохраните его в файл test.php в корневом каталоге документов веб-сервера (подкаталог htdocs/ в каталоге установки Apache) и направьте браузер по адресу http://localhost/test.php. Вы должны увидеть страницу с информацией о сборке PHP (рис. А.8).

Внимательно изучите список расширений, чтобы удостовериться, что включены расширения MySQL, GD, SimpleXML, Zip и memcached (необязательно). Если какие-то из расширений отсутствуют, пересмотрите процедуру установки, а также документацию по установке, поставляемую с программным обеспечением, чтобы выяснить, что пошло не так.

Установка пароля суперпользователя MySQL

После первой установки MySQL доступ к серверу баз данных есть только у администратора MySQL, также известного как 'root'. По умолчанию этому пользователю присвоен пустой пароль, что обычно считается *Плохой Идеей*. Поэтому вы должны как можно скорее исправить ситуацию, установив для него пароль с помощью утилиты mysqladmin из дистрибутива MySQL. В UNIX используется следующий синтаксис:

```
[root@host]# /usr/local/mysql/bin/mysqladmin -u root password 'new-password'
```

В Windows для этих целей можно использовать MySQL Server Instance Configuration Wizard, который позволяет устанавливать или сбрасывать пароль администратора MySQL (подробности см. разделе в «Установка в Windows»).

Новый пароль применяется сразу, без необходимости перезапускать сервер.

Выводы

Являясь популярными приложениями с открытым исходным кодом, MySQL, Apache и PHP доступны для широкого разнообразия платформ и архитектур как в двоичной форме, так и в виде исходных кодов. В данном приложении был показан процесс установки и настройки этих программных компонентов для создания окружения разработки на PHP для двух наиболее распространенных платформ: UNIX и Windows. Здесь также было показано, как настроить систему для автоматического запуска этих компонентов при загрузке, и приведены несколько простых советов касательно безопасности MySQL.

Чтобы узнать больше о процессах установки, кратко описанных в этом приложении, или получить подробные советы и помощь в решении проблем, посетите следующие страницы:

- Примечания по установке MySQL: http://dev.mysql.com/doc/refman/5.1/en/installing-binary.html.
- Основные указания по компиляции Apache в UNIX: http://httpd.apache.org/docs/2.2/install.html.
- Примечания по установке двоичных дистрибутивов Apache для Windows: http://httpd.apache.org/docs/2.2/platform/windows.html.

- Инструкции по установке PHP в Windows: http://www.php.net/manual/en/install.windows.php.
- Инструкции по установке PHP в UNIX: http://www.php.net/manual/en/install.unix.php.
- Инструкции по установке PHP в Mac OS X: http://www.php.net/manual/en/install.macosx.php.

Викрам Васвани

Zend Framework: разработка веб-приложений на PHP

Перевел с английского Р. Тетерин

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Кривцов*
Ведущий редактор	*Ю. Сергиенко*
Художественный редактор	*Л. Адуевская*
Корректор	*Н. Викторова*
Верстка	*Е. Егорова*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 30.11.11. Формат 70x100/16. Усл. п. л. 34,830. Тираж 1500. Заказ 26925.
Отпечатано по технологии CtP в ОАО «Первая Образцовая типография», обособленное подразделение «Печатный двор».
197110, Санкт-Петербург, Чкаловский пр., 15.

САЛД

САНКТ-ПЕТЕРБУРГСКАЯ АНТИВИРУСНАЯ ЛАБОРАТОРИЯ ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные программные продукты

ZEND FRAMEWORK Разработка веб-приложений на PHP

Zend Framework — это, говоря словами Эрнеста Хемингуэя, «праздник, который всегда с тобой». Задуманный и реализованный как надежная и функциональная библиотека компонентов для разработчиков на PHP, он позволяет быстро и эффективно решать множество распространенных задач, связанных с разработкой приложений, таких как создание и проверка форм ввода, обработка XML, генерация динамических меню, разбивка данных на страницы, работа с веб-сервисами и многое-многое другое!

Тема: Программирование для Интернета

Уровень пользователя: начинающий/опытный

ISBN: 978-5-459-00826-5

ПИТЕР

Заказ книг:
197198, Санкт-Петербург, а/я 127
тел.: (812) 703-73-74, postbook@piter.com
61093, Харьков-93, а/я 9130
тел.: (057) 758-41-45, 751-10-02, piter@kharkov.piter.com
www.piter.com — вся информация о книгах и веб-магазин

Mc Graw Hill