THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Practical CakePHP Projects

Learn how to create dynamic web sites with CakePHP by working through practical examples from the experts.

Kai Chan and John Omokore
with Richard K. Miller

Apress®

Practical CakePHP Projects

Kai Chan and John Omokore
with Richard K. Miller

Apress®

Practical CakePHP Projects

Copyright © 2009 by Kai Chan and John Omokore with Richard K. Miller

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1578-3

ISBN-13 (electronic): 978-1-4302-1579-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

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

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editor: Steve Anglin
Technical Reviewer: David Golding
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Richard Dal Porto
Copy Editor: Marilyn Smith
Associate Production Director: Kari Brooks-Copony
Production Editor: Candace English
Compositor: Patrick Cunningham
Proofreader: Martha Whitt
Indexer: Brenda Miller
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com`.

For Rita
—Kai Chan

For Comfort
—John Omokore

For Marian
—Richard K. Miller

Contents at a Glance

About the Authors xiii
About the Technical Reviewer xiv
Acknowledgments xv
Introduction xvii

CHAPTER 1 Cake Fundamentals 1
CHAPTER 2 Blogging 29
CHAPTER 3 E-Commerce 47
CHAPTER 4 A Message Forum Web Service 85
CHAPTER 5 Google Maps and the Traveling Salesman 131
CHAPTER 6 Mashing Twitter with the Google Translator 173
CHAPTER 7 Unit Testing and Web Testing 213
CHAPTER 8 A Cake Control Panel 237
CHAPTER 9 Translating Stories 271
CHAPTER 10 Adding Automagic Fields 307
CHAPTER 11 Cake Tags 317
CHAPTER 12 Dynamic Data Fields 329
CHAPTER 13 Captcha 359

INDEX 371

Contents

About the Authors . . . xiii
About the Technical Reviewer . . . xiv
Acknowledgments . . . xv
Introduction . . . xvii

CHAPTER 1 Cake Fundamentals . . . 1

Cake Features . . . 1
The Ingredients of Cake . . . 2
The Model-View-Controller Design Pattern . . . 2
Rapid Application Development . . . 2
PHP 4+ . . . 3
Object-Oriented Programming . . . 3
Dissecting Cake . . . 5
Cake's Directory Structure . . . 5
The Cake Naming Conventions . . . 6
Models . . . 8
Model Creation . . . 8
Data Validation . . . 13
Views . . . 15
Controllers . . . 18
Cake Components . . . 19
Helpers . . . 21
Plugins . . . 22
Vendors . . . 25
Summary . . . 27

CHAPTER 2 Blogging . . . 29

Creating the Database . . . 29
Reviewing the Application Structure . . . 31
Creating the Post Model . . . 31

Creating the Posts Controller 32
Listing the Posts 32
Adding a Post 35
Updating a Post 38
Unpublishing a Post 40
Publishing a Post 41
Deleting a Post 42
Creating an RSS Feed 43
Summary 46

CHAPTER 3 **E-Commerce** 47

The Online Shop Layout 47
Two Site Layouts 48
Layout of the Main Content 48
The User Journey 49
Setting Up the Shop Database 50
Interacting with the Online Shop Database 52
The Category Model 53
The Categories Controller 56
The Product Model 59
The Products Controller 59
The Cart Model 63
Handling User Requests 67
The AppController Class 67
The Home Page 69
The Carts Controller 69
The Order Model 75
The Google Checkout Button 79
The PayPal Submit Button 82
Summary 83

CHAPTER 4 **A Message Forum Web Service** 85

Our Take on Web Services 85
Web Service Elements 85
REST and HTTP 87
Result Return Formats 88
Application Requirements 88
Threads and Posts 89
Web Service Requests 90
Layout 91

Application Structure 91
JSON Web Services 92
Our Application Controller 94
Fetch a Message 97
Fetch Several Messages 100
Fetch the Threads 105
Post Messages 110
Process a Message 116
Process a Search Request 120
Writing the API Documentation 125
Summary 129

CHAPTER 5 Google Maps and the Traveling Salesman 131

Hello Map! 131
Google Maps Explained 133
Geocoding 133
Google Map Events 134
Map Interface Elements 134
Overlays 134
Driving Directions 135
Application Requirements 136
Application Structure 138
Cake Models 140
The Interface 141
The Global Layout 141
Home Page 143
Travel Mappr Manager 146
Finding Locations 148
The Traveling Salesman Algorithm 150
Plotting the Journey 156
Journey Data 160
Saving a Journey 160
Saving Tags 167
Retrieving and Editing a Journey 168
Viewing a Journey 169
Summary 172

CHAPTER 6 **Mashing Twitter with the Google Translator** 173

The Twitter API 173
The Google Ajax Language API 175
Application Requirements 176
Application Structure 178
Cake Models 181
Internationalization and Localization 184
Web Services 186
Caching 188
Caching Views 189
Caching Models 189
Caching Twitter and Google Translations 189
Caching and the Application Layout 190
Changing Languages 190
Changing Locales 191
The Controllers 192
The Twittertwister Controller 192
The TwitterRequest Controller 194
The TwitterStatus Controller 196
The AppController 209
Summary 211

CHAPTER 7 **Unit Testing and Web Testing** 213

Getting Programming Done 213
Our Case Study: An App Like In/Out 214
Creating the Application 214
Adding Username Validation 219
Using Cake's Unit Testing Framework 221
Installing SimpleTest 222
Creating Your Own Unit Tests 224
Using Assert Methods 231
Testing the Entire MVC System 232
Web Testing 232
Creating Web Tests 232
Web Testing Any Application 234
Test-Driven Development 235
Summary 235

CHAPTER 8 **A Cake Control Panel** 237

Application Requirements 237
The Authentication and ACL Components 238
The Authentication Component 238
The Access Control List Component 238
Component Setup 241
Control Panel Application Controllers 242
The Control Panel Controller 242
The Actions Controller 244
The Groups Controller 251
The Users Controller 259
Testing the Control Panel 266
Summary 270

CHAPTER 9 **Translating Stories** 271

Application Structure 271
The Translate Behavior 272
Stories 273
Baking Cake 273
Adding Stories 277
Administering Stories 279
Translating Stories 279
Viewing Stories 282
Deleting Stories 284
Listing Stories 284
Translation Pagination 288
Locale and Language Selection 292
Setting Locale by Browser 292
Setting Locale by Language Code 293
Setting Locale by Hand 293
User Authentication 293
Logging In 294
Logging Out 296
Baked Code 297
Summary 306

CHAPTER 10 **Adding Automagic Fields** 307

Cake's Built-in Magic Fields 307
Writing a Custom Behavior. 308
Building Custom Magic Fields 309
Access Data Field. 310
Record Order Data Field 311
Locking Data Field 313
Summary. 315

CHAPTER 11 **Cake Tags** 317

Content and Data Separation. 317
View Template 319
Cake View Class Extension 321
Cake Plugins. 323
Yahoo! Maps. 325
Summary. 328

CHAPTER 12 **Dynamic Data Fields** 329

Traditional Product Searching 329
The Dynamic Data Approach 330
Considerations for Using the Dynamic Data Approach 331
The Product Database Design 332
Baking for This Application. 337
Building the Product Search Feature 338
Creating the Product Search Form 338
Processing the Search 347
Adding a Product 353
Creating Table Entries. 353
Entering Product Data 354
Summary. 358

CHAPTER 13 **Captcha** 359

Captcha Implementations. 359
Captcha Types 360
ASCII Art Captcha. 361
A Captcha Component 362
The ASCII Art Component Class. 362
The Captcha Controller 365
Summary. 369

INDEX 371

About the Authors

KAI CHAN started his computing career in the late 1980s. His current interests include programming methodology, the Semantic Web, data visualization, and enterprise systems. Kai holds a Computer Science bachelor's degree and a master's degree in Computer Graphics. He is a cofounder of the Azzian MVC CMS framework. Together with John Omokore and others, he runs a software and training company in London, specializing in various large-scale projects, from SAP to e-commerce web sites. When he has a spare moment, he likes tennis, squash, and long-distance running.

JOHN OMOKORE is a developer, technical consultant, writer, and trainer. John has programming experience in many technologies, including Linux, PHP, MySQL, and Ajax. He has worked on market research data analysis, database development, and related systems. He received his bachelor's degree in Mathematics and is pursuing a postgraduate degree in software engineering at Oxford University in England. John provides consulting and web development services to corporate organizations around the world. He's a cofounder of AlternativeBrains and the Azzian MVC CMS framework and sits on the board of many companies. John lives outside London with his wife, two children, and some animals. His career interests include open source scripting languages, OOP programming, and the use of SAP in large-scale industries (chiefly oil and gas). When not scripting, he enjoys playing chess and squash, visiting the gym, and a bit of socializing.

RICHARD K. MILLER graduated from Brigham Young University with a degree in Business Management but has been interested in technology since he began computer programming at age 10. His experience includes web programming, Internet marketing, and new media strategies. He is the developer of several MediaWiki extensions and WordPress plugins, including the widely used What Would Seth Godin Do plugin.

About the Technical Reviewer

DAVID GOLDING began developing web sites in 1999 and first started using CakePHP on a bet he couldn't complete a web application in five minutes. He is the author of *Beginning CakePHP: From Novice to Professional* (Apress, 2008) and has taught CakePHP even while it was still in early stages of development. David has a degree in European Studies from Brigham Young University and continues work in religious studies and history as a graduate student at Claremont Graduate University. He lives with his wife, Camille, and his son, Kenny, in Southern California.

Acknowledgments

When we first decided to write this book, we really didn't think it would be that difficult a task. After all, we've been coding and writing documentation for years and years. Now having written the book, we can honestly say it has been one of the hardest projects we've done since we wrote our first-ever Hello World program. As such, with tears streaming from our eyes, we would wholeheartedly like to thank all the people involved. It all sounds like a cliché, but it's all true. Thank you to the team at Apress, the Cake Software Foundation, colleagues, friends, families, and neighbors. In no particular order, we would like to thank them individually. They are Steve Anglin, Richard Dal Porto, Matt Wade, Marilyn Smith, Joohn Choe, David Golding, Nancy Wright, Richard K. Miller, Rita Woo, Terry Wells, Dan Jackson, Candace English, and God.

Kai Chan and John Omokore

Thank you to Kai Chan and John Omokore for allowing me to take part in this book. I've enjoyed working with them and the entire Apress team. Thanks to David Golding for getting me involved. I'm thankful for good parents, family, friends, and colleagues, and to God.

Richard K. Miller

Introduction

First off, thank you for picking up this book. Whether you are standing in a bookshop or reading this at home, we assume you probably have a strong interest in developing web sites. In the past few years, the number of web site frameworks has increased dramatically. This is especially true for PHP-based frameworks. Many people have chosen to adopt CakePHP (Cake, for short) for various reasons, such as these:

- PHP programmers are widely available. Most projects have tight deadlines, and you want team members who can quickly pick up a new piece of technology.
- CakePHP is easy to learn. You want a powerful tool that you can easily master.
- CakePHP has good support. Developers frequently post and reply to messages on the Cake forum. And there are always some good discussions happening on the Cake IRC. (To see for yourself, simply download mIRC from `http://www.mirc.com/`, connect to the server `irc.freenode.net`, and join the `#cakephp` channel.)

When you are developing a site using Cake, you often find yourself trawling through tutorials online to see how things are done. We've done that ourselves many times. However, despite the power of the Internet, we still like to look through books. And we think you will find this book a great help in your Cake development endeavors, in addition to all of the material available online.

Most of the applications in this book have been written as a result of some real-world development we have done in the past. We focus on projects that we think are relevant to the future of web development.

Let's take mashups, for example. We should all take an interest in this ever-expanding area of web development. We can honestly say that any successful online web site in the future will need to easily communicate with other applications. Application designers will need to bear this in mind. Matters such as search engine optimization need to be built into the application itself. Cake allows us to think in terms of the high-level architecture instead of the nuts and bolts of a web application.

Who Should Read This Book

Practical CakePHP is a book mainly for developers. To get the most from it, you should be comfortable with a number of web technologies and programming concepts. These include PHP, SQL, HTML, JavaScript, object-oriented programming, and design patterns, as well as the general principles of web development. If you are at the forefront of web development, then this book is for you!

If our book sounds a little too advanced for you, we recommend that you do some preliminary reading. We suggest the following books:

- *Beginning PHP and MySQL: From Novice to Professional, Third Edition,* by W. Jason Gilmore (Apress, 2008)
- *Beginning CakePHP: From Novice to Professional* by David Golding (Apress, 2008)

How This Book Is Organized

Each chapter in this book has been chosen so it will cover the core features in Cake, plus some of the minor features as well. The following is a rough breakdown of what each chapter includes.

- Chapter 1, "Cake Fundamentals," gives you an introduction to Cake. If you are new to the CakePHP framework, this is the place to start.
- Chapter 2, "Blogging," provides you with a simple blogging application. It's perfect for beginners who want to know what a Cake application looks like. If there are two chapters in the book that need to be read in sequence, they are Chapters 1 and 2.
- Chapter 3, "E-Commerce," gives you greater insight into the way Cake is used in a common application. We walk through implementing an online shop using the Cake framework.
- Chapter 4, "A Message Forum Web Service," covers the development of a web services API. We guide you through creating a clean API, so any third party can access your application using standard protocols.
- Chapter 5, "Google Maps and the Traveling Salesman," shows you how the Google Maps API is used with Cake. One of the main features of this chapter's application relates to the classic traveling salesman problem: a salesman needs to visit a number of cities only once and return to where he started.
- Chapter 6, "Mashing Twitter with the Google Translator," emphasizes the importance of web services in modern web application development. In true Web 2.0 and Cake fashion, this chapter's application mashes the Google Ajax Language API with the Twitter API to provide automatic translation of Twitter messages.
- Chapter 7, "Unit Testing and Web Testing," covers one of the hottest topics among web professionals. Cake 1.2 devotes a large section to testing, and this chapter shows you how to take advantage of Cake's integrated unit testing features.
- Chapter 8, "A Cake Control Panel," covers Cake's access control lists and security features. We develop a web-based front end that allows administrators to manage user security.

- Chapter 9, “Translating Stories,” provides you with the knowledge to tackle Cake’s internationalization and localization features. We develop an application in which news stories are available in other languages, with an administration area where translators can translate stories from a base language to another language.
- Chapter 10, “Adding Automagic Fields,” demonstrates extending Cake’s use of automagic fields like `created`, `modified`, and `title`. We create three new automagic fields.
- Chapter 11, “Cake Tags,” shows you our take on an established technology where XML tags are used as a wrapper to coding logic. Using Cake, we develop our own HTML-based tags to display two Yahoo maps.
- Chapter 12, “Dynamic Data Fields,” extends the e-commerce chapter with a special product-filtering technique. We take a dynamic data approach to product searches.
- Chapter 13, “Captcha,” shows how ASCII Art can be used as a Captcha test. In this chapter’s project, the Captcha test is housed in a Cake component so it can be used by other applications.

How to Contact the Authors

The authors can be contacted as follows:

- Kai Chan can be contacted at `kai.chan@edgeable.com`.
- John Omokore can be contacted at `john@omokore.com`.
- Richard K. Miller can be contacted at `richard@richardkmiller.com`.

CHAPTER 1

Cake Fundamentals

Using a framework of some sort has now become the order of the day for building large-scale web applications. Organizations have found that using an in-house framework for web projects enhances code reuse, scalability, quick project turnarounds, and security.

New and evolving frameworks provide rapid application development tools to promote the adoption of particular programming languages. Many frameworks derived from PHP have been popular with programmers in the open source community. CakePHP—Cake for short—is currently one of the fastest-growing rapid application development frameworks. When you are developing large web applications or creating components that you will reuse in many applications, you'll find Cake to be a great help.

In this chapter, we'll highlight some of the concepts, technologies, and tools that Cake relies on, including the PHP scripting language, the Model-View-Controller design pattern, and object-oriented programming techniques. We will also outline the default folder structures and naming conventions and introduce some Cake best practices. And, of course, we'll demonstrate how to write some Cake code.

This chapter will serve as a quick reference that will provide you with a solid foundation on which to build your knowledge of the framework throughout the rest of the book.

Cake Features

Why should you use Cake when there are so many other frameworks in town? There is a number of good reasons for the popularity of Cake PHP. It has a short learning curve in comparison to other frameworks, because Cake is easy to use and understand. Also, because there are so many PHP programmers, Cake has a large community. New users can find many projects to refer to and use.

Here are some features of Cake that make web application development with it easy and fast:

- It uses the Model-View-Controller (MVC) framework for PHP.
- Its database connectivity support includes MySQL and PostgreSQL, as well as many other database platforms.
- Cake is easy to install on most platforms, including Unix and Windows.
- Its MIT license is more flexible than other licenses.
- It uses easy and flexible templating (which allows PHP syntax, with helpers).

- Cake has view helpers to assist in the insertion of often-repeated snippets of HTML and forms code, Ajax, JavaScript, and so on.
- It has components for handling e-mail, authentication, access control, localization, security, sessions, and request handling.
- Cake provides utility classes to manipulate resources such as sets, files, folders, XML, and many others.
- Your URLs are optimized for search engines.

Note For a complete and up-to-date list of Cake features; see the official web site at `http://cakephp.org`. You can also find many discussions regarding how Cake compares with other frameworks, such as Ruby on Rails, symfony, Zend Framework, and CodeIgniter. For a comparison of Cake with the aforementioned frameworks, check `http://frinity.blogspot.com/2008/06/why-choose-cakephp-over-other.html`.

The Ingredients of Cake

In this section, we'll delve into the core concepts and technologies employed by Cake, starting with the MVC design pattern.

The Model-View-Controller Design Pattern

Cake supports the MVC design pattern, which aims to modularize an application into three parts:

- The *model* represents the data for the application.
- The *view* represents the presentation.
- The *controller* ties the model and view together and deals with user input.

Familiarity with the MVC pattern is a plus, but this book does not assume you have any prior knowledge of MVC. This chapter covers how Cake employs the MVC concept.

Rapid Application Development

Along with MVC, Cake took on the philosophy of rapid application development (RAD), sometimes also known as rapid prototyping. RAD is basically a method of decreasing the time taken to design software systems by using many prebuilt skeleton structures. This provides developers with many advantages, including easier maintenance, code reuse, more efficient teamwork, and quick project turnaround. RAD also provides the ability to make rapid changes based on client feedback, decreasing the dangers of feature creep.

Additionally, you can find a lot of off-the-shelf open source code, which you can easily plug into your Cake applications. A great place to start is `http://bakery.cakephp.org`.

PHP 4+

PHP 4+ refers to PHP version 4 and above. PHP has become one of the most important server-side scripting languages on the Web. It is currently a predominant language for the development of web applications. It provides web developers the functionalities to quickly create dynamic web applications. PHP has come a long way since PHP 3 was first introduced more than a decade ago.

The adoption of the Cake framework assumes knowledge of PHP 4. The official PHP manual, at `http://www.php.net`, provides a complete reference on PHP.

■ **Note** A common problem faced in life with a new adventure is where to go for the right information in order to avoid the mistakes of predecessors. If you are just starting out with PHP, you can refer to the many online PHP forums and repositories, such as the popular PEAR library and the ever-growing `http://www.phpclasses.org` web site.

Object-Oriented Programming

Object-oriented programming (OOP) can be described as a method of implementation in which the parts of a program are organized as a collection of objects, each of which represents an instance of a class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. For example, a `Dog` object `says() 'woof woof'`, while a `Cat` object `says() 'meow meow'`, and they both inherit `says()` from the `Pets` class.

The Cake framework supports the three key principles of object-oriented development: encapsulation, inheritance, and polymorphism.

For the simple magic called *encapsulation*, Cake's implementation of one object is protected, or hidden away, from another object to eliminate interference. However, there must be some interaction with other objects in the application, or the object is useless. As in most OOP applications, an object in the Cake framework provides an *interface* to another object to enable this interaction. Listing 1-1 shows the default database configuration class, called `DATABASE_CONFIG`, which encapsulates `$default` and `$shop` database connection arrays.

Listing 1-1. *The Cake Database Configuration Class*

```
class DATABASE_CONFIG {
    var $default = array(
            'driver' => 'mysql',
            'persistent' => 'true',
            'host' => 'localhost',
            'login' => 'admin',
            'password' => 'superadmin',
            'database' => 'userdb',
            'prefix' => ''
```

```
    var $shop = array(
            'driver' => 'mysql',
            'persistent' => 'true',
            'host' => 'localhost',
            'login' => 'user',
            'password' => 'userme',
            'database' => 'shopdb',
            'prefix' => 'sp'
);
```

By default, Cake internally interfaces with the `$default` connection database. It uses its array parameters for its default database connection unless you explicitly specify a different database connection by assigning the `$useDbConfig = '$shop'` property in a model class. This explicit interface will enable some interaction with the tables in the `shop` database.

Cake's support for *inheritance* cannot be overemphasized. It wraps a lot of database manipulation and other utility functions in its default classes in a manner that enables an object to take on the functions of another object and extend or tailor those functions so you don't repeat the same code. We consider this act of charity as one of the greatest benefits to developers, as it undoubtedly ensures fast application development. Therefore, you need to spend some time sharpening your knives by reading a Cake cheat map or its online API (`http://api.cakephp.org`) to understand what your objects will inherit.

In a controller genealogy, user-defined controller objects inherit from the `AppController` object. The `AppController` inherits from `Controller` object, which extends the `Object` class. A controller class can be derived from the `AppController` class, as shown in Listing 1-2. (Controllers are discussed in more detail in the upcoming sections about Cake models, views, and controllers.)

Listing 1-2. *The Application Controller Class*

```
class ProductsController extends AppController {
    function beforeFilter() {
    }
}
```

This default class contains the `beforeFilter()` method, which can be overridden in any class that extends the `AppController` class, such as a user-defined controller class. In Listing 1-2, `ProductsController` extends the `AppController` class.

And lastly, Cake implements *polymorphism* and ensures that functions within an object can behave differently depending on the input. It basically creates the ability to respond to the same function call in many different ways.

The Cake framework creates many reusable objects. You can use these objects without knowing their internal workings. This is one of the key benefits of using Cake.

■ **Note** For more information about OOP in relation to PHP, refer to the PHP manual at `http://www.php.net/oop5`.

Dissecting Cake

Before you start baking a Cake application, you will need to download the Cake framework from `cakephp.org` and install it on your computer. Remember that Cake is based on the PHP scripting language, so you need to have PHP up and running first. If you will be using information stored in a database, you will need to install the database engine. All our examples assume the MySQL database.

Cake's Directory Structure

When you unpack Cake, you will find the following main folder structures:

- `app`: Contains files and folders for your application. The `app` folder is your development folder, where your application-specific folders and files reside.
- `cake`: Contains core Cake libraries. The `cake` folder contains the core libraries for CakePHP. You should not touch these libraries unless you really know what you are doing.
- `docs`: Contains Cake document files such as the read me, copyright, and change log text files. You can store your own documentation in this folder.
- `vendors`: Contains third-party code. The `vendors` folder can contain third-party libraries, such as the Swift Mailer package for sending e-mail messages.

Separating the default Cake core library folder from the application folder makes it possible for you to have many different applications sharing a single Cake installation. With this folder structure, you can easily upgrade your existing version of Cake without affecting any applications you have written. Table 1-1 details Cake's default folder structure.

Table 1-1. *The Cake Default Folder Structure*

Directory	Description
`app/`	The parent folder for your application
`config/`	Contains configuration files for global structures such database connections, security, and access control
`controllers/`	Contains your application controllers files (e.g., `user_controller.php`)
`components/`	Contains your user-defined component files
`/index.php`	Allows you to deploy Cake with `/app` as the `DocumentRoot`
`locale/`	Contains locale files that deal with internationalization
`models/`	Contains the model files
`plugins/`	Contains the plugin files
`tests/`	Contains the test folders and files
`tmp/`	Used for caches and logs
`vendors/`	Contains third-party libraries

Continued

Table 1-1. *Continued*

Directory			Description
	views/		Contains view folders and files for presentation (e.g., the .ctp files)
		elements/	Elements, which are bits of views, go here
		errors/	Custom error pages
		helpers/	Helpers
		layouts/	Application layout files
	webroot/		The DocumentRoot for the application
		css/	Contains the application style sheet files
		files/	Contains any files
		img/	Contains graphics
		js/	Contains JavaScript files
cake/			Contains Cake core libraries
vendors/			Contains third-party libraries for all applications

The Cake Naming Conventions

Like similar frameworks, Cake employs naming conventions instead of configuration files for many of its workings, such as for its MVC structure. It is good practice to understand and employ the Cake conventions. You can override some of the rules later, when you become a proficient Cake baker.

Cake has naming conventions for the four core objects: controllers, models, views, and tables. It also provides global constants and functions.

Controller Naming

Controller class names must be plural and must have `Controller` appended, as in `ProductsController`. If the object has more than one word, the second word must also begin with an uppercase letter, as in `OnlineProductsController`. Do not use underscores to separate words.

File names must be plural, with `_controller` appended and the `.php` extension, as in `products_controller.php`. If the object has more than one word, the subsequent words must be delimited with underscores, as in `online_products_controller.php`.

Model Naming

Model class names are singular, as in `Product`. If the object has more than one word, the second word must also begin with an uppercase letter (camel case), as in `OnlineProduct`.

File names are singular, with the `.php` extension, as in `product.php`. If the object has more than one word, the subsequent words are delimited with underscores, as in `online_product.php`.

View Naming

View file names take on the action name in the controller. For example, if the object has a method `ProductsController::upgrade()`, the path is `app/views/products/upgrade.ctp`.

Table Naming

Database table names should be plural, with words delimited with underscores, as in `country_codes`. You can override this naming convention by setting the `$useTable` property to your preferred table name. For example, you could set the following:

```
$useTable = 'mytable';
```

where `'mytable'` is the name of a table in a database.

Global Constants

The global constants are categorized into three major parts:

- *Core defines*: For example, `CAKE _SESSION_STRING` defines the Cake application session value.
- *Web root configurable paths*: For example, `WEBROOT_DIR` defines the web root folder where resources such as image and CSS files are stored.
- *Paths*: For example, `VIEWS` defines the parent folder for the presentation files (views).

For example, you can have the following code snippet to read the `example.ctp` page as an array from the `VIEWS` folder:

```
$page = VIEWS . 'example.ctp';
$file = file($page);
```

Global Functions

The global functions serve as wrappers for some utility functions. For example, the following function code snippet performs a simple search and replace operation to add style to `$text`.

```
$word = "sweet";  // search string - The needle!
$text = "This Cake is as sweet as honey"; // The haystack
// r(...) is the global function
r('$word', "<div class='red'>$word</div>", 'text');
```

The controller should contain most of the business logic, like this:

```
$shopping_basket_balance = $net_price + $tax;
```

Quite often in real applications, the business logic is split into separate parts. However, in Cake, the business logic is often separated into components or vendors, as discussed later in this chapter.

Note It is advisable to familiarize yourself with the global constants and functions to avoid reinventing the wheel. To see a complete list of Cake's various classes and functions, visit `http://api.cakephp.org`.

Models

The model is the first of the MVC concepts. Communicating with data stores such as tables, iCal events, structured files, LDAP records, and so on is an inevitable aspect of any large-scale web application, especially when it involves a large number of users. The actions of manipulating data stored in a data store are best done within a model. The model should be involved with just fetching and saving data from data stores. For example, table queries should be placed in the model.

Model Creation

Models are declared using the keyword `class`, followed by the name you wish to give to the model. Just like any PHP class, a user-defined model may contain some properties and methods specific to the implementation of that model as determined by the business requirement.

A user-defined model class should follow the Cake naming convention and predefined rules; for example, a `Product` model class should extend the `AppModel` class. The `AppModel` class extends the `Model` class, which defines Cake's model functionality. For example, the `Product` model class in Listing 1-3 invariably inherits all of the `Model` class properties and methods.

Listing 1-3. *A Sample Model Class*

```
class Product extends AppModel {
}
```

Though the `Product` class in Listing 1-3 appears empty, it is heavily loaded with some of Cake's properties and methods. We will bring some of these useful properties and methods into the limelight throughout this book.

The `AppModel` class is originally defined in the `cake/` directory. To create your own, place it in `app/app_model.php`. This allows methods to be shared among the models. The `Model` class, which `AppModel` extends, is a standard Cake library defined in `cake/libs/model.php`. Model default methods such as the `find()` method are defined in the `Model` class stored in `cake/libs/model/model.php`. Don't start tweaking Cake's default `Model` class until you become an expert baker.

Note Refer to the cheat sheet in at `http://cakephp.org/files/cakesheet.pdf` before writing a query method in your `Model` class definition. Alternatively, check the Cake API at `http://api.cakephp.org`. This effort will save you from rewriting existing functionalities and enhance rapid application development. For example, Cake provides a `find('all')` query to retrieve some or all information from a database table.

Let's use an example to demonstrate the four well-known types of operations that you will normally perform on a database (collectively known as CRUD):

- Create a record and insert it into a database table.
- Retrieve records from one or more database tables.
- Update tables.
- Delete records.

First, we will create a table named `departments` and insert some sample data with the SQL shown in Listing 1-4.

Listing 1-4. *The Table Schema for departments*

```
CREATE TABLE IF NOT EXISTS `departments` (
 `id` int(10) unsigned NOT NULL auto_increment,
`name` varchar(255) default NULL,
`region` varchar(255) default NULL,
  PRIMARY KEY (`id`)
) ;

INSERT INTO `departments` (`id`, `name`,`region`) VALUES
(1, 'Customer Services','UK'),
(2, 'Sales','UK'),
(3, 'Press Office','UK'),
(4, 'Investor Relations','US'),
(5, 'Human Resources',NG),
(6, 'Partnership Opportunities','US'),
(7, 'Marketing','UK'),
(8, 'Online Marketing','US');
```

Listing 1-4 contains some records about the name and region of the departments. This table is simple and self-explanatory.

Now that we have a database, we'll perform the first of the CRUD operations. We'll create some records, by using the default `save()` method provided in a model class. Using this method comes at the price of ensuring that the format of the data to be passed to it as a parameter must adhere to the Cake preformatted array structure. Let's take a look at a sample data structure in Listing 1-5.

Listing 1-5. *Cake's Expected $this->data Format*

```
Array
(
    [0] => Array
        (
            [Department] => Array
                (
                    [id] => 9
                    [name] => Warranties
                    [region] => Russia
                )

        )

    [1] => Array
        (
            [Department] => Array
                (
                    [id] => 10
                    [name] => Website
                    [region] => UK
                )
        )
)
```

In Listing 1-5, we've preformatted two additional records to be added to our `departments` database table. This structure, stored in a PHP variable such as `$data` or `$this-data`, will save its values to matching fields in the `departments` database table. To commit the data in this structure into this table, the `save()` method is at your service, but the format of `$this->data` argument is crucial to the success of the operation.

Now that we've created the expected data structure, let's define the `Department` model class to use this preformatted data and commit the two additional records into the `departments` table. The `Department` model class is shown in Listing 1-6.

Listing 1-6. *The Department Model Class*

```
<?php
class Department extends AppModel {

    var $name = 'Department';
    var $useTable = 'departments';

    function saveMessage($data) {
        if ($this->save($data)) {
            return true;
        } else {
            return false;
        }
    }
?>
```

In Listing 1-6, first we declare the `Department` class that extends `AppModel`. Next are the properties, starting with the `$name` property assigned the value `Department`. This property is necessary if you are running on anything less than PHP 5. The `$useTable` property specifies the name of the table required for data access or manipulation in the model. Although it isn't required, if omitted, Cake will use a table with the name of the model. For example, if the model name is `department`, Cake will use the `departments` table for the model by default. It is important to explicitly specify which database table you are using, especially if Cake's table naming convention is not followed.

The `saveMessage()` model function call should be done in a controller class. We'll discuss the controller in more detail later in this chapter. In our imaginary controller class, to invoke the `saveMessage()` method defined in Listing 1-6, we need the following statement:

```
$this->Department->saveMessage($data);
```

This method accepts as a parameter the preformatted array information called `$data`, as defined in Listing 1-5. Using an `If` statement, if the `$data` passed to Cake's `save()` model function is committed to the `departments` database table, a Boolean `true` value is returned; if not, `false` is returned. You can also save data into a database table in this manner by using Cake's `create()` model function.

When submitting an HTML form created using the `$form` object in a view, Cake automatically structures the form fields data submitted to a controller in a format that is similar to that shown in Listing 1-5.

Next, let's delve into the retrieve part of CRUD operations, or data access. To be meaningful, most data-access operations are filtered using some criteria. We're going to add a `getDepartment()` method to the `Department` model class, as shown in Listing 1-7.

Listing 1-7. *Retrieving Records Using $region='US' Criteria with the find() Method*

```
function getDepartment($region=null) {

    return $this->find('all',array('conditions'=>array('region'=>$region)));
}
```

In Listing 1-7, we define a `getDepartment()` method that accepts `$region` as its parameter. This method employs the service of the Cake's `find()` method to retrieve some department information based on `$region` as its parameter. To search for a department in the United States, we'll create the following in a controller class:

```
$this->Department->getDepartment('US');
```

This statement will retrieve and format all the departments in the US region, as shown in Listing 1-8.

Listing 1-8. *Structure of the Return Department Data Where Region Equals US*

```
Array
(
    [0] => Array
        (
            [Department] => Array
                (
                    [id] => 4
                    [name] => Investor Relations
                    [region] => US
                )

        )

    [1] => Array
        (
            [Department] => Array
                (
                    [id] => 6
                    [name] => Partnership Opportunities
                    [region] => US
                )
        )
    [2] => Array
        (
            [Department] => Array
                (
                    [id] => 8
                    [name] => Online Marketing
                    [region] => US
                )
        )

)
```

■ **Note** The formatted array data in Listing 1-8 might appear completely different when there are associations between the `Department` model class and other model classes that are connected to database tables. In case of associations, the array will include array data from tables of associated models. You will come across preformatted associated data in Chapter 3.

The `find()` function is one of the most useful Cake functions for data access. It has the following format:

```
find( conditions[array], fields[array], order[string], recursive[int] )
```

This method also accepts the parameters, in the order listed in Table 1-2.

Table 1-2. *The find() Function Parameters*

Name	Description	Default Value
`type`	Can be set to `all`, `first`, `count`, `neighbors`, or `list` to determine what type of data-access operation to carry out	`first`
`conditions`	An array of conditions specified as key and value	`null`
`fields`	An array of fields of key and value to retrieve	`null`
`order`	To specify whether to order fields in ascending (`ASC`) or descending (`DESC`) order	`null` (no SQL `ORDER BY` clause will be used if no `order` field is specified)
`page`	To determine the page number	`null`
`limit`	To limit the page result	`null`
`offset`	The SQL offset value	`null`
`recursive`	Whether to include the associated model	`1`

You can use many other Cake predefined model methods, such as the `query()` method or the `read()` method, which returns a list of fields from the database and sets the current model data (`Model::$data`) with the record found. Or you can create your own user-defined methods to manipulate data specific to the table a model object uses.

Listing 1-9 shows an alternative way of retrieving information about the departments. This listing will return exactly the same records from the `departments` table as the one shown in Listing 1-8 when used in our imaginary controller. The difference is that Listing 1-8 uses the `find()` method, while Listing 1-9 uses the `query()` method to access data.

Listing 1-9. *Retrieving Records with the query() Method*

```
function getDepartment($region=null) {

    $sql = "SELECT * FROM `departments` WHERE `region` = $region";
    return $this->query($sql);
}
```

One advantage of using the `query()` method is that you can put an already defined SQL statement, from a legacy system, into this method without going through the trouble of dividing the query parameters into parts, as you would need to do to use the `find()` method.

Data Validation

Data validation is an essential part of ensuring integrity and accuracy of data submitted by the user, such as via a web form. Cake has built-in validation mechanisms. You specify the validation rules in a model, and Cake automatically applies the rules when a web form is connected to that model. These rules can also be applied to XML data.

First, let's add a simple validation rule to our `Department` model using the `$validate` array, as shown in Listing 1-10. The validation rule array is basically an associative array. The keys are the names of the form fields to validate, and the corresponding values represent the rules attached to the form fields. We'll make use of this rule later, in the "Views" section.

Listing 1-10. *The Validation Rule for the Department Model*

```
var $validate = array( 'region' => array(
                                    'alphaNumeric' => array(
                                        'rule'=>'alphaNumeric',
                                        'required'=>true,
                                        'message'=>'Enter a region.'
                                        )
                                    )
                    );
```

A field can have multiple validation rules. The `$validate` array in Listing 1-10 defines a rule for the `region` field in our `Department` model. If a user does not submit a valid `region` field, the model will return an error to the controller and quit committing the data to the `departments` database table. The `message` key deals with the error messages during validation. To display the error message on a form, use the form helper's `error` function:

```
<?php echo $form->error('region');?>
```

This will display the error message "Enter a region" if a user enters a nonalphanumeric value in the `region` input field.

Apart from the rules employed in Listing 1-10, Cake provides a number of built-in validation rules to check the validity of form inputs and ensure the integrity of information you want to store. Table 1-3 lists a few of the built-in rules provided by Cake.

Table 1-3. *Some of Cake's Built-in Validation Rules*

Rule	Description	Example
`cc`	Checks for a valid credit card number	`'rule'=>array('cc','fast')`
`date`	Checks for a valid date	`'rule'=>'date'`
`email`	Checks for a valid e-mail address	`'rule'=>'email'`
`ip`	Checks for a valid IP address	`'rule'=>'ip'`
`phone`	Checks for a valid phone number	`'rule'=>array('phone', null, 'uk')`

For a complete list of the validation constants in your Cake build, see the predefined rules in the `cake/libs/validation.php` file.

You can extend the list of the rules by adding your own user-defined rules. In Listing 1-11, we define a simple custom rule to check if a value is a string.

Listing 1-11. *A Custom Rule Called String*

```
function string($check) {

    return is_string($check);
}
```

Before you apply a custom rule, such as the `string` rule shown in Listing 1-11, add it to the `cake/libs/validation.php` file, and then simply add the rule to your model `$validate` array:

```
var $validate = array('name'=>'string');
```

This will ensure that the `name` field is a valid string. In upcoming chapters, you will come across more validation rules.

The model object is robust and provides a lot of functionality for database manipulation. However, part of the data retrieved by a model is required for web surfers' consumption. When a user makes a URL request, some response is expected to be displayed in a view, which we'll look at in the following section.

Views

Now that we have some validation rules, let's build an HTML form to ask users to enter department information. The task of building a web form is done in a view.

Views are presentation pages. The HTML or XML documents on the Web are views to the users. However, views can be anything, especially if Cake is used to output other formats like RSS, PDF, and so on (which is certainly possible with the `RequestHandler` component and parsing extensions in the router). Views render information to the users. Views are composed of a mixture of HTML and PHP code.

By default, a view should be stored under the controller name folder. For example, the view for the `add()` method in the `DepartmentsController` is stored as `app/views/departments/add.php`.

Data from the controller is passed to the view by using the `set()` method in a controller.

■ **Note** Views should be involved only with displaying output. For example, this is where you will see HTML tags and XML tags. Business logic, such as `$shopping_basket_balance = $net_price + $tax;`, should not be in the view. However, the following is OK in a view: `If ($shopping_basket_balance > 1000 ) { echo 'You are eligible for a discount'; }`.

Let's build the view to add information to our `departments` database table. We are going to use another utility provided by Cake to build forms: the `$form` object. Listing 1-12 shows the add view.

Listing 1-12. *The Add View for the departments Table*

```
<h1>Add Department</h1>
<?=$form->create('department',array('action'=>'add'));?>
<p>Name:
<?=$form->input('Department.name',array('size'=>'120'));?>
<?=$form->error('name');?></p>
<p>Region:
<?=$form->input('Department.region',array('size'=>'40'));?>
<?=$form->error('region');?></p>
<?=$form->end('Save')?>
```

The view code created using the `$form` object is stored in `/app/views/department/add.ctp`. Remember that the validation rule for this form is created at Listing 1-10.

To display the add view to a user, we need a controller object with a function called `add()`, which tries to do exactly what it says: add the form data to our `departments` database table.

Listing 1-13 shows the action `add()` method of the `DepartmentsController`.

Listing 1-13. *The add() Action in the Departments Controller*

```
<?php
class DepartmentsController extends AppController {

   function add() {

        if (!empty($this->data)) {

            $this->Department->create();
            if ($this->Department->save($this->data)) {
                $this->Session->setFlash(➥
__('The Department data has been saved', true));
                $this->redirect(array('action'=>'add'));
            } else {
                $this->Session->setFlash(➥
__('The Department data could not be saved. Please, try again.', true));
            }
        }
    }
?>
```

Now that we've built a web form using the HTML helper and created a controller to handle the `add()` action, let's demonstrate how data is passed to a view. We'll start by creating a show action in our `Departments` controller class. The action method is shown in Listing 1-14.

Listing 1-14. *The show() action in the Departments Controller*

```
function show($region) {

    $this->set('data', $this->Department->getDepartment($region));
}
```

In Listing 1-14, the `show()` method accepts the `$region` data as a parameter, and then retrieves departmental data and uses the `set()` method to prepare the data for the view in Listing 1-15. A view is always named after an action. For example, the `show()` action in the `DepartmentsController` of Listing 1-14 will have a view file stored in `app/views/department/show.ctp`.

Listing 1-15. *A View for the Action show($region) of DepartmentsController*

```
<h1 class="first">

    Our Department: -

    <?php

        foreach($data as $department) {
            echo $department ['Department'] ['name'];
        }
    ?>

</h1>
<p>Our Departments are led by Dr. Cake .</p>
```

The view in Listing 1-15 will display the name of the department (for example, Sales) in the header section of the web page presented to the user.

Tip If you have a number of data items to display in a view, such as `$title = "Practical Cake Projects"; $department = "Sales"; and $region = "UK";`, using `$this->set(compact('title','department','region'));` will enable you to use only one `set()` function in your controller class to pass all the information to your view. You can then access the individual variable in your view; for example, `<?php pr($title); ?>`.

An essential part of any framework is the part that handles requests. In the MVC structure, this is handled by the controller.

Controllers

As you've seen, a controller is a class with methods called *actions*. These actions or methods contain most of the logic that responds to user requests in an application. For example, if a user wants to know the number of departments in a particular region, the user needs to access the `show()` method of `DepartmentsController` defined in Listing 1-10, by typing the following URL in a browser address bar:

```
localhost/departments/show/US
```

The default structure for accessing a Cake URL is to first specify the controller and then the action. In the preceding URL, `departments` is the controller, `show` is the action, and `US` is a parameter.

By convention, a Cake request should be structured in the following manner:

```
http://[mydomain.com]/[Application]/[Controller]/[Action]/[Param1]/…[ParamN]
```

Note The `index()`method is the default access point to a controller when a method is not explicitly specified in a user's request. For example, you can load the `index()` method with codes that will invoke the welcome page of your application. However, do not forget to create a view, or you will get a warning from Cake stating that you should create a view for the action.

Your application's controller classes are expected to extend the `AppController` class, which in turn extends a core `Controller` class, which is a standard Cake library. The `AppController` class is defined in `/app/app_controller.php`, and it should contain methods that are shared between two or more controllers.

These controllers can include any number of actions. The `AppController` serves as a global class that can contain properties and methods common to all the user-defined controllers in an application. For example, you can have a method to detect and extract the IP address of a user, and then use the value of this address to determine the flow of the application. Earlier, in Listing 1-2, we used the default controller method called `beforeFilter()` in our controller class to reference the method defined in the `AppController` class stored in `app/app_controller.php`. Another simple example is to set a default page title for an application, as in this example:

```
<?php
class AppController extends Controller {
    var $pageTitle = 'Chapter 1 – A Bakery Application';
```

Since our user-defined controller extends the `AppController`, using the statement `$this->PageTitle` within our controller gives us access to the string `'Chapter 1 – A Bakery Application'` assigned to the `pageTitle` property in the `AppController` class.

The `$uses` property is an important property within the controller. It works similarly to the `require_once` statement in PHP. Basically, once you have created a model (such as `Department`) and you want to use the model in a controller, you need to include it in the `$uses` array. For example, where `Department` and `Trade` are existing models, you can have the following statement in your controller:

```
$uses = array('Department', 'Trade');.
```

Cake Components

Components are classes defined to carry out specific application tasks to support the controller. Cake comes with many built-in components, such as `Acl` (for user and group access control), `Auth` (for user and group authentication), `Email`, `Session`, and `RequestHandler`. Components can also be user-defined. In fact, in large web applications, you will most likely need to build some of your own components to be used by several controllers. All the components that you develop should be stored in the folder `app/controllers/components`. Components follow the same Cake conventions as controllers.

From a programmer's point of view, components enable you to extend the functionality of Cake. If you find that your component is quite useful and you possess the free open source spirit, you can and should post it on the Cake web site, where there is a public repository of components.

To demonstrate, we'll dive straight in and create our own simple component—a utility to convert an array to an object. Listing 1-16 shows the code to create this component.

Listing 1-16. *A Component to Convert an Array to an Object*

```
<?php
class ArrayToObjectComponent extends Object{

    function startup(&$controller) {
        $this->Controller = $controller;
    }

    function convert($array, &$obj) {
        foreach ($array as $key => $value) {
            if (is_array($value)) {
                $obj->$key = new stdClass();
                $this->array_to_obj($value, $obj->$key);
            } else {
                $obj->$key = $value;
            }
        }

        return $obj;
    }
}
?>
```

The ArrayToObjectComponent class in Listing 1-16 contains two basic functions, which are stored in app/controllers/components/array_to_object.php:

- The first function, called startup(), is used to instantiate the controller object. This enables all other functions within the component to access information contained in the parent controller. It's basically a callback method used to bring the controller object into the component.
- The second function, convert(), is our user-defined function. It does the work of accepting an array of data and returning the array as an object. You can use this component whenever you want to convert an array to an object.

Everything inside a component should be generic. Do not put controller-specific code, such as a database table name, into components.

You can use components within controllers or other components. To use a component—whether it is a built-in one or one you have created—you need to first declare the component within the $components array in a user-defined controller, another component, or in the AppController class. For example, to use the component in Listing 1-16, include the following statement:

```
var $components = array('ArrayToObject')
```

In Listing 1-17, we make references to the built-in Session component and our ArrayToObject component in the DepartmentsController class.

Listing 1-17. *Using Components in DepartmentsController*

```
<?php
class DepartmentsController extends AppController
{
    var $uses = array( 'Department');
    var $components = array( 'Session', 'ArrayToObject');

    function display() {
        $arrData = array();
        $arrData  =  $this->find('all');
        pr($this->ArrayToObject->convert(arrData, &$obj));
    }
}
?>
```

In Listing 1-17, we convert the result of the data retrieved from our departments database table from an array to an object. First, we use the $uses array to reference the Department model. We then use the $component array to reference the Session component, which is a built-in Cake component, and the ArrayToObject component, which is our user-defined component. Next, we create a display() function that contains a declaration of an array variable called $arrData. We retrieve the department data using the default find() function, store the result in the array, and then pass the array to the convert() method of the ArrayToObject component. Finally, we use the Cake pr() global function to print the resulting object.

Helpers

Cake helpers are classes that help to decrease development time by providing shortcuts to generate presentational elements. Earlier, we used the Cake form helper, which helps with form element creation and data handling. Helper files should be stored in the `app/views/helpers` folder. Table 1-4 briefly describes some of Cake's built-in helpers. For full documentation of the Cake helpers, check the Cake API at `http://api.cakephp.org`.

Table 1-4. *Some of Cake's Built-in Helpers*

Helper	Description
HTML	Helps automate creating HTML elements and also enables dynamic generation of HTML tags by accepting and parsing variables. This helper is called in a view by using the `$html` object. To include a reference to HTML helper, use the variable `$helpers = array('Html');`. HTML helper functions output HTML elements such as `charset`, `css`, `div`, `docType`, `image`, `link`, `meta`, `nestedList`, `para`, `style`, `tableHeaders`, `tableCells`, and so on.
Form	Helps in form creation and processing. Use the `$form` object together with its functions to create form elements. For example, to create a form input element, use the `$form->input()` function. To start the form tag, use the `$form->create()` function. Other form input element functions include `label`, `checkbox`, `dateTime`, `hidden`, `radio`, `textarea`, and so on. There are many options that can be used in form element functions, such as `maxLength`, to set the maximum length of an HTML attribute.
Ajax	Helps to simplify Ajax tasks. It requires the statement `$javascript->link(array('prototype'))` in the view, which references the Prototype JavaScript framework, in order to work properly in a view. You can get a copy of Prototype from `http://www.prototypejs.org/download`.
JavaScript	Helps to simplify JavaScript tasks, such as to create a JavaScript Object Notation (JSON) object from an array, using `$javascript->object()`. To attach an event to an element, use `$javascript->event()`.
Paginator	Helps to format data into multiple pages or to sort data based on some parameters. For example, to create a link to the next set of paginated results, use the `$paginator->next()` function.
Session	Provides functions to deal with session management. For example, to render messages, use `$session->flash()`. To read all values stored in a given session, use `$session->read()`.
Text	Provides functions to deal with text or string handling. For example, to remove whitespace from the beginning and/or end of text, use the `$text->trim()` function.
Time	Helps manage dates and times. For example, to check if a given date/time string is today, use the `$time->isToday()` function.
XML	Helps with XML manipulation. To create XML elements, use the `$xml->elem()` function. This helper can also be used to convert a result set into XML.

To reference the common helpers that you need in your application, you can specify the following statement in your `AppController` class:

```
var $helpers = array('Html','Form','Ajax','JavaScript');
```

This will ensure that the `$javascript->link()` function in a layout works properly.

You may need to create your own helper or tweak an existing helper class to provide additional functionality that is not yet supplied by Cake. As an example, Listing 1-18 creates a simple helper called `/app/views/helpers/break.php`. This helper will print a variable and insert a new break after printing.

Listing 1-18. *A Sample Custom Break Helper*

```
class BreakHelper extends AppHelper {
    function newline($val) {
        return $this->output("$val<br />");
    }
}
```

We can use this helper in our `Departments` controller object to insert a break whenever we use the controller's `print()` function, as shown in Listing 1-19.

Listing 1-19. *Using the Sample Break Helper*

```
<?php
class DepartmentsController extends AppController {
    var $name = 'Departments';
    var $helpers = array('Break');

    function print($val) {
        return $this->Break->newline($val);
    }
}
?>
```

First, in Listing 1-19, we reference the break helper by declaring `var $helpers = array('Break');`. Next, we define a `print()` function that accepts a `$val` parameter. This function contains the statement that invokes the break helper's `newline($val)` method, and consequently returns the result with a newline after it.

Plugins

With Cake, you can create a complete MVC package called a *plugin*, which you can integrate into other Cake applications. A plugin is a mini-application with its own controllers, models, views, and other Cake resources. Cake does not have any built-in plugins. You can use third-party plugins, or better still, build your own.

Here, we will create a basic feedback plugin that will provide mailing facility. It will have the following directory structure:

```
/app
    /plugins
        /feedback
            /controllers
            /models
            /views
            /feedback_app_controller.php
            /feedback_app_model.php
```

where

- `/controllers` contains plugin controllers.
- `/models` contains plugin models.
- `/views` contains plugin views.
- `/feedback_app_controller.php` is the plugin's `AppController`, named after the plugin.
- `/feedback_app_model.php` is the plugin's `AppModel`, named after the plugin.

Note You must create both an `AppController` and an `AppModel` for a plugin to work properly. If you forget to define the `FeedbackAppController` class and the `FeedbackAppModel`, Cake will throw a "Missing Controller" error.

The feedback plugin's `AppController` is stored in `app/plugins/feedback_app_controller.php`, and its corresponding `AppModel` class is stored in `app/plugins/feedback_app_model.php`, as shown in Listing 1-20.

Listing 1-20. *Feedback App Classes for the Feedback Plugin*

```
<?php
class FeedbackAppController extends AppController {
    //..
}
?>

<?php
class FeedbackAppModel extends AppModel {
    / /..
}
?>
```

Now, let's create the FeedbackSendController for our feedback plugin. The code in Listing 1-21 is stored in app/plugins/feedback/controllers/feedback_send_controller.php.

Listing 1-21. *The FeedbackSendController to Invoke the send() Method*

```
<?php
class FeedbackSendController extends FeedbackAppController {
    var $name = 'Feedback';
    var $uses = array( 'Feedback');

    function send($toEmail) {

        $this->set( "result", false );

        if ( $this->Feedback->sendEmail($toEmail) ) {
            $this->set( "result", true );
        }
    }
}
?>
```

Next, we'll create and store the FeedbackSendModel class in the app/plugins/feedback/models/feedback_send_model.php file, as shown in Listing 1-22.

Listing 1-22. *The FeedbackSendModel That Uses the PHP mail() Function to Send a Message*

```
<?php
class FeedbackSendModel extends FeedbackAppModel
{
    function sendEmail($recipient) {
        // Here's where we try to send an email message.
        if( mail($recipient, 'Hi', 'What is baking', 'From: sugar@Cake.com') ) {
            return true;
        }
        return false;
    }
}
?>
```

Next, let's create a simple feedback plugin view stored in the app/plugins/feedback/views/feedback_send/send.ctp file, as shown in Listing 1-23.

Listing 1-23. *The Feedback Email View*

```
<h1>Feedback Email</h1>
<?php
if ( $result ) {
    echo 'Thank you for your feedback.';
} else {
    echo 'Sorry! Our system is down at the moment. Please try again later.';
}
?>
}
?>
```

Now that we have installed the feedback plugin, we can use it. To access the plugin within a Cake application, you can add name of the plugin, then the action, then the parameter to the URL, as follows:

`http://localhost/feedback/feedbackSend/feedbackSend/send/[`*`emailparam`*`]`

You can have a default controller with the name of your plugin. If you do that, you can access it via `/[plugin]/action`. For example, a plugin named `users` with a controller named `UsersController` can be accessed at `http://[your domain]/users/add` if there is no plugin called `AddController` in your `[plugin]/controllers` folder.

Plugins will use the layouts from the `app/views/layouts` folder by default. You will see how to override layouts in Chapter 3.

You can access a plugin within controllers in your Cake application by using the `requestAction()` function:

```
$sent = $this->requestAction(array('controller'=>'FeedbackSend','action'=>'send'));
```

Vendors

Many modern frameworks adopt the Don't Repeat Yourself (DRY) principle. Lazy (or maybe efficient) programmers don't like to reinvent the wheel! To allow us to sleep in a little longer, Cake has provided a `vendors` folder. This is where we store third-party applications that don't have any relationship with Cake, such as the phpBB message board application and the Swift Mailer mailing application. This comes in handy, considering the number of utility scripts and programs available in various PHP repositories such as `http://www.phpclasses.org`.

Cake's technique of including external scripts is as simple as using the following function:

```
App::import('Vendor','file',array('file'=>'fileName.php'));
```

The third parameter of the function accepts an array of file names. Usually, `fileName.php` is the startup file of the third-party application.

The `App::import()` function can be used in controllers, models, and views of Cake applications. However, it is important that the call is made before any class definition. The `vendors` folder provides a standard way to include third-party applications.

As an example, let's create a `spillout.php` script, as shown in Listing 1-24. This script will serve as our third-party script that we'll use in our `ScreenController` later. This script should be stored in the `app/vendors` folder.

Listing 1-24. *A Script to Display the Content of a File on the Screen*

```
<?php
class ReadFile {
    protected $file;

    public function __construct($fileName) {
        $this->file = $fileName;
    }

    private function spill() {
        return readfile($this->file);
    }
}
?>
```

Listing 1-24 will simply read the content of the `welcome.html` file and send it to the screen for a user's consumption. We can import this script into our `ScreenController` as shown in Listing 1-25.

Listing 1-25. *The ScreenController to Use a Script as a Vendor*

```
<?php
    App::import('Vendor','file',array('file'=> 'spillout.php'));
    class ScreenController extends AppController {

    function index() {
        $output = new ReadFile('welcome.html');
        $output->spill();
    }
}
?>
```

In Listing 1-25, we use the `import()` function to load the content of the `spillout.php` file, and then we create an instance of the `ReadFile` class using the `welcome.html` file as its parameter and store it in `$output`. Finally, we send the content of the file to the screen using `$output->spill();`.

Summary

In this chapter, we briefly introduced you to the main features of Cake. We explained how it is a RAD platform, with the MVC design pattern forming the base foundation. We covered how the Cake MVC structure works, with business logic stored in controllers and components, data access in models, and presentational markup in the view. Additionally, we showed how Cake reduces development time with helpers, plugins, and vendors.

After reading this chapter, you should have an overview of how Cake structures a web application. You should feel confident that learning Cake is one of the best decisions for anyone interested in PHP programming and with a need to write rapid web applications. But do note, sometimes it may be better to write basic methods like "Hello World" in a simple PHP script, rather than using Cake, so that you don't end up killing an ant with a sledge hammer.

In the following chapters, we'll present full-fledged Cake applications, beginning with a simple blogging application.

CHAPTER 2

Blogging

The Web has revolutionized the way we communicate with friends and strangers. We now freely exchange media content, such as textual information, graphics, audio, and video. One of the ways to exchange such information is known as *blogging*. Blogging uses HTML forms for tasks such as submitting posts, uploading content, and so on.

In the 1990s, blogging started like a kiddie joke, with individuals posting their personal stuff online. Since then, there has been an explosion of blogging web sites. Nowadays, movie stars, politicians, and corporate organizations such as Microsoft host their own blogging sites to communicate their ideas.

This chapter describes how to build your own blogging application. But why would you bother to develop such an application when you can use one of the many free or low-cost solutions, such as Blogger, Movable Type, Textpattern, WordPress, TypePad, or LiveJournal (to name a few)?

The ready-made blogger solutions have common interface design features. Developing your own blogging application allows you to customize the site, giving it a unique look, excluding unnecessary features, and adding features that are not supplied with the prebuilt sites.

In this chapter, we'll build our own blog application, which will enable us to list, add, edit, delete, publish, and unpublish posts. We will use Cake's form helper to automate some tasks, such as to generate form elements, validate user-submitted data, and repopulate invalid form elements with submitted data. We will insert the post data into an XML file to provide RSS service to those with an RSS reader (or aggregator).

To create this blog application, you need a web server that supports PHP and a database server to store some information. If you are new to the concepts and the workings of Cake, read Chapter 1 before continuing with this chapter.

Creating the Database

Building web sites that allow user interactivity sometimes requires working with persistent data, which can be stored in relational databases or local file systems. This requirement applies to building our blog application, as we need to manage the post records. As mentioned in the previous chapter, we'll use the MySQL database server for the examples in this book. We'll use Cake's objects and their methods that allow us to store and retrieve data from a database. For information about how to configure Cake's database connection parameters and connect to a database, see Chapter 1.

Our Blog database will contain a single table named posts. This table will store records of posts. The records include fields for an ID to provide a unique reference for each post, the title of a post, the post's content, the dates that a post was created and modified, and whether or not a post should be published (displayed to the public). Listing 2-1 shows the SQL to create the posts table.

Listing 2-1. *The SQL Statement to Create the posts Table*

```
CREATE TABLE IF NOT EXISTS `posts` (
  `id` int(11) NOT NULL auto_increment,
  `title` varchar(50) default NULL,
  `content` text,
  `created` datetime default NULL,
  `modified` datetime default NULL,
  `published` tinyint(1) NOT NULL default '1',
  PRIMARY KEY  (`id`)
);
```

The SQL schema shown in Listing 2-1 will handle basic post information. If you like, you can add more fields, such as a summary field to store summaries of posts.

Now, let's insert some sample post information into our posts table, using the following SQL statements:

```
INSERT INTO `posts`➥
(`id`, `title`, `content`, `created`, `modified`, `published`) VALUES
(1, 'Another day Still Looking', 'My Lion ran off',➥
'2008-06-19 18:26:11', '2008-06-19 18:26:11', 1),

(2, 'A good day', 'The Lion is back in one piece.',➥
'2008-06-19 18:31:50', '2008-06-19 18:31:50', 1),

(3, 'Thank GOD', 'Everything belongs to my father',➥
'2008-06-20 18:42:11', '2008-06-20 18:42:11', 1);
```

If you added other fields, be sure to insert their corresponding values with SQL INSERT statements.

Tip Cake will automatically populate the created and modified fields in a table with the current date information (in our example, the dates when saving and updating posts). In Cake, these fields are called automagic model fields.

Reviewing the Application Structure

Before we start to build the blog application, let's take a brief look at the folder structure and files that will form part of the application. Table 2-1 assumes that other default Cake folders and files also exist in the same environment.

Table 2-1. *The Blog Application Structure*

Directory		Description
app/		The parent folder for the blog application
	config/	Amended database.php file to include our blog database parameters
	controllers/	posts_controller.php file, which contains all the actions, such as add post
	models/	post.php file to deal with our blog application data
	views/posts	index.ctp, add.ctp, edit.ctp, and delete.ctp files

We'll create these files and explain their contents in upcoming sections. For details on Cake's complete file system structure, refer to Chapter 1.

Creating the Post Model

The Post object manages the post data. By using Cake's naming convention, we'll be able to take advantage of the functionality inherently provided by Cake. We'll use that functionality to interact with the posts database table, and access and manipulate the post records. The Post model class, shown in Listing 2-2, is stored in app/models/post.php.

Listing 2-2. *The Post Object That Handles the Post Data (app/models/post.php)*

```
<?php
class Post extends AppModel
{
    var $name = 'Post';
    var $validate = array( 'title'=>array(
                              'alphaNumeric'=>array(
                                  'rule'=>'alphaNumeric',
                                  'required'=>true,
                                  'message'=>'Enter a title for this post',
                                  )
                           ),
                           'content'=>array(
                              'alphaNumeric'=>array(
                                  'rule'=>'alphaNumeric',
                                  'required'=>true,
                                  'message'=>'Enter some content for your post',
                                  )
                           )
                       );
}?>
```

The `Post` model class consists of the `$name` property, used to handle PHP 4 backward-compatibility, and the `$validate` array property, which contains the validation rules.

In the `$validate` array, each element's key corresponds to the name of the input element to be validated (for example, `title`), and its value defines the rules to apply against the input before the post data is saved to the `posts` table—when the post form is submitted. Listing 2-2 shows the validation rules. We check that the `title` and the `content` fields of the form are not empty when the form is submitted. We also check whether the values submitted are alphanumeric. If not, the corresponding error messages set in the validation array against the `'message'` keys will be displayed.

The `PostsController` object, which we will create next, will use the `Post` model object to access information from the `posts` table, ensure the integrity of the submitted post information, and then commit the post into the `posts` database table.

Creating the Posts Controller

Now that the `Post` model class is created, we need a `PostsController` object to manage all the post actions. These actions include listing all the posts and providing the user interfaces for adding and editing post data. The controller calls the `Post` model object created in Listing 2-2 to handle the post data as required.

The `PostsController` class will contain the methods listed in Table 2-2.

Table 2-2. *The PostsController Class Actions*

Method	Description
`index()`	Lists all the posts from the `posts` table and handles the RSS feed for posts
`add()`	Invokes the add post page and saves validated posts to the `posts` table
`edit()`	Invokes the edit post page
`disable()`	Disables a published post
`enable()`	Enables a disabled post so it's published
`delete()`	Removes a post record from the `posts` table

Listing the Posts

The `PostsController` class, shown in Listing 2-3, extends the `AppController` class. This file is stored in `app/controllers/posts_controller.php`.

Listing 2-3. *The PostsController to Define Post Actions (app/controllers/posts_controller.php)*

```
<?php
class PostsController extends AppController
{
    var $name = 'Posts';
    function index() {
        $posts = $this->Post->find('all');

        $this->set(compact('posts'));
    }
```

First, we add the `index` method, which displays the list of posts. By default, this method is called if no other action is called explicitly during a URL request. Along with showing all the published posts, the index page contains links that will enable users to perform operations such as edit, publish, unpublish, and delete a post record.

In Listing 2-3, the `index` method contains two simple statements. The first uses the `Post` model object with its default `find` method to pull all the posts from the `posts` database table and then store the results in an array called `$posts`. The second prepares and sets the `$posts` records so that the `views/posts/index.ctp` file, shown in Listing 2-4, can display the list of all the posts from the `$posts` variable.

Listing 2-4. *The View for the Post List (views/posts/index.ctp)*

```
<div id="center_content">
    <h2>Post Listings</h2>
    <p>Here is a list of the existing posts.</p>
    <div>
    </div>
    <?php
        if ( isset( $posts ) && is_array( $posts ) ) {
    ?>
        <table>
            <tr>
                <td>
                    <b>ID</b>
                </td>
                <td>
                    <b>title</b>
                </td>
                <td>
                    <b>content</b>
                </td>
                <td>
                    <b>Last Modified</b>
                </td>
                <td>
                    <b>published</b>
                </td>
                <td colspan="2"><b>  Action</b></td>
            </tr>
```

```
            <?php foreach ($posts as $post) : ?>
            <tr>
                <td><?php echo $post['Post' ][ 'id' ]; ?></td>
                <td><?php echo $post['Post' ][ 'title' ]; ?></td>
                <td><?php echo $post[ 'Post' ][ 'content' ]; ?></td>
                <td><?php echo $post[ 'Post' ][ 'modified' ]; ?></td>
                <td>
                    <?php echo $html->link(ife(
                            '$post['Post']['published'] == 1',
                            'Published',
                            'Unpublished'),
                            '/posts/'.ife('$post['Post']['published'] == 1',
                            'disable','enable').'/'.$post['Post']['id']
                            );
                    ?>
                </td>
                <td>
                    <?php echo $html->link(
                                    'Edit',
                                    '/posts/edit/'.$post['Post']['id']);?>
                </td>
                <td>
                    <?php echo $html->link(
                                    'Delete',
                                    '/posts/delete/'.$post['Post']['id']);?>
                </td>
            </tr>
            <? endforeach;?>
            <?php
                if ( sizeof( $posts ) == 0 ) {
            ?>
                <tr style="background-color: #cccccc;">
                    <td colspan="6">
                        <span style="font-size: 17px;">
                            No post found.
                        </span>
                    </td>
                </tr>
            <?php
                }
            ?>
        </table>

        <br/>

    <?php
        }
    ?>
</div>
```

The `index.ctp` file in Listing 2-4 starts with the headings of the web page that displays the posts list. We then insert a section of PHP code immediately after these headings. If the `$posts` variable set by the `PostsController` object contains some post records, we first display the headings for the individual elements in the variable. Next, using a `foreach` loop statement to loop through the `$posts` variable, we display a list item for each post. Finally, if the `$post` variable is empty, we simply display the message `No post found`. Figure 2-1 shows an example of a post listing.

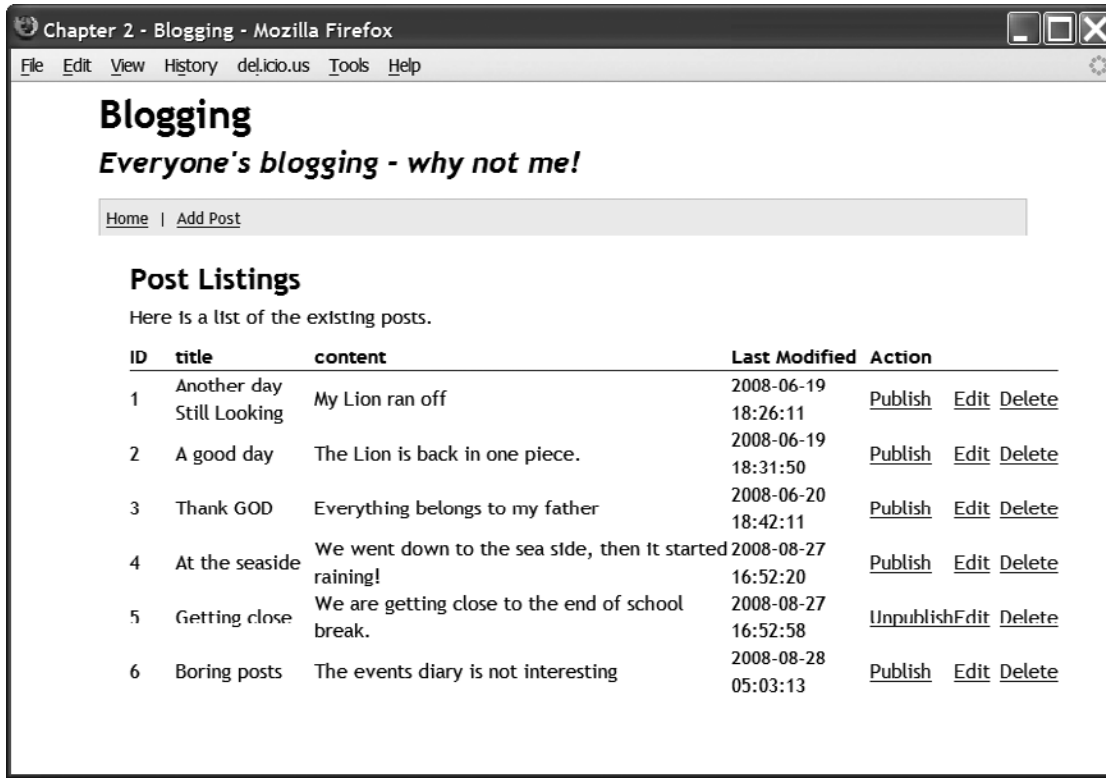

Figure 2-1. *Viewing the post listings page*

This list provides an interface to directly manage individual posts. As shown on the right side of Figure 2-1, the page has links to trigger the publish, edit, and delete actions.

Adding a Post

The next method we need to implement in the `PostsController` class is the `add` method, as shown in Listing 2-5. This method, as the name implies, handles adding post data.

Listing 2-5. *The add Method for Adding Post Data*

```
function add() {

    $actionHeading = 'Add a Post!';
    $actionSlogan = 'Please fill in all fields. Feel free to add your➥
post and express your opinion.';

    $this->set(compact('actionHeading','actionSlogan'));

    if (!empty($this->data)) {
        $this->Post->create();
        if ($this->Post->save($this->data)) {
            $this->Session->setFlash(__('The Post has been saved', true));
            $this->redirect(array('action'=>'index'));
        } else {
            $this->Session->setFlash(➥
__('The Post could not be saved. Please try again.', true));
        }
    }
}
```

In the `add` method, the first two statements set the heading and slogan for the add view page. This is necessary because we are going to use a single element view to display the forms to add and edit posts. Elements in Cake enable you to reuse views.

Next, we check if the add post form has been submitted. If the form has not been submitted, the add view is displayed. If the submitted data (`$this->data`) is not empty, using the `save` method of the `Post` model object, the application will attempt to create a new post record. The `save` method automatically uses the validation rules defined in Listing 2-2 to check the integrity of the submitted post. If the post does not pass the validation rules, the error message is set, using the `setFlash` method of the `Session` object. Otherwise, the post is saved to the database table, and the success message is set for display in the view.

Next, we'll create the add view and store the codes in `views/posts/add.ctp` file. The content of the `add.ctp` file is simply the following code snippet:

```
<?php echo $this->element('add_or_edit');?>
```

The `$this->element()` method accepts the name of a file stored in the `views/elements` folder (`add_or_edit` in this case), without the file extension (without `.ctp`). It simply transfers the content of `add_or_edit.ctp` into the `add.ctp` file. The resulting source code for the add view is shown in Listing 2-6.

Listing 2-6. *The Add View That Provides an Interface to Add a Post (app/views/posts/add.ctp)*

```
<fieldset>
    <legend> <?php __('Add a Post!');?> </legend>
    Please fill in all fields.
    <?php
        echo $form->create('Post');
        echo $form->error( 'Post.title' );
        echo $form->input( 'Post.title',
                        array( 'id' => 'posttitle', 'label' => 'Title:',➥
'size' => '50', 'maxlength' => '255', 'error' => false ) );

        echo $form->error( 'Post.content' );
        echo $form->input( 'Post.content',
                        array( 'id' => 'postcontent', 'type'=>'textarea',➥
'label' => 'Content:', 'rows' => '10', 'error' => false ) );

        echo $form->end( array( 'label' => ' Submit Post ' ) );
    ?>
</fieldset>
```

In Listing 2-6, we start by displaying the heading of the interface for adding a post. We then insert a PHP opening code tag to house the creation of the form using Cake's form helper functionality. First, the `$form->create()` method defines the start tag for our form. Its `'Post'` string argument represents the action that will be invoked, such as the URL to which the form data will be submitted. Note that if the method attribute is not specified, the `POST` method is the default request method.

Next, we start to add the required form input elements using the `$form->error()` method, which deals with the error handling of the form. Its argument, `Post.title`, is a string that represents the name of the input element, where `Post` is the model name, followed by a dot (`.`), and `title` holds the value of the post's title.

Next is the `$form->input()` method to generate the text input element called `title`, whose first argument is in the same argument format as the error input element. The second argument of the text input element is an associative array of HTML text input element attributes.

Following that is the code that generates the `textarea` input element called `content` using an argument format similar to the `title` input element discussed previously.

Finally, after we've added the form elements, we can add the form closing tag using the form helper method `$form->end()`. It also accepts an associative array of HTML submit input element attributes.

Figure 2-2 shows an example of an add post form when a user tries to submit a blank form. The error messages are displayed.

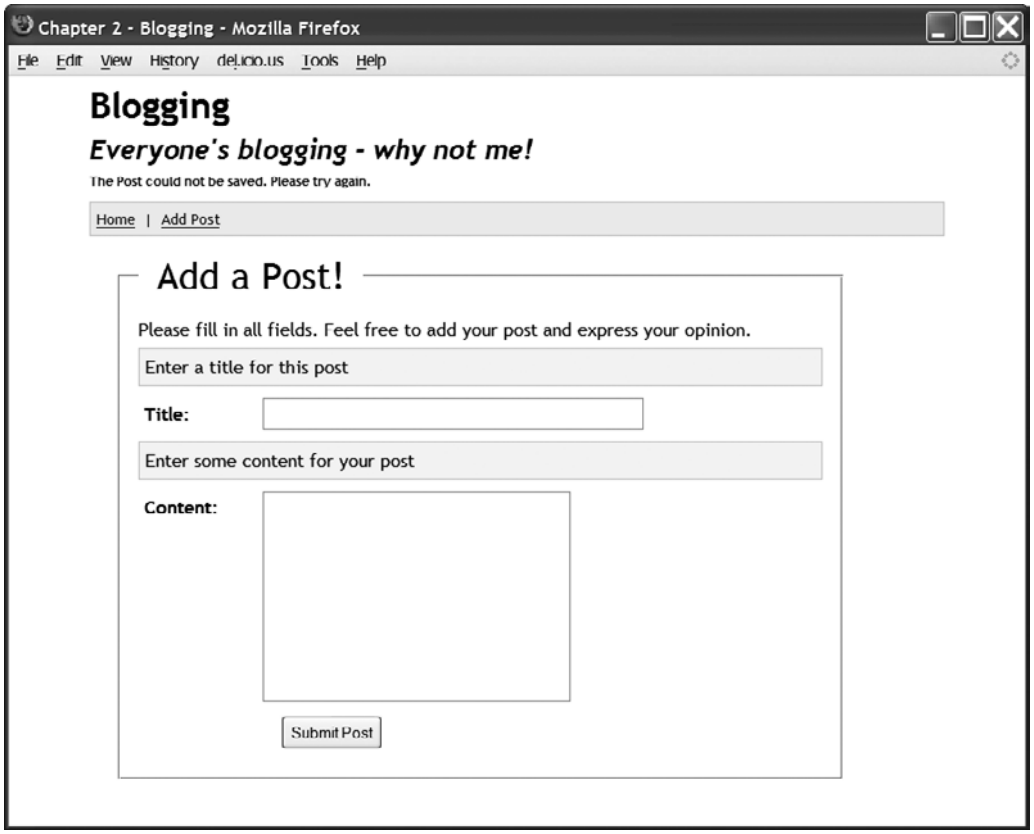

Figure 2-2. *Error messages appear when you try to submit a blank add post form.*

Updating a Post

Sometimes, we are not completely satisfied with our posts and would like to make some amendments. We will create the `edit` action of the `PostsController` to handle this task. It uses a supplied post ID (in our case `$id`) to retrieve the details of a post from the `posts` database table and repopulates the edit form with the information. This `edit` method code snippet is shown in Listing 2-7.

Listing 2-7. *The edit Action That Handles a Post Edit Request*

```
function edit($id = null) {

    $actionHeading = 'Edit a Post!';
    $actionSlogan = 'Please fill in all fields. Now you can amend your post.';

    $this->set(compact('actionHeading','actionSlogan'));

    if (!$id && empty($this->data)) {
        $this->Session->setFlash(__('Invalid Post', true));
        $this->redirect(array('action'=>'index'));
    }

    if (!empty($this->data)) {
        if ($this->Post->save($this->data)) {
            $this->Session->setFlash(__('The Post has been saved', true));
            $this->redirect(array('action'=>'index'));
        } else {
            $this->Session->setFlash(__➥
('The Post could not be saved. Please try again.', true));
        }
    }

    if (empty($this->data)) {
        $this->data = $this->Post->read(null, $id);
    }
}
```

In Listing 2-7, we first set the heading and slogan for the edit view page. The next step we take is to check whether `$id` and `$this->data` (form data) are empty. If so, an error message is stored in our `Session` object, and the request is redirected to the blog home page. If the submitted form data is not empty, Cake will try to commit the edited post information to the `posts` database table and then flash appropriate messages upon success or failure. Finally, if only the submitted data is empty, a post's information is pulled with the `Post` model `read` method using the supplied `$id` as the criterion.

Next, we'll create the edit view and store the codes in the `views/posts/edit.ctp` file. The content of this file is exactly the same as that of the `add.ctp` file:

```
<?php echo $this->element('add_or_edit');?>
```

Here, we've reused the `add_or_edit.ctp` element file again to produce the source code for the edit post view, taking advantage of the elements resource of Cake. In principle, the views for adding and editing posts are the same, except for their page headings.

Figure 2-3 shows an example of the edit view for post ID 1.

Figure 2-3. *The view to update post ID 1*

We've built our post forms using one of the key features of Cake: form helpers. These helpers have automated the tasks of generating our form elements, validating the form input, and repopulating the submitted data.

Unpublishing a Post

When you don't want a post record to be displayed on the home page, you can disable the record. Given a post `$id`, the `disable` method disables the appropriate post, as shown in Listing 2-8.

Listing 2-8. *The disable Action to Disable a Published Post*

```
function disable($id=null) {
    $post = $this->Post->read(null,$id);
    if (!$id && empty($post)) {
        $this->Session->setFlash(➥
__('You must provide a valid ID number to disable a post.',true));
        $this->redirect(array('action'=>'index'));
    }
    if (!empty($post)) {
        $post['Post']['published'] = 0;
        if ($this->Post->save($post)) {
            $this->Session->setFlash(__('Post ID '.$id.' has been disabled.',true));
```

```
        } else {
            $this->Session->setFlash(__('Post ID '.$id.' was not saved.',true));
        }
        $this->redirect(array('action'=>'index'));
    } else {
        $this->Session->setFlash(__('No Post by that ID was found.',true));
        $this->redirect(array('action'=>'index'));
    }
}
```

In Listing 2-8, using the request `$id`, a post record is retrieved from the `posts` database table and stored in the `$post` array variable. If the `$id` value is null or the `$post` variable is empty, we use the `Session` object to set the appropriate message and redirect to the blog home page. If there is a valid `$id` and the `$post` is not empty, we set the post published element to `0` and update the `posts` database table. Finally, the `Session` object sets the appropriate message, and then we redirect to the blog home page.

Publishing a Post

The `enable` method does the opposite of the `disable` method. It uses the supplied post ID (`$id`) to determine which post record to enable, as shown in Listing 2-9.

Listing 2-9. *The enable Action to Enable (Publish) a Disabled Post*

```
function enable($id=null) {
    $post = $this->Post->read(null,$id);
    if (!$id && empty($post)) {
        $this->Session->setFlash(➥
__('You must provide a valid ID number to enable a post.',true));
        $this->redirect(array('action'=>'index'));
    }
    if (!empty($post)) {
        $post['Post']['published'] = 1;
        if ($this->Post->save($post)) {
            $this->Session->setFlash➥
(__('Post ID '.$id.' has been published.',true));
        } else {
            $this->Session->setFlash(__('Post ID '.$id.' was not saved.',true));
        }
        $this->redirect(array('action'=>'index'));
    } else {
        $this->Session->setFlash(__('No Post by that ID was found.',true));
        $this->redirect(array('action'=>'index'));
    }
}
```

In Listing 2-9, the `enable` method contains the same code sections as that of the `disable` method in Listing 2-8. The difference is in the `enable` method, the statement `$post['Post']['published']` is set to `1`.

Deleting a Post

We can do some housekeeping by removing posts that are no longer needed. We'll use the `delete` method to remove posts from the `posts` database table. Listing 2-10 shows the code for the `delete` method.

Listing 2-10. *The delete Action to Remove Posts*

```
function delete($id = null) {
        if (!$id) {
            $this->Session->setFlash(__('Invalid id for Post', true));
            $this->redirect(array('action'=>'index'));
        }
        if ($this->Post->del($id)) {
            $this->Session->setFlash(__('Post deleted', true));
            $this->redirect(array('action'=>'index'));
        }
    }
?>
```

When the `delete` action is invoked by requesting to delete a post, the dialog box shown in Figure 2-4 appears. Clicking the OK button will permanently remove the selected post record from the database table.

Figure 2-4. *The delete confirmation dialog box*

Creating an RSS Feed

Since we've decided to share our posts with the rest of the world, we can reduce the stress on our blog database by creating an RSS feed—a static XML file that will be updated whenever the posts change.

RSS is an abbreviation for Really Simple Syndication. It is an XML web content syndication format that can be read by using news reader software (or through some online sites and scripts). For the blog application, using an RSS feed means that people can get updated information about your blog without needing to visit your blogging web site. Scripts such as robots can fetch your RSS feed so that your users can be kept informed of changes to your posts.

Essentially, what we'll create is an XML file of the information stored in the `posts` database table. Remember that our code will be writing to a static file stored somewhere on your web site, so it is important to ensure read-write privileges for the document.

You can generate an RSS feed in several ways, such as with the Document Object Model (DOM), XMLWriter, SimpleXML, and so on. However, Cake provides a helper to handle RSS feeds, so you don't need to worry about how to micromanage them. The RSS helper creates standards-compliant RSS feeds. You invoke its functionality with `var $helpers = array('Rss');`.

For the RSS feed for our blog application, we first need to include the following in our `app/config/routes.php` file:

```
Router::parseExtensions( 'rss' );
```

This informs the router to parse out file extensions from the URL. For example, `http://localhost/posts.rss` would yield a file extension of `rss`. It is used by `RequestHandler` components to automatically switch to alternative layouts and templates in order to load the `RssHelper` with `.rss` content.

Caution Don't forget to add `Router::parseExtensions( 'rss' );` to your `app/config/routes.php` file and also ensure that you add `RequestHandler` to your `PostsController`'s `$component` variable. Things will not go well if the two instructions are not added. However, if you already have `Router::parseExtensions`, just add the `rss` string argument. `Router::parseExtensions` accepts many arguments.

Now we need to perform the following tasks:

- Determine the elements (for example, the data fields) that we want to appear in our RSS feed.
- Create the layout for our RSS/XML view.
- Create the post RSS view.

The first task requires that we tweak the `index` method in the `PostsController` class by adding extra lines of code to it. In this case, we will add the channel array data and then fetch the ten most recent post records for public consumption. The content of our new `index` method is shown in Listing 2-11.

Listing 2-11. *The New index Method of the PostsController Modified for RSS*

```
function index() {
    $posts = $this->Post->find('all');

    $channelData = array(
            'title' => 'Current posts | The blogger',
            'link' => array(
                  'controller' => 'posts', 'action' => 'index', 'ext' => 'rss'),
            'url' => array('controller' => 'posts', 'action' => 'index',➥
'ext' => 'rss'),
            'description' => 'The current posts in our blog',
            'language' => 'en-uk'
                );

    $posts = $this->Post->find('all',
                           array('limit' => 10, 'order' => 'Post.created'));

    $this->set(compact('channelData', 'posts'));
    }
```

In Listing 2-11, the channel and posts information are set for the views and layouts, which we'll create next.

The layout file, `app/views/layouts/rss/defaut.ctp`, will contain the following code snippet:

```
<?php
    echo $rss->header();

    $channel = $rss->channel(array(), $channelData, $items);

    echo $rss->document(array(), $channel);
?>
```

Since Cake allows views to pass variables to the layout, we've set the `$items` variable in the layout, instead of following the normal convention of setting variables in a controller.

Next, we will create the RSS view in `app/views/posts/rss/index.ctp`. This file will contain a function called `rss_transform`, which accepts the `$items` (for example, post data) as an argument, and as its name implies, returns a transformed array version of the post data. Listing 2-12 shows the content of the RSS view file.

Listing 2-12. *The Content of the RSS View (app/views/posts/rss/index.ctp)*

```
<?php
    function rss_transform($item) {
        return array('title' => $item['Post']['title'],
            'link' => array('controller' => 'posts', 'action' => 'view',➥
'ext' => 'rss', $item['Post']['id']),
            'guid' => array('controller' => 'posts', 'action' => 'view',➥
```

```
'ext' => 'rss', $item['Post']['id']),
            'description' => strip_tags($item['Post']['content']),
            'pubDate' => $item['Post']['created'],
            );
    }
    $this->set('items', $rss->items($posts, 'rss_transform');
?>
```

In Listing 2-12, the `$rss->items` helper method converts our post data into an XML format and uses the `set` method to pass the converted XML to the RSS layout. Here, the layout is going to do the work of rendering the output when you make the URL request `posts/index.rss`. Your request should display an XML version of the posts as fetched in the `PostsController index` action. When you choose to view the posts, you can select whether to save a copy of the RSS feed file or display it on the screen. Figure 2-5 shows an example of how this looks in Firefox.

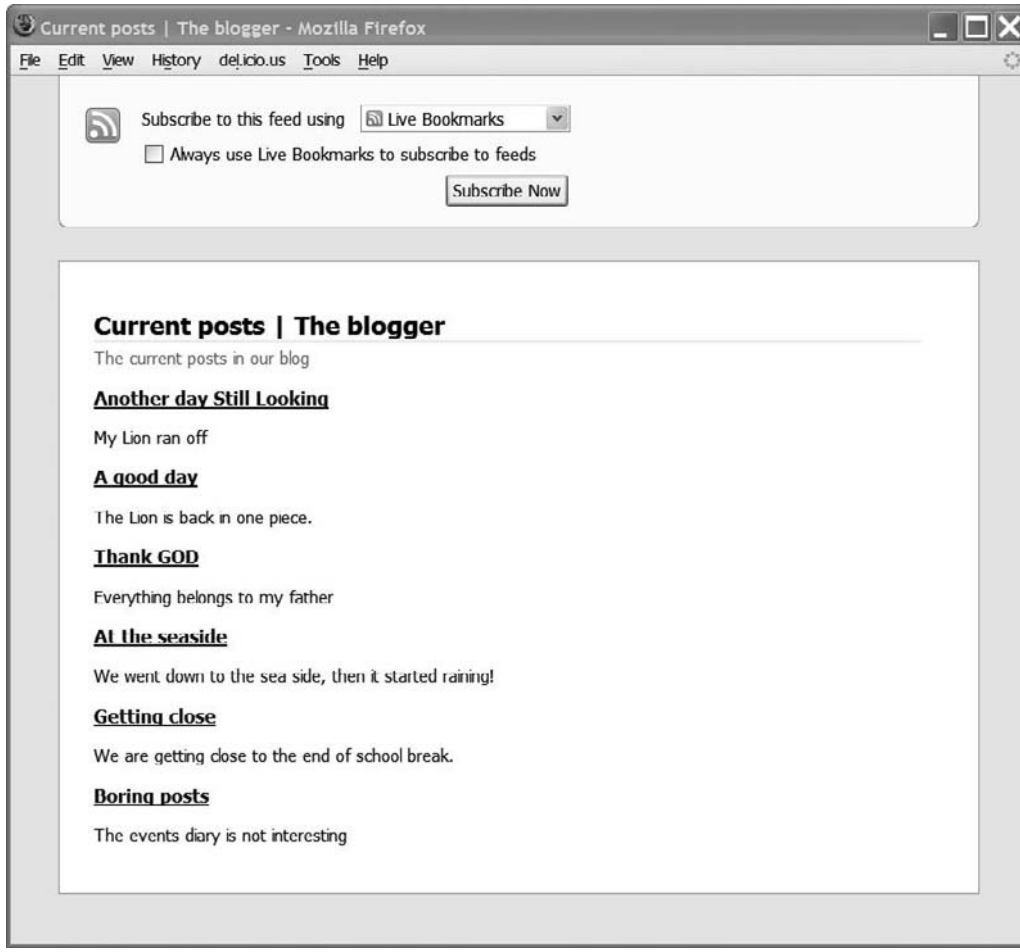

Figure 2-5. *Fetching posts from an RSS feed*

Summary

This chapter described how to build a blogging application using Cake. We first created a `posts` database table and populated it with some sample posts. We took advantage of Cake's components by creating our posts controller and model classes to perform the necessary tasks: list, add, edit, delete, save, enable, disable, and access data. The corresponding action views were created using the `$form` object methods. Finally, we created an RSS feed, with the help of the RSS helper, to present some of our posts to the public.

The modularity of the blogging application allows it to be extended by adding more actions to the controller class. For example, you could add an action to upload images.

CHAPTER 3

E-Commerce

The buzzword e-*commerce*, short for electronic commerce, simply refers to the processing and recording of online transactions. To boast a competitive edge and increase profit, businesses and individuals that peddle services or products must not just have a web presence but also endeavor to sell their merchandise online. There are numerous e-commerce applications based on the PHP scripting language, such as the free, open source osCommerce and Magento. Here, we're going to implement an online shop using the Cake framework.

First, we'll design a skeletal look and feel for our shop, and then we'll create our shop database in MySQL. Next, we'll populate the database with some category and product information. We'll then use Cake features to allow users to select categories and products, add selected products to their shopping cart, click to check out, and, finally, make payments using the popular Google Checkout or PayPal payment system.

This chapter assumes that you're familiar with PHP, MySQL, and Cake. It also assumes that you have set up a development environment that supports these technologies.

The Online Shop Layout

A typical online shop layout is divided into five sections (elements). We'll follow that design and use the Cake view elements listed in Table 3-1. We'll use Cake's features to help us develop these elements of our shop application before stitching them together into the standard layout.

Table 3-1. *View Elements of Online Shop Layout*

Division Name	Description
[Header]	Contains header information, such as the logo and banner
[Left Column]	Displays an expandable category level, sublevel, and product lists
[Center Column]	The main content area of the web site; what it contains depends on the visitor action, so it might show a product description, best-selling products, and so on
[Right Column]	Contains the mini-basket or other elements
[Footer]	Displays summary information about a shop, such as the copyright, contact information, shortcut links, and so on

Tip To include the left navigation element stored in `app/views/elements/menu.php` in a view, use the code `$this->elements('menu');`.

Two Site Layouts

In this example, we're going to use two different layouts:

- Most of the web pages will show the five sections of the web site as illustrated in Figure 3-1. This first layout is stored in the `app/views/layouts/default.ctp` file as the application default layout.
- The second layout will exclude the right column by collapsing the center and the right columns into one column. This layout stored in `app/views/layouts/checkbase.ctp`. We'll use this layout to display the `checkout` action of the orders controller.

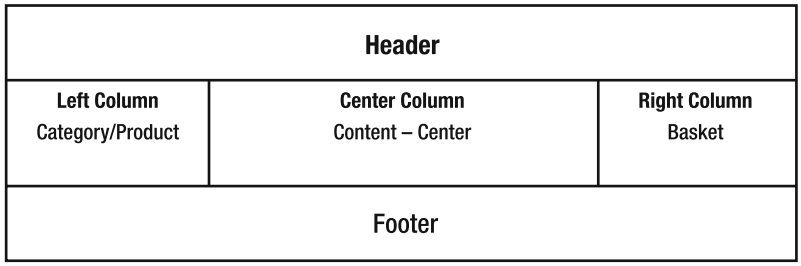

Figure 3-1. *The default layout*

Note One of the aims of this book is to show you how to properly organize elements (code snippets) of an application within the Cake's structure in order to enjoy the benefit of Cake's code reuse and ease of maintenance. However, in this chapter, we will not discuss how our online shop folders and files are organized, as this organization is identical to the structure explained in Chapter 1.

Layout of the Main Content

When a user requests a web page, the `default.ctp` file takes care of the overall look and feel of the web site, which is shown in Figure 3-1. If you want to adapt the main layout of the shop application, just edit the file or create another layout, such as a `checkbase.ctp` file, to suit your needs. However, do not forget to include the `$layout = 'checkbase';` statement in the method of your controller object that requires this new look.

The following code displays only the main content area of the layout:

```
<div id="main_content_container">
    <div id="leftnav"><?php echo $this->element(menu');?></div>
        <div id="main_body_container">
            <?php echo $content_for_layout; ?>
        </div>
    <div id="right_nav_container">
        <?php echo $this->element('basket');?>
    </div>
</div>
```

The `$this->element(menu')` method is used to include the navigation view stored in the `app/views/elements/menu.ctp` file. This file contains the view logic that generates the category tree menu that is displayed in the left column of the layout. This navigation section of the page is expected to display the product categories at all times. The `$content_for_layout` variable includes the view rendered in response to a user action. The `$this->element('basket')` method displays the mini-basket content in the right column of the layout.

The User Journey

Like any successful dynamic web site, an online shop requires planning and a data store of some sort before jumping into the implementation of the application. These tasks include database design, program flow design, resource planning, and so on. Since we are fortunate to have an existing database schema to use for this example, we will skip the first step of categorizing and normalizing our shop data and move on to explaining the program flow, or user journey.

The online shopping process starts with a customer surfing the Internet and ends with the customer parting with some cash, which ends up in some businessman's online account.

This basic flow for our online shop example looks like this:

1. A user visits our shop.
2. The user browses the categories and products.
3. The user views the product details.
4. The user adds products to the shopping basket.
5. The user clicks to check out and pay.
6. The customer receives an e-mail confirmation notice.

In our example, users do not need to register. They browse categories and products, pay for selected items, and leave the shop.

During the user journey, some vital transaction information is collected and stored in database tables. Needless to say, the tracked data is used for transaction completion and for making other business decisions.

Setting Up the Shop Database

First, we need to connect our application to a MySQL database server. Our configuration, stored in the `app/config/database.php` file, includes the following array definition:

```
var $default = array('driver' => 'mysql',
        'persistent' => false,
        'host' => 'localhost',
        'login' => 'user',
        'password' => 'password',
        'database' => 'shop',
        'prefix' => '',
);
```

Do not forget to replace the current database access information with your local username and password. If you need more information about how to configure a database connection, see Chapter 1.

We'll create three database tables for our online shop application:

- The `categories` table stores product categories.
- The `products` table stores the product descriptions.
- The `carts` table stores selected product items.

Note We've said that we are not going to collect personal user information in our online shop application. So how do we identify a user? We will create and use a session ID for every unique user request. For information about implementing authentication, see Chapter 8. Also, some common online shop tables, such as the `order` and `order_item` tables, are left out of this example. Any table schema not presented in this section is skipped for the sample implementation of this application.

As indicated in Figure 3-1, the left column of our shop layout will present a navigation menu tree that is generated from the product categories data. The `categories` table and data are created by the SQL statements shown in Listing 3-1.

Listing 3-1. *The categories Table Schema*

```
CREATE TABLE IF NOT EXISTS `categories` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `parent_id` int(11) NOT NULL default '0',
  `name` varchar(50) character NOT NULL,
  `description` varchar(200) character NOT NULL,
  `image` varchar(255) character NOT NULL,
  PRIMARY KEY  (`id`),
  KEY `cat_parent_id` (`parent_id`),
  KEY `cat_name` (`name`)
);
```

```
INSERT INTO `categories` (`id`, `parent_id`, `name`, `description`, `image`) VALUES
(17, 0, 'Jazz', 'Everything from 1890s', ''),
(12, 0, 'Classical', 'From Medieval to Contemporary', ''),
(13, 17, 'Dizzy Gillespie', 'The Trumpeter Master', ''),
(14, 12, 'Mozart', 'The Old Favourite', '');
```

Next, our categories need to relate to some product information, which we'll put in a `products` table. Listing 3-2 presents the SQL statements to create this table and populate it with some sample data.

Listing 3-2. *The products Table Schema*

```
CREATE TABLE IF NOT EXISTS `products` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `category_id` int(10) unsigned NOT NULL default '0',
  `name` varchar(100) character NOT NULL default '',
  `description` text character NOT NULL,
  `price` decimal(9,2) NOT NULL default '0.00',
  `qty` smallint(5) unsigned NOT NULL default '0',
  `image` varchar(200) character default NULL,
  `thumbnail` varchar(200) character default NULL,
  `created` datetime NOT NULL default '0000-00-00 00:00:00',
  `modified` datetime NOT NULL default '0000-00-00 00:00:00',
  PRIMARY KEY  (`id`),
  KEY `cat_id` (`category_id`),
  KEY `name` (`name`)
);

INSERT INTO `products` (`id`, `category_id`, `name`, `description`, ➥
`price`, `qty`, `image`, `thumbnail`, `created`, `modified`) VALUES
(1, 13, 'Dizzy 1990s', 'Best of Dizzy Gillespie in the 1990s', 12.00, ➥
10, NULL, '1.jpg', '0000-00-00 00:00:00', '0000-00-00 00:00:00'),
(2, 14, 'Mozart for Lovers', 'Relax with your loved one with this ➥
double CD.', 15.00, 5, NULL, NULL, '0000-00-00 00:00:00', '0000-00-00 00:00:00'),
(22, 13, 'Dizzy and Stan', 'Live with Dizzy Gillespie and Stan Getz.', ➥
13.00, 10, NULL, '1.jpg', '0000-00-00 00:00:00', '0000-00-00 00:00:00');
```

When a user selects a category, the products linked to this category are displayed. For example, when you select the category called Jazz, you'll see the Dizzy 1990s item and the Dizzy and Stan item listed as products under this category.

Next on the user journey, when a user has selected the type of music she wants to buy and she is happy with the vibes, she adds her selections to a shopping basket. The user-selected items are stored in the `carts` database table. This table and some sample data are created using the SQL in Listing 3-3.

Listing 3-3. *The carts Table Schema*

```
CREATE TABLE IF NOT EXISTS `carts` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `product_id` int(10) unsigned NOT NULL default '0',
  `qty` mediumint(8) unsigned NOT NULL default '1',
  `ct_session_id` char(32) NOT NULL,
  `created` datetime NOT NULL default '0000-00-00 00:00:00',
  PRIMARY KEY  (`id`),
  KEY `pd_id` (`product_id`),
  KEY `ct_session_id` (`ct_session_id`)
);

INSERT INTO `carts` (`id`, `product_id`, `qty`, `ct_session_id`, `created`) VALUES
(52, 1, 2, '32cabb9d6fb31404a5b736830eac6aa3', '2008-09-24 15:46:53'),
(51, 2, 3, '32cabb9d6fb31404a5b736830eac6aa3', '2008-09-24 15:29:01');
```

■ **Tip** Some online shops store configuration information in a table. Alternatively, this information can be stored using Cake's configuration method—for example, `Configure::write('Shop.name', 'Sound Empire');`. To access the shop name, use Cake's configuration `read` method, as in `Configure::read('Shop.name')`. As you can imagine, a shop administrator would find it more difficult to update these configuration parameters than to work with information stored in a database table. Modifying the parameters would require physical file system access and consequently some manipulation fuss.

Interacting with the Online Shop Database

Now that we're connected to the database, created the required tables, and populated them with some sample data, our application needs the information stored in the database in order to provide the initial content of the web pages. For example, the site navigation requires the category data stored in the `categories` table to do its bit. So who handles the tasks of accessing and manipulating information stored in the database? You'll be happy to learn that Cake provides default model functions that serve as shortcuts to database operations. These prebuilt functions allow for easy and fast application development. Now let's start digging out some information.

First, we'll create our custom objects to handle the display of an expandable category level, sublevel, and product lists.

The Category Model

Let's create the `Category` model class to interact with the `categories` table data created in Listing 3-1. The beginning of this model is shown in Listing 3-4.

Listing 3-4. *The Beginning of the Category Model (app/models/category.php)*

```
<?php
class Category extends AppModel {
    var $name = 'Category';
    var $hasMany = array('Product');
```

In this model, we have two important properties:

- The `$name` property will serve as a reference to this model object within the controllers of this application when needed.
- The association property `$hasMany` simply defines the relationship between the `categories` table and the `products` table. In this case, we define that a `categories` record can relate to many `products` records. In our application, the category `Jazz` is associated with the products `Dizzy 1990s` and `Dizzy and Stan` (see Listings 3-1 and 3-2). This property is required to ensure an association between the `Category` and `Product` model objects.

Now that the `Category` model class is attached to our `categories` table data and it has established a relationship with the `products` table data, we can create some custom methods in the model class to interact with the `categories` and `products` tables. These custom methods will internally use Cake's default properties and methods, such as the `find` method for database create, read, update, and delete (CRUD) operations.

The `Category` model class will contain three methods to provide information in the final format needed to generate the products category navigation: `getCategories`, `buildCategories`, and `getChildCategories`.

The first method, `getCategories`, returns the entire category list. This method is shown in Listing 3-5.

Listing 3-5. *The Category Model's getCategories() Method*

```
function getCategories($field='Category.id',$direction='ASC') {
    return $this->find('all',array('order'=>$field.' '.$direction));
}
```

The method in Listing 3-5 is employed whenever we require the list of all the product categories from the `categories` table, in ascending order of the category IDs. We've provided some parameters, `$field` and `$direction`, so that it's possible to use this function in more than one way elsewhere in the application.

Next in this model class is the `buildCategories` method, which returns the currently selected category and its children categories. This method is shown in Listing 3-6.

Listing 3-6. *The Category Model's buildCategories() Method*

```
function buildCategories($categories, $parentId)
{
    $ChildCategories = array();
    $ids = array();
    foreach ($categories as $category) {
        if ($category['Category']['parent_id'] == $parentId) {
            $ChildCategories[] = $category['Category'];
        }
        $ids[$category['Category']['id']] = $category['Category'];
    }
    $HoldParentId = $parentId;
    while ($HoldParentId != 0) {
        $parent    = array($ids[$HoldParentId]);
        $currentId = $parent[0]['id'];
        $HoldParentId = $ids[$HoldParentId]['parent_id'];
        foreach ($categories as $category) {
            if ($category['Category']['parent_id'] == $HoldParentId && ➥
!in_array($category['Category'], $parent)) {
                $parent[] = $category['Category'];
            }
        }
        array_multisort($parent);
        $n = count($parent);
        $ChildCategories2 = array();
        for ($i = 0; $i < $n; $i++) {
            $ChildCategories2[] = $parent[$i];
            if ($parent[$i]['id'] == $currentId) {
                $ChildCategories2 = array_merge($ChildCategories2,
                                                $ChildCategories);
            }
        }
        $ChildCategories = $ChildCategories2;
    }
    return $ChildCategories;
}
```

This method takes two arguments: an array of the entire category records from the `categories` table (`$categories`) and a category's parent ID (`$parentId`). The `getCategories` method in Listing 3-5 supplies the `$categories` array data used as the first argument in the `buildCategories` method in Listing 3-6.These two `Category` model methods, together with the `$this->passedArgs['c']` obtained from the user request, are used in the `CategoriesController` class method called `menu` to generate the application category navigation presented in the left column of the web pages. The `CategoriesController` is discussed in the next section.

In Listing 3-6, first we declare two array variables: `$ChildCategories` and `$ids`. Using a `for` loop over `$categories`, if the current category's parent ID equals that passed as an argument (`$parentId`), we add the current category to the `$ChildCategories` array variable. We also add

the current category to the $ids array, making sure that the $ids array key is the category ID of the current category. This is followed by a while loop to create a parent category and append children categories appropriately. Finally, the function formats and returns the current category list, which includes only the currently selected category and its children. This function is made so it can also handle deep category levels (more than two levels).

Next, we create the getChildCategories method, as shown in Listing 3-7. This method returns a list of the entire category IDs that belong to the children of a specified category.

Listing 3-7. *The Category Model's getChildCategories() Method*

```
function getChildCategories($categories, $id, $recursive = true)
{
    if ($categories == NULL) {
        $categories = $this->getCategories();
    }
    $n     = count($categories);
    $child = array();
    for ($i = 0; $i < $n; $i++) {
        $catId    = $categories[$i]['id'];
        $parentId = $categories[$i]['parent_id'];
        if ($parentId == $id) {
            $child[] = $catId;
            if ($recursive) {
            $child = array_merge($child,
                                 $this->getChildCategories($categories, $catId));
            }
        }
    }
    return $child;
}
```

The arguments supplied to the getChildCategories method contain the following information, in the order presented here:

- $categories: All the category lists stored in the categories table.
- $id: A category ID that requires its children category IDs.
- $recursive: Whether it should include all levels deep of children categories in the operation. Its default value is the Boolean true.

Next in this method we use an if statement to check for a valid category list, if the getChildCategories method is not supplied with a valid category argument. This means that if there is no category information from the categories table stored in the $categories array, we use the getCategories method to generate the entire category list and assign the result to the $categories variable.

Next, we count the number of elements in the $categories array and assign the result to the $n variable. We then declare an array variable called $child, which will contain a list of the children category IDs.

We then loop over the $categories array variable with a for loop and assign a category ID and its parent ID to $catId and $parentId, respectively. Next, using an If statement, we check that the current parent ID ($parentId) we've extracted is the one for which we want to obtain children category IDs. If the $parentId is the same as the one passed to the getChildCategories method as an argument ($id), we add the current category ID to the $child array variable. By default, this method will use itself to perform the same tasks based on the current category ID, unless the $recursive argument is set to Boolean false. Lastly, this method will return the $child array variable containing all children category IDs of a specified category parent ID.

The getChildCategories method in Listing 3-7 is used within the ProductsController class method called lists to provide this function with a list of category IDs. This function uses the list of category IDs, formatted as {1,2,3}, to fetch a list of all products that belong to these category IDs, presented in the middle column of the web page. The ProductsController is discussed a little later in this chapter.

The Categories Controller

Now that we've created the Category model, let's delve into the CategoriesController class, as shown in Listing 3-8.

Listing 3-8. *The CategoriesController Class (app/controllers/categories_controller.php)*

```
<?php
class CategoriesController extends AppController {
    var $name = 'Categories';
    function getAll() {
        return $this->Category->getCategories();
    }
    function menu() {
        $categories = $this->getAll();
        return $this->Category->buildCategories($categories,
                                                $this->passedArgs['c']);
    }
}
?>
```

In the CategoriesController, we make use of the services provided by the methods defined in the Category model, discussed in the previous section. Additionally, one of the properties inherited from the AppController is the $uses array. This array includes an element that provides a reference to the Category model.

Next, we declare the getAll method, which simply provides the list of all the categories information from the categories table. This method returns the result of the $this->Category->getCategories() statement when called by another object.

The last method provided by the CategoriesController is menu. As the name implies, it helps with the creation of the categories navigation (menu) shown in the left column of the web site. The first statement of this method stores the categories information in the $categories array using the getAll method previously defined in this class. This array variable, as well as the current category ID (supplied by $this->passedArgs['c']), is then passed to the buildcategories() method to create an array formatted for the menu view. The array returned by the buildcategories method on the home page of the web site is structured as follows:

```
Array
(
    [0] => Array
        (
            [id] => 12
            [parent_id] => 0
            [name] => Classical
            [description] => From Medieval to Contemporary
            [image] =>
        )

    [1] => Array
        (
            [id] => 17
            [parent_id] => 0
            [name] => Jazz
            [description] => Everything from 1890s
            [image] =>
        )
)
```

This array structure contains two first-level category records; that is, categories with their parent IDs equal to an integer value of `0` (zero).

Next, we call the `menu` method of the `CategoriesController` within the `app/views/elements/menu.ctp` file to supply this view script with the categories information required to display the category navigation menu. The view takes the category list (stored in `$categories`) and displays its content in Cascading Style Sheets (CSS) `<ul>` and `<li>` tags in the left column of the layout. Since the category navigation is central to this application—that is, it must always be displayed—we've placed the following code snippet in the `default.ctp` layout file to include this element of the web site at all times:

```
<div id="leftnav"><?php echo $this->element(menu');?></div>
```

The content of the `menu.ctp` view file is shown in Listing 3-9.

Listing 3-9. *The Category Navigation View for the Left Column (app/views/elements/menu.ctp)*

```
<ul>
<li><?=$html->link('All Category','/');?></li>
<?php
$categories = $this->requestAction("/categories/menu/c:$catId/p:$pdId/ ");
foreach ($categories as $category) {
    extract($category);
    $level = ($parent_id == 0) ? 1 : 2;
    $url   = '/carts/index/cat_id:' . $id;
    if ($level == 2) {
        $name = "~~~" . $name;
    }
    $listId = '';
```

```
    if ($id == $catId) {
        $listId = ' id="current"';
    }
?>
<li<?php echo $listId; ?>><?=$html->link($name, $url);?></li>
<?php
}
?>
</ul>
```

In Listing 3-9, the first line begins with the `<ul>` opening tag. Next, using the `$html` helper object, we created a link to display the message `'All Category'`.

The first line of the `<?php` opening tag uses Cake's `requestAction` function. It requests the `menu` method of the `CategoriesController` object to provide a list of categories information (stored in the `$categories` array variable) that match the category and the product IDs passed to this method using the `c:$catId` and `p:$pdId` parameters, respectively.

Finally, using a `foreach` loop, the `$categories` array content is extracted to generate the category navigation that appears in the left column throughout the application. Figure 3-2 shows the application response when the Classical menu item is selected by a user, with the submenu item Mozart displayed.

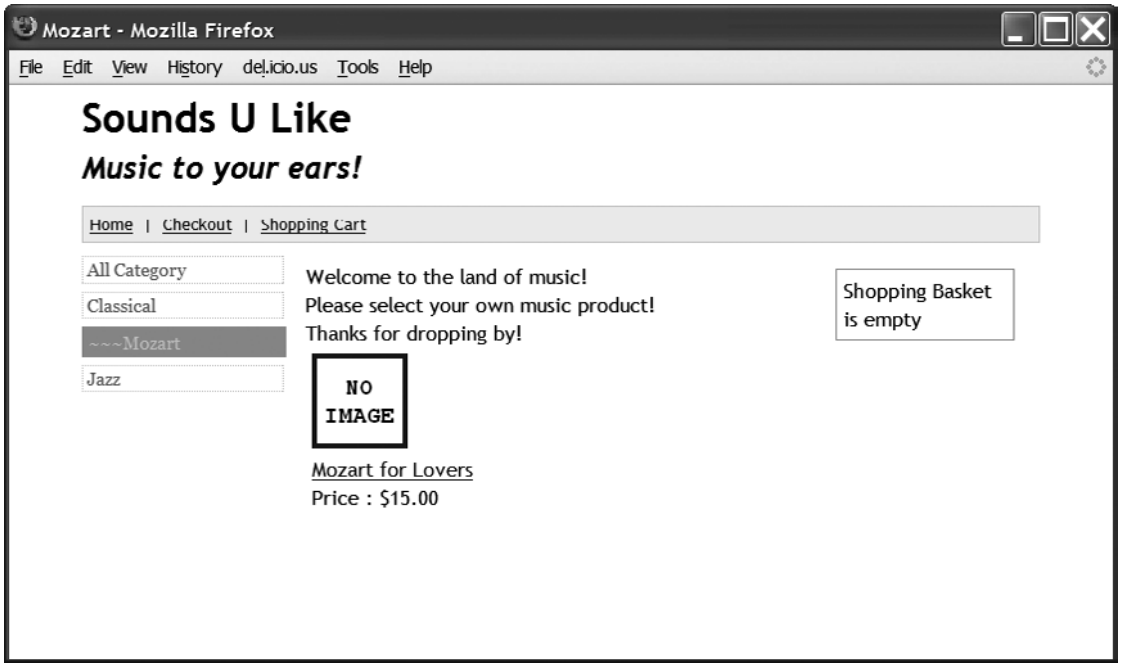

Figure 3-2. *The screen showing levels of categories*

The Product Model

Our next task is to implement the `Product` model class, which as the name suggests, provides product information from the `products` table. We will begin with the class properties, as shown in Listing 3-10.

Listing 3-10. *The Beginning of the Product Model (app/models/product.php)*

```
<?php
class Product extends AppModel
{
    var $name = 'Product';
    var $belongsTo = array('Category');
```

In the `$name` property, the `$belongsTo` array variable tells the application that a product record belongs to a category. (The `$name` property works as described in the earlier discussion of the `Category` model.)

Next is the `lists` method, shown in Listing 3-11. This method provides all the product records based on one or more category IDs in the ascending order of product category ID.

Listing 3-11. *The Product Model's lists() Method*

```
function lists($catIds = NULL) {
    if(is_array($catIds)) {
    $results = $this->find( 'all', array(
        'conditions'=>array('Product.category_id'=>$catIds),
        'order' => 'Product.category_id ASC' ));
    return $results;
    }
}
```

The Products Controller

Now that we've created the `Product` model class, we need an object to provide some data and services to other objects in our shop application. The `ProductsController` class, shown in Listing 3-12, defines the `lists` and `view` methods to provide these services when called by other objects or scripts within the application.

Listing 3-12. *The ProductsController Class (app/controllers/products_controller.php)*

```
<?php
class ProductsController extends AppController {
    var $name = 'Products';
    function lists() {
        $categories = $this->Category->find( 'all',
                                        array( 'order' => 'Category.id ASC' ));
        $categories = $this->Category->buildCategories($categories,
                                        $this->passedArgs['c']);
        $children_ids  = $this->Category->getChildCategories($categories,
                                           $this->passedArgs['c'], true);
```

```
        $allCatIds = array_merge(array($this->passedArgs['c']), $children_ids);
        return $this->Product->lists($allCatIds) ;
    }

    function view() {
        $result = $this->Product->find( 'all',
                                array('conditions'=>array('Product.id'=>
                                            $this->passedArgs['pd_id'] )));
        if (empty ($result) ) {
            $this->redirect(array('controller'=>'/ecommerce', 'action'=>'index'));
        }
        $this->set('product', $result);
    }
}
?>
```

The `lists` method in Listing 3-12 returns a list of products based on one or more category IDs. The first statement in this method retrieves all the categories information and stores the result in the `$categories` array. This array is then used as the first argument of the `buildCategories` method of the `Category` model to provide a filtered set of category information, stored in the `$categories` array variable. The second argument, `$this->passedArgs['c']`, of this method determines the family of category information that is required.

Next is the `getChildCategories` method, which accepts the newly created `$categories` array as the first argument, the current category ID (stored in `$this->passedArgs['c']`), and a Boolean `true`. We talked about this method earlier in the discussion of the `Category` model class (Listing 3-7). Here, this method returns the list of all children category IDs that belong to the category ID supplied as second argument of the method. Next, using the PHP `array_merge` function, the current category ID and that of the children are merged together and stored in the `$allCatIds` array variable.

Finally, the `$allCatIds` variable is passed to the `lists` method of the `Product` model, which returns a list of all products that belong to the category IDs available in the `$allCatIds` variable. The `lists` method of the `ProductsController` is called or triggered when a user clicks a category. This method is called in the `app/views/elements/products.ctp` view file, which is shown in Listing 3-13.

Listing 3-13. *The Product List View for the Center Column (app/views/elements/products.ctp)*

```
<?php
$products = $this->requestAction("/products/lists/c:$catId/");
foreach($products as $product): ?>
<div class="product_container">
<?php
    if ( $product['Product']['thumbnail'] ) {
        $thumbnail = '/img/products/' . $product['Product']['thumbnail'];
    } else {
        $thumbnail = 'no-image-small.png';
    }
```

```
        e( $html->link(
            $html->image($thumbnail, array( 'border' => '0' )),
            array('controller' =>'products',
            'action' => "view/cat_id:$catId/pd_id:".$product['Product']['id']
            ),
            array('escape' => false))); ?>
    <br>
    <? echo $html->link( $product['Product']['name'], ➥
"/products/view/cat_id:$catId/pd_id:".$product['Product']['id'] ); ?>
    <br>Price : $<?=$product['Product']['price'];
?>
</div>
<? endforeach;?>
```

In Listing 3-13, the first line uses Cake's `requestAction` function to request the `lists` method of the `ProductsController` object to provide a list of product information that matches the category ID passed to this method via the `c:$catId` parameter. The resulting list of products is stored in the `$products` array variable. Finally, using a `foreach` loop, the content of the `$products` array is displayed in the center of the web page. The resulting view is shown in Figure 3-3.

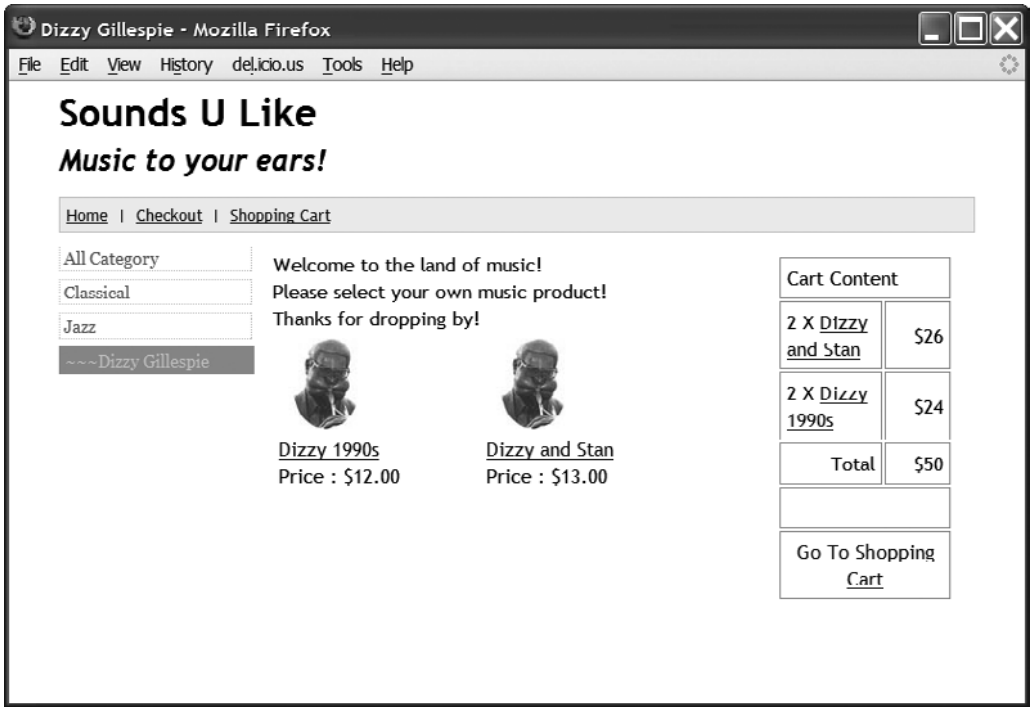

Figure 3-3. *The view rendered when a category ID is selected*

The last method in Listing 3-12, `view`, uses Cake's `set` method to automagically pass a full product description to the `app/views/products/view.ctp` file. Using the default `find` method against the `Product` object, our `view` method retrieves a record of a product from the `products` table based on a specified product ID (retrieved from `$this->passedArgs['pd_id']`). This product information is stored in the `$result` array variable. If `$result` is empty, then the application redirects the user to the home page; otherwise, the current product information is displayed on the page.

If the product ID is set, then the content of the product details view stored in the `app/views/elements/product_details.ctp` file will be displayed in the center column. This view file is shown in Listing 3-14.

Listing 3-14. *The Product Details View for the Center Column (app/views/elements/product_details.ctp)*

```
<?php
    if ( $product[0]['Product']['image'] ) {
        $pd_thumbnail = '/img/products/' . $product[0]['Product']['image'];
    } else {
        $pd_thumbnail = 'no-image-small.png';
    }
    e($html->image($pd_thumbnail,
                   array( 'border' => '0', 'width'=>'150', 'height'=>'150' )));
?>
<br />
<strong><?php echo $product[0]['Product']['name']; ?></strong>
<br />
Price : <?php echo Configure::read('Shop.currency'); ?>
<?php echo $product[0]['Product']['price']; ?><br>
<?php
    if ($product[0]['Product']['qty'] > 0) {
        e($html->link('Add to Shopping Basket',
           '/carts/add/cat_id:'.$catId.'/pd_id:'.$product[0]['Product']['id']));
    } else {
        e('Out Of Stock');
    }
?>
<br />
<?php
    e( nl2br($product[0]['Product']['description']));
?>
```

In Listing 3-14, we use the `$product` array variable passed in using `$this->set('product', $result)` from the `view` method of the `ProductsController`. It displays the product details in the center of the web page, as shown in the example in Figure 3-4.

The `ProductsController` and other controllers employ the services of the `Product` and `Category` models. Reference to these models is available in the `AppController` class, as discussed a little later in the chapter.

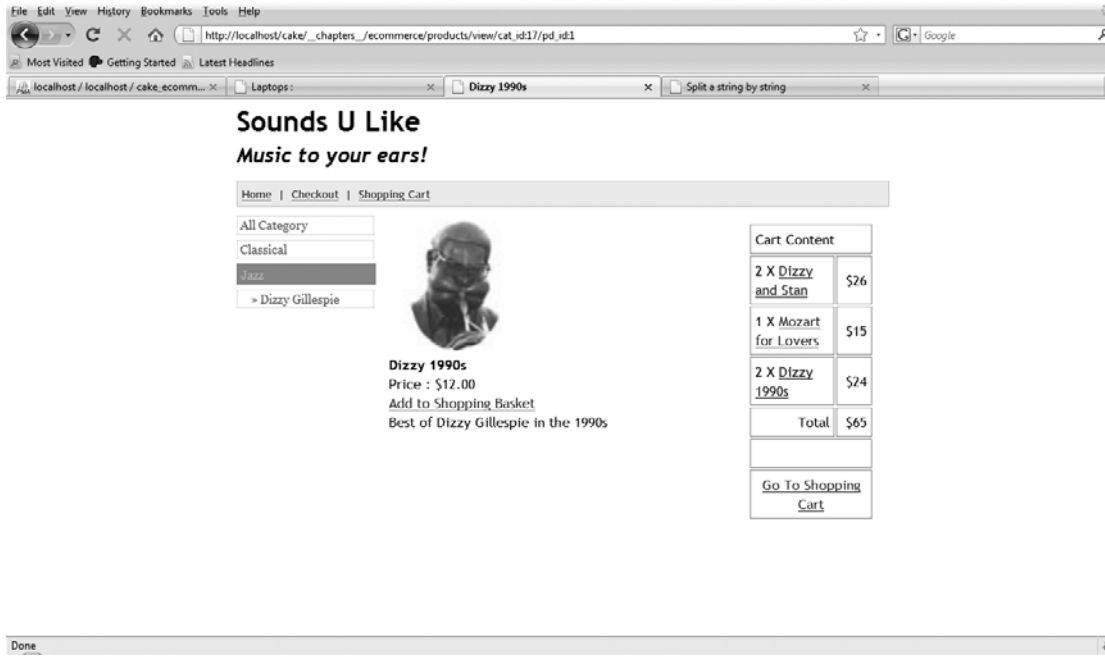

Figure 3-4. *The index view rendered when a product is selected*

The Cart Model

Next, we'll implement the `Cart` model class. Along with supplying the content of the shopping cart to the application, the `Cart` model provides functionalities to manipulate the cart, as well as the checkout section of the application. We will begin with the class properties, shown in Listing 3-15.

Listing 3-15. *The Beginning of the Cart Model (app/models/cart.php)*

```
<?php
class Cart extends AppModel
{
    var $name = 'Cart';
    var $hasMany = array('Product');
```

The `Cart` class starts with the class definition, as usual, and then establishes its relationship with the `Product` model object using `var $hasMany = array('Product');`. This simply states that a cart can contain one or more products.

We'll create the following methods in the Cart model:

- getCart
- isCartEmpty
- addCart
- updateCart
- cleanUp
- emptyBasket
- doUpdate
- getCartContent

The getCart method supplies all the information stored in the carts table. This method is shown in Listing 3-16.

Listing 3-16. *The Cart Model's getCart() Method*

```
function getCart($pid, $sid) {
    return $this->find( 'all', array(
    'conditions'=>array('Cart.product_id'=>$pid,
                        'Cart.ct_session_id'=>$sid),
                        'order' => 'Cart.id ASC' ));
}
```

The getCart method takes two arguments: $pid (product ID) and $sid (session ID). These are used in Cake's find method to filter the provided carts table records. The product ID is extracted from a user's URL request. The session ID is obtained from Cake's Session object. These variables are extracted in the beforeFilter method of the AppController object, as you will see later in this chapter.

Next is the isCartEmpty method, which checks whether there is at least one record in the carts table. This method is shown in Listing 3-17.

Listing 3-17. *The Cart Model's isCartEmpty() Method*

```
function isCartEmpty($sid = NULL) {
    $result = $this->find('first',
                array('conditions'=>array('Cart.ct_session_id'=>$sid),
                                          'recursive' => 0 ));
    if ( empty( $result ) ) {
        return true;
    }
        return false;
}
```

The isCartEmpty method accepts $sid (session ID) as its argument and uses the find method to check whether the session ID exists in the carts table. If this session ID exists in the carts table, true is returned; otherwise, false is returned, and the message "Shopping Basket is empty" is displayed. The status of the basket section, on the right side of the web site, depends on the value returned by this method.

Next, we need the function to add a product to the carts table, addCart, as shown in Listing 3-18.

Listing 3-18. *The Cart Model's addCart() Method*

```
function addCart($product_id, $sid) {
    $this->data[ 'Cart' ][ 'product_id' ] = $product_id;
    $this->data[ 'Cart' ][ 'qty' ] = 1;
    $this->data[ 'Cart' ][ 'ct_session_id' ] = $sid;

    $this->save();
}
```

The addCart method takes a product ID and the session ID as its arguments. Here, we use Cake's model method save to add the product ID, quantity, and session ID. We do not need to fill in the cart table's created field (see Listing 3-3), as this is one of Cake's magic fields and will be filled in automatically by the model when the save method is called.

Now we need a function to update the quantity of a product in the carts table. This is the updateCart, as shown in Listing 3-19.

Listing 3-19. *The Cart Model's updateCart() Method*

```
function updateCart($product_id, $sid) {
    $sql = "UPDATE carts
    SET qty = qty + 1
    WHERE ct_session_id = '$sid' AND product_id = $product_id";
    $this->query( $sql );
}
```

This is triggered when a user clicks the Add to Shopping Basket link on the product description page. If the displayed product ID and session ID match that of the argument, the record quantity is incremented by one.

Next, we include a function to remove old cart records, cleanUp. Sometimes customers add products to the shopping basket, but don't bother to either complete the transaction or empty the basket. The method shown in Listing 3-20 will handle cleaning up abandoned carts' records.

Listing 3-20. *The Cart Model's cleanUp() Method*

```
function cleanUp() {
    $threeDaysAgo = date('Y-m-d H:i:s',
                         mktime(0,0,0, date('m'),
                                       date('d') - 3,
                                       date('Y')));

    $delete_condition = "Cart.created < '$threeDaysAgo'";
    $this->deleteAll( $delete_condition, false);
}
```

The cleanUp method deletes records that were added to the basket three days ago from the carts table.

Next is the emptyBasket method, shown in Listing 3-21. This method is used as the delete function that is triggered when a user clicks the Delete button on the checkout page.

Listing 3-21. *The Cart Model's emptyBasket() Method*

```
function emptyBasket($cartId = NULL) {
    if ($cartId) {
      $this->delete( $cartId );
    }
}
```

The emptyBasket method deletes a record from the carts table based on the cart ID.

We also need to be able to update the quantity of products a customer wants to buy. This functionality is provided by the doUpdate method, as shown in Listing 3-22.

Listing 3-22. *The Cart Model's doUpdate() Method*

```
function doUpdate($newQty, $catId) {

    // Update product quantity
    $this->data[ 'Cart' ][ 'qty' ] = $newQty;
    $this->id = $catId;

    $this->save();
}
```

The doUpdate method accepts two arguments: $newQty supplies the new quantity of products the customer wants to purchase, and $catId specifies the ID of the carts table record to be updated.

Finally, we add the getCartContent method to find out what's in the cart, as shown in Listing 3-23.

Listing 3-23. *The Cart Model's getCartContent() Method*

```
function getCartContent( $sid ) {
    $cartContent = array();
    $sql = "SELECT ct.id, ct.product_id, ct.qty, pd.name, pd.description,
            pd.price, pd.thumbnail, pd.category_id
            FROM carts ct, products pd, categories cat
            WHERE
            ct_session_id = '$sid' AND
            ct.product_id = pd.id AND
            cat.id = pd.category_id";
    $results = $this->query( $sql );
    foreach ($results as $result ) {
        $cartContent[] = $result;
    }
    return $cartContent;
}

}
?>
```

The getCartContent method returns the contents of the carts table where the session ID field value matches the argument $sid. The result, stored in the $cartContent array, contains additional product and category information relating to the retrieved cart's contents. If there is no match, an empty array variable is returned.

Handling User Requests

Now that we have created all the functionality needed to interact with our database, we can begin the process of building our application's controllers to handle user requests. We will need to decipher which action should be invoked and employ Cake's MVC tricks to tie these actions to views that render the appropriate application display. We will start by creating the master controller, called the AppController class. Our online shop controllers will borrow some properties and methods from this master controller to save the time of creating them in each individual controller.

The AppController Class

By default, a Cake controller class extends the AppController class. You've already been introduced to the app/app_controller.php class file in Chapter 1. Our shop AppController class will contain all the common functionality required to centralize request handling and extract session IDs. The global properties and methods defined here are automatically available to the objects of requests or other classes that extend AppController.

Listing 3-24 shows the AppController class properties.

Listing 3-24. *The Beginning of the AppController Class (app/app_controller.php)*

```
<?php
class AppController extends Controller {
    var $pageTitle = 'Chapter 3 - Ecommerce';
    var $sid;
    var $catId;
    var $pdId;

    var $uses = array('Product', 'Category', 'Cart');
    var $helpers = array( 'Form', 'Html',  'Session', 'Javascript'  );
    var $components = array( 'Session', 'RequestHandler', 'Shop' );
```

The $pageTitle property sets an initial page title for this application, which can change according to a user's page request. The $sid property will contain a session ID to identify our application user. As we mentioned earlier, to avoid repetition of code within this book, we've decided not to address authentication in this application. Hence, we've used the $sid variable to represent and track a user from page to page.

The $catId and $pdid properties will be assigned the user-requested category and product values, respectively, when available. If these values are not available, we set the properties to an integer value of 0. Next in Listing 3-24, we include some of Cake's built-in properties, such as the $uses array to require the Product, Category, and Cart model objects. Then we include some helpers and components, which were introduced in Chapter 1.

Next, we override the `beforeFilter` method, which is triggered first by default, before any other action in our `AppController`, as shown in Listing 3-25.

Listing 3-25. *The AppController's beforeFilter() Method*

```
function beforeFilter() {

    if ( isset( $this->passedArgs['cat_id'] ) &&
         (int)$this->passedArgs['cat_id'] != 1 )  {
        $this->catId = (int)$this->passedArgs['cat_id'];
    } else {
        $this->catId = 0;
    }

    if ( isset( $this->passedArgs['pd_id'] ) &&
         $this->passedArgs['pd_id'] != '' )  {
        $this->pdId = (int)$this->passedArgs['pd_id'];
    } else {
        $this->pdId = 0;
    }

    $data = $this->Session->read();
    $this->sid = $data['Config']['userAgent'];
    $this->set('catId', $this->catId);
    $this->set('pdId', $this->pdId);
    $this->set('sid', $this->sid);
    $this->setPageTitle();
}
```

In Listing 3-25, the overridden method checks the URL of a user request to extract and then determine the category ID and product ID values. The `beforeFilter` method ensures that during the course of the application, the `$catId` and `$pdId` variables contain some integer values. For example, with this URL:

```
http://localhost/cake/__chapters__/ecommerce/
```

the values of these two properties will default to `0`. If the URL is as follows the value of the current category ID will be `12` and the product ID will be `0`.

```
http://localhost/cake/__chapters__/ecommerce/ecommerce/index/cat_id:12
```

In this case, the `$this->passedArgs['cat_id']` array stores the requested value of `cat_id:12` of the URL.

The last statement of the `beforeFilter` method, `$this->setPageTitle()`, ensures that as a user moves from page to page, the current page title is reflected.

The `setPageTitle` method, which sets the current page title, is shown in Listing 3-26.

Listing 3-26. *The AppController's setPageTitle() Method*

```
function setPageTitle() {
    if ( $this->pdId > 0 ) {
        $result = $this->Product->find('all',
                    array('conditions'=>array('Product.id' => $this->pdId )));
        $this->pageTitle = $result[0]['Product']['name'];
    } elseif ( $this->catId > 0 ){
        $result = $this->Category->find('all',
                    array('conditions'=>array('Category.id' => $this->catId )));
        $this->pageTitle = $result[0]['Category']['name'];
    }
}
```

If the current product ID value (`$this->pdId`) is greater than zero, the value of `$this->pdId` is used to query the `products` table in order to extract the corresponding product name. The `$pageTitle` property is then assigned the appropriate product name. If the current category ID value is greater than zero, then the category ID (`$this->catId`) is used to pull the corresponding category name from the `categories` table, and finally, this category name is used as the page title. If the function fails to set a page title, the `$pageTitle` remains `Chapter 3 - Ecommerce`, as defined in the property section of the `AppController`.

Now that we have defined the properties and functionalities that we require in our `AppController` class, let's tie things together to enable users to navigate our shop.

The Home Page

The starting point is the definition of our preferred home page. The `app/config/routes.php` file contains information about handling user requests for our application. We use the following line in our `routes` file:

```
Router::connect('/', array('controller' => 'carts', 'action' => 'index'));
```

This defines a controller (`carts`) and an action (`index`) within the `CartsController` class that serve as our shop's home page. When a user types the URL `http://localhost/cake/__chapters__/ecommerce/` in a web browser address bar or clicks the Home link on the web site, the request is handled by the `index` action of the `CartsController` object.

We'll create the `CartsController` in the following section.

The Carts Controller

Listing 3-27 shows the carts controller for our shop application.

Listing 3-27. *The CartsController Class (app/controllers/carts_controller.php)*

```
<?php
class CartsController extends AppController {
    var $name = 'Carts';
    function index() {
    }
}
?>
```

The CartsController class begins by extending the AppController class. It then sets its $name property to Carts to ensure backward compatibility with PHP 4.

By default, the index action is mapped to the app/views/carts/index.ctp view file. Remember that the layout of our shop contains a middle section, or center column. Our index.ctp view handles what is rendered in this area of the web site layout. The content of this view file is shown is shown in Listing 3-28.

Listing 3-28. *The View for the Center Column Content (app/views/carts/index.ctp)*

```
<?php
    e( 'Welcome to the land of music!
    <br />
    Please select your own music product!
    <br />
    Thanks for dropping by!<br />');
    if ($pdId) {
        e( $this->element('product_details'));
    } else if ($catId) {
        e( $this->element('products'));
    } else {
        e( $this->element('categories'));
    }
?>
```

In Listing 3-28, we first render our welcome message using the e() function, which is a Cake convenience method for PHP's echo statement. You can change this message to suit your needs. Next, we use the current values of the product ID ($pdId) and category ID ($catId), as defined in the beforeFilter method of the AppController class (Listing 3-25), to logically control the view that is finally displayed to web surfers.

When a user visits our shop for the first time, the values of the product ID and category ID will be set to zero, as defined in the beforeFilter method of the AppController, since the user has not yet selected a category or a product from the home page. For this scenario, the view logic will render the content stored at app/views/elements/categories.ctp. We will also have this scenario when a user clicks the Home link of the web site. The content of this view is shown in Listing 3-29.

Listing 3-29. *The Category List View for the Center Column (app/views/elements/categories.ctp)*

```
<?php
$categories = $this->requestAction("/categories/getAll");
foreach($categories as $category): ?>
<div class="product_container">
<?
    if ( $category['Category']['image'] ) {
        $cat_thumbnail = '/img/products/' . $category['Category']['image'];
    } else {
        $cat_thumbnail = 'no-image-small.png';
    }
```

```
        e( $html->link(
            $html->image($cat_thumbnail, array( 'border' => '0' )),
            array('action' => '/index/cat_id:' . $category['Category']['id']),
            array('escape' => false))); ?>
    <br>
    <? e( $html->link( $category['Category']['name'], ➥
"/carts/index/cat_id:".$category['Category']['id'] )); ?>
</div>
<? endforeach;?>
```

In Listing 3-29, we also use Cake's `requestAction` method to trigger the `getAll` method of the `CategoriesController`. The result of this method is a list of all the category information, which is stored in the `$categories` array variable. This variable is then looped over to display the list of categories, as shown in Figure 3-5.

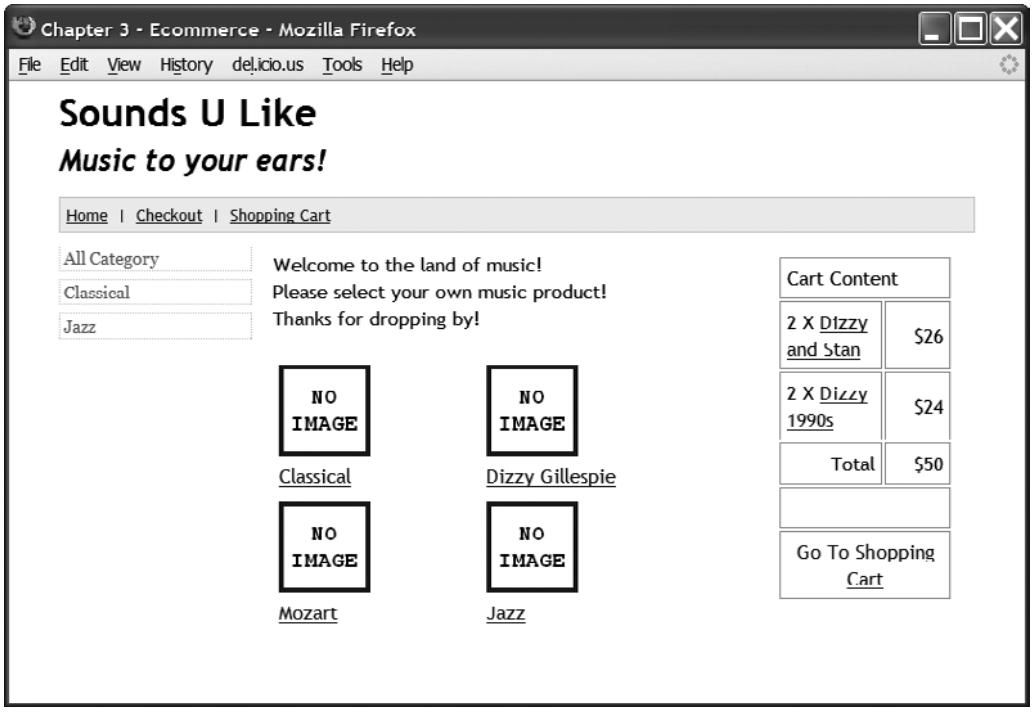

Figure 3-5. *The view showing the list of categories*

Now, we're finished with the left column content—the category navigation—and the center column content. Let's move to the right column, which displays the current shopping basket (cart) content (see Figure 3-5). If there is no information in the `carts` table, we'll display the message "Shopping Basket is empty" in this area of the page.

The view code to generate the basket content is stored at `app/views/elements/basket.ctp`. This code is shown in Listing 3-30.

Listing 3-30. *The Basket View for the Right Column (app/views/elements/basket.ctp)*

```
<table id="minicart">
<?php
$cartContents = $this->requestAction("/carts/getMiniCart/s:$sid");
if ( !empty($cartContents) && is_array($cartContents) ) {
$Total = 0;
?>
<tr>
<td colspan="2">Cart Content</td>
</tr>
<?php
foreach($cartContents as $cartContent) {
    $Total += $cartContent['pd']['price'] * $cartContent['ct']['qty'];
?>
<tr>
<td>
<?=$cartContent['ct']['qty'];?> X
<?=$html->link($cartContent['pd']['name'],
'/products/view/pd_id:'.$cartContent['ct']['product_id'].➥
'cat_id:'.$cartContent['pd']['category_id']);?></td>
<td width="30%" align="right"><?=Configure::read('Shop.currency');?>
<?=$cartContent['pd']['price'] * $cartContent['ct']['qty'];?></td>
</tr>
<?php
    } ?>
<tr>
<td align="right">Total</td>
<td width="30%" align="right">
<?=Configure::read('Shop.currency');?>
<?=$Total;?>
</td>
</tr>
<tr>
<td colspan="2"> </td></tr>
<tr>
<td colspan="2" align="center"><?=$html-> ➥
link(' Go To Shopping Cart','/carts/view');?></td>
</tr>
<?php
} else {
echo '<tr><td width="150">Shopping Basket is empty</td></tr>';
}
?>
</table>
```

Before we delve into the code in Listing 3-30, let's add the product titled Mozart for Lovers to our basket by clicking the Add to Shopping Basket link on the product description page. Adding a product to the basket triggers the `CartsController`'s `add` method, shown in Listing 3-31.

Listing 3-31. *The CartsController's add() Method*

```
function add() {
    $data = $this->Product->findById( $this->pdId );
    if( empty ( $data ) ) {
        $this->redirect('/');
    } else {
        if ($data['Product']['qty'] <= 0) {
            $this->Session->setFlash('The product you requested is ➥
no longer in stock');
            $this->redirect('/');
        }
    }
    $sessionData = $this->Cart->getCart($this->pdId, $this->sid);
    if ( empty($sessionData)) {
        $this->Cart->addCart($this->pdId, $this->sid);
    } else {
        $this->Cart->updateCart($this->pdId, $this->sid);
    }
    $this->Cart->cleanUp();
    $this->redirect(array('controller'=>'Products',
                    'action'=>"view/cat_id:$this->catId/pd_id:$this->pdId"));
}
```

In Listing 3-31, the first statement uses Cake's `findById` model method, which accepts a product ID (`$this->pdId`) to retrieve a product's information from the `products` table. This product information is stored in the `$data` array variable. If `$data` is empty, the application redirects to the home page.

Next, the script checks if the product quantity is less than or equal to zero. If so, the application uses the Cake `Session` object's `setFlash` method to hold the message `'The product you requested is no longer in stock'` in the `Session` object, and then redirects to the home page.

Then the application calls the `getCart` method of the `Carts` model object in order to check the existence of this product in the shopping basket. The result of this method is stored in the `$sessionData` array variable. If the product does not exist in the shopping basket against the current session ID, the `addCart()` method of the `Carts` model object is called to insert a new product into the `carts` table. If the product does exist in the shopping basket, the `updateCart` method is called to increment the quantity of the product in the `carts` table by one.

Next, the script seizes the opportunity to do some housecleaning by calling the `cleanUp` method to remove records that are few days old from the `carts` table. Finally, the application redirects the user to the current product's description page.

Now that we've added a product to the `carts` table, let's jump back into the basket view code in Listing 3-30. First, we use the `requestAction` method to get all the content of the cart from the `carts` table. This Cake method accepts the string that triggers the `CartsController`'s `getMiniCart` method, which accepts the session ID (via `s:$sid`) as an argument and assigns the result to the `$cartContents` variable. Next, if the `$cartContents` variable is an array and is not empty, the `$Total` variable is assigned a value of zero. Finally, using the `foreach` loop over `$cartContents`, the current content of the basket now includes the newly added product Mozart for Lovers, costing $15, as shown in Figure 3-6.

Figure 3-6. *The view showing Mozart for Lovers in the shopping basket in the right column*

Checking our user journey, our next action is to proceed to the shopping basket, to check that we are happy with our current product selections. If not, we will amend the selections or completely remove unwanted products from our shopping basket. This interface allows for cart-manipulation processes, such as updating the product quantity, deleting the product, and so on. Let's suppose the user has clicked the Go to Shopping Cart link in Figure 3-6, and modified the cart content by removing the Mozart for Lovers and Dizzy 1990s items from her shopping basket. The new content of the shopping basket is shown in Figure 3-7.

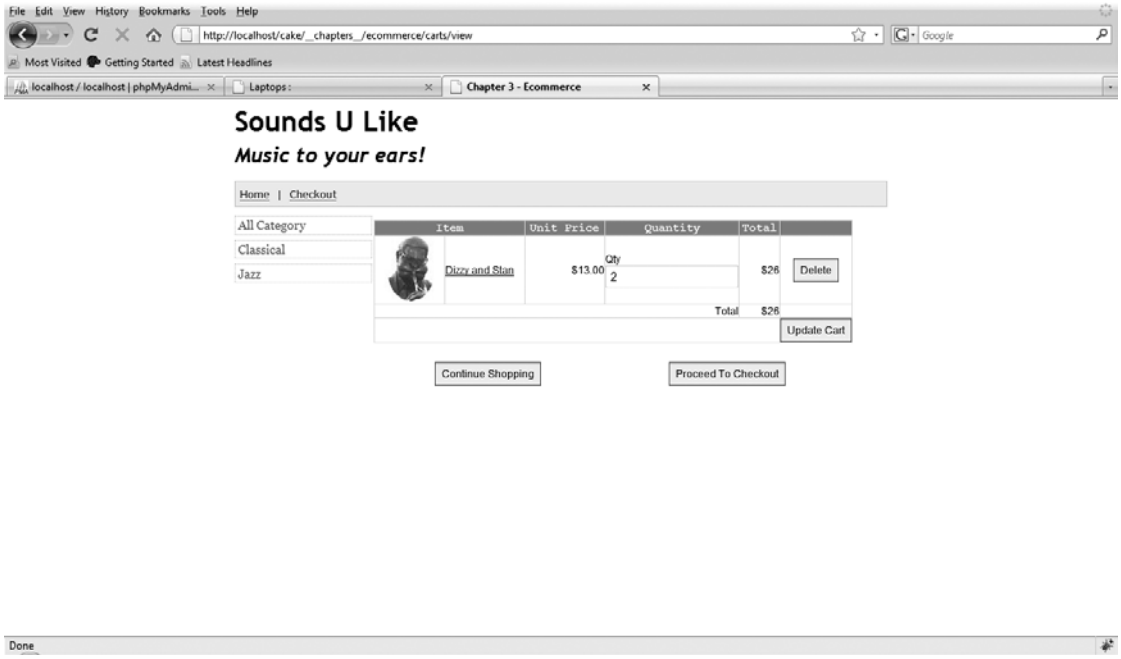

Figure 3-7. *The view showing a sample cart content*

The Order Model

Next, we're going to deal with the orders. We'll start by creating the `Order` model class, shown in Listing 3-32. Note that this class does not interact with any database table.

Listing 3-32. *The Order Model Class (app/models/order.php)*

```
<?php
class Order extends AppModel
{
    var $name = 'Order';
    var $useTable = false;
    var $validate = array(
            'name' => array( 'rule' => array( 'between', 2, 255 ),
            'required' => true,
            'message' => 'Please enter a name.' ),
            'address' => array( 'rule' => array( 'between', 4, 255 ),
            'required' => true,
            'message' => 'Please enter an address.' ),
            'comment' => array( 'rule' => array( 'between', 5, 255 ),
            'required' => true,
            'message' => 'Please enter a comment.' ));
}
?>
```

In Listing 3-32, we define the `$validate` array to ensure input data integrity for the `name`, `address`, and `comment` form elements.

Next, we're going to click the Proceed to Checkout button. Our action will trigger the `checkout` method of the `OrdersController` object, which is defined in Listing 3-33.

Listing 3-33. *The OrdersController Class (app/controllers/orders_contoller.php)*

```
<?php
class OrdersController extends AppController {
    var $name = 'Orders';
    var $uses = array('Order');

    function checkout() {
        $data = $this->passedArgs['cts'];
        $this->set('data', $data);
        $this->layout = 'checkbase';
    }

    function confirm() {
        $this->layout = 'checkbase';
        if (!empty($this->data)) {
            $orders = $this->data;
            $carts = $this->Cart->find('all', array(
            'conditions'=>array('Cart.id'=>$this->data['Order']['cts']),
            'recursive' => 1 ));
            $this->set(compact('carts', 'orders'));
    }
    }
}
?>
```

As usual, the `OrdersController` class references the `Orders` model object using the `$uses = array("Order");` statement. Next, we define the `checkout` method, which first extracts the current cart's content IDs from `$this->passedArgs['cts']` array, and then stores the result in the `$data` array. The `$data` array is passed to the `app/views/orders/checkout.ctp` view file, shown in Listing 3-34, for display. The last statement of this method changes the default layout to `checkbase`.

Listing 3-34. *The Checkout View (app/views/orders/checkout.ctp)*

```
<fieldset>
    <legend>
        <?php echo $this->pageTitle = __('<strong>Checkout</strong>', true); ?>
    </legend>
    <?php
        echo $form->create('Order', array('url'=>'/orders/confirm/') );
        echo $form->error( 'Order.name' );
```

```
        echo $form->input(  'Order.name', array( 'id' => 'ordername',
                            'label' => 'Name:', 'size' => '30',
                            'maxlength' => '255', 'error' => false ) );
        echo $form->error( 'Order.address' );
        echo $form->input(  'Order.address', array( 'id' => 'orderaddress',
                            'type'=>'textarea', 'label' => 'Address:',
                            'rows' => '3', 'error' => false ));
        echo $form->error( 'Order.comment' );
        echo $form->input(  'Order.comment', array( 'id' => 'ordercomment',
                            'type'=>'textarea', 'label' => 'Comment:',
                            'rows' => '5', 'error' => false ) );
        echo $form->error( 'Order.payment' );
        echo $form->input(  'Order.payment', array('type'=>'radio',
                            'options'=>array(1=>'Google', 2=>'Paypal')));
        echo $form->input( 'cts', array('type'=>'hidden', 'value'=>$data) );
        echo $form->end( array( 'label' => ' Confirm Order ' ) );
    ?>
</fieldset>
<br />
<?php echo $this->element('checkout'); ?>
```

In Listing 3-34, the code is divided into two main parts. The first part uses Cake's `$form` object to render the checkout form. The second part renders the current content of the shopping basket, similar to the basket view code (Listing 3-30). Listing 3-34 produces the checkout form shown in Figure 3-8.

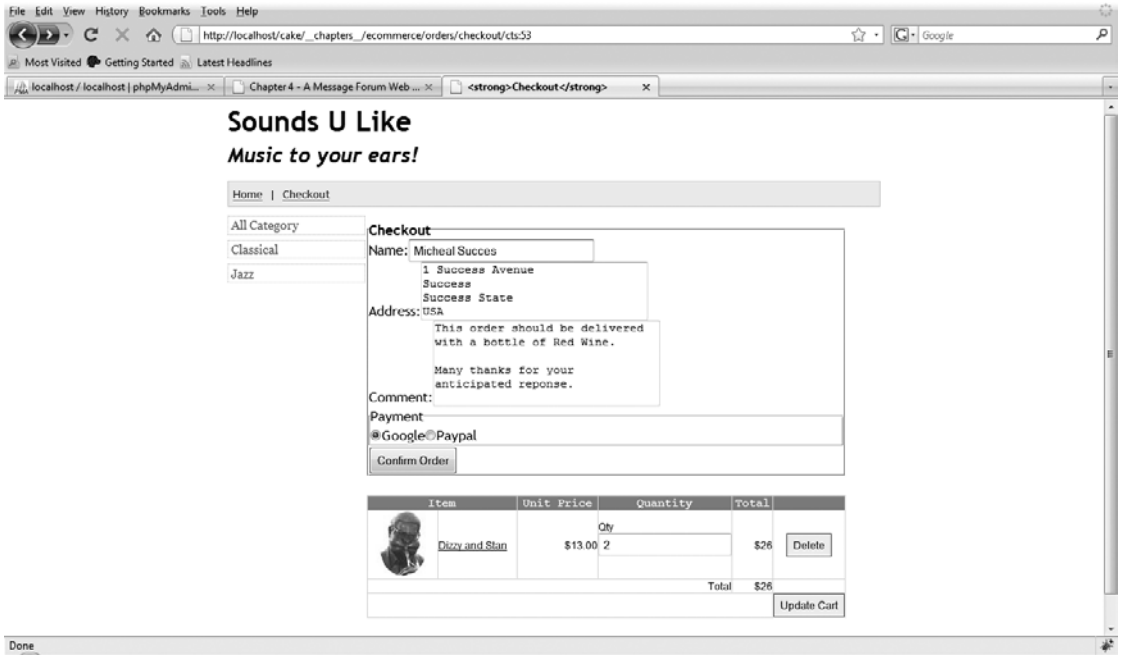

Figure 3-8. *The checkout page*

When the user clicks the Confirm Order button on the checkout page, the `confirm` method of the `OrdersController` is triggered. If the submitted form data (stored in `$this->data`) is not empty, then the form data is stored in the `$orders` array. Next, we retrieve the current cart content based on the cart ID (extracted from `$this->data['Order']['cts']`), and the result is stored in the `$carts` array variable.

Note It is important to include the statement `var $belongsTo = array('Product');` in the `Cart` model class, because we'll need to submit some vital product information (such as product name, price, and description) to our chosen payment gateway. This statement will ensure that the application includes related product information whenever `carts` table records are retrieved.

Finally, using Cake's `set` and `compact` functions, the two array variables are passed on to the `app/views/orders/confirm.ctp` file. The structure of the array variables is shown in Listing 3-35.

Listing 3-35. *The Structure of $order and $cart Array Variables*

```
Array
(
    [Order] => Array
        (
            [name] => Micheal Succes
            [address] => 1 Success Avenue
                         Success
                         Success State
                         USA
            [comment] => This order should be delivered with a bottle of Red Wine.
                         Many thanks for your anticipated reponse.
            [payment] => 1
            [cts] => 53
        )
)

Array
(
    [0] => Array
        (
            [Cart] => Array
                (
                    [id] => 53
                    [product_id] => 22
                    [qty] => 2
                    [ct_session_id] => 32cabb9d6fb31404a5b736830eac6aa3
                    [created] => 2008-09-25 23:30:18
                )
```

```
            [Product] => Array
                (
                    [id] => 22
                    [category_id] => 13
                    [name] => Dizzy and Stan
                    [description] => Live with Dizzy Gillespie and Stan Getz.
                    [price] => 13.00
                    [qty] => 10
                    [image] => 1.jpg
                    [thumbnail] => 1.jpg
                    [created] => 0000-00-00 00:00:00
                    [modified] => 0000-00-00 00:00:00
                )
        )
)
```

The array structures in Listing 3-35 are passed on to the `confirm.ctp` view file, as shown in Listing 3-36.

Listing 3-36. *The OrdersController's confirm() Method View*

```
<?php
    if($order['Order']['payment'] == 1) {
        echo $this->element('google_checkout');
    } else {
        echo $this->element('paypal_checkout');
    }
?>
```

If the order payment method selected equals `1`, then the Google Checkout payment form is displayed. If not, the PayPal form is presented.

The Google Checkout Button

Google needs no introduction when it comes to the world of Internet. However, you should be aware that to use Google services, you need to have a Google account, which you probably already do if you use Google Mail, Google Docs, Google Adwords, or any of the numerous other Google offerings.

Shoppers are often frustrated by needing to fill out lengthy online forms before making payments. It is easy to encourage users to check out via the Google Checkout button, which is secure and convenient. When this button is selected, the `$carts` and `$order` array variables are passed to the `app/views/elements/google_checkout.ctp` file, shown in Listing 3-37.

Listing 3-37. *Google Checkout View (app/views/elements/google_checkout.ctp)*

```
<?php
    echo $form->create('Order',
        array(  'url'=>'https://checkout.google.com/api/checkout/v2/➥
checkoutForm/Merchant/xxxxxxxxxxxxxxx',
                'accept-charset'=>'utf-8' ) );

    $i = 0;

    foreach($carts as $cart) {

        $x = $i+1;

        echo $form->hidden( "item_name_$x",
                        array( 'value'=>$cart['Product']['name']));
        echo $form->hidden( "item_description_$x",
                        array( 'value'=>$cart['Product']['description']));
        echo $form->hidden( "item_quantity_$x",
                        array( 'value'=>$cart['Cart']['qty']));
        echo $form->hidden( "item_price_$x",
                        array( 'value'=>$cart['Product']['price']));
        echo $form->hidden( "item_currency_$x",
                        array( 'value'=>Configure::read('Item.currency')));
        echo $form->hidden( "ship_method_name_$x",
                        array( 'value'=>Configure::read('Order.shipmethod')));
        echo $form->hidden( "ship_method_price_$x",
                        array( 'value'=>Configure::read('Order.shipprice')));
        echo $form->hidden( "tax_rate",
                        array( 'value'=>Configure::read('Order.taxrate')));
        echo $form->hidden( "tax_us_state",
                        array( 'value'=>Configure::read('Order.taxstate')));
    }

    echo $form->hidden('_charset_');

    echo $html->image(  'http://checkout.google.com/buttons/checkout.gif? ➥
merchant_id=xxxxxxxxxxxxxxx&w=180&h=46&style=white&variant=text&loc=en_US',
                        array(  'name'=>'Google Checkout',
                                'alt'=>'Fast checkout through Google',
                                'height'=>'46',
                                'width'=>'180') );

    echo $form->end( array( 'label' => ' Confirm Order ' ) );
?>
```

The view file shown in Listing 3-37 generates a page similar to the one shown in Figure 3-9.

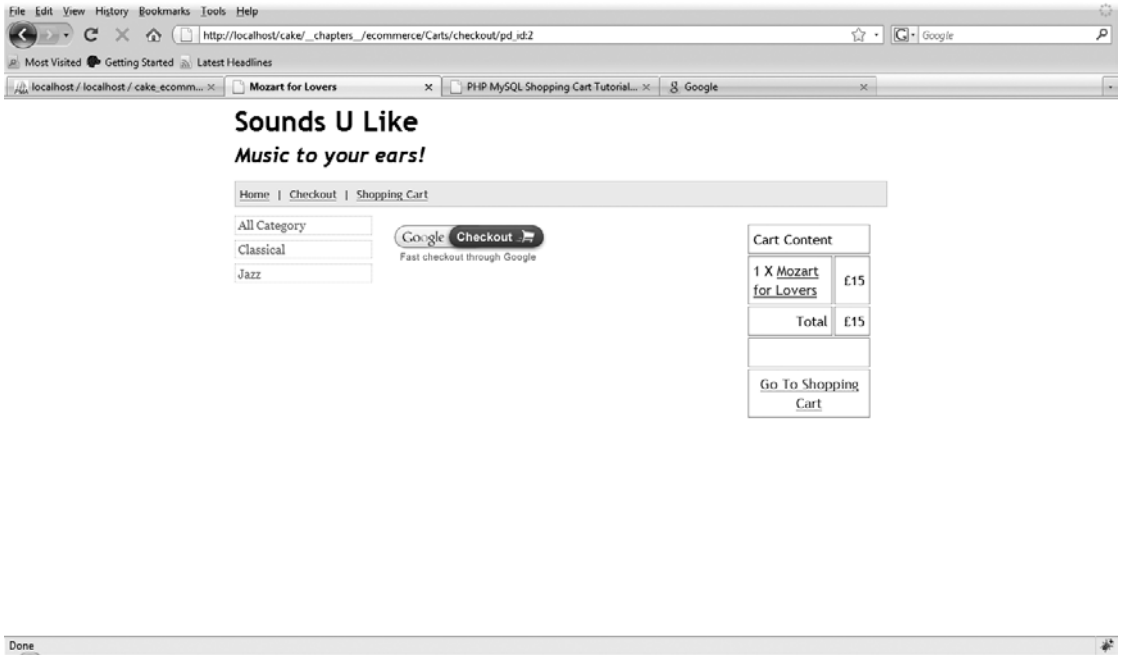

Figure 3-9. *The Google checkout page*

Clicking the Google Checkout button will redirect the user to the Google sandbox environment, as shown in Figure 3-10.

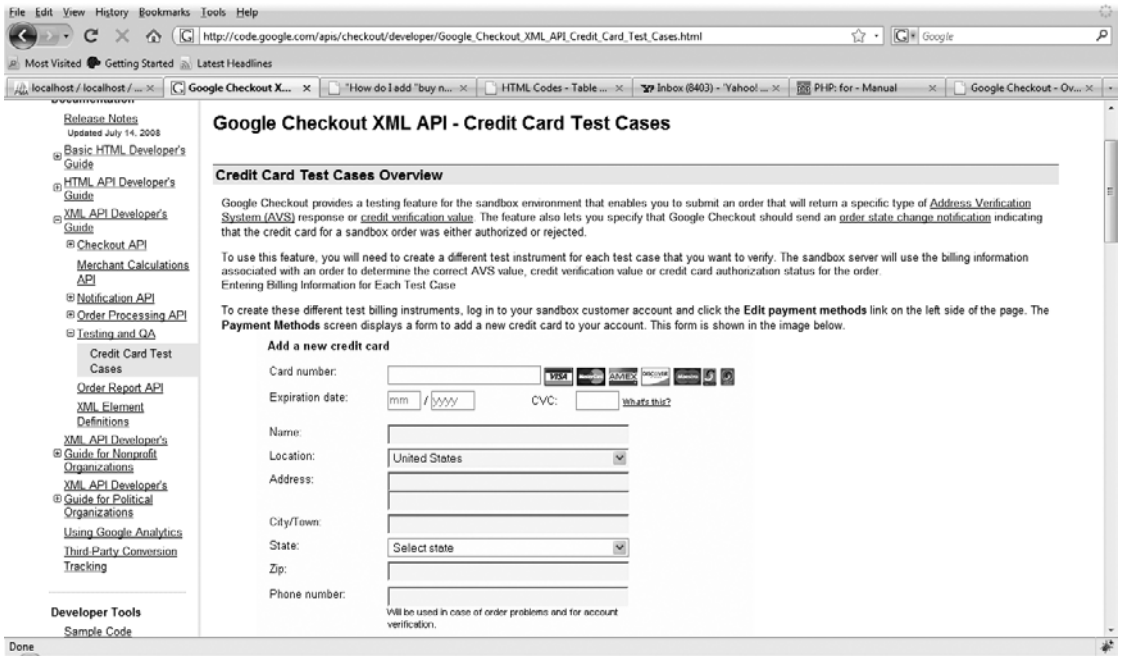

Figure 3-10. *The Google test sandbox*

For more information about processing online sales using Google's payment system, Google Checkout buttons, and the Google sandbox, visit the following sites:

- https://checkout.google.com/sell/
- http://code.google.com/apis/checkout/developer/Google_Checkout_Basic_HTML_Google_Checkout_Buttons.html
- http://code.google.com/apis/checkout/developer/Google_Checkout_XML_API_Credit_Card_Test_Cases.html.

The PayPal Submit Button

PayPal can be regarded as the father of online payment gateways. It's ubiquitous when it comes to online payment choices. Here, we are going to create the payment form, in the app/views/elements/paypal_checkout.ctp file, as shown in Listing 3-38. You can use the Buy-it-Now button form generator on PayPal to create a similar payment form.

Listing 3-38. *The PayPal View (app/views/elements/paypal_checkout.ctp)*

```
<?php
    echo $form->create('Order',
                       array('url'=>'https://www.paypal.com/cgi-bin/webscr',
                             'id'=>'payPalForm') );
    echo $form->hidden("item_number", array( 'value'=>"The Music Club"));
    echo $form->hidden("cmd", array( 'value'=>"_xclick"));
    echo $form->hidden("no_note", array( 'value'=>"1"));
    echo $form->hidden("business", array( 'value'=>"sales@practicalcakephp.com"));
    echo $form->hidden("currency_code", array( 'value'=>"USD"));
    echo $form->hidden("return", array( 'value'=>"http://practicalcakephp.com"));
    $i = 0;
    foreach($carts AS $cart) {
    $x = $i+1;
    echo $form->hidden("item_number", array( 'value'=>"The Music Club"));
    echo $form->hidden("item_description_$x",
                       array( 'value'=>$cart['Product']['description']));
    echo $form->hidden("item_quantity_$x", array( 'value'=>$cart['Cart']['qty']));
    echo $form->hidden("item_price_$x", array( 'value'=>$cart['Product']['price']));
    }
    echo $form->end( array( 'label' => ' Submit ' ) );
?>
```

The PayPal view is passed the $carts and the $order array variables.

For further information about the numerous services provided by PayPal, visit http://www.paypal.com.

Summary

In this chapter, we went through the process of building an online shop. For this demonstration, we've kept its features to a minimum. However, users can navigate the product category menu, add products to a shopping basket, review the basket contents, and proceed to the checkout form.

We started this chapter by looking at a typical shop layout, the shop user journey, and the creation of the database and tables needed for the shop application. We then proceeded to build all the model classes that interact with the database tables, such as those for categories, products, carts, and so on.

To handle user requests, we built the `AppController` class with properties and methods that provide functionality common to other controller classes. This helps us to avoid "reinventing the wheel" and to extend other controller classes, such as the category controller of the application. We then built the other controller classes to handle the application requests.

Finally, we created checkout forms to handle payment transactions. We used a Google sandbox as our payment test environment and also created a PayPal form as an alternative payment option.

Building a comprehensive e-commerce web site obviously involves more than what we covered in this chapter. For example, you would want to add an administration area to the web site to facilitate management.

CHAPTER 4

A Message Forum Web Service

In this chapter, we'll build a Cake-based forum. While there are a lot of popular open source forums such as phpBB, we like the idea of rolling our own. It's a fantastic learning process, and we get to deal with a lot of new subjects that don't come up in our day jobs.

So that our forum will stand out from the crowd, we need a unique selling point. That will be a web service API for our forum. Web services are quite a common feature in many modern web applications, but not that common in many of the forums.

Our Take on Web Services

The term *web services* can have many meanings. For example, simply entering the URL `http://www.cakephp.org/` into your browser can be called a web service request, since you're using HTTP `GET`. The World Wide Web Consortium (W3C) has an official definition for web services (`http://www.w3.org/2002/ws/Activity`):

> *Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks.*

This meaning is quite general, indicating that web services are a way for computers to talk to each other, which may or may not include the Web.

Before we get to creating our forum, let's clear up what we mean by web services.

Web Service Elements

There are many elements relating to web services. We start with a short explanation of each element to provide a foundation for our particular angle on web services.

- *API*: This acronym stands for application programming interface. In the world of computers, much like the definition of web services, this is quite a general term. For us, it means a set of published functions or methods that can be directly accessed via URLs on the server, such as `http://www.example.com/getWidgets.json` or `http://www.example.com/getWidgets/json`.
- *SOAP*: This stands for Simple Object Access Protocol, and is one of the main protocols of a web service. It piggybacks onto the HTTP protocol, considered too complex and verbose by many developers. Taking the previous example, our SOAP request would look something like this:

```
POST / HTTP/1.1
Host: www.example.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:getWidgets xmlns:m="Some-URI">
           <type>square</type>
       </m:getWidgets>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- *HTTP*: In a sense, HTTP (Hypertext Transfer Protocol) and SMTP (Simple Mail Transfer Protocol) are the two main protocols people use on the Internet. They surf the Web using the HTTP protocol and read e-mail messages using the SMTP protocol. Many developers have adopted HTTP as their protocol for developing their web services; specifically, just the `GET` and `POST` methods within that protocol. This is how we will be developing our API in this chapter's application.
- *XML-RPC*: The Extensible Markup Language Remote Procedure Call was created by David Winer, one of the pioneers in modern web services and blogging. It's similar to SOAP, but simpler. This protocol is not frequently used by developers. The major web applications that support XML-RPC include Flickr and Amazon S3.
- *REST*: This stands for representational state transfer. The term was coined by Roy Fielding, one of the main authors of HTTP. It is not a protocol, but a set of statements about how distributed media should be organized, with the Web being a key example.

Figure 4-1 illustrates how many web developers see web services. On the left side, we have the clients. They can be applications on other servers; desktop applications, which include browsers; and other devices, like mobile cell phones. These clients will most often use HTTP (`GET`, `POST`, and so on) to send and request data to and from the server, shown on the right. The way the clients talk to the server is the protocol: XML-RPC, SOAP, RSS, and so on. These protocols are quite specific, since there are standards attached. For example, in RSS, you must end all requests with the `.rss` extension. The format you will receive will be in a specific format, as defined by the official standards body.

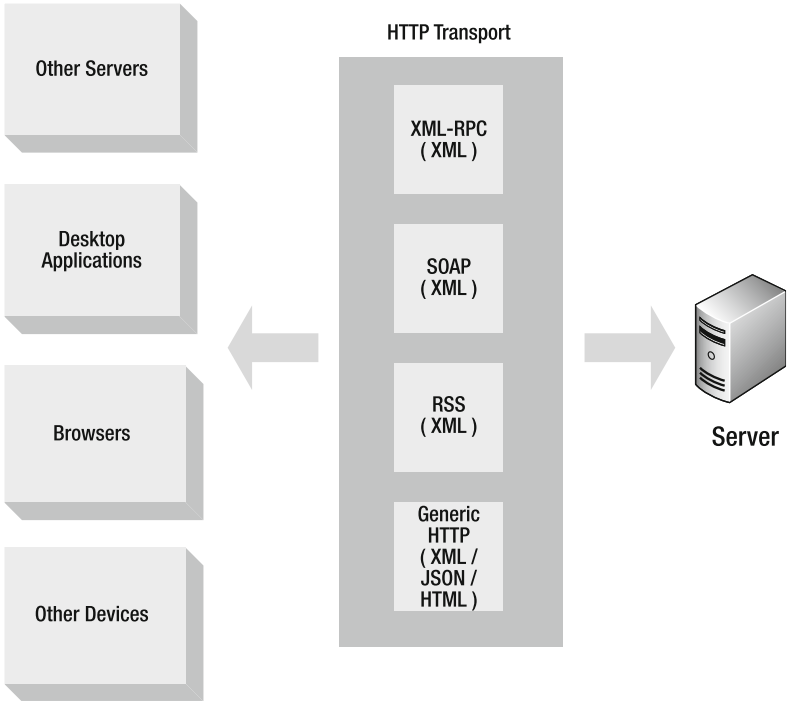

Figure 4-1. *Elements of web services*

However, you can develop your own proprietary protocol using the features of HTTP. Many web service providers have gone this route—Google Maps, Flickr, and Twitter, to name a few.

Mashing different services together is never an easy task. For example, an image will have a different context depending on the application. The use of REST complicates the picture. In Figure 4-1, it sits within generic HTTP, as we regard it as a specific way of using that protocol.

REST and HTTP

Many people have advocated the use of the REST principle with HTTP for web services. In particular, the use of the HTTP methods should conform to the W3C standard. For example, `GET` methods should not alter any data. Something like `http://www.example.com/editUser/?id=1&name=new_name` should not be allowed. In REST, this would look like `http://www.example.com/User/1`. The new name would be supplied as a key/value pair—`name=new_name`—in the request body, and instead of using `GET`, we would be using `PUT`.

In a sense, we see HTTP as being a diluted form of REST. It can get quite confusing, as some meanings can overlap. We see the confusion surrounding REST much as we see the complexity of SOAP. Many developers find REST difficult to use.

To use REST in its full meaning, you must conform to the correct use of the HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE`. Most browsers support only `GET` and `POST`. You can use the other methods via `XMLHttpRequest` in Ajax scripting, but that just won't be enough to conform to the principles of REST. To end our discussion of REST, the web services that we're going to write in this chapter will not conform to the principles of REST.

Note The four most important HTTP requests are GET, POST, PUT, and DELETE. GET requests a resource, such as an HTML file. (Requesting the resource must not cause a side effect, such as deleting a record). POST sends data to the resource to be processed. It creates a new resource, updates the resource, or both. We'll consider the PUT operation to be the same as the POST. Elliotte Rusty Harold gives a good explanation at http://www.elharo.com/blog/software-development/web-development/2005/12/08/post-vs-put/. Finally, DELETE just deletes a resource, such as a user account.

We've always liked the principle of KISS, which means we prefer to Keep It Short and Simple. Logical standards like SOAP sometimes are just too logical. As such, like many web APIs, we're just opting for plain old HTTP GET and POST as the two modes that developers can use to access the innards of our forum. Referring to our diagram in Figure 4-1, we're using the generic HTTP method.

Result Return Formats

When we make a web service request, the result can be returned in many formats depending on the client request, including HTML, JSON, XML, and RSS.

You can even specify the result to be returned as a comma-separated list or as a JavaScript-ready document output. For our application, we'll use JSON. Developers can easily use our web services via Ajax or on the server using curl or wget.

Application Requirements

One of the main focal points of our forum application is the API. Therefore, we'll start by thinking about the methods that we will be exposing to the public. The API requests will come from other applications, not individuals using the application via an interface, so we must consider that in our planning.

In our forum application, we want to include some common features that are found in all forums:

- Post messages
- Reply to messages
- View messages
- Search messages

Since our own interface is basically also a client, we need to look at our own front end as if it were a third-party client application that is calling the processing scripts from a distant server. We know there are several ways to make a URL request. The following are the ones we are interested in:

- Requesting a URL via curl or wget; they may also want it returned in a particular format
- Requesting a URL via a browser using HTTP GET or POST
- Requesting a URL via Ajax scripting using HTTP GET or POST; that is, the XMLHttpRequest (XHR) within the browser makes the HTTP call

Some of our actions may support only GET and/or POST. This is similar to many of the other web APIs.

We must have a standard way to respond to requests. For example, in HTTP, the response code 200 is OK, and the response code 404 is resource or page not found. Furthermore, some requests return data, while others carry out particular actions like saving data. We must be able to return data in a consistent manner given different requests.

Threads and Posts

For our forum, we need to decide how discussions should be organized. We simply define a discussion (commonly called a *thread*) as the messages and the subsequent messages (replies) to those messages. Let's look at some of the ways in which messages could be organized.

Organized by Date

Using this method, each message is organized by date order, regardless of to which message it is replying. Users can then see the most up-to-date messages as they come in.

Organizing by date is problematic because we won't know which message a message is replying to. We can overcome this by including the message, but do we include the whole message or just part of it? Perhaps we could allow the users to select the parts they are replying to?

Listing 4-1 shows a code snippet that allows a user to quote part of a message so other users can refer to it.

Listing 4-1. *JavaScript Code to Get Selected Text*

```
<input type="button" onmousedown="displaySelected();">

<script>

function displaySelected(){

   var selectedText = '';

   if ( window.getSelection ) {
        selectedText = window.getSelection();
   }
   else if (document.selection){
        selectedText = document.selection.createRange().text;
   }

   return selectedText;

}

</script>
```

You might want to use the code in Listing 4-1 if you decide to extend the forum application. It simply retrieves the text a user has selected with the mouse. On line 9, we check whether the user has selected any text on Firefox/Gecko-based browsers. If so, we place the text in to the `selectedText` variable. Line 12 checks for text selection in Internet Explorer/Trident-based browsers.

Organized by First Post

Using the by-first-post organization, we show only new messages, essentially the first messages, as they come in. We won't show the replies. If users want to read the replies, they need to drill down; that is, make another request to a different Cake action.

If a user replies to an old message, it won't be displayed. It will remain relative to that old message. Users will know of that reply only when they drill down to it.

An obvious problem is that active discussions will be followed only by those who participated in it at the early stages, when the discussions were at the top of the page. When new messages arrive, the old messages get pushed down. This is not desirable, since we want many people contributing to a discussion as long as possible. After all, if people are just asking questions on the forum, very few people would bother to read the forum.

Organized by Replies

You can also organize the threads by the number of replies, so the thread that has the greatest number of replies is shown first. If no one has contributed to the thread for a while, it will still stay at the top.

The problem with this organization is that current and active threads are not given priority. So we end up putting too much emphasis on a topic that may no longer be of interest to most people.

Organized by Last Post

Another organization possibility is a slightly different take on the by-first-post organization. Using this method, we order a discussion by its last post, but we show the first post.

This approach has a number of advantages. If a new message comes in, it floats to the top. However, if a reply to an old message is posted, the first post of that discussion floats to the top. In this way, we encourage people to participate in ongoing discussions. If the discussion has been exhausted, it simply floats back down the page.

Organizing the posts by the last post seems to be the most sensible method. In fact, most forums organize discussions this way, and so will our application.

Web Service Requests

Our application will support the following five API requests:

- Fetch one message
- Fetch many messages
- Fetch threads
- Process a message
- Search for messages

These requests will be the ones that we will be publishing in our API documentation. They will all support Ajax from a browser, ordinary HTTP requests from a browser, and `curl` or `wget` calls from a server.

Layout

The API is an important component to the application. However, we're not just publishing an API. The application will also be a working forum where users can post messages. In that sense, we're building a client as well as the API services within the same application.

Our application front end will have all the essential features that any forum user will expect. A user can post messages, reply to messages, view topics, and search for messages. The layout of the interface will be similar to that of the other projects in this book. We start with a header, followed by a navigation bar, and then the main content area. When a user goes straight to the site, they will be presented with a view of the current threads, as shown in Figure 4-2.

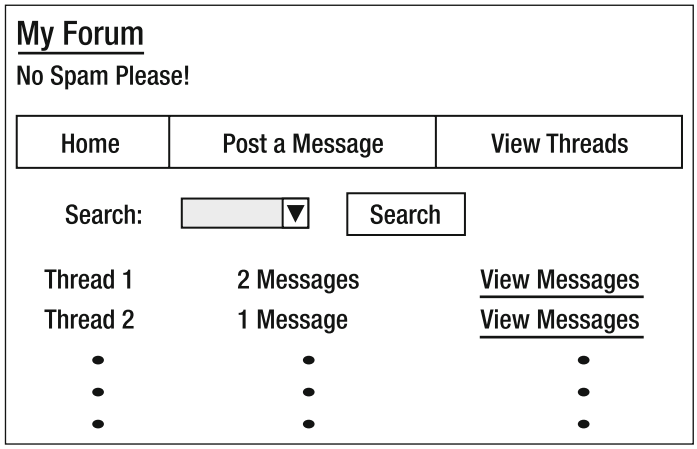

Figure 4-2. *A rough sketch of the forum application layout*

Application Structure

In the previous section, we talked about how we're going to organize our messages according to threads. This relationship is a one-to-many association, where one thread ties together many messages.

The two main database tables in our application are `messages` and `threads`. The fields in these tables are shown in Tables 4-1 and 4-2.

Table 4-1. *The Fields in the messages Table*

Field Name	Data Type	Description
id	char(36)	Primary key (note it's a UUID and not an auto-increment field; see the action class MfMessageProcess for an explanation)
name	varchar(255)	Name of the user posting the message
email	varchar(255)	E-mail address of the user
message	mediumtext	The message body itself
reply_to	char(36)	If the message is a reply to another message, this is the reply message ID
Subject	varchar(255)	The subject of the message (replies will have a RE: prefix)
t_created_at	datetime	Automagic field (but we fill this in ourselves; see the action class MfMessageProcess for an explanation)
thread_id	char(36)	The thread this message belongs to

Table 4-2. *The Fields in the threads Table*

Field Name	Data Type	Description
id	char(36)	Primary key (again, UUID and not an auto-increment field)
first_message_id	char(36)	Used so we can easily pick out the first message when it needs to be displayed in the threads listing
last_message_date	datetime	Used so we can order the threads correctly according to the last message date
message_num	int(11)	The number of messages within this thread

■ **Note** As we haven't used database transactions in our application, the number in the message_num field of the threads table may not always be completely accurate, but that's not critical at this stage, as it's used only for display purposes.

You'll notice that the threads table includes a few metadata fields to contain data that describes the thread. This helps us to cut down on the number of SQL calls needed to list our threads. Each time a new message is added to a thread, the code updates the last_message_date field with the date of that message. This way, we don't need to look in the messages table for the last message that was posted within a thread.

JSON Web Services

We have specified that we'll be using the JSON format for all our API returns. To get this part of our application working, we need to do some setup.

The Router class must be aware of the JSON extension. We take care of this by adding the following line in the /app/config/routes.php file:

```
Router::parseExtensions( 'json' );
```

Following this, we need to include the RequestHandler component by adding it into our $components variable in our /app/app_controller.php file, so that it's available to all controllers. We also need the Session component. Our $components variable will look like this:

```
var $components = array( 'Session', 'RequestHandler' );
```

Now the RequestHandler will automatically map JSON requests to the correct layout and view. When a JSON request comes in, it will pick the layout file within the folder /app/views/layout/json instead of /app/views/layout. Our JSON layout file is shown in Listing 4-2.

Listing 4-2. *JSON Layout File (/app/views/layout/base.ctp)*

```
<?php

    // This is used when the call extension is .json
    header( "Pragma: no-cache" );
    header( 'Content-Type: text/x-json' );
    header( "X-JSON: ".$content_for_layout );

    $controller = $this->name;
    $action = $this->action;
    $datetime = date( "Y_M_j__G_i_s_T" );
    $file_name = $controller.'_'.$action.'_'.$datetime;

    header( 'Content-Disposition: attachment; filename="'.$file_name.'.json"' );

    echo $content_for_layout;
?>
```

Any script that requests a JSON return can use the HTTP headers Content-Type and X-JSON to identify whether or not it is a JSON return. However, the $content_for_layout isn't guaranteed to be JSON. We must manually put that into a JSON format.

Once the layout has been picked out, the RequestHandler will look in a json folder within the views folder that corresponds to the controller, similar to how the layout folder structure works. In our application, all the API methods have the same json view. For example, in the view file app/views/mf_fetch_threads/json/index.ctp for the MfFetchThreads controller, we just make use of JavaScript helper to format our result in a JSON format, as shown here:

```
<?php
    echo $javascript->object( $result );
?>
```

Our Application Controller

Throughout the development of the forum application, we've placed a lot of attention on our web services API. We're going to continue with that line of thought by using a version of the Command design pattern, adapted for the Web, in our controller.

The Command pattern is an object-oriented class design, where we encapsulate each action in a class of its own. Class names are usually nouns, because they represent objects. However, in the Command pattern, the names are verbs, and each class must implement an execution method, traditionally called `execute()`; in our case, it will be called `index()`. (See `http://en.wikipedia.org/wiki/Command_pattern` for more information about the traditional implementation of the Command pattern.)

Why implement the actions as a class? The advantages are not so obvious in a web environment or with small applications. The following are the advantages of this approach:

- Each action class is treated as an API request action. It has a higher status than the verb object.
- The classes are smaller and more manageable.
- Supporting functions for each action are encapsulated within the action's own class. For example, each class has its own validation method.

To use the Command pattern in our controller, we start with a base controller, which acts as the parent class of all the action classes. Code that is common to all the action classes can be placed here. This base class is shown Listing 4-3.

Listing 4-3. *Base Controller (mf_controller.php)*

```
 1:    <?php
 2:
 3:    class MfController extends AppController {
 4:
 5:        var $name = 'Mf';
 6:
 7:        var $uses = array( 'Thread', 'Message', ➥
'MfSearchProcess', 'MfFetchMessage',
 8:                            'MfFetchMessages', 'MfFetchThreads' );
 9:
10:        function _checkAjax() {
11:
12:            if ( $this->RequestHandler->isAjax() ) {
13:
14:                // This must exist: app/views/layouts/json/ajax.ctp
15:                $this->RequestHandler->renderAs( $this, 'json' );
16:            }
17:        }
18:
19:        function index() {
20:
21:        }
22:    }
23:    ?>
```

The `_checkAjax` method on line 10 is used by the action classes to check whether the API call is an Ajax request. If it is an Ajax request, we render the Ajax layout file in `app/views/layouts/json/ajax.ctp`.

Following this, each action class extends the `MfController` class. An example is shown in Listing 4-4. There are quite often other supporting functions that help the main `index` function.

Listing 4-4. *Example of an Action Class That Extends the Main MfController Class*

```
 1:    <?php
 2:
 3:    include_once( 'mf_controller.php' );
 4:
 5:    class MfExampleController extends MfController {
 6:
 7:        var $name = 'MfExample';
 8:
 9:        var $result = array(    'result' => '',
10:                                'message' => '',
11:                                'errors' => '',
12:                                'data' => ''
13:                                );
14:
15:        function beforeFilter() {
16:
17:            $this->_checkAjax();
18:        }
19:
20:        function _validation() {
21:
22:            $result = true;
23:
24:            // Do validation
25:
26:            return $result;
27:        }
28:
29:        function index() {
30:
31:            if ( $this->_validation() ) {
32:
33:                // Do business logic
34:            }
35:        }
36:
37:        function beforeRender() {
38:
39:            $this->set( 'result', $this->result );
40:        }
41:    }
42:
43:    ?>
```

When the controller is called, it first checks whether it's an Ajax call in the `beforeFilter` on line 15. Next, we go in to the main `index` action on line 29. If the API call validates, we do the business logic. When we're ready to display the results, we package it in the `beforeRender` action on line 37.

Taking this hypothetical example, when we request a Cake URL, it will look like this:

```
http://www.example/MfExample/index/param:1/
```

Now that we've talked about how the controllers are set up, we can show you how we've actually implemented it. Table 4-3 shows all the controller actions we have written.

Table 4-3. *Our Application Controllers*

Controller File	Description
`mf_controller.php`	Parent controller to all action classes
`mf_fetch_post_controller.php`	Fetch a message
`mf_fetch_posts_controller.php`	Fetch several messages
`mf_fetch_threads_controller.php`	Fetch the threads
`mf_post_form_controller.php`	Post messages
`mf_post_process_controller.php`	Process a message
`mf_search_process_controller.php`	Process a search request

Now, if we were to do this in the traditional Cake way, there would be only two files: `message_controller.php`, to contain all actions relating to messages, and `thread_controller.php`, to contain all actions relating to threads. The `MessageController` would contain actions that fetch one or more posts and handle message posting, and it would contain various supporting functions. In this case, by separating out the actions as classes, we gain better management.

All the action classes have their own corresponding model, as shown in Table 4-4.

Table 4-4. *Action Controllers and the Corresponding Models*

Controller	Model	File
`MfController`	No associated model	
`MfFetchMessageController`	`MfFetchMessage`	`app/models/mf_fetch_message.php`
`MfFetchMessagesController`	`MfFetchMessages`	`app/models/mf_fetch_messages.php`
`MfFetchThreadsController`	`MfFetchThreads`	`app/models/mf_fetch_threads.php`
`MfMessageFormController`	`MfMessageForm`	`app/models/mf_message_form.php`
`MfMessageProcessController`	`MfMessageProcess`	`app/models/mf_message_process.php`
`MfSearchProcessController`	`MfSearchProcess`	`app/models/mf_search_process.php`

Next, we'll dive right into the heart of each controller and its view, starting with the `MfFetchMessageController` controller.

Fetch a Message

The `MfFetchMessageController` is used to find a particular message using its `id`. The `findById` method executes a simple SQL `SELECT` statement to the database and fetches the single message record. The code is shown in Listing 4-5.

Listing 4-5. *Fetch One Message Controller (mf_fetch_message_controller.php)*

```
 1:    <?php
 2:
 3:    include_once( 'mf_controller.php' );
 4:
 5:    class MfFetchMessageController extends MfController {
 6:
 7:        var $name = 'MfFetchMessage';
 8:
 9:        var $result = array(    'result' => '',
10:                                'message' => '',
11:                                'errors' => '',
12:                                'data' => ''
13:                                );
14:
15:        function beforeFilter() {
16:
17:            $this->_checkAjax();
18:        }
19:
20:        function _validation() {
21:
22:            $result = true;
23:
24:            $this->data = array( 'MfFetchMessage' );
25:            $this->data[ 'MfFetchMessage' ][ 'messageId' ] = '';
26:
27:            if ( isset( $this->passedArgs[ 'messageId' ] ) ) {
28:                $this->data[ 'MfFetchMessage' ][ 'messageId' ] = ➥
$this->passedArgs[ 'messageId' ];
29:            }
30:
31:            $this->MfFetchMessage->set( $this->data );
32:
33:            if ( !$this->MfFetchMessage->validates() ) {
34:
35:                $result = false;
36:
37:                $this->result[ 'result' ] = '0';
38:                $this->result[ 'message' ] = ➥
"There are some problems with your request.";
```

```
39:                 $this->result[ 'errors' ] = ➥
$this->MfFetchMessage->validationErrors;
40:             }
41:
42:             return $result;
43:         }
44:
45:         function index() {
46:
47:             if ( $this->_validation() ) {
48:
49:                 $message = $this->Message->findById( ➥
$this->data[ 'MfFetchMessage' ][ 'messageId' ] );
50:
51:                 if ( $message == '' ) {
52:                     $this->result[ 'result' ] = '0';
53:                     $this->result[ 'message' ] = ➥
'There was a problem fetching the message. Please try again later.';
54:                 }
55:                 else {
56:
57:                     $this->result[ 'result' ] = '1';
58:                     $this->result[ 'message' ] = ➥
'Message fetched successfully.';
59:                     $this->result[ 'data' ] = $message;
60:                 }
61:             }
62:         }
63:
64:         function beforeRender() {
65:
66:             $this->set( 'result', $this->result );
67:         }
68:     }
69:
70:     ?>
```

Each action class returns a set of key/value pairs to the caller. On lines 9 through 12, there are four standard ones that will always be returned:

- `result`: Whether the request was successful or not: `1` or `0`
- `message`: The human-readable message that goes with the result
- `errors`: A set of key/value pairs, where the key is the parameter and the value is the error message
- `data`: If the request is for data, where to look for it

Most of the action classes have three methods: `beforeFilter`, `index`, and `beforeRender`. The `beforeFilter` method checks whether it's an Ajax call. The parent `_checkAjax()` method looks like this:

```
function _checkAjax() {

    if ( $this->RequestHandler->isAjax() ) {

        // This must exist: \app\views\layouts\json\ajax.ctp
        $this->RequestHandler->renderAs( $this, 'json' );
    }
}
```

We have manually mapped Ajax calls to JSON returns. If it is an Ajax call, the layout `\app\views\layouts\json\ajax.ctp` is used. We would like to return data to callers in a standard way, of course. However, this is not always possible, Ajax and server-side calls can handle formatted data like JSON, but then HTML messes up the format.

The main `index()` method is where all the action happens. We have created a `_validation()` method. Any parameter coming into the action class must be validated, and we make use of Cake's validation as much as we can. To do this, we create a model specifically for our action class called `MfFetchMessage`, as shown in Listing 4-6.

Listing 4-6. *MfFetchMessageController Model (app/models/mf_fetch_message.php)*

```
<?php

class MfFetchMessage extends AppModel
{
    // Mainly for PHP4 users
    var $name = 'MfFetchMessage';

    var $useTable = false;

    var $validate = array(  'messageId' => array(
                               'rule' => array( 'between', 36, 36 ),
                               'required' => true,
                               'message' => 'Please provide a message id  ➥
of 36 characters in length.' )
                            );
}

?>
```

Once the validation has passed, we go ahead and carry out the business logic within `index`. When that's done, the `beforeRender()` method is called. This method sets the `$result` variable for the views. Of course, we could have just as easily set the result in `index()`, but following the idea of the Command pattern, we're keeping code decoupled and standardized.

Our view for the `MfFetchMessageController` is quite simple. It takes `$result[ 'data' ]` and formats it in HTML, ready to be displayed. It's worth highlighting that the JSON output

also makes use of the `$result` variable, but in that case, it formats the output in JSON notation instead of HTML. Our HTML view is shown in Listing 4-7.

Listing 4-7. *Fetch One Message View (app/views/mf_fetch_message/index.ctp)*

```
<?php

    if ( isset( $result[ 'data' ][ 'Message' ] ) ) {

        echo '<div class="message">';

            echo '<div class="message_header">';
                echo '<h3>'.$result[ 'data' ][ 'Message' ] ➥
[ "subject" ].'</a></h3>';
                echo ' <h6>By '.$result[ 'data' ][ 'Message' ] ➥
[ "email" ].'</h6>';
            echo '</div>';

            $message_id = $result[ 'data' ][ 'Message' ][ "id" ];
            $thread_id = $result[ 'data' ][ 'Message' ][ "thread_id" ];

            echo '<div class="message" short_message="" ➥
full_message="" fetched="0" id="message_'.$message_id.'">';

                echo $result[ 'data' ][ 'Message' ][ "message" ];

                $reply_link = $html->link( 'Reply', ➥
'/cake/__chapters__/message_forum/MfMessageForm/index/ ➥
reply_to:'.$message_id.'/thread_id:'.$thread_id.'/' );
                echo '<div>'.$reply_link.'</div>';

            echo '</div>';

        echo '</div>';
    }
?>
```

Fetch Several Messages

At present, our `MfFetchMessagesController`, shown in Listing 4-8, helps us to find queries based on the thread ID.

■ **Note** We can quite easily see the fetch several messages action being deprecated, in favor of using the `MfMessageProcessController` as a generic "find any messages" class. We'll leave that as an exercise for the reader.

Listing 4-8. *Fetch Several Messages Controller (mf_fetch_messages_controller.php)*

```
<?php

include_once( 'mf_controller.php' );

class MfFetchMessagesController extends MfController {

    var $name = 'MfFetchMessages';

    // The message results
    var $messages = array();

    // The message results ordered with indent indicator
    var $messagesOrdered = array();

    var $result = array(    'result' => '',
                            'message' => '',
                            'errors' => '',
                            'data' => ''
                            );

    function beforeFilter() {

        $this->_checkAjax();
    }

    function _validation() {

        $result = true;

        $this->data = array( 'MfFetchMessages' );
        $this->data[ 'MfFetchMessages' ][ 'threadId' ] = '';

        if ( isset( $this->passedArgs[ 'threadId' ] ) ) {
            $this->data[ 'MfFetchMessages' ][ 'threadId' ] = ➥
$this->passedArgs[ 'threadId' ];
        }

        $this->MfFetchMessages->set( $this->data );

        if ( !$this->MfFetchMessages->validates() ) {

            $result = false;

            $this->result[ 'result' ] = '0';
            $this->result[ 'message' ] = ➥
"There are some problems with your request.";
```

```
 45:                $this->result[ 'errors' ] = ➥
$this->MfFetchMessage->validationErrors;
 46:            }
 47:
 48:            return $result;
 49:        }
 50:
 51:        function index() {
 52:
 53:            if ( $this->_validation() ) {
 54:
 55:                $conditions = array ();
 56:
 57:                if ( isset( $this->data[ 'MfFetchMessages' ] ➥
[ 'threadId' ] ) ) {
 58:
 59:                    $threadId = ➥
$this->data[ 'MfFetchMessages' ][ 'threadId' ];
 60:                    $conditions[] = array( ➥
"Message.thread_id =" => $threadId );
 61:                }
 62:
 63:                $messages = $this->Message->find( 'all',
 64:                             array( 'conditions' => $conditions,
 65:                                    null,
 66:                                    'order' => 'Message.t_created_at ASC'
 67:                                    ) );
 68:
 69:                if ( $messages == '' ) {
 70:
 71:                    $this->result[ 'result' ] = '0';
 72:                    $this->result[ 'message' ] = ➥
'There was a problem fetching the threads. Please try again later.';
 73:                }
 74:                else {
 75:
 76:                    $this->messages = $messages;
 77:
 78:                    $this->_sortMessages();
 79:
 80:                    $this->result[ 'result' ] = '1';
 81:                    $this->result[ 'message' ] = ➥
'Messages fetched successfully.';
 82:                    $this->result[ 'data' ] = $this->messagesOrdered;
 83:                }
 84:            }
 85:
```

```
 86:            $this->set( 'result', $this->result );
 87:        }
 88:
 89:        function beforeRender() {
 90:
 91:            $this->set( 'result', $this->result );
 92:        }
 93:
 94:        function _sortMessages( $start_id = '', $level = '' ) {
 95:
 96:            static $stopRun = 1;
 97:
 98:            // You never know!
 99:            if ( $stopRun++ > 1000 ) {
100:                return;
101:            }
102:
103:            for( $idx=0; $idx<sizeof( $this->messages ); $idx++ ) {
104:
105:                if ( !isset( $this->messages[$idx][ 'Done' ] ) ) {
106:
107:                    // Found a root message
108:                    if ( $this->messages[$idx][ 'Message' ] ➥
[ 'reply_to' ] == $start_id ) {
109:
110:                        $this->messages[$idx][ 'Message' ] ➥
[ 'indent' ] = $level;
111:
112:                        $this->messagesOrdered[] = ➥
$this->messages[$idx];
113:
114:                        $message_id = ➥
$this->messages[$idx][ 'Message' ][ 'id' ];
115:
116:                        // That's done; let's remove it
117:                        $this->messages[$idx][ 'Done' ] = '1';
118:
119:                        $this->_sortMessages( $message_id, $level.'.' );
120:                    }
121:                }
122:            }
123:        }
124:    }
125:
126:    ?>
```

As you can see, the structure of this class is similar to that of the `MfFetchMessageController`. In this case, we have an extra private method `_sortMessages()` on line 94, which sorts our message recursively according to replies. We have used a simple full stop to signify the depth of the message within the thread. For peace of mind, we've placed a stopper on line 99 just in case the method goes into an infinite recursive loop.

Again, our view for the `MfFetchMessages` controller simply takes the `$result[ 'data' ]`, loops through the data, and formats the output for HTML display. We've used the indent indicator to indent our messages from the left side of the browser. As we are multiplying the number of full stops by pixel value, the greater the depth, the further away it will be displayed from the left margin (see lines 7 and 9 in Listing 4-9). You can just as easily use `em`, as that probably makes it more accessible. But hey, we just wanted to make it obvious that you can use pixels instead.

Our view is shown in Listing 4-9.

Listing 4-9. *Fetch Multiple Messages View (app/views/mf_fetch_messages/index.ctp)*

```
 1:    <h2>Messages</h2>
 2:
 3:    <?php
 4:
 5:        foreach ( $result[ 'data' ] as $current_message ) {
 6:
 7:            $indent = strlen( $current_message[ 'Message' ][ "indent" ] )*20;
 8:
 9:            echo '<div style="margin-left: '.$indent.'px;">';
10:
11:            echo '<div class="message_header">';
12:            echo '<h3>'.$current_message[ 'Message' ][ "subject" ].'</a></h3>';
13:            echo ' <h6>By ➥
'.$current_message[ 'Message' ][ "email" ].'</h6>';
14:
15:            echo '</div>';
16:
17:            $message_id = $current_message[ 'Message' ][ "id" ];
18:            $thread_id = $current_message[ 'Message' ][ "thread_id" ];
19:
20:            echo '<div class="message_message" ➥
short_message="" full_message="" fetched="0" id="message_'.$message_id.'">';
21:                    echo $current_message[ 'Message' ][ "message" ];
22:                    echo '<div> ➥
<a href="/cake/__chapters__/message_forum/MfMessageForm/index/ ➥
reply_to:'.$message_id.'/thread_id:'.$thread_id.'/">Reply</a></div>';
23:                echo '</div>';
24:
25:             echo '</div>';
26:        }
27:
28:    ?>
```

Fetch the Threads

We have used the pagination feature within Cake to separate the threads into pages. Listing 4-10 shows the `MfFetchThreadsController`.

Listing 4-10. *Fetch the Threads Controller (mf_fetch_threads_controller.php)*

```
 1:    <?php
 2:
 3:    include_once( 'mf_controller.php' );
 4:
 5:    class MfFetchThreadsController extends MfController {
 6:
 7:        var $name = 'MfFetchThreads';
 8:
 9:        var $paginate = array(  'limit' => 20,
10:                                'order' => array( ➥
'Thread.last_message_date' => 'DESC' ) );
11:
12:        var $result = array(    'result' => '',
13:                                'message' => '',
14:                                'errors' => '',
15:                                'data' => ''
16:                                );
17:
18:        function beforeFilter() {
19:
20:            $this->_checkAjax();
21:        }
22:
23:        function _validation() {
24:
25:            $result = true;
26:
27:            return $result;
28:        }
29:
30:        function index() {
31:
32:            if ( $this->_validation() ) {
33:
34:                $threads = $this->paginate( 'Thread' );
35:
36:                if ( $threads == '' ) {
37:                    $this->result[ 'result' ] = '0';
38:                    $this->result[ 'message' ] = ➥
'There was a problem fetching the threads. Please try again later.';
39:                }
```

```
40:                 else {
41:                     $this->result[ 'result' ] = '1';
42:                     $this->result[ 'message' ] = ➥
'Threads fetched successfully.';
43:                     $this->result[ 'data' ] = $threads;
44:                 }
45:             }
46:         }
47:
48:         function beforeRender() {
49:
50:             $this->set( 'result', $this->result );
51:         }
52:     }
53:
54:     ?>
```

Adding pagination is essentially a three-step process. Referring to our controller class, first you set up the options in the controller via the `$paginate` variable on line 9. In our case, we limit the number of records returned to 20. We also order the threads according to the `last_message_date` field in descending order. Next, instead of using the Cake `findBy[Field Name]` method, we use the `paginate` method. This takes the model name as its main parameter (see line 34).

The third and final step in using pagination is at the bottom of the view, shown in Listing 4-11. Cake provides a paginate helper (see line 43), which is included by default once the `paginate` method is used. We simply call the `prev` and `next` methods, which will generate the necessary HTML links.

Listing 4-11. *Fetch the Threads View (app/views/mf_fetch_threads.php)*

```
 1:     <h2>Threads</h2>
 2:
 3:     <?php
 4:         echo $javascript->link( 'mf_fetch_threads/index' );
 5:     ?>
 6:
 7:     <?php
 8:
 9:         foreach ( $result[ 'data' ] as $current_message ) {
10:
11:             $message_id = $current_message[ 'Message' ][ "id" ];
12:             $thread_id = $current_message[ 'Message' ][ "thread_id" ];
13:             $message_num = $current_message[ 'Thread' ][ "message_num" ];
14:
15:             echo '<div class="thread_header">';
16:             echo '<h3>'.$current_message[ 'Message' ] ➥
[ "subject" ].'</a></h3>';
```

```
17:           echo ' <h6>Started By ➥
'.$current_message[ 'Message' ][ "name" ].'</h6>';
18:
19:                // Number of messages
20:                echo ' | '.$message_num.' messages';
21:
22:                // Open link
23:                echo ' | <span id="open_link_'.$message_id.'"> ➥
<a href="javascript: void(0);" ➥
onclick="getMessage(\''.$message_id.'\',\''.$thread_id.'\');">Open</a></span>';
24:
25:                // View thread
26:                echo ' | <a ➥
href="/cake/__chapters__/message_forum/MfFetchMessages/index/threadId:'.➥
$thread_id.'/">List Messages</a>';
27:
28:                // Loading
29:                echo ' | <div id="loading_'.$message_id.'"></div>';
30:
31:            echo '</div>';
32:
33:            // Display first message
34:            echo '<div class="thread_message" short_message=""
35:           full_message="" fetched="0" id="message_'.$message_id.'"></div>';
36:        }
37:
38:    ?>
39:
40:    <hr class="paginator_line">
41:
42:    <?php
43:        if ( isset( $paginator ) ) {
44:
45:            echo $paginator->prev( '« Previous ', null,
46:                                   null, array( 'class' => 'disabled' ) );
47:            echo ' ';
48:            echo $paginator->next( ' Next »', null,
49:                                   null, array( 'class' => 'disabled' ) );
50:            echo ' ';
51:            echo $paginator->counter();
52:        }
53:    ?>
```

Line 4 in Listing 4-11 creates an HTML `script` tag, which will call the JavaScript file located at `app/webroot/js/mf_fetch_threads/index.js`. This file is shown in Listing 4-12.

Listing 4-12. *JavaScript File Used to Support the Fetching of Threads (app/webroot/js/mf_fetch_threads/index.js)*

```
 1:    function getMessage(message_id,thread_id) {
 2:
 3:        var full_message = ➥
$('message_'+message_id).getAttribute( 'full_message' );
 4:
 5:        if ( full_message != '' ) {
 6:            $('message_'+message_id).innerHTML = full_message;
 7:            return false;
 8:        }
 9:
10:        new Ajax.Request( '/cake/__chapters__/message_forum/MfFetchMessage/ ➥
index/messageId:'+message_id+'/.json',
11:            {   asynchronous:true,
12:                evalScripts:true,
13:                onComplete: function(response,json){
14:
15:                    if ( json[ 'result' ] == '1' ) {
16:
17:                        message_links = ➥
json[ 'data' ][ 'Message' ][ 'message' ];
18:                        message_links += '<div>';
19:                        message_links += '<a href="javascript: ➥
void(0);" onclick="closeMessage(\''+message_id+'\');">Close</a> |';
20:                        message_links += ➥
'<a href="/cake/__chapters__/message_forum/MfFetchMessages/index/ ➥
threadId:'+thread_id+'/">List Messages</a> |';
21:                        message_links += ➥
'<a href="/cake/__chapters__/message_forum/MfMessageForm/index/ ➥
reply_to:'+message_id+'/thread_id:'+thread_id+'/">Reply</a>';
22:                        message_links += '</div>';
23:
24:                        $('message_'+message_id).innerHTML = message_links;
25:
26:                        $('message_'+message_id).setAttribute( ➥
'full_message', $('message_'+message_id).innerHTML );
27:                        $('loading_'+message_id).innerHTML = '';
28:                    }
29:                },
30:                onLoading: function() {
31:
32:                    $('message_'+message_id).setAttribute( ➥
'short_message', $('message_'+message_id).innerHTML );
33:                    $('loading_'+message_id).innerHTML = ➥
'<img src="/cake/__chapters__/message_forum/img/ajax-loader.gif" alt="" /> ➥
loading, please wait.';
34:                }
35:                } );
36:    }
```

```
37:
38:    function closeMessage(message_id) {
39:
40:        var short_message = $('message_'+message_id).➥
getAttribute( 'short_message' );
41:        $('message_'+message_id).innerHTML = short_message;
42:    }
```

When a user clicks the Open link (see line 23 in Listing 4-11), the `getMessage` function on line 1 in Listing 4-12 is called. This function makes an Ajax call to the `MfFetchMessage` method (see line 10). That method will return the message in JSON format. We start the parsing on line 13. We finish the process by displaying the message together with some links, on line 24. A sample of the parsed output is shown in Figure 4-3.

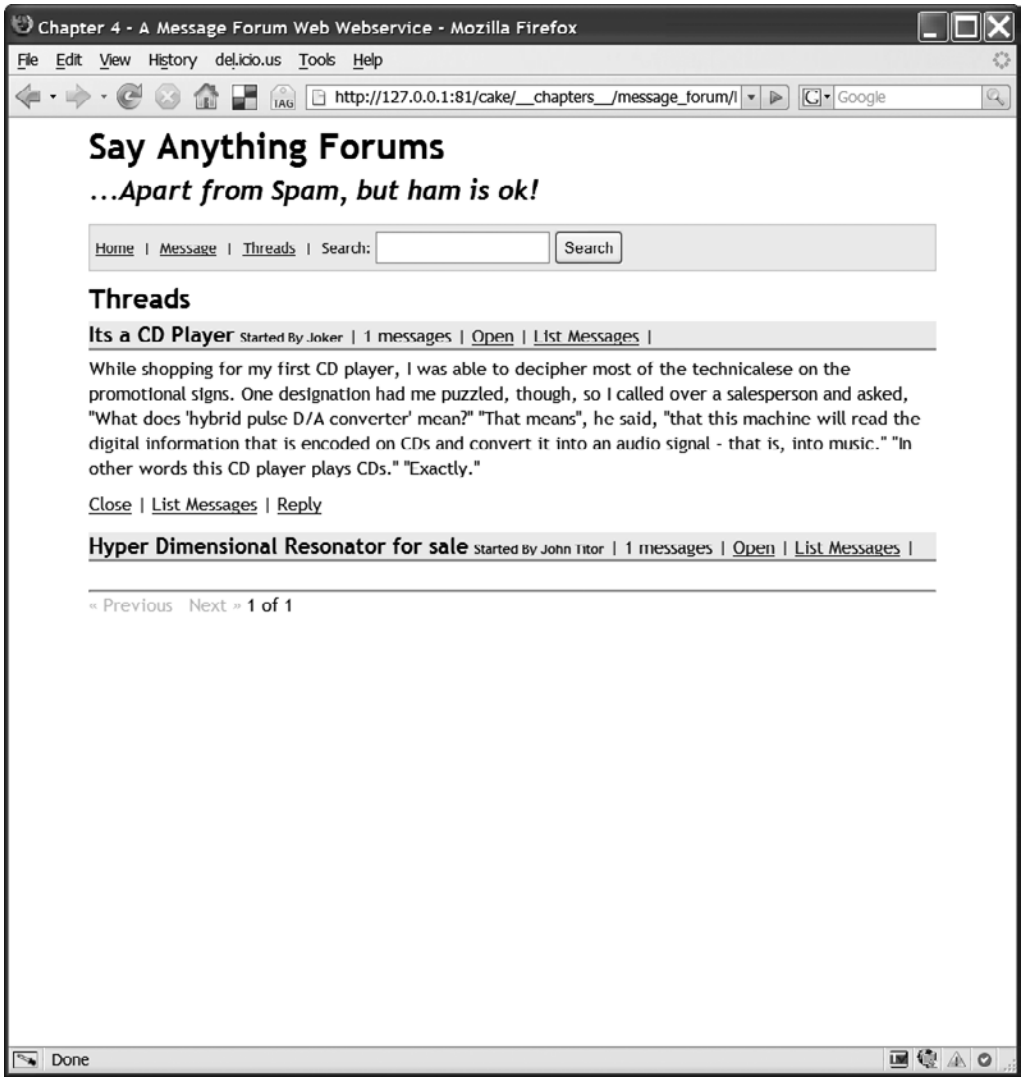

Figure 4-3. *Displaying threads*

Post Messages

Our message posting form, shown in Listing 4-13, posts messages using Ajax by default. However, if the user has JavaScript turned off, it will degrade to standard HTML posting. Because it's not part of our API method calls, its structure doesn't mirror the API action classes. However, in the future we may include this as an API method call where users can use our form instead of writing their own. They can conveniently include the form in the sidebar of their web site—in a blog, for example.

Listing 4-13. *Post a Message Controller (mf_message_form_controller.php)*

```
<?php

include_once( 'mf_controller.php' );

class MfMessageFormController extends MfController {

    var $name = 'MfMessageForm';

    function index() {

        if ( isset( $this->passedArgs['reply_to'] ) ) {

            $this->data['Message']['reply_to'] = $this->passedArgs['reply_to'];

            $replyMessageId = $this->data['Message']['reply_to'];
            $replyMessage = $this->Message->findById( $replyMessageId );

            if ( !empty( $replyMessage ) ) {

                $this->data['Message']['subject'] = ➥
'RE: '.$replyMessage[ 'Message' ][ 'subject' ];
            }
        }

        if ( isset( $this->passedArgs['thread_id'] ) ) {

            $this->data['Message']['thread_id'] = $this->passedArgs['thread_id'];
        }
    }
}

?>
```

The view for the form is pretty sparse, as shown here:

```
<?php
    echo $this->element( 'message_form' );
?>
```

As you can see, the view for the message form is separated off into an element, as shown in Listing 4-14.

Listing 4-14. *Message Posting Form as an Element (/app/views/elements/message_form.ctp)*

```
 1:    <?php
 2:        if ( $session->check( 'Message.flash'  ) ) {
 3:            echo '<div class="hilight">';
 4:            $session->flash();
 5:            echo '</div>';
 6:        }
 7:    ?>
 8:
 9:    <!-- TinyMCE -->
10:    <?php
11:        echo $javascript ➥
->link( 'jscripts/tinymce/jscripts/tiny_mce/tiny_mce' );
12:    ?>
13:    <script type="text/javascript">
14:        tinyMCE.init({
15:            mode : "textareas",
16:            theme : "advanced",
17:            theme_advanced_buttons1 : ➥
"bold,italic,underline,separator,strikethrough,justifyleft, ➥
justifycenter,justifyright, justifyfull,bullist,numlist, ➥
undo,redo,link,unlink",
18:            theme_advanced_buttons2 : "",
19:            theme_advanced_buttons3 : "",
20:            theme_advanced_toolbar_location : "top",
21:            theme_advanced_toolbar_align : "left",
22:            theme_advanced_statusbar_location : "bottom",
23:            extended_valid_elements : ➥
"a[name|href|target|title|onclick],img[class|src|border=0|alt| ➥
title|hspace|vspace|width|height|align|onmouseover|onmouseout|name], ➥
hr[class|width|size|noshade],font[face|size|color|style], ➥
span[class|align|style]",
24:            content_css : "/cake/__chapters__/message_forum/css/global.css"
25:
26:        });
27:    </script>
28:    <!-- /TinyMCE -->
29:
30:    <?php
31:        echo $javascript->link( 'elements/message_form' );
32:    ?>
33:
34:    <div id="page_message"></div>
35:
```

```
36:    <div id="message_form">
37:
38:    <?php
39:        echo $ajax->form(   null,
40:                            'post',
41:                            array(  'url' => '/MfMessageProcess/index',
42:                                    'complete' => 'updateForm(request,json);',
43:                                    'before' => 'beforeMessage();'
44:                                    )
45:                            );
46:    ?>
47:
48:    <fieldset style="border:1px solid black; background-image:url( ➥
'/cake/__chapters__/message_forum/app/webroot/img/background.gif' );">
49:        <legend> <?php __('Add a Message!');?> </legend>
50:        Please fill in all fields. Remember, spam and be damned!
51:        <?php
52:            echo $form->error( 'Message.name' );
53:            echo $form->input( 'Message.name', array( ➥
'id' => 'messagename', 'label' => 'Name:', 'size' => '50', ➥
'maxlength' => '255', 'error' => false ) );
54:
55:            echo $form->error( 'Message.email' );
56:            echo 'Your email will not be displayed.';
57:            echo $form->input( 'Message.email', array( ➥
'id' => 'messageemail', 'label' => 'Email:', 'size' => '50', ➥
'maxlength' => '255', 'error' => false ) );
58:
59:            echo $form->error( 'Message.subject' );
60:            echo $form->input( 'Message.subject', array( ➥
'id' => 'messagesubject', 'label' => 'Subject:', 'size' => '50', ➥
'maxlength' => '255', 'error' => false ) );
61:
62:            echo $form->error( 'Message.message' );
63:            echo $form->input( 'Message.message', array( ➥
'id' => 'messagemessage', 'type'=>'textarea', 'label' => 'Message:', ➥
'rows' => '20', 'error' => false ) );
64:
65:            echo $form->hidden( 'Message.reply_to' );
66:            echo $form->hidden( 'Message.thread_id' );
67:        ?>
68:    </fieldset>
69:
70:    <?php echo $form->end( array( 'label' => ' Submit Message ' ) );?>
71:
72:    </div>
```

There are a number of interesting things happening in this element. Before we go any further, we have to admit that this simple form turned out to be quite complex because of the use of Ajax and how it can degrade in the absence of JavaScript.

First, we have included the TinyMCE what you see is what you get (WYSIWYG) web editor. It was either that or the FCKeditor, which is just as good.

Note TinyMCE (`http:/tinymce.moxiecode.com`) and FCKeditor (`http:/www.fckeditor.net`) are two of the most popular web-based open source text editors. You will generally use them to replace the HTML `textarea` tag. They provide editing capabilities much like any desktop word processing editor.

Installing TinyMCE is quite easy. First, download the package from the TinyMCE web site (`http://tinymce.moxiecode.com`). Then unzip the code into the folder `/app/webroot/js/jscripts/`.

Next, we need to initialize the editor. This is done from lines 10 to 27 in Listing 4-14. Later, when the form is saved, we also need to trigger TinyMCE to save the content. This is done in the JavaScript helper file (see Listing 4-15, line 47).

In Listing 4-14, we include our JavaScript helper file in the line `echo $javascript->link( 'elements/message_form' );` on line 31. The helper file, shown in Listing 4-15, is stored in `/app/webroot/js/elements/message_form.js`. We have created an `elements` folder within `/app/webroot/js`. This allows us to easily pick out which JavaScript file goes with which `.ctp` file, as we have named the JavaScript file to be the same as the `.ctp` file that included it.

Listing 4-15. *JavaScript Helper Functions for the Message Posting Form (/app/webroot/js/elements/message_form.js)*

```
 1:
 2:    function updateForm(request,json) {
 3:
 4:        var msg_render = '';
 5:        var result = json['result'];
 6:        var message = json['message'];
 7:        var errors = json['errors'];
 8:
 9:        $('page_message').innerHTML = '<div class="hilight">'+message+'</div>';
10:
11:        $$('.submit input')[0].disabled = false;
12:
13:        // Remove all previous error messages
14:        var current_errors = $$('.error-message');
15:
16:        for( var i=0; i<current_errors.length; i++ ) {
17:            var error_tag = current_errors[i];
18:            error_tag.parentNode.removeChild(error_tag);
19:        }
20:
```

```
21:         // If there are error messages, update these
22:         if ( errors != '' ) {
23:
24:             for( var field_id in errors ){
25:
26:                 // The error message in a div
27:                 var div_error = document.createElement( "div" );
28:                 div_error.className = "error-message";
29:                 div_error.innerHTML = errors[field_id];
30:
31:                 form_ele = $('message'+field_id);
32:                 parent_form_ele = form_ele.parentNode;
33:
34:                 parent_form_ele.insertBefore(   div_error,
35:                                                 form_ele.previousSibling );
36:             }
37:         }
38:
39:         // If it was successful, then we clear the form
40:         if ( result == 1 ) {
41:             $('message_form').innerHTML = '';
42:         }
43:     }
44:
45:     function beforeMessage() {
46:
47:         tinyMCE.triggerSave();
48:
49:         $('page_message').innerHTML = ➥
'<div class="hilight">Posting, please wait ... ➥
<img src="/cake/__chapters__/message_forum/img/ajax-loader.gif" alt="" /> ➥
</div>';
50:
51:         $$('.submit input')[0].disabled = true;
52:     }
53:
```

In the `updateForm` method in Listing 4-15, we basically parse the form process return results, which is in the standard format we described earlier, where the results contain the keys, message, errors, and data. We first remove any errors that were previously there. We do this simply by using the `removeChild` method on line 18.

We then display any errors by looping through the `errors` key value and displaying it before the field. The Cake Ajax form can easily do this step for us, but it displays the error messages after the field, and we want the user to read the error first and then see the field to which it refers. We use the DOM method `insertBefore` to insert a `DIV` tag, which contains the error message before the field. The web page `http://developer.mozilla.org/en/docs/DOM:element.insertBefore` has some good information about DOM methods.

If there are no errors, the form has been posted successfully, and we then clear the form to avoid any trigger-happy user. Additionally, we can display some other useful information there at a later date—maybe some advertisement relating to the posted message, for example.

Before posting the Ajax request, we call the `beforeMessage()` method on line 45. In that function, we need to manually get TinyMCE to save our message. This happens only because we are using Ajax. If you're using ordinary HTML to post, you won't need to carry out this step with TinyMCE.

After that, we simply add some user interface sugar, starting with a spinning Ajax loader image. Then we disable the submit button.

Returning to our message posting form in Listing 4-14, our next step simply uses the Ajax and form helpers to add some form elements (see line 38). An important point to note is the handling of the error messages relating to the fields. This point applies only when the user browser is not running JavaScript; that is, HTML is being used to post messages. We have turned off the automatic display of the error messages relating to Cake's rendering of the input tags. This is because we want the error messages to be displayed before the input fields. To add the error messages before the field, we manually display the error messages, as in this example:

```
echo $form->error( 'Message.name' );
```

A sample form for adding a message is shown in Figure 4-4.

■ **Note** The Ajax helper is a wrapper for Prototype's methods. In the Ajax helper, you can use the `update` option to specify which `div` container to update the return result. If you don't use the `update` option, Cake will use Prototype's `Request` method. If you do specify an `update` option, it will use the `Updater` method.

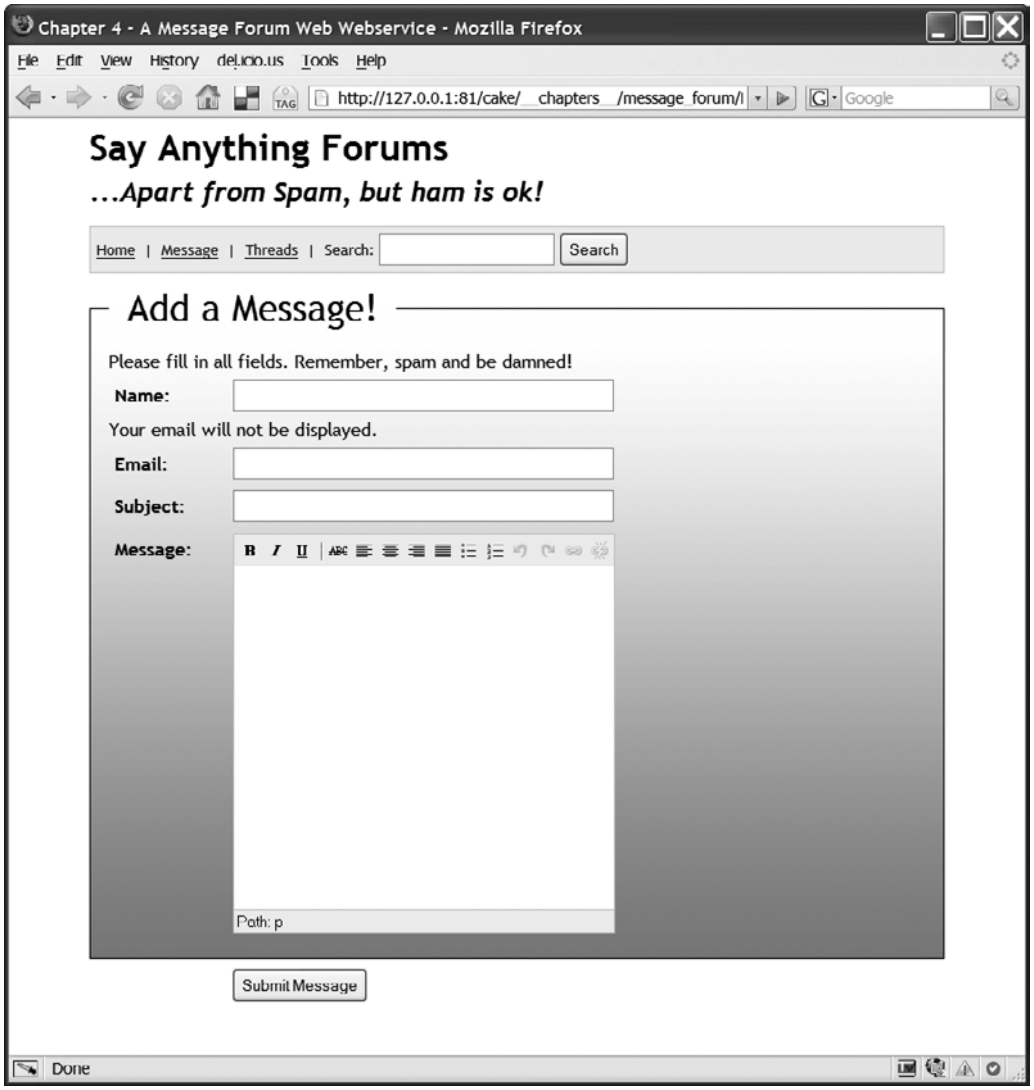

Figure 4-4. *Posting a message*

Process a Message

In the processing or saving of a message, we use a model that directly maps to a table. This is unlike some of the earlier controller actions, where the models were essentially wrapper classes for validation, as `MfMessageProcess` is an API method that we're exposing. As shown in Listing 4-16, it follows the same class structure as the other API methods.

Listing 4-16. *Save a Message Controller (mf_message_process_controller.php)*

```
<?php

include_once( 'mf_controller.php' );

class MfMessageProcessController extends MfController {

    var $name = 'MfMessageProcess';

    var $result = array(    'result' => '',
                            'message' => '',
                            'errors' => '',
                            'data' => ''
                            );

    function beforeFilter() {

        $this->_checkAjax();
    }

    function _validation() {

        $result = true;

        if ( !empty( $this->data ) ) {

            $this->Message->set( $this->data );

            if ( !$this->Message->validates() ) {

                $result = false;

                if ( $this->RequestHandler->requestedWith() == 'form' ) {
                    $this->Session->setFlash( __( ➥
'Sorry, there was a problem with your form details, see below.', true ) );
                }

                $this->result[ 'result' ] = '0';
                $this->result[ 'message' ] = ➥
"There are some problems with your request.";
                $this->result[ 'errors' ] = ➥
$this->Message->validationErrors;
            }
        }

        return $result;
    }

```

```
45:         function index() {
46:
47:             if ( $this->_validation() ) {
48:
49:                 $message_date = date( 'y-m-d H:i:s', mktime() );
50:                 $message_id = String::uuid();
51:                 $thread_id = String::uuid();
52:                 $origThreadId = $this->data[ 'Message' ][ 'thread_id' ];
53:
54:                 if ( $this->data[ 'Message' ][ 'thread_id' ] ) {
55:                     $thread_id = $this->data[ 'Message' ][ 'thread_id' ];
56:                 }
57:
58:                 $this->Message->create();
59:
60:                 $this->data[ 'Message' ][ 't_created_at' ] = $message_date;
61:                 $this->data[ 'Message' ][ 'thread_id' ] = $thread_id;
62:                 $this->data[ 'Message' ][ 'id' ] = $message_id;
63:
64:                 if ( $this->Message->save( $this->data ) ) {
65:
66:                     if ( $this->RequestHandler->requestedWith() == 'form' ) {
67:                         $this->Session->setFlash( ➥
__( 'Your message has been posted!', true ) );
68:                     }
69:
70:                     $this->result[ 'message' ] = ➥
'Your message has been posted!';
71:                     $this->result[ 'result' ] = '1';
72:                     $this->result[ 'errors' ] = '';
73:
74:                     // Save the thread
75:                     if ( empty( $origThreadId ) ) {
76:
77:                         $threadData = array();
78:                         $threadData[ 'first_message_id' ] = $message_id;
79:                         $threadData[ 'last_message_date' ] = $message_date;
80:                         $threadData[ 'id' ] = $thread_id;
81:                         $threadData[ 'message_num' ] = '1';
82:
83:                         $this->Thread->create();
84:                         $thread_save = $this->Thread->save( $threadData );
85:
86:                         if ( !$thread_save ) {
87:
88:                             $this->Message->delete( $this->data );
89:                         }
90:                     }
```

```
 91:                    else {
 92:                        $this->Thread->threadPlusOne( $origThreadId );
 93:                    }
 94:
 95:                    // Everything seems OK. Let's do a redirect after message.
 96:                    // Is it coming from a standard HTML message form
 97:                    if ( $this->RequestHandler->requestedWith() == 'form' ) {
 98:                        $this->redirect( array( ➥
'controller' => 'MfMessageForm',
 99:                                                'action' => 'index' ) );
100:                        exit();
101:                    }
102:                }
103:                else {
104:
105:                    if ( $this->RequestHandler->requestedWith() == 'form' ) {
106:                        $this->Session->setFlash( ➥
__( 'Sorry, there was a problem with your form details, see below.', true ) );
107:                    }
108:
109:                    $this->result[ 'message' ] = ➥
'Sorry, there was a problem with your form details.';
110:                    $this->result[ 'result' ] = '0';
111:                    $this->result[ 'errors' ] = ➥
$this->Message->validationErrors;
112:                }
113:            }
114:        }
115:
116:        function beforeRender() {
117:
118:            $this->set( 'result', $this->result );
119:        }
120:    }
121:
122:    ?>
```

There isn't much in the process view for HTML requests. It simply displays the message form again, as follows:

```
<?php
    echo $this->renderElement( 'message_form' );
?>
```

In Listing 4-16, we start off with some Cake validation. Once validation is all good (on line 47), we do the business stuff of saving the message. If a user is posting a new message, we also need to create a thread record for it. However, if the message wasn't saved for some reason, we need to remove it from the thread record that we have already saved.

We have used UUIDs to help us with two coding tasks (see lines 50 and 51):

- It allows us to easily delete the thread record if the message record didn't save. This isn't strictly needed, as we can always get the ID quite easily once the thread has been saved.
- Since a thread needs the ID of the last message posted (or the first one if it's a fresh post), and a message needs the ID of the thread, it makes sense for us to generate the IDs within the code, rather than rely on the database to give us IDs. We save on the number of queries we make and have fewer lines of code.

Another feature within the code in Listing 4-16 is the redirect after the post, which is quite a common practice nowadays in order to avoid repeated posts. However, this feature isn't needed in an Ajax call, so we filter this out on lines 32, 66, and 105.

Process a Search Request

Again, the `MfSearchProcessController` follows the structure of the other API methods, as shown in Listing 4-17.

Listing 4-17. *Search for Messages Controller (mf_search_process_controller.php)*

```
  1:    <?php
  2:
  3:    include_once( 'mf_controller.php' );
  4:
  5:    class MfSearchProcessController extends MfController {
  6:
  7:        var $name = 'MfSearchProcess';
  8:
  9:        var $paginate = array(  'limit' => 20,
 10:                                'order' => array( ➥
'Message.t_created_at' => 'asc' ) );
 11:
 12:        var $result = array(    'result' => '',
 13:                                'message' => '',
 14:                                'errors' => '',
 15:                                'data' => '',
 16:                                'search_term' => ''
 17:                                );
 18:
 19:        var $search_term = '';
 20:
 21:        function _validation() {
 22:
 23:            $result = true;
 24:
 25:            if ( isset( $this->data[ 'MfSearchProcess' ]['search_term'] ) ) {
 26:
```

```
 27:                $search_term = ➥
$this->data[ 'MfSearchProcess' ]['search_term'];
 28:            }
 29:            elseif ( $this->Session->read( "search_term" ) ) {
 30:
 31:                $search_term = $this->Session->read( "search_term" );
 32:                $this->data[ 'MfSearchProcess' ]['search_term'] = ➥
$search_term;
 33:            }
 34:            elseif ( isset( $this->passedArgs['search_term'] ) ) {
 35:
 36:                $search_term = $this->passedArgs['search_term'];
 37:                $this->data[ 'MfSearchProcess' ]['search_term'] = ➥
$search_term;
 38:            }
 39:
 40:            $this->MfSearchProcess->set( $this->data );
 41:
 42:            // Now check search term
 43:            if ( !$this->MfSearchProcess->validates() ) {
 44:
 45:                $result = false;
 46:
 47:                $this->result[ 'result' ] = '0';
 48:
 49:                // Since there's only 1 field, for the minute
 50:                // we'll assume it's from the search term
 51:                $validation_error = '';
 52:
 53:                if ( isset( $this->MfSearchProcess ➥
->validationErrors[ 'search_term' ] ) ) {
 54:                    $validation_error = $this ➥
->MfSearchProcess->validationErrors[ 'search_term' ];
 55:                }
 56:
 57:                $message = __( ➥
'Sorry, there was a problem with your search form. '.$validation_error, true );
 58:
 59:                $this->result[ 'message' ] = $message;
 60:                $this->result[ 'errors' ] = ➥
$this->MfSearchProcess->validationErrors;
 61:            }
 62:
 63:            $this->result[ 'search_term' ] = ➥
$this->data[ 'MfSearchProcess' ]['search_term'];
 64:            $this->search_term = ➥
$this->data[ 'MfSearchProcess' ]['search_term'];
 65:
```

```
 66:            return $result;
 67:        }
 68:
 69:        function index() {
 70:
 71:            $messages = array();
 72:
 73:            if ( $this->_validation() ) {
 74:
 75:                $conditions = array();
 76:
 77:                if ( $this->search_term ) {
 78:
 79:                    $this->Session->write( "search_term", ➥
$this->search_term );
 80:
 81:                    $search_term = $this->search_term;
 82:
 83:                    $conditions[] = ➥
array( "MATCH(email,subject,message) AGAINST ('$search_term')" );
 84:                }
 85:
 86:                $messages = $this->paginate( ➥
'Message', array( "or" => $conditions ) );
 87:
 88:                if ( $messages == '' ) {
 89:                    $this->result[ 'result' ] = '0';
 90:                    $this->result[ 'message' ] = ➥
'There was a problem with the search. Please try again later.';
 91:                }
 92:                else {
 93:
 94:                    $this->result[ 'result' ] = '1';
 95:                    $this->result[ 'message' ] = ➥
'Search results fetched successfully.';
 96:                    $this->result[ 'data' ] = $messages;
 97:                }
 98:            }
 99:        }
100:
101:        function beforeRender() {
102:
103:            $this->set( 'result', $this->result );
104:        }
105:    }
106:
107:    ?>
```

We first validate to see whether we have a search term on line 73. Once that passes, we use Cake's pagination feature to split the results into pages.

We have used a dummy model `MfSearchProcess` as a way to validate our search term using Cake's validation function. We need to store the search term in session so it can be used during the pagination when a user selects a different page.

On line 83, we have used MySQL's `MATCH AGAINST` operator in our queries. This gives a wider range of accurate matches. To use the MySQL `MATCH AGAINST` operator in your code, you will need to create a full-text index on the database fields using this command:

```
CREATE FULLTEXT INDEX full_text_1 ON messages (email,subject,message);
```

■ **Note** Avoid using the `LIKE` operator. The `LIKE` operator is quite expensive, as it must scan all the fields where the operator is used. Also, it doesn't match variations of the word. For example, if the search term is *running*, it won't search for *run* or *runner*. MySQL has many other search methods besides the standard `LIKE`. See `http://dev.mysql.com/doc/refman/5.0/en/fulltext-search.html` for details.

The view for our search function is shown in Listing 4-18.

Listing 4-18. *View Template for the Search Action Class (/app/views/mf_search_process/index.ctp)*

```
<h2>Search Results</h2>

<?php
    if ( $result[ 'result' ] == '0' ) {
        echo '<div class="hilight">';
        echo $result[ 'message' ];
        echo '</div>';
    }
?>

<?php

    if ( $result[ 'data' ] ) {

        foreach ( $result[ 'data' ] as $current_message ) {

            echo '<div class="message_header">';
                echo '<h3>'.$current_message[ 'Message' ] ➥
[ "subject" ].'</a></h3>';
                echo ' <h6>By '.$current_message[ 'Message' ] ➥
[ "email" ].'</h6>';

            echo '</div>';
```

```
            $message_id = $current_message[ 'Message' ][ "id" ];
            $thread_id = $current_message[ 'Message' ][ "thread_id" ];

            echo '<div class="message_message" short_message="" ➥
full_message="" fetched="0" id="message_'.$message_id.'">';

                echo $text->highlight(   $current_message[ 'Message' ] ➥
[ "message" ],
                                    $result[ 'search_term' ],
                                    '<span class="highlight_search">\1</span>'
                                    );

                echo '<div><span id="open_link_'.$message_id.'"> ➥
<a href="/cake/__chapters__/message_forum/MfFetchMessages/index/ ➥
threadId:'.$thread_id.'/">View in thread</a></div>';
            echo '</div>';
        }
    }
    else {
        echo '<div class="hilight">';
        echo 'No Results';
        echo '</div>';
    }

?>

<hr class="paginator_line">

<?php
    if ( isset( $paginator ) ) {

        $paginator->options(array( ➥
'update' => 'main_content_container', 'indicator' => 'loading'));

        echo $paginator->prev( '« Previous ', null, null, ➥
array( 'class' => 'disabled' ) );
        echo ' ';
        echo $paginator->next( ' Next »', null, null, ➥
array( 'class' => 'disabled' ) );
        echo ' ';
        echo $paginator->counter();
    }
?>
```

The view loops through the message results and formats the output. In the pagination helper methods at the bottom, we have used Ajax for the `next` and `prev` links. In Cake, we simply add the following line to Ajaxify the links:

```
$paginator->options(array( 'update' => 'main_content_container',
                           'indicator' => 'loading'));
```

Tip We have used the text helper to highlight our search term. But don't forget there are also some other basic functions in the Cake API that may help you out in other situations. For example, in parts of this application, we have used the `__()` helper function instead of using `echo` or even the Cake shorthand `echo` function `function e()`. See `http://api.cakephp.org/basics_8php.html` for details.

Writing the API Documentation

For many developers, writing documentation is never a pleasant experience. But for this chapter, you could say it's the end point. Without the documentation, there's no reason for writing the API. Why bother to structure our actions around the Command pattern? It would have been easier to just write each action within one or two controllers.

The following shows the documentation for the Web Forum API as it would appear on our forum web site.

Welcome to our message forum API. We provide five different methods for you to use. All our method returns use the JSON format. Each request will return four or more keys. The standard four keys returned on every request are as follows:

- `result`: If 1 is returned, this means the request was processed successfully. If 0 is returned, then see `message` and `errors` key.
- `message`: A human-friendly return message. This will complement the `result` key.
- `errors`: This will contain error messages relating to the result. It will be given in key/value pairs, where the `key` is the name of the parameter and `value` contains the error message.
- `data`: If the request returns data, it will be held in this key.

You must end all URL requests in `.json`. An example request would look like this:

```
http://www.example.com/MfFetchMessage/ ➥
index/Id:488bb046-2ae8-43c2-af41-0154cbdd56cb/.json
```

And the return response may look like this:


```
{"result":"1","message":"Message fetched successfully.","errors":"","data": ➥
{"Message":{"id":"488bb046-2ae8-43c2-af41-0154cbdd56cb","name":"John Titor", ➥
"email":"sfsd","message":"<p>Time travel machine. Good working order. Used ➥
only once. 10,000 Ningis ono. if you're interested, please call me at ➥
314 159 2653. <\/p>","reply_to":"","subject":"Hyper Dimensional Resonator ➥
for sale","t_created_at":"2008-07-27 00:16:22", ➥
"thread_id":"488bb046-5004-44e2-9dbd-0154cbdd56cb"}, ➥
"Thread":{"id":"488bb046-5004-44e2-9dbd-0154cbdd56cb", ➥
"first_message_id":"488bb046-2ae8-43c2-af41-0154cbdd56cb", ➥
"last_message_date":"2008-07-27 00:16:22","message_num":"1"}}}
```


The API documentation follows.

MfFetchMessage

Get one forum message.

Arguments

`messageId` (Required)—The message ID

HTTP Method

`POST`

Syntax

```
http://[domain]/MfFetchMessage/index/messageId:[message_id]/.json
```

Return

Standard return keys

MfFetchMessages

Note the extra letter `s`. Get more than one message from the forum. At present, we filter only by the thread ID.

Arguments

`threadId` (Required)—The thread ID

HTTP Method

POST

Syntax

```
http://[domain]/MfFetchMessages/index/threadId:[thread_id]/.json
```

Return

Standard return keys

MfFetchThreads

Get the forum threads. This returns a set of paginated messages. The messages are sorted according to the date of the last message that was posted within a thread. Each page has 20 messages. At present, this amount is fixed.

Arguments

`page` (Optional)—The page number starting from 1. Numeric.

HTTP Method

POST

Syntax

```
http://[domain]/MfFetchMessages/index/page:[page_number]/.json
```

Return

Standard return keys

MfMessageProcess

Post a message onto the forum.

Arguments

- `name` (Optional)—Sender's name. Maximum 255 characters.
- `email` (Required)—Sender's e-mail address. Maximum 255 characters.
- `subject` (Required) —Message subject. Maximum 255 characters.
- `message` (Optional) —Message body. Maximum 16,777,215 characters.

HTTP Method

POST

Syntax

```
http://[domain]/MfMessageProcess/index.json
```

Return

Standard return keys

MfSearchProcess

Search for messages. The results are paginated with 20 messages per page.

Arguments

- searchTerm (Required)—Search term. Maximum 255 characters.
- page (Optional)—Page number. Numeric.

HTTP Method

POST

Syntax

```
http://[domain]/MfSearchProcess/index.json
```

Return

Standard return keys, plus extra search_term key, which contains the search term

Summary

In this application, we've covered several Cake topics, including the use of Ajax, JSON web service returns, Cake pagination, the use of validation in a model with no associated table, and how to include the TinyMCE browser editor. The highlight of the chapter, however, was the building of a web service API.

The application still has a number of features we should add before it can be used in a commercial environment. Here are some suggestions that you can use to further develop the forum application:

- This is an obvious one. Stop spammers! There are many techniques to prevent spam, or at least keep it down to a minimum. In the final chapter in this book, we implement a Captcha authentication test. You can easily integrate this into the forum application. However, this puts the responsibility on every user. If you don't like interrupting the usability of the site, you can employ a filter, like a Bayesian text filter, instead. There are several PHP versions of this filter floating about. For other techniques, see `http://en.wikipedia.org/wiki/Forum_spam`.
- Users should be able to upload images to the forum.
- Users should be able to e-mail updates on a thread. This is particularly important for users who have posted questions.
- Add user accounts. You could then allow only registered users to post messages. However, this might be a barrier to the use of the forum, since it's one more hurdle for users to jump over. Fewer users posting messages means fewer people will find anything interesting on the forum.
- We have to admit that our application has one slightly uncomfortable area. Our client action classes like `mf_post_form_controller.php` and our API action classes like `mf_search_process_controller.php` more or less live within the same domain. They share the same folders and same parent controllers. It's as if they were part of the API, but, of course, they are not. You should separate them out, either into separate controllers or separate folders.

CHAPTER 5

Google Maps and the Traveling Salesman

Some friends of ours are about to go on a big European vacation. They talked about how they've used Google Maps to help them find the location of hotels, restaurants, and local attractions. They wanted to keep friends and families updated of their progress, but didn't want to use e-mail. Furthermore, they weren't sure in which order to visit the places. Being far more enthusiastic than they were, we said we would write an application for them that they can use to enter their destinations, make comments, and plan their journey.

In our application, we're going to be covering many topics. Since we're going to be using Google Maps, client-side JavaScript will be employed. We also want to store the locations and the comments they make. Naturally, we'll be creating some Cake models to represent these data entities. We will also be building a straightforward controller to hold the functions that will manage the locations and comments.

One of the main features of the application relates to a classic computing puzzle called the "traveling salesman problem" (or in our case, the traveling tourist problem). Namely, a salesman needs to visit a number of cities only once, but return back to the same place as where he started. Devise an algorithm to find the shortest route for the whole trip. This part of the application will be done client side using a simple algorithm, but the calculated route will be stored on the database.

Now, to get started, we'll first talk about Google Maps.

Hello Map!

Google Maps is pretty easy to use, so we can dive straight in with the Google Map equivalent of the Hello World program, as shown in Listing 5-1.

Listing 5-1. *A Simple Google Maps Example*

```
 1:    <!DOCTYPE html "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 2:    <html xmlns="http://www.w3.org/1999/xhtml">
 3:    <head>
 4:    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
 5:    <title>Google Maps JavaScript API Example</title>
 6:    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;➥
key=insert_your_key_here" type="text/javascript"></script>
```

```
 7:    <script type="text/javascript">
 8:    function initialize() {
 9:    if ( GBrowserIsCompatible() ) {
10:    var map = new GMap2( document.getElementById( "map_container" ) );
11:    map.setCenter( new GLatLng( 51.49937, -0.14421 ), 13 );
12:    }
13:    }
14:    </script>
15:    </head>
16:    <body onload="initialize()" onunload="GUnload()">
17:    <div id="map_container" style="width: 600px; height: 400px"></div>
18:    </body>
19:    </html>
```

The code in Listing 5-1 displays a 600 × 400 pixel Google Map on the page. The location of the map is set to the coordinates for London, specified by the `setCenter()` function on line 11. The output from the code in Listing 5-1 is shown in Figure 5-1.

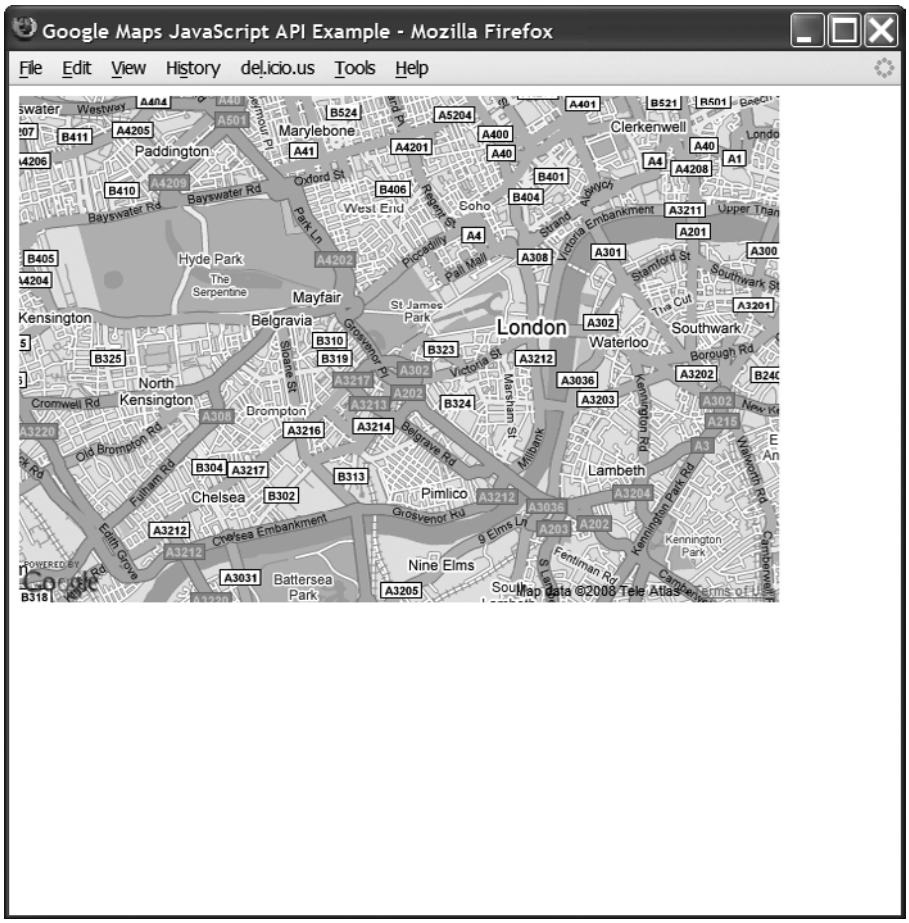

Figure 5-1. *Google Maps showing London*

To get started, we need to include the JavaScript Google Maps API file. This is done using the `<script>` tag on line 6. Within the tag, you must provide your own Google Maps API key. You can get a key (which is free) at `http://maps.google.com`.

Tip If you are developing on a local machine, you can still get a key. For example, you can use `http://127.0.0.1` or `http://localhost`, depending on which local URL you are using.

Once the page has been loaded via the `onload` event on line 16, the `initialize()` function is called.

The `initialize()` method, as defined on line 8, first checks whether we have a compatible browser. If we do, we then go ahead and create an object that will represent a map on the web page. This is done using the `GMap2` class. Additionally, we also need to provide an area where it can display the map; in our case, it's a `<div>` tag with an ID called `map_container` on line 17.

We're not quite finished yet though. In order for a map to be displayed, we need to tell the map object to display a location, and this is done using the `setCenter()` function on line 11, which sets the location and displays the map at the same time. And that's it! Wasn't that easy?

When the browser is closed or when a different page is brought up, the `onunload` event calls `GUnload()`. This Google Map function closes down unwanted connections and is used to avoid memory leaks.

Google Maps Explained

Google Maps is almost exclusively a client-side JavaScript API. Although you can make some server-side calls, that is not encouraged. After all, Google collects better statistics if the calls are made from the browser rather than from the server.

Note To effectively use Google Maps, you need to have some basic knowledge of JavaScript, particularly the object-oriented areas of the language. Two sites to get you started are `http://www.w3schools.com/jS/js_obj_intro.asp` and `http://developer.mozilla.org/en/docs/Introduction_to_Object-Oriented_JavaScript`.

Before we start, let's go over the main features of the API that are relevant to our application.

Geocoding

Geocoding is the process of converting textual locations such as street addresses or place names into geographic coordinates. This is important for a number of reasons. We can locate exact positions on a map, enabling us to place markers accurately. It eliminates confusion where two or more places have the same name.

The Google Maps API includes a class that provides us with a geocoding service. This is the `GClientGeocoder` class.

Google Map Events

Within browsers, JavaScript can be programmed to execute code depending on certain events. In traditional terminology, this is called *event-driven programming*. These events come in many forms: they can be mouse-based, keyboard-based, or even based on other events, such as when a page is completely loaded.

Event-based programming is particularly important in an Ajax environment. Since calls are mostly asynchronous, things don't happen in a nice serial fashion.

■ **Note** Before the use of Ajax, just about every activity on a browser was synchronous. Users had to wait for feedback whenever they generated an event, such as by clicking a link or submitting a form. Essentially, you could make only one call to the server at any one time. With the use of `XMLHttpRequest` in Ajax, we can now make many calls to the server within a single web page, which makes them all asynchronous. Events can happen independently of each other.

Google Maps defines its own specialized events, which are handled by the API itself. These are separate from the DOM events within a browser, which are more generic. For example, a `click` event can be attached to any instance of a map, so when a user clicks a map, a call is made to a function, which opens a window with information regarding the location that the user has clicked and additionally stores the coordinates of the location with a call to the server using Ajax. Map events are handled by the `GEvent` object. We will register events using the `addListener()` static method.

Map Interface Elements

In the Google Maps API, you can add interface control elements, which allow you to interact with the map. Examples include buttons that you can use to move the map around, instead of using the mouse, and a sliding control bar to zoom in and out of the map. In Google Maps terminology, these are known as *controls*. All the control elements subclass the `GControl` class. You can define your own control by subclassing this class. For example, you can create a button that makes an Ajax call to your database server, fetching any journey near that location.

Overlays

Objects that move with the map are called *overlays*. These can be pushpins marking the location of a point or graphical lines that show route directions. The API provides several built-in overlay classes, which are listed in Table 5-1.

Markers are interactive in that they can be dragged across the map and placed in a new location. Each marker has an icon attached to it. You can define your own icon or use the default one.

A large number of markers can slow down the display of the map. As a result, the API has a marker manager class called `MarkerManager`.

Table 5-1. *Map Overlay Objects*

Overlay	Description
GMarker	This class is used to mark a point on the map.
GPolyline	This class is used to lay down lines on the map. There is also a GPolygon class, where the lines form a closed area.
GGroundOverlay	With this class, as well as drawing lines on the map, you can add images on top of them.
GTileLayerOverlay	With this class, the map itself is an overlay. You can modify the map itself with your own version of the map. However, you probably won't be using this very often. The pop-up bubble speech window is a special overlay; only one can exist in any one map instance.

Driving Directions

One useful feature in the Google Maps API is the ability to map out a travel route via several locations. These routes are then marked in blue on the map. This is done via the GDirections class. A very useful feature is the ability of the class to take locations either as textual names or latitude/longitude coordinate points. As an example, Listing 5-2 shows plotting a route from New York to Anchorage. Figure 5-2 shows the route map itself.

Listing 5-2. *Using the Google Maps GDirections Class to Find a Route*

```
 1:    <html>
 2:
 3:    <head>
 4:        <title>Google Maps GDirections Class</title>
 5:
 6:        <script src="http://maps.google.com/maps?file=api&v=2.x&➥
key= insert_your_key_here " type="text/javascript"></script>
 7:
 8:        <script type="text/javascript">
 9:
10:            function initialize() {
11:              map = new GMap2(document.getElementById("map_canvas"));
12:              directions = new GDirections(map);
13:              directions.load("from: New York, USA to: Anchorage");
14:            }
15:
16:        </script>
17:
18:    </head>
19:
20:    <body onload="initialize()">
21:        <div id="map_canvas" style="width: 400px; height: 480px;"></div>
22:    </body>
23:
24:    </html>
```

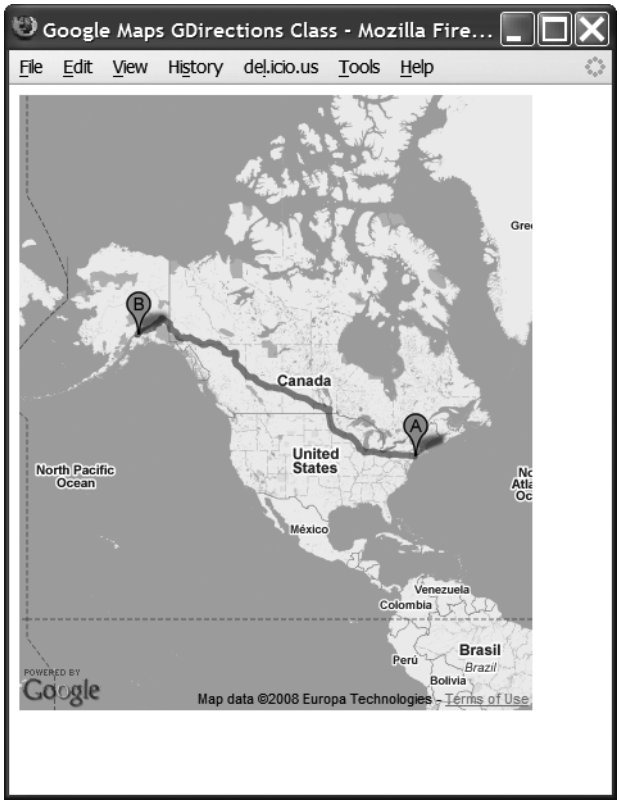

Figure 5-2. *Route from New York to Anchorage*

This `GDirections` class will be an important component of the project. When a query is sent off to the Google servers, it returns a number of useful items of information. One of which is the distance between the locations. This is very important to us, because it will provide the distance via the roads on the map rather than the straight linear distance between locations. However, if the user were planning to fly, the straight line distance would be more appropriate. In that case, we could easily calculate the distance using the latitude/longitude information.

OK, now that we have explained how Google Maps work, we can start our application by gathering the requirements.

Application Requirements

Our travel application will allow our friends and also the general public to plan their travel journeys and make comments on the places they visit. It will be a web site where they can keep friends and families up-to-date on their travels.

So that our application can be as successful as possible, we will employ a bit of *user-centered design* philosophy, a term originally coined by Donald Norman. We will think of the application in terms of users or personas, rather than application functions. Sometimes these needs also correlate directly with business objectives.

Using our friends as our average persona, we map out their needs and the conclusions that we can draw from them. A user interface can then be created based on these conclusions.

We run through the ways in which our friends will use the site. We start the process by asking them some broad questions. These are mapped out as follows:

- *Scenarios/Needs*: We're going to Europe soon, and we want to keep a blog or journal of some kind to keep family and friends updated.
- *Conclusion*: Users want to be able to enter destinations and make comments about those destinations. We need to save the destinations so they can make comments while they are on their journey. We can continue to draw conclusions from this simple need, but that is probably enough for us.
- *Scenarios/Needs*: We're not sure which places to visit first.
- *Conclusion*: There are numerous conclusions we can draw from this statement. For example, we can map out a user journey based on different personas: art lovers, party people, or a combination of personas. Or we could map a user journey based on age groups, such as for students. But in this case, we will simply employ the traveling salesman algorithm to find the shortest route between each destination.

We will base our interface on what we know from this brief user-centered design exercise.

So far, we have covered the broad issues of the application. Next, we will map out a functional specification that we can use for our Cake application. The functions will include the following:

- There will be a consistent navigation bar at the top, since the focus is on the application.
- Users will be able to plan their journey on the first page of the application by entering their destination.
- There will be a page where they can retrieve a saved journey and make comments.
- There will be a page where families and friends can view their comments.

In addition to these functions, we will add a further twist by allowing users to add tags to their journeys. This will be a comma-separated list of strings or tags, which you can associate with a journey.

From the specification, we have sketched out the simple layout of the application in Figure 5-3.

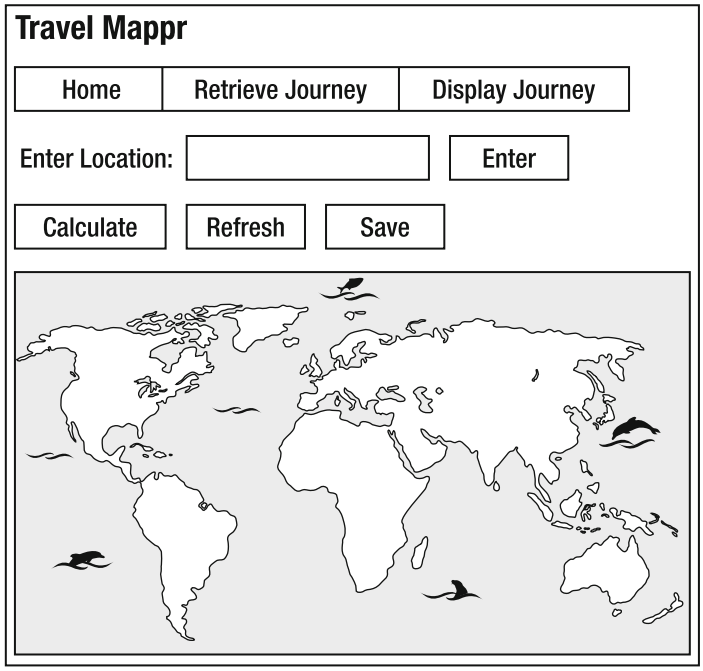

Figure 5-3. *Sketch of our home page*

Application Structure

We will set out to create our journeys controller, which will represent the journeys within the application. It will be responsible for the server-side needs of the application, saving and retrieving user journeys, for example.

Along with the journeys controller, there will be three related models. Remember that the model class names follow the name of the controller. The model class names we will use are `Journey`, `Location`, and `Tag`.

Table 5-2 shows the relationships among the different tables. This will give us an overall picture of how the different data elements are related to each other.

Table 5-2. *Main Database Table Relationships*

Table	Relationship
`journeys`	Each journey is composed of different locations. Also, each journey can have more than one tag.
`locations`	One location belongs to only one journey.
`tags`	One tag belongs to only one journey.

The information in Table 5-2 can also be shown graphically in an entity-relationship diagram, as in Figure 5-4.

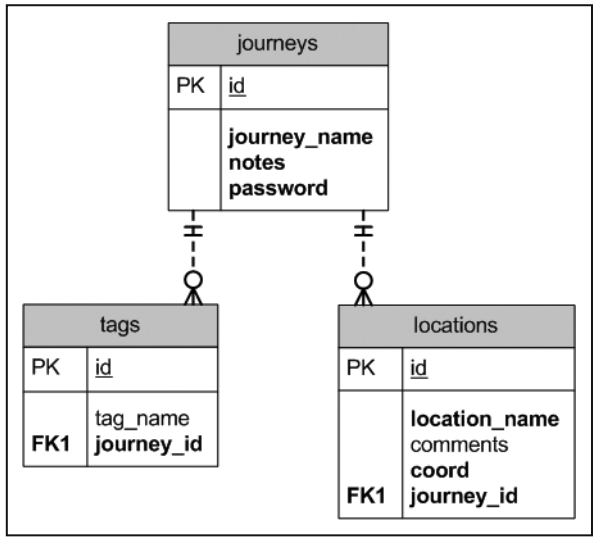

Figure 5-4. *Entity-relationship diagram of application tables*

Tip Data schemas are quite valuable during development. They help you to visualize the relationships among elements, identify problems, and maybe even improve on the relationships. In our example, we see that a route has one tag line associated with it, but later on, we may also want tags associated with locations. But would this add too much complexity to the application? Always be aware of feature bloat—adding unnecessary features that may be used only by a small percentage of users.

From the data schema, we can create our database tables. Listing 5-3 shows the structure of the `journeys` table.

Listing 5-3. *The journeys Table Schema*

```
CREATE TABLE `journeys` (
  `id` int(11) NOT NULL auto_increment,
  `journey_name` varchar(255) NOT NULL,
  `notes` text NOT NULL,
  `password` varchar(255) NOT NULL,
  PRIMARY KEY  (`id`)
);
```

The `journey_name` field stores the journey name that the user enters. The `notes` field is for comments about the journey itself. Editing a journey can be done only when a correct password is used, so we have created a `password` field.

Listing 5-4 shows the structure of the `locations` table.

Listing 5-4. *The locations Table Schema*

```
CREATE TABLE `locations` (
  `id` int(11) NOT NULL auto_increment,
  `location_name` varchar(255) NOT NULL,
  `comments` varchar(255) default NULL,
  `coord` varchar(255) NOT NULL,
  `journey_id` int(11) NOT NULL,
  PRIMARY KEY  (`id`)
);
```

The name of the location that the user enters is held in `location_name`. The `comments` field stores the comments for a particular location. The `coord` field is longitude and latitude information taken from geocoding the location name. Finally, `journey_id` is the foreign key to the `journeys` table.

Listing 5-5 shows the `tags` table structure.

Listing 5-5. *The tags Table Schema*

```
CREATE TABLE `tags` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `tag_name` varchar(100) default NULL,
  `journey_id` varchar(255) NOT NULL,
  PRIMARY KEY  (`id`)
);
```

The `tags` table is pretty simple. The `tag_name` field holds a particular tag name for a journey. The `journey_id` field is the foreign key to the `journeys` table.

Cake Models

From the schema in Listings 5-3, 5-4, and 5-5, we created the Cake models, as shown in Listings 5-6, 5-7, and 5-8.

Listing 5-6. *The Journey Model (/app/models/journey.php)*

```
class Journey extends AppModel {

    var $name = 'Journey';

    // The journey data validation
    var $validate = array( 'journey_name' => array(
                           'rule' => array( 'between', 1, 255 ),
                           'required' => true,
                           'message' => 'Your journey name
                                         must be between 1 and 255
                                         characters long.' ) );
```

```
    // Each journey has many locations and also many tags.
    var $hasMany = array(   'Location' => array( 'className' => 'Location' ),
                            'Tag' => array( 'className' => 'Tag' )
                            );
}
```

Listing 5-7. *The Location Model (/app/models/location.php)*

```
class Location extends AppModel {

    var $name = 'Location';

    var $belongsTo = array('Journey');
}
```

Listing 5-8. *The Tag Model (/app/models/tag.php)*

```
class Tag extends AppModel
{
    var $name = 'Tag';

    var $belongsTo = array('Journey');
}
```

All three models are quite straightforward, except `Journey`, which contains a `hasMany` association that reflects the schema. A simple validation code has also been added, requiring each journey entry into the database to have a `journey_name` within it.

The Interface

Like most modern web applications, a large part of our application is written in JavaScript that sits within the browser. Most of the action will initially occur in the browser itself. The server-side code comes into play when the user decides to save or retrieve the journey information.

The Global Layout

We start off by creating the global layout file in `/app/views/layouts/default.ctp`, as shown in Listing 5-9.

Listing 5-9. *The Global Layout File (/app/views/layouts/default.ctp)*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>

    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>

    <!-- page title -->
    <title><?php echo $title_for_layout; ?></title>

    <!-- page css -->
    <?php echo $html->css( 'site' ); ?>

    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;➥
key= insert_your_key_here" type="text/javascript"></script>

    <?php
        echo $javascript->link( 'scriptaculous-js-1.8.1/lib/prototype' );
        echo $javascript->link( '➥
scriptaculous-js-1.8.1/src/scriptaculous.js?load=effects' );

        // include site.js
        echo $javascript->link( 'site' );
    ?>

</head>

<body onload="initialize()" onunload="GUnload()">

    <div id="center_content">

        <div class="header_wrapper">
            <h1>Travel Mappr</h1>
            <h4><i>Because life is a journey :)</i></h4>
        </div>

        <div class="nav_1">
            <?php echo $html->link( 'Home', '/base' ); ?>
             | 

            <?php echo $html->link( 'Retrieve Journey',
                                    '/journeys/retrieve_form' ); ?>
             | 

```

```
44:                <?php echo $html->link( 'Display Journey',
45:                                        '/journeys/display_journey' ); ?>
46:            </div>
47:
48:            <div id="main_content_container">
49:                <?php echo $content_for_layout; ?>
50:            </div>
51:
52:        </div>
53:
54:    </body>
55:    </html>
```

Note You can create many layouts in the `layouts` folder and change between them in the same view. Just set the `layout` variable in the controller action. For example, if you created a new layout file `/layouts/vanilla.ctp`, you can change the layout from within your action with this statement: `$this->layout = 'vanilla';`.

Let's go through the important lines in Listing 5-9. We start by setting the title in line 10. Next, we include our CSS file for our application in line 13.

On line 15, we bring in the Google Maps API. Remember to use your own key that correlates to your domain name.

Following this, we use Cake's JavaScript inclusion method on lines 18 and 19 to include the Prototype and script.aculo.us JavaScript libraries. We also include our site-wide JavaScript file `site.js`, located in `/app/webroot/js/`.

Within the main `body` tag, the page starts off with a simple header followed by a navigation bar using Cake's HTML helper to create the HTML links, as follows:

- `Home`: This simply takes a user back to the home page.
- `Retrieve Journey`: Allows a user to retrieve a saved trip from the database.
- `Display Journey`: A user can display any journey.

To finish, we echo the `$content_for_layout` variable on line 49. This takes the rendered controller action view content and inserts it into that position.

Home Page

Now that the global layout has been set up, we can create our home page view. We'll use Cake's designated default home page `home.ctp`, in the folder `/app/views/pages/.`, shown in Listing 5-10.

In our application, users must be able to plot out a journey quite easily. On the home page, we will create a form where users can create their journeys the moment they enter the site.

Listing 5-10. *The Default View (/app/views/pages/home.ctp)*

```
<p id="journey_helper_message">
    You currently have no journey planned,
    enter a starting location to begin.
</p>

<div>

    <p>
        Location Name:

        <?php echo $form->text( 'LocationName' ); ?>
        <?php echo $form->button( 'Enter a Location',
                                  array('id'=>'location_name_button'));
        ?>
    </p>

</div>

<div id="map_functions_wrapper">

    <!-- find the best route -->
    <?php
        echo $form->button( 'Calculate Journey',
                            array(  'id'=>'find_route_button' ) );
    ?>

    <!-- start a new route -->
    <?php
        echo $form->button( 'Start Again',
                            array(  'id'=>'start_again_button' ) );
    ?>

    <!-- save the route -->
    <?php
        echo $form->create( null, array(    'id' => 'add_form',
                                            'url' => '/journeys/add_form',
                                            'class' => 'function_form'
                                            ) );

        // used to store the location in json format
        echo $form->hidden( 'locations' );

        echo $form->submit( 'Save Journey', array( 'div' => false ) );

        echo $form->end();
```

```
46:         ?>
47:
48:     </div>
49:
50:     <!-- This is where our Google Map will be displayed -->
51:     <div id="map_canvas"></div>
```

In Listing 5-10, we haven't included any form-posting elements apart from the Save Journey button on line 43. All of the other buttons relate entirely to client-side code and will be used in conjunction with the Google Maps functionality. The first button, Enter a Location, on line 12, will save and display the location in Google Maps locally. The Calculate Journey button, on line 23, will use the Google Maps API and JavaScript to calculate the shortest journey. The Start Again button, on line 29, will clear the slate for a new journey to be entered.

When this home page code is processed with the default layout, we get the output shown in Figure 5-5 (with Paris set as the starting location in this example). The code and functionality behind all these buttons will be explained in the next section.

Figure 5-5. *The application home page*

Travel Mappr Manager

Now we're going to talk about the Travel Mappr manager class in `/app/webroot/js/site.js`. The class will be called `TravelMapprManager`, and it will handle all the client-side functionality relating to the application. `site.js` is called from within the default layout `default.ctp`, as shown on line 21 in Listing 5-9. We use Prototype's API to help us with the creation of the class itself.

■ **Note** You may have noticed that Listing 5-10 doesn't contain any `onclick` events. If you are used to writing `onclick` or `onchange` events within HTML code, consider using Prototype's `Event.observe` method instead. It allows you to separate any JavaScript code from the content.

Listing 5-11 shows the main skeleton structure of our `TravelMapprManager` class in the `site.js` file. We have purposely left out many other functions, as commented on line 75. These will be described individually.

Listing 5-11. *The Main JavaScript File Used in Our Application (/app/webroot/js/site.js)*

```
 1:    var TravelMapprManager = Class.create( {
 2:
 3:        // id of container
 4:        map_container : '',
 5:
 6:        /* the current map */
 7:        map : null,
 8:
 9:        /* geocoding location */
10:        geocoder : null,
11:
12:        /* user entered locations */
13:        user_journey : new Array(),
14:
15:        initialize : function( map_container ) {
16:
17:            this.map_container = map_container;
18:
19:            // start the map
20:            Event.observe( window, 'load', this.displayMap.bind(this) );
21:
22:            // observe the map buttons
23:            Event.observe( document, 'dom:loaded',➥
this.initObservers.bind(this) );
24:
25:        },
26:
27:    displayMap : function() {
28:
```

```
29:             if ( GBrowserIsCompatible() ) {
30:
31:                 if ( $( this.map_container ) ) {
32:
33:                     // create a map instance
34:                     this.map = new GMap2( $( this.map_container ) );
35:
36:                     // add map controls
37:                     this.map.addControl(new GLargeMapControl());
38:                     this.map.addControl(new GMapTypeControl());
39:
40:                     // center the map at a certain location
41:                     this.map.setCenter( new GLatLng( 48.85656, 2.35097 ), 8 );
42:
43:                     // set up a geocoding instance
44:                     this.geocoder = new GClientGeocoder();
45:
46:                     // unload map
47:                     Event.observe( window, 'unload', GUnload );
48:                 }
49:             }
50:         },
51:
52:     initObservers : function() {
53:
54:             if ( $('location_name_button') ) {
55:                 $('location_name_button').observe(
56:                     'click', this.findLocation.bindAsEventListener(this) );
57:             }
58:
59:             if ( $('find_route_button') ) {
60:                 $('find_route_button').observe(
61:                     'click', this.findBestJourney.bindAsEventListener(this) );
62:             }
63:
64:             if ( $('start_again_button') ) {
65:                 $('start_again_button').observe(
66:                     'click', this.startAgain.bindAsEventListener(this) );
67:             }
68:
69:             if ( $('add_form') ) {
70:                 $('add_form').observe( 'submit',
71:                     this.saveJourney.bindAsEventListener(this) );
72:             }
73:         },
74:
75:         // other functions.
76:     )
```

At the beginning of Listing 5-11 are some variables to hold various journey-related values. When `Class.create` is called, the `initialize` method on line 15 is called. This loads and displays the map and also starts observing the map buttons we have created.

Once the DOM has been fully loaded using the `dom:loaded` event on line 23, the buttons get the `click` event attached to them.

The `findLocation()` function will locate the place in Google Maps and display it in the map canvas and also save the details locally in a JavaScript variable.

The `findBestJourney()`function will carry out the algorithm for finding the shortest journey using the shortest neighbor algorithm.

The `startAgain()`function will clear reset variables and allow a user to enter a new journey.

Finally, once a journey has been plotted, the `saveJourney()` function will observe the submit event in the save journey form and post the journey details to the server for saving.

In the following sections, we'll describe each of the missing functions in Listing 5-11, starting with the function that helps us to find a Google map location.

Finding Locations

When a user enters a location name and clicks the Enter a Location button, the `findLocation()` function, shown in Listing 5-12, is fired.

Listing 5-12. *JavaScript to Find a Google Map Location*

```
 1:    findLocation : function() {
 2:
 3:            loc_name = $( 'LocationName' ).value;
 4:
 5:            if ( loc_name == '' ) {
 6:                alert( "Please enter a location name." );
 7:                return;
 8:            }
 9:
10:            // we only allow a maximum number of locations
11:            if ( this.user_journey.length >= 20 ) {
12:                alert( "    Sorry! We have reached the maximum
13:                            number of locations." );
14:                return;
15:            }
16:
17:            // do geocoding; find the longitude and latitude of the location
18:            if ( this.geocoder ) {
19:
20:                var current_o = this;
21:
22:                this.geocoder.getLatLng(
23:                    loc_name,
24:                    function( point ) {
25:
```

```
26:                         if ( !point ) {
27:                             alert( loc_name + " not found" );
28:                         } else {
29:
30:                             // store the location
31:                             current_o.storeLocation( loc_name, point );
32:
33:                             // center the location on the map and
34:                             // add pushpin marker
35:                             current_o.map.setCenter( point, 13 );
36:                             var marker = new GMarker( point );
37:                             current_o.map.addOverlay( marker );
38:                         }
39:                     }
40:                 );
41:             }
42:         },
43:
44:     storeLocation : function( loc_name, point ) {
45:
46:             var new_loc = new Array();
47:
48:             new_loc['coord'] = point.lat()+','+point.lng();
49:             new_loc['loc_name'] = loc_name;
50:
51:             this.user_journey.push( new_loc );
52:
53:             // update the journey message
54:             this.updateJourneyMessage()
55:         },
```

This function starts with a couple of error-handling lines. Then we make the geocoding API call `geocoder.getLatLng` on line 22. If the Google API call finds the location, we store it using our `storeLocation()` function on line 44. We simply hold the location name and coordinate in an array and push it into a global array. Finally, we set the map to show the location and add a pushpin marker, starting from line 35.

Figure 5-6 shows an example of what a user should see after entering a location.

In the next section, we'll look at the algorithm for our traveling salesman/tourist problem. It's the shortest distance between locations. Even so, it's pretty long, so you may want to jump to the next section and come back to the algorithm later.

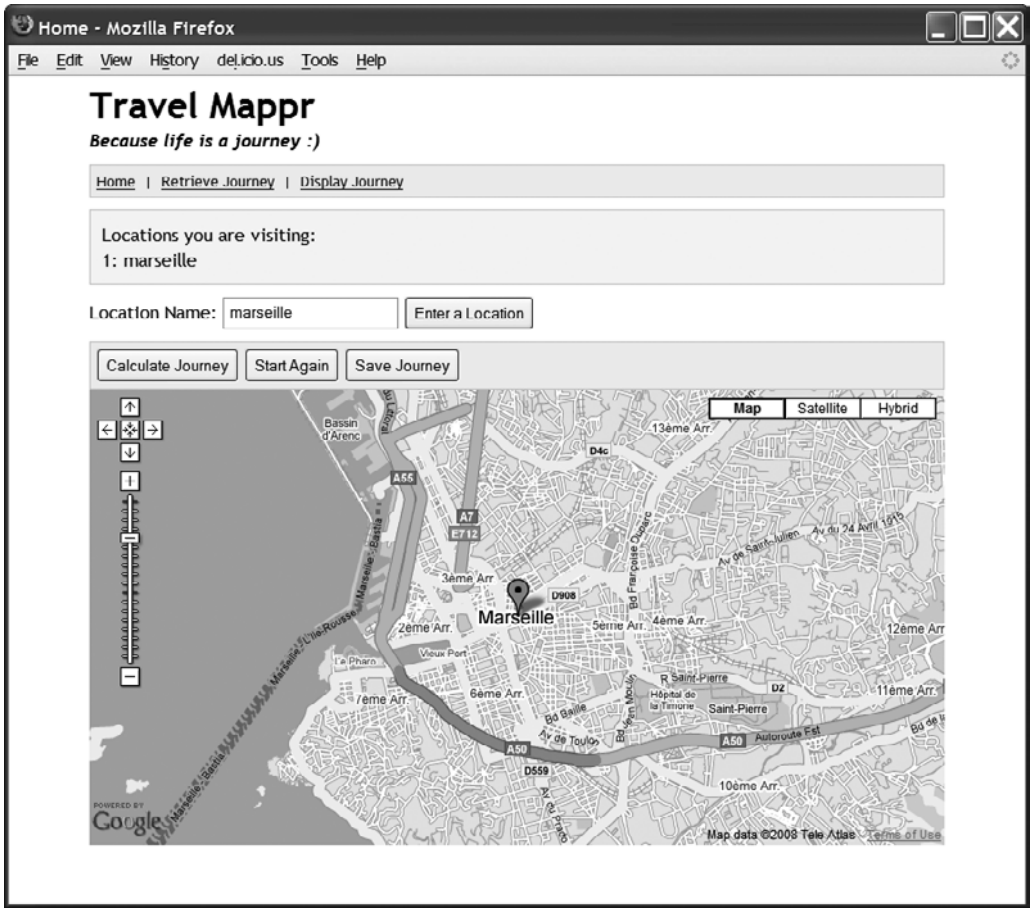

Figure 5-6. *User enters one location*

The Traveling Salesman Algorithm

When a user has entered a set number of locations and clicked the Calculate Journey button, we find the best journey with the "traveling salesman" algorithm. As the algorithm is quite large, we have split the code into three parts. We start the algorithm in Listing 5-13.

Listing 5-13. *The Traveling Salesman Algorithm Part 1*

```
journey_combination : new Array(),

findBestJourney : function() {

```

```
 5:         // we won't calculate a journey if there is no journey to calculate
 6:         if ( this.user_journey.length < 2 ) {
 7:             alert( "Please enter at least 2 locations." );
 8:             return;
 9:         }
10:
11:         // we won't calculate the journey again if it has already been done.
12:         if ( this.best_journey.length == this.user_journey.length ) {
13:             return;
14:         }
15:
16:         // get all the journey combinations
17:         var num_locs = this.user_journey.length;
18:
19:         for ( var x = 0; x < num_locs; x++ ) {
20:
21:             from_here = this.user_journey[x];
22:
23:             for ( var y = x+1; y < num_locs; y++ ) {
24:
25:                 to_here = this.user_journey[y];
26:
27:                 current_journey = new Array();
28:
29:                 current_journey[ "from" ] = from_here;
30:                 current_journey[ "to" ] = to_here;
31:                 current_journey[ "journey_distance" ] = '';
32:
33:                 this.journey_combination.push( current_journey );
34:             }
35:         }
36:
37:         if ( this.getJourneyDistance() ) {
38:             this.calcTSP();
39:         }
40:
41:         return false;
42:     },
```

The variable at the top of Listing 5-13 holds all the journey combinations that the user has entered. The `findBestJourney()` function starts on line 3 with a couple of lines of error checking. Next, we work out all the combinations between the locations on line 17 to line 35. After that, we call `getJourneyDistance()` on line 37, which in turn makes the Google geocoding requests to fetch the distance for each pair of locations in `journey_combination`. Once that is done, we can start calculating the shortest distances between locations.

We continue from the previous listing with Listing 5-14.

Listing 5-14. *The Traveling Salesman Algorithm Part 2*

```
    getJourneyDistance : function() {

        // Start getting the journey distance. Once we have found a journey
        // without any distance, let's just pick that one and fetch the
        // journey distance from Google.
        var do_journey = -1;

        for ( var x = 0; x < this.journey_combination.length; x++ ) {

            if ( this.journey_combination[x][ 'journey_distance' ] == '' ) {
                // no journey distance here, so let's get it
                var do_journey = x;
                break;
            }
        }

        if ( do_journey >= 0 ) {

            // we found a journey to do

            directions = new GDirections();

            GEvent.addListener( directions, "load", function() {

                // Get and fill journey distance. We always just
                // take the first journey
                this.journey_combination[do_journey]➥
[ 'journey_distance' ] = directions.getJourney(0).getDistance().meters;

                // now let's find the next one
                this.getJourneyDistance();
            } );

            direction_journey = 'from: ';
            direction_journey +=➥
this.journey_combination[do_journey]['from']['coord'];
            direction_journey += 'to: ';
            direction_journey +=➥
this.journey_combination[do_journey]['to']['coord'];

            directions.load( direction_journey );
        }

        return true;
    },
```

Notice that the `getJourneyDistance()` function in Listing 5-14 is recursive. The Ajax geocoding call to the Google servers is asynchronous. We create a `GDirections` object on line 21, make the call to fetch the geocoding data, and then quickly carry on with the next location. The `GEvent.addListener` statement will automatically get the distance for us when the reply comes back.

OK, just to recap, we first work out all the combinations from one location to another and store this in the `.journey_combination` array. Next, we make geocoding requests to fetch the distance for all the location pairs in `.journey_combination`. Once all that is complete, we can start working out the shortest round-trip.

Working out the shortest round-trip starts with the `calcTSP()` function in Listing 5-15. Before we start, there are two important variables to mention: `visited` and `best_journey`. From their names, you can probably guess what they hold: locations already visited and the final path that we have worked out.

Listing 5-15. *The Traveling Salesman Algorithm Part 3*

```
    // cities already visited
    visited : new Array(),

    // for holding the best journey
    best_journey : new Array(),

    calcTSP : function() {

        var stopInfin = 0;

        // while there is a next location to visit
        while ( this._visitNextCity() ) {

            // temp var for holding the best (shortest)
            // nearest neighbor so far
            var nearest_neighbour = -1;

            if ( stopInfin > 20 ) { return; }
            stopInfin++;

            start_here = this._getNextNeighbour();

            // get all neighbors that have not been visited from the
            // "start_here" location
            var neighbours = this._getNeighbours( start_here );

            for ( var x = 0; x < neighbours.length; x++ ) {

                if ( nearest_neighbour == -1 ) {
                    nearest_neighbour = neighbours[x];
                }

```

```
                // now find the shortest journey
                if (    neighbours[x][ 'journey_distance' ] <
                        nearest_neighbour[ 'journey_distance' ] ) {
                    nearest_neighbour = neighbours[x];
                }
            }

            // we should now have the next nearest neighbor
            this.best_journey.push( nearest_neighbour['to'] );
            this._markVisited( nearest_neighbour['to'] );
        }

        this._plotBestJourney();
    },

    _visitNextCity : function() {

        if ( this.visited.length == this.user_journey.length ) {
            return false;
        }

        return true;
    },

    _getNextNeighbour : function() {

        var next_city = '';

        if ( this.best_journey.length == 0 ) {

            // init. We take the first journey as the starting point
            start_here = this.journey_combination[0]['from'];
            this.best_journey.push( start_here );
            this._markVisited( start_here );

            next_city = start_here;
        }
        else {

            // we pick the last city
            var last_loc = this.best_journey.length-1;
            next_city = this.best_journey[last_loc];
        }

        return next_city;
    },

```

```
 80:    _markVisited : function( loc ) {
 81:
 82:        this.visited.push( loc['loc_name'] );
 83:    },
 84:
 85:    _locVisited : function( loc_name ) {
 86:
 87:        for ( var x = 0; x < this.visited.length; x++ ) {
 88:
 89:            if ( loc_name == this.visited[x] ) {
 90:                return true;
 91:            }
 92:        }
 93:
 94:        return false;
 95:    },
 96:
 97:    /*
 98:    * Get all neighbors not visited
 99:    */
100:    _getNeighbours: function( from_loc ) {
101:
102:        var result = new Array();
103:
104:        for ( var x = 0; x < this.journey_combination.length; x++ ) {
105:
106:            var next_loc = new Array();
107:            next_loc[ 'from' ] = from_loc;
108:            next_loc[ 'to' ] = (➥
this.journey_combination[x]['from']['loc_name'] ==➥
from_loc['loc_name'] ) ? this.journey_combination[x]['to'] :➥
this.journey_combination[x]['from'];
109:            next_loc[ 'journey_distance' ] =➥
this.journey_combination[x]['journey_distance'];
110:
111:            // check whether the location has been visited already
112:            if ( !this._locVisited( next_loc[ 'to' ][ 'loc_name' ] ) ) {
113:                result.push( next_loc );
114:            }
115:        }
116:
117:        return result;
118:    },
```

Our `calcTSP` function starts with an outer loop on line 12. If there is a next location to visit, we carry on with the algorithm. Within the loop, we first pick a starting location using `_getNextNeighbour` on line 21. Next, we work out the nearest neighbor to that point using the `_getNeighbours()` function on line 25. This gets all the neighbors that haven't already been

visited. From this, we use a simple bubble sort to work out the shortest distance to the next location. When we come around the loop the next time, we pick the last city we came from as the next starting point and start over again. Once we have visited all the locations, we plot the journey path on the map via the `_plotBestJourney()` function on line 45, which we will cover in the next section.

There are many functions that support `calcTSP`. These functions are described in Table 5-3.

Table 5-3. *Supporting Functions to calcTSP*

Function	Description
`_visitNextCity`	On line 48 in Listing 5-15. This function tells us whether there are any more locations to visit. Once there are no more locations to visit, we have found our shortest route.
`_getNextNeighbour`	On line 57 in Listing 5-15. This function provides us with the next location to visit, which is always the destination location of the last journey.
`_markVisited`	On line 80 in Listing 5-15. Once a location has been visited, this function marks that location, so that we do not visit the same location.
`_locVisited`	On line 85 in Listing 5-15. This tells us whether we have visited a location.
`_getNeighbours`	On line 100 in Listing 5-15. This gets all the neighbors that we have not visited. Using the result returned, we can then find the neighbor with the shortest route to the location.

Plotting the Journey

Using the Google Maps `GDirections` class, we can plot the path between the locations. Using the `loadFromWaypoints` method in the `GDirections` class, a blue path is marked on the map itself. The code is shown in Listing 5-16.

Listing 5-16. *Plotting the Journey*

```
    _plotBestJourney : function() {

        var new_journey = new Array();

        for ( var x = 0; x < this.best_journey.length; x++ ) {

            direction_journey = this.best_journey[x]['coord'];
            new_journey[x] = direction_journey;
        }

        // add the starting point back as starting position
        new_journey.push( this.best_journey[0]['coord'] );

        directions = new GDirections( this.map );

        var current_o = this;

```

```
18:         // remove the default red markers
19:         GEvent.addListener( directions, "load", function() {
20:             current_o.map.clearOverlays();
21:         } );
22:
23:         // remove the last marker, so the first one would show up
24:         GEvent.addListener( directions, "addoverlay", function() {
25:             var num_markers = directions.getNumGeocodes();
26:             current_o.map.removeOverlay(➥
directions.getMarker(num_markers-1) );
27:         } );
28:
29:         directions.loadFromWaypoints( new_journey );
30:     },
```

Most of this code is devoted to organizing the locations in a format that will be suitable for use for the `loadFromWaypoints()` function. We first get the shortest route from the `best_journey` variable by looping through it on line 5. Then, on line 12, we trace a route from the end point back to the start so we have a loop. On line 14, we create a `GDirections` object for the plotting of the route. Next, starting from line 19, we need to carry out some housekeeping functions. Google automatically creates pushpin markers when we plot a location. We need to clear these, because the `GDirections` object creates additional markers that show the numeric order of the locations.

It is worth noting that for a lot of countries, the path and distance use the roads as the mode of travel, which is the route that we would prefer. However, in countries where Google doesn't have any road information, the straight path between the locations is used.

And that is just about it. We finish `/app/webroot/js/site.js` with the class-creation code shown here:

```
new TravelMapprManager( 'map_canvas' );
```

We're not quite finished yet on the client side, as there are two more buttons: Start Again and Save Journey. Start Again is pretty straightforward, as it just clears the variables and refreshes the map. Save Journey is a little more interesting. As shown in Listing 5-17, when `saveJourney` is called, we first check whether there is a journey to save. If there is, we basically create a JSON string format from the calculated route and insert it as a value in the hidden tag element with `id locations`, after which the form is submitted to the server. Using the journey we plotted earlier, Listing 5-18 shows the data format of the locations as a JSON string.

■ **Note** What is a JSON string format? JSON stands for JavaScript Object Notation. It is a data format much like XML or even CSV. The popularity of its use came as a result of Ajax, since the format is native to JavaScript. It is also less verbose than XML.

Listing 5-17. *The saveJourney Function*

```
checkJourneyExists : function() {

    if ( this.best_journey.length == 0 ) {
        alert( 'No journey to save, create a journey first,\nor➥
maybe the journey has not been calculated yet.' );
        return false;
    }

    return true;
},

saveJourney : function(e) {

    if ( this.checkJourneyExists() ) {

        // create the json notation
        xml_loc = '{';

        for ( var x = 0; x < this.best_journey.length; x++ ) {
            xml_loc += '"' + x + '":';
            xml_loc += '{'

            xml_loc += '"id":';
            xml_loc += '"';

            xml_loc += ',';

            xml_loc += '"comments":';
            xml_loc += '"';

            xml_loc += ',';

            xml_loc += '"coord":';
            xml_loc += '"' + this.best_journey[x]['coord'] + '"';

            xml_loc += ',';

            xml_loc += '"loc_name":';
            xml_loc += '"' + this.best_journey[x]['loc_name'] + '"';

            xml_loc += '}'

            if ( x+1 < this.best_journey.length ) { xml_loc += ','; }
        }
```

```
        xml_loc += '}';

        $('locations').value = xml_loc;

        return true;
    }
    else {

        Event.stop(e);

        return false;
    }
}
```

Listing 5-18. *Our Journey Locations in a JSON String*

```
{

"0":{   "id":"",
        "comments":"",
        "coord":"43.298344,5.383221",
        "location_name":"Marseille" },

"1":{   "id":"",
        "comments":"",
        "coord":"48.856667,2.350987",
        "location_name":"Paris" },

"2":{   "id":"",
        "comments":"",
        "coord":"40.416741,-3.70325",
        "location_name":"Madrid"}

}
```

We can now plot a journey and see how it looks on screen. Let's assume our friends are going to visit three locations: Marseille in France, Madrid in Spain, and Paris in France again. After entering the three locations, our map application will look like the one shown in Figure 5-7.

Now that we have completed the client side of the application, we'll jump across to the server side and look at how a journey is saved into the database.

Figure 5-7. *A journey with three locations*

Journey Data

For our application, we need to save the journey details and also the tags associated with a journey. We also need to give users the ability to retrieve journeys for viewing and editing. Without further ado, we'll show you how these functions are done.

Saving a Journey

All our actions are contained in the one and only `JourneysController` in the application. Within that controller, we save a journey using the `add_form()` action, as shown in Listing 5-19. This is the target action for the Save Journey button on the client side. It parses the journey name, tags, and comments and also the journey details including the destination comments. The detail of the destinations arrives in the JSON format. We could have chosen XML or even some comma/semicolon type proprietary format. However, the JSON format seems to be the

de facto format nowadays for web client/server data exchange. We set two variables to be used by the view: `locations` and `journey_id`.

Listing 5-19. *Saving Journeys*

```
function add_form() {

    // get the locations from the hidden form element
    $locations = json_decode( $this->data['locations'], true );
    $journey_id = null;
    $this->set(compact('locations','journey_id'));


}
```

Now let's look at the view that goes with the `add_form()` action. The `journey_id` is used to decide whether we are saving a new journey or editing an existing one. The same form is used to add or edit a journey. The view for `add_form()` action is shown in Listing 5-20. It is stored in `/app/views/journeys/add_form.ctp`. The output of the view is shown in Figure 5-8.

Note We have noticed that as applications get more complex, they often branch out into separate add or edit action/view pairs. Be careful, as the code in the controller and view can get out of hand. Use components in the controller and view helpers or elements in the view where you can spot common code. In fact, we were pretty close to having two views, as we weren't sure whether to have the password field when editing a journey.

Listing 5-20. *The View for the add_form Action*

```
 1:    <?php
 2:       if ( !empty( $user_message ) )
 3:          echo $user_message;
 4:    ?>
 5:
 6:    <p>Fields marked with * are needed.</p>
 7:
 8:    <div>
 9:
10:    <?php
11:
12:        // Decide whether we're adding a journey or editing
13:        if ( $journey_id ) {
14:            echo $form->create( 'Journey',
15:                                array(➥
'url' => '/journeys/edit/'.$journey_id ) );
16:        }
```

```
        else {
            echo $form->create( 'Journey',
                                array( 'url' => '/journeys/add' ) );
        }

        echo '<div class="form_fields">';
        echo '<h5>Journey Name *:</h5>';
        echo $form->input( 'name', array(➥
'div' => false, 'label' => false ) );
        echo '</div>';

        echo '<div class="form_fields">';
        echo '<h5>Password:</h5>';
        echo $form->password( 'password', array(➥
'div' => false, 'label' => false ) );
        echo '  <p>Used to retrieve your journey. If you do not enter your own
                password, one will be generated for you.</p></div>';

        echo '<div class="form_fields">';
        echo '<h5>Tags:</h5>';
        echo $form->input( 'tags', array(➥
'div' => false, 'label' => false ) );
        echo '<p>Please separate tags with commas.</p></div>';

        echo '<div class="form_fields">';
        echo '<h5>Notes:</h5>';
        echo $form->textarea(   'notes',
                                array(  'div' => false, 'label' => false,
                                        'rows' => '7', 'cols' => '60'
                                        ) );
        echo '</div>';

        // locations
        echo $map->generateFields( $locations );

        echo '<div class="form_fields">';
        echo $form->submit( 'Save Journey' );
        echo '</div>';

        echo $form->end();
    ?>

    </div>
```

Figure 5-8. *The add_form view*

In the `add_form()` action, we decode the JSON locations details. These are then passed onto the view for display. In the view, we manage the layout of the elements ourselves by setting the `div` and `label` parameters to `false`. Each journey needs to have a password so a user can come back and edit a journey. In practice, you would probably create a user account and attach journeys to a particular user. However, to keep the application simple, we have attached a password to each journey instead.

We have created a view helper named `MapHelper`, which is stored in `/app/views/helpers/map.php`. In the `MapHelper` class, shown in Listing 5-21, we have a `generateFields()` method on line 5, which is used to help display the comment `textarea` tags that go with each location. Within the helper, we can use other helpers by including them in the `$helpers` array.

Listing 5-21. *The View Helper Class MapHelper (/app/views/helpers/map.php)*

```
 1:    class MapHelper extends AppHelper {
 2:
 3:        var $helpers = array( 'Form' );
 4:
 5:        function generateFields( $locations ) {
 6:
 7:            $result = '';
 8:
 9:            for( $idx=0; $idx<sizeof( $locations ); $idx++ ) {
10:
11:                $id = $locations[$idx][ 'id' ];
12:                $coord = $locations[$idx]['coord'];
13:                $loc_name = $locations[$idx]['loc_name'];
14:                $comments = $locations[$idx]['comments'];
15:
16:                $json = json_encode( $locations[$idx] );
17:
18:                $result .= '<div class="form_fields">';
19:                $result .= '<h5>Location: '.$loc_name.'</h5>';
20:                $result .= $this->Form->textarea(   'notes',
21:                                array(  'value' => $comments,
22:                                        'name' =>➥
23:    "data[Journey][locations]["."$idx."][comments]",
24:                                        'div' => false, 'label' => false,
25:                                        'rows' => '5', 'cols' => '50'
26:                                        ) );
27:
28:                $result .= '    <input type="hidden"
29:                                name="data[Journey][locations]['.$idx.'][data]"
30:                                value=\''.$json.'\' />';
31:                $result .= '</div>';
32:            }
33:
34:            return $this->output($result);
35:        }
36:    }
```

In Listing 5-20, the form is posted to the `add()` action on line 19. This is the action in the `JourneysController` that saves the journey when a user has entered the journey details. The `add()` action is shown in Listing 5-22.

Listing 5-22. *Saving the Journey Data*

```
function add() {

    // If no data is supplied, we just render the journey form
    if ( empty( $this->data ) ) {
        $this->redirect( '/', null, true );
    }
    else {

        // Whether the save was successful
        $save_result = 1;

        if ( $save_result ) {

            // Check password
            $password = $this->data[ 'Journey' ][ 'password' ];
            if ( empty( $this->data[ 'Journey' ][ 'password' ] ) ) {
                $password = rand( 1, 1000 );
            }

            // Save journey
            $journey = array();
            $journey[ 'name' ] = $this->data[ 'Journey' ][ 'name' ];
            $journey[ 'notes' ] = $this->data[ 'Journey' ][ 'notes' ];
            $journey[ 'password' ] = md5( $password );
            $save_journey_result = $this->Journey->save( $journey );

            // Journey didn't save properly
            if ( !$save_journey_result ) {
                $save_result = 0;
            }
        }

        if ( $save_result ) {

            // Save locations
            $save_loc_result = $this->_save_locations(
                                $this->data[ 'Journey' ][ 'locations' ],
                                $this->Journey->id
                                  );

            // Locations didn't save properly
            if ( !$save_loc_result ) {
                $save_result = 0;
            }
        }

```

```
47:             if ( $save_result ) {
48:
49:                 // Save tags
50:                 $save_tag_result = $this->_save_tags(
51:                                         $this->data[ 'Journey' ][ 'tags' ],
52:                                         $this->Journey->id
53:                                            );
54:
55:                 // Tags didn't save properly
56:                 if ( !$save_tag_result ) {
57:                     $save_result = 0;
58:                 }
59:             }
60:
61:             if ( $save_result ) {
62:
63:                 // Now render the success message view
64:                 $this->set( 'journey_id', $this->Journey->id );
65:
66:                 $this->set( 'password', $password );
67:
68:                 $this->render( 'add_success' );
69:             }
70:             else {
71:
72:                 $this->set( 'user_message',
73:                       'Please correct the form errors as shown below.' );
74:
75:                 // We have to reform the location data
76:                 $locations_for_form = $this->_reformat_locations(
77:                                       $this->data['Journey']['locations']
78:                                             );
79:                 $this->set( 'locations', $locations_for_form );
80:
81:                 $this->set( 'journey_id', '' );
82:
83:                 $this->render( 'add_form' );
84:             }
85:         }
86:     }
```

On line 5, if no data is supplied, we simply redirect the user back to the home page, as we regard that as an error. If data is supplied, we save the three data sets into the tables `journeys`, `locations`, and `tags`. Note that in a production environment, these three units of code should be within a transaction block. If one fails, the previous save actions should be rolled back.

If everything has been saved OK, we display the `add_success` view on line 68. If not, we render the form again with error messages.

We have used two protected actions/functions to save the locations and tags: `_save_locations` and `_save_tags`. The underscore prefix tells Cake that they are protected actions and should not be executed via any URL request.

Saving Tags

The tag data that goes with a journey is a set of comma-separated strings. These strings are split up and entered into the `tags` table separately. The tag-saving protected action is shown in Listing 5-23.

Note We could have placed the tags within a single database field in the `journeys` table and called it `tags`. However, this would not conform to the first normal form in database normalization, where each field must contain single, not multiple, values. We would run into trouble when we wanted to query the tags within a route.

Listing 5-23. *Saving the Journey Tags*

```
 1:    function _save_tags( $tags, $journey_id ) {
 2:
 3:        $tags_a = explode( ",", $tags );
 4:
 5:        for( $idx=0; $idx<sizeof( $tags_a ); $idx++ ) {
 6:
 7:            $db_tag = array();
 8:            $db_tag[ 'tag' ] = trim( $tags_a[$idx] );
 9:            $db_tag[ 'journey_id' ] = $journey_id;
10:
11:            // We need to create a new tag before saving another
12:            $this->Tag->create( $db_tag );
13:            $save_result = $this->Tag->save( $db_tag );
14:
15:            if ( !$save_result ) {
16:                return false;
17:            }
18:        }
19:
20:        return true;
21:    }
```

In Listing 5-23, we simply use the PHP `explode()` function to split the tags on line 3 and then loop through the strings and save the tags individually. When we are editing a journey, the saving or updating of the tags is done slightly differently, in that we first delete all the tags relating to the journey and then we save the updated tags as if they were new. We find this technique to be simpler to maintain and easier to read and reuse than the alternative method, where we update any tags that have been changed and delete any tags that have been removed.

Retrieving and Editing a Journey

The code to retrieve a journey is pretty simple. The action is shown in Listing 5-24. Yes, it's empty, because it's just a simple form with no other dependencies. The main view elements are shown in Listing 5-25. We have cut some of the HTML markup to simplify the view.

Listing 5-24. *The Controller Action to Retrieve a Journey*

```
function retrieve_form() {
}
```

Listing 5-25. *The View to Retrieve a Journey*

```
echo $form->create( 'Journey', array( 'url' => '/journeys/retrieve' ) );
echo $form->input( 'journey_id', array( 'div' => false, 'label' => false ) );
echo $form->input( 'password', array( 'div' => false, 'label' => false ) );
echo $form->submit( 'Retrieve Journey' );
```

The retrieve form simply renders a traditional HTML form with two input elements and a submit button.

The target of the form is the `retrieve()` action. The outline of the code is shown in Listing 5-26. It's very similar to the `add` action. If no data is supplied, we simply redirect the user back to the retrieve form. If data is supplied, we use Cake's `find()` model method to find the journey based on the journey ID and encrypted MD5 password. If a journey is found, we'll use the `add_form` action to display the results. If not, we will render the retrieve form with an error.

Listing 5-26. *Retrieving a Journey*

```
 1:    function retrieve() {
 2:        // If no data is supplied, we redirect user back to the retrieve form
 3:        if ( empty( $this->data ) ) {
 4:             $this->redirect( '/journeys/retrieve_form',null,true);
 5:
 6:        }
 7:        else
 8:        {
 9:            $journey_id = $this->data[ 'Journey' ][ 'journey_id' ];
10:            $password = $this->data[ 'Journey' ][ 'password' ];
11:            $journey = $this->Journey->find(
12:                                  array( 'id' => $journey_id,
13:                                         'password' => md5( $password ) ) );
14:
15:            if ( $journey ) {
16:
17:                // Name of the journey
18:                $this->data['Journey']['name'] = $journey['Journey']['name'];
19:
```

```
                // Tags relating to the journey
                $this->data['Journey']['tags'] =➥
$this->_implode_tag( $journey['Tag'] );

                // Notes of the journey
                $this->data['Journey']['notes'] =➥
$journey['Journey']['notes'];

                // Locations of the journey
                $this->set( 'locations', $journey['Location'] );

                // Journey ID
                $this->set( 'journey_id', $journey_id );

                $this->render( 'add_form' );
            }
            else
            {
                $this->set( 'user_message',
                            '   Sorry, we couldn\'t
                                find your journey, or your password is
                                incorrect!' );

                $this->render( 'retrieve_form' );
            }
        }
    }
```

Viewing a Journey

Viewing a journey is similar to retrieving a journey, except we can only view the journey details (and not edit them). The action is shown in Listing 5-27.

Listing 5-27. *The Action to Display a Journey*

```
    function display_journey( $get_journey_id = '' ) {

        $journey_id = $get_journey_id;

        if ( $this->data[ 'Journey' ][ 'journey_id' ] ) {
            $journey_id = $this->data[ 'Journey' ][ 'journey_id' ];
        }

        // If no data is supplied, we redirect user back to the retrieve form
        if ( $journey_id ) {

            $journey = $this->Journey->findById( $journey_id );

```

```
            if ( $journey ) {
                $this->set( 'journey', $journey );
            }
            else {
                $this->set( 'user_message',
                            '   <div class="error-message">Sorry, we
                                couldn\'t find the journey!</div>' );
            }
        }
    }
```

The action handles both POST and GET scenarios. In the GET action, journeys can be displayed where the journey ID is within the URL request, as in /journeys/display_journey/1. In a POST action, the journey ID is stored within the $this->data variable.

In this action, we have used Cake's findBy<fieldName>(string $value) methods to retrieve the journey. The view is pretty straightforward and is shown in Listing 5-28. In this case, we have combined the display form with the view inserted below it. An example of the output is shown in Figure 5-9.

Listing 5-28. *The Journey View (/app/views/journeys/display_journey.ctp)*

```
<?php

   if ( !empty( $user_message ) )
      echo $user_message;
?>

<p>Fields marked with * are needed.</p>

<div>

<?php

    echo $form->create( 'Journey',
                        array( 'url' => '/journeys/display_journey' ) );

    echo '<div class="form_fields">';
    echo '<h5>Journey ID: *<h5>';
    echo $form->input( 'journey_id', array( 'div' => false, 'label' => false ) );
    echo '</div>';

    echo '<div class="form_fields">';
    echo $form->submit( 'Display Journey' );
    echo '</div>';

    echo $form->end();
?>

</div>
```

```
<!-- display journey here -->
<?php

    if ( isset( $journey ) ) {

        echo '<div class="display_journey_container">';
        echo '<h3>Your Journey Details</h3>';

        // journey name
        echo '<div class="form_fields">';
        echo '<h4>Journey Name:</h4>';
        echo $journey[ 'Journey' ][ 'name' ];
        echo '</div>';

        // journey notes
        echo '<div class="form_fields">';
        echo '<h4>Journey Notes:</h4>';
        echo str_replace( chr(10), '<br />', $journey[ 'Journey' ][ 'notes' ] );
        echo '</div>';

        // tag
        $tag = $journey[ 'Tag' ];
        $tag_str = '';
        for( $idx=0; $idx<sizeof( $tag ); $idx++ ) {

            $tag_str .= $tag[$idx][ 'tag' ];
            if ( $idx+1 < sizeof( $tag ) ) { $tag_str .= ','; }
        }

        echo '<div class="form_fields">';
        echo '<h4>Journey Tags:</h4>';
        echo $tag_str;
        echo '</div>';

        // locations
        $locations = $journey[ 'Location' ];

        for( $idx=0; $idx<sizeof( $locations ); $idx++ ) {

            echo '<div class="form_fields">';
            echo '<h3>Location: '.$locations[$idx][ 'loc_name' ].'</h3>';
            echo $locations[$idx][ 'comments' ];
            echo '</div>';
        }

        echo '</div>';
    }
?>
```

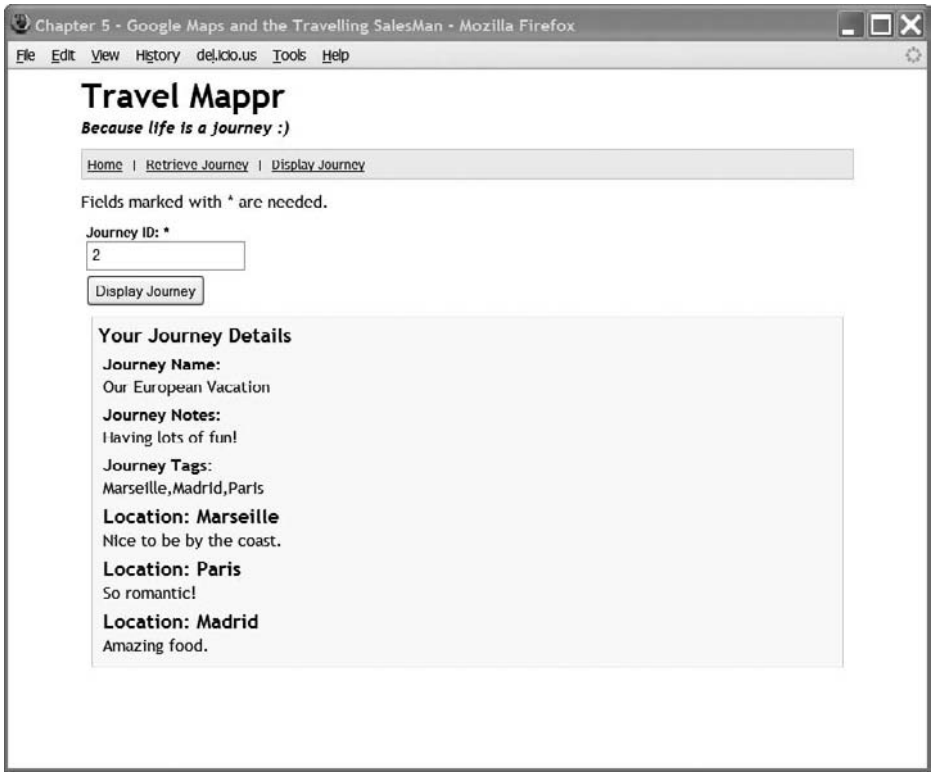

Figure 5-9. *Viewing a journey*

Summary

In this chapter, we have put together a simple travel log application. On the front end, we have used Google Maps to plot and work out a journey plan using purely JavaScript. On the server, we've created a journeys controller to handle saving and retrieving data. In the Cake models, we've used `hasMany` and `belongsTo` associations, as well as some simple validation. It's worth noting that some actions have more than one view.

Friends and family members can easily see what the travelers are up to by retrieving a journey, either via a URL or by using the display journey form. As a result of the feedback we got from our traveling friends, we have more tweaks and changes to implement, which we'll leave to you as an exercise. These are their comments:

- We would like to add a tag cloud so we can see what people are doing in general.
- We want to be able to just click the map instead of entering place names.
- We want to be able to enter comments for one destination only.
- We want to allow other people to enter comments. Also, each comment should be a separate, new entry.
- We want to see what other people are doing in similar destinations—something like a public gallery.

CHAPTER 6

Mashing Twitter with the Google Translator

For our fast-paced modern lifestyle, Twitter has filled a gap that fits between text messaging and blogging. You have a spare minute and want to let your pals know what you are doing or thinking, Twitter fits that need nicely. Twitter is a social networking and micro-blogging web application. You can post short messages to tell your friends and everyone else what you're up to right now. These short messages are referred to as *statuses*, *updates*, or *tweets*.

We can see Twitter being quite an addiction: Twitterdiction! In fact, we often view the Twitter public timeline, which is a listing of what people around the world are doing at that moment. However, quite a number of messages are in a foreign language. If a message has a cute picture, we cut and paste it into the Google Translate web site (`http://translate.google.com`) to see what it says. Wouldn't it be nice if we could do this translation automatically? Here's a stroke of luck: Google now has a language translation web service within the Google Ajax Language API. And in true Web 2.0 and Cake fashion, we can easily mash them together and bake some Cake!

In this chapter, we're going to cover quite a number of Cake topics. Twitter comes with an API that we can call from the server. The Google Ajax Language API can be called from the browser via Ajax or the server, but in this chapter, we'll be using only the server-side method.

So, let's get to creating our application, which we'll call Twitter Twister.

The Twitter API

The Twitter API comes with many methods. These methods are submitted via the principles of representational state transfer (REST). In most cases, this will be HTTP `POST` or `GET`—what most people have been using since their early days of web development. Twitter will also return the appropriate HTTP status code.

There are a few more items to be aware of when using the API:

- Twitter limits the number of requests to 70 requests per 60 minutes. Essentially, if you make one request per minute, you'll be OK.
- As users can post in many languages, the returned data from Twitter is encoded in UTF-8. As such, we must develop our application with this in mind.
- When Twitter returns an error, it will do so in the format you requested.

Within the Twitter API, we'll be using only one of the methods: the `public_timeline`. This returns the 20 most recent statuses from the server. This is the only method in the API that doesn't require authentication.

Calling the public timeline is easy. Using Cake's `HttpSocket`, you can fetch the most recent 20 statuses as follows:

```
App::import( 'HttpSocket' );

$http = new HttpSocket();

$request = array( 'uri' => ➥
'http://www.twitter.com/statuses/public_timeline.xml' );

$body = $http->request($request);
```

As you can see, requesting the public timeline is a simple case of supplying the `request` method with the URL. The `$body` variable will contain details of the 20 statuses in XML format, which we can easily handle in Cake or PHP. Listing 6-1 shows an example.

Listing 6-1. *Twitter XML Status Example*

```
 1:    <status>
 2:        <created_at>Thu Jul 10 21:31:17 +0000 2008</created_at>
 3:        <id>122039345</id>
 4:        <text>This is my first twitter!</text>
 5:        <source><a href="http://www.thisismyhomepage.com/">➥
my home page</a></source>
 6:        <truncated>false</truncated>
 7:        <in_reply_to_status_id>0123034190</in_reply_to_status_id>
 8:        <in_reply_to_user_id>123456</in_reply_to_user_id>
 9:        <favorited />
10:        <user>
11:            <id>9876543</id>
12:            <name>Me Myself</name>
13:            <screen_name>JustMe</screen_name>
14:            <location>London</location>
15:            <description>Take the Red Pill</description>
16:            <profile_image_url>http://s3.amazonaws.com/twitter_production/➥
17:    profile_images/9876543/mugshot.jpg</profile_image_url>
18:            <url>http://www.thisismyhomepage.com/</url>
19:            <protected>false</protected>
20:            <followers_count>111</followers_count>
21:        </user>
22:    </status>
```

On line 4, note the `text` tag, which is the tag that contains the Twitter status. Additionally, the tags `text`, `screen_name`, and `description` (on lines 4, 13, and 15, respectively) are the three XML tags that may contain foreign/multibyte characters. However, in our application, to keep things simple, we're just dealing with the `text` tag.

Tip Some of the normal string manipulation functions do not always work on multibyte characters. For example, `strlen` may return different length sizes depending on whether the string is single or multibyte. Cake has a useful class called `Multibyte`, which contains some string-manipulation functions specifically written for multibyte characters, similar to PHP's `mbstring` module.

Now that we have covered the basics of the Twitter API, we can show you how the Google Ajax Language API works.

The Google Ajax Language API

The Google Ajax Language API is part of the Google Ajax API set of products. We'll call it the Google Translator for short, as that's its main feature.

The API allows you to translate or detect language text. The API commands can be called via Ajax, through other non-JavaScript environments like Flash, or on the server. For further details on using the Ajax calls, visit the Google Language API web site at (`http://code.google.com/apis/ajaxlanguage/`).

Caution It must be said that the quality of the Google Ajax Language API translation can be quite poor. However, the service will serve our purposes, since we just want to get an idea of what's in the Twitter statuses.

To use the commands on the server side, again, we can employ Cake's `HttpSocket` class. For example, to translate "hello world," we use the code in Listing 6-2.

Listing 6-2. *Translating "Hello World"*

```
App::import( 'HttpSocket' );

$http = new HttpSocket();

$request = array(
    'uri' => 'http://ajax.googleapis.com/ajax/services/language/translate?➥
v=1.0&q=hello%20world&langpair=en%7Cja',
    'header' => array(
    'Referer' => 'http://'.env('SERVER_NAME')
    )
);

$body = $http->request($request);
```

In Listing 6-2, we use the `translate` command and pass three parameters to it:

- `v`, for version `1.0` of the API
- `q`, for the `hello%20world` text we want to translate
- `langpair`, for the source language English (`en`) and target language Japanese (`ja`), separated by an entity-encoded vertical bar symbol: `%7C`

All the API commands respond in the JSON format, which we can easily decode using the PHP command `json_decode`.

Google's API documentation advises developers to make sure an HTTP referer header is always in place when making requests. This is shown in line 8 of Listing 6-2. Additionally, an API key should be sent. You can obtain a key from either of the following signup pages:

- Google Ajax Search (`http://code.google.com/apis/ajaxsearch/signup.html`)
- Google Maps (`http://code.google.com/apis/maps/signup.html`)

One useful feature is the language detection. If you do not know the original language that you want to translate, you can omit the source language in the `langpair` parameter. So, for example, we could simply write `langpair=%7Cja`. Now how cool is that!

■ **Note** At the time of writing, the Google Ajax Language API is still quite new, and there don't appear to be any request limits. However, we can all be quite certain this will change at some point in the future.

Application Requirements

Even though we are essentially developing this application for fun, we still need to establish some requirements and consider usability.

The broad requirement is to give users the ability to view the current public timeline in any language by using the Google Translator to translate the status text. This simple statement gives us a starting point for scoping the application.

Scoping an application can be a lengthy task, involving many aspects, such as content, function, layout, wireframes, and site maps—just to mention a few. Since our application is quite small and should be fun, we will concentrate on the following simple functional requirements:

- Twitter has a usage limit. Considering the number of people who use Twitter, we definitely need to cache results. We will take advantage of Cake's cache helper to assist us.
- Even though there currently doesn't seem to be a usage limit on the Google Translator, we still need to cache results and save on bandwidth and the time it takes for the translation round-trip.
- Some people may want to view the translation on other devices. Since it is quite easy for Cake to provide data in different formats, we will add an RSS web feed service as well. We will use Cake's RSS helper.

- As we are translating the Twitter statuses, it makes sense to have the other parts of the site available in other languages as well. Using Cake's internationalization and localization classes, we can easily support this.
- Since we will be caching the results, we can also provide the ability to view past statuses.

Here are the details of these requirements:

- Each Twitter request we make will generate 20 statuses.
- A user comes to the site without specifying any language. The most recent 20 statuses in their original language are displayed on the first page. Since this happens to every user, we must always cache the results. We will be making requests to Twitter every 60 seconds. As such, we will cache the results every 60 seconds.
- We want the ability for any user to view past statuses as well as current statuses. Therefore, we will have a background process that will make Twitter requests and cache the results every 60 seconds, regardless of whether anyone is making a request to view statuses. In this way, we have separated the viewing of the statuses and the process that requests Twitter statuses.
- A user can select any language to view the current or past statuses. This language selection will also determine the language in which the rest of the site will be displayed.

In order to emphasize the raw nuts and bolts of the application, the interface layout will be kept as simple as possible. There will be two web pages:

- The first and main page will list 20 statuses in the selected language. We have sketched out a rough paper-and-pencil prototype in Figure 6-1. The top part consists of a header and tag line followed by a navigation area. We also allow the user to change the language at any point via a drop-down list on the right side of the page.

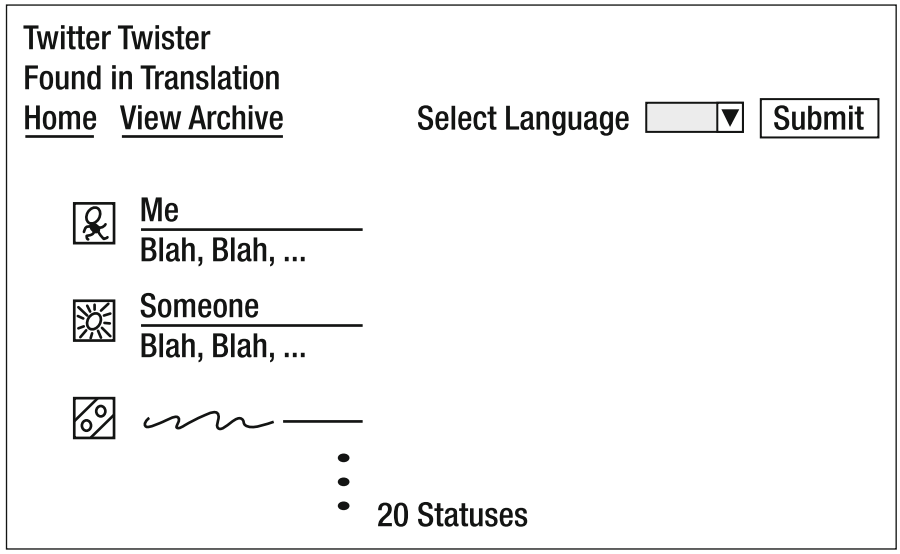

Figure 6-1. *Sketch of the status listing page*

- The second page will list the Twitter archives, which we have saved in the database. As shown in Figure 6-2, it will just list the date and time of each Twitter request. Each item in the list will be a link to the first view statuses page. This page will also have the drop-down list to allow the user to change the language.

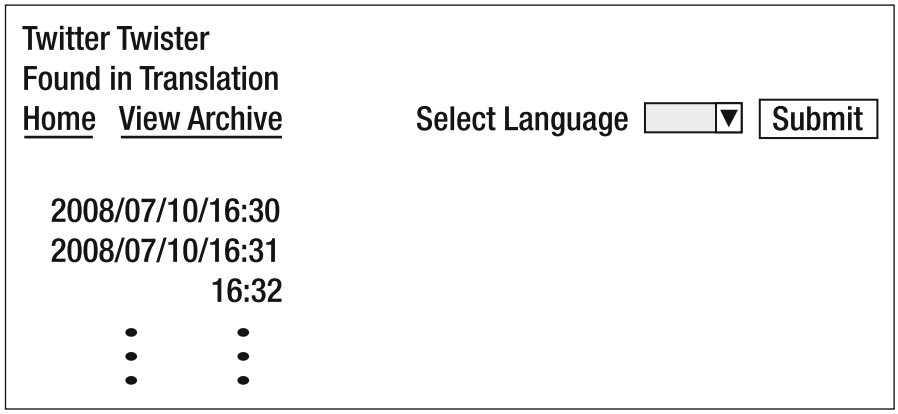

Figure 6-2. *Sketch of the archive listing page*

Application Structure

Each Twitter request we make will need to be stored in a database table. Remember that a single Twitter request to the public timeline returns 20 statuses. This naturally maps into a one-to-many relationship. Listing 6-3 shows the `CREATE TABLE` statements for both the `twitter_requests` and `twitter_statuses` tables.

Listing 6-3. *The twitter_requests and twitter_statuses Table Schemas*

```
CREATE TABLE `twitter_requests` (
  `id` int(11) NOT NULL auto_increment,
  `request_time` timestamp NOT NULL,
  PRIMARY KEY  (`id`)
);

CREATE TABLE `twitter_statuses` (
  `id` int(11) NOT NULL auto_increment,
  `twitter_request_id` int(11) NOT NULL,
  `t_created_at` timestamp NOT NULL,
  `t_id` varchar(50) collate latin1_general_ci NOT NULL,
  `t_text` varchar(255) character set utf8 collate utf8_unicode_ci NOT NULL,
  `t_source` varchar(255) collate latin1_general_ci NOT NULL,
  `t_truncated` varchar(255) collate latin1_general_ci NOT NULL,
  `t_in_reply_to_status_id` varchar(255) collate latin1_general_ci NOT NULL,
  `t_in_reply_to_user_id` varchar(255) collate latin1_general_ci NOT NULL,
```

```
  `t_favorited` varchar(255) collate latin1_general_ci NOT NULL,
  `t_user_id` varchar(50) collate latin1_general_ci NOT NULL,
  `t_user_name` varchar(255) collate latin1_general_ci NOT NULL,
  `t_user_screen_name` varchar(255) character set utf8 collate➥
utf8_unicode_ci NOT NULL,
  `t_user_location` varchar(255) collate latin1_general_ci NOT NULL,
  `t_user_description` varchar(255) character set utf8 collate➥
utf8_unicode_ci NOT NULL,
  `t_user_profile_image_url` varchar(255) collate latin1_general_ci NOT NULL,
  `t_user_url` varchar(255) collate latin1_general_ci NOT NULL,
  `t_user_protected` varchar(255) collate latin1_general_ci NOT NULL,
  `t_user_followers_count` varchar(50) collate latin1_general_ci NOT NULL,
  PRIMARY KEY (`id`)
);
```

In the `twitter_requests` table, the `request_time` field simply holds the time of the request. The `twitter_statuses` table will mainly hold all the data that has been returned by the Twitter request. Most fields are stored in the `latin1_general_ci format` collation. However, the `t_text`, `t_description`, and `t_user_screen_name` fields are stored in `utf8_unicode_ci`, since they will contain foreign characters. Notice that we're not using `utf8_unicode_bin`, as we may want to add search functionality to the fields later on; users expect searches to be case insensitive—`ci`.

The structure of how the translation will work is slightly trickier. When a Twitter request is made, the statuses returned will contain different languages. Do we also want to automatically translate them into all the other available languages? Pondering this point, we come up with a half yes and half no answer. Yes, we do want to translate them into other languages, but no, not immediately after fetching them from Twitter. This job should be spread over a period of time instead.

When a user does want to view the statuses in another language, whether they are current or past statuses, we will translate them on the fly and then cache the results. Thus, the first user who chooses to view a Twitter request in a specific language will need to endure the time delay that it takes to translate the 20 statuses. If we were running this application for real, we wouldn't want this to happen, as users would start to talk about how slow the site is to use.

When a status does get translated, the results are stored in the `twitter_translations` table, shown in Listing 6-4. This gives us a one-to-many relationship between `twitter_statuses` and `twitter_translations`, with one status having many different translations.

Listing 6-4. *The twitter_translations Table Schema*

```
CREATE TABLE `twitter_translations` (
  `id` int(11) NOT NULL auto_increment,
  `lang_from` varchar(10) collate latin1_general_ci NOT NULL,
  `lang_to` varchar(10) collate latin1_general_ci NOT NULL,
  `t_text_translation` varchar(255) character set utf8 collate➥
utf8_unicode_ci NOT NULL,
  `twitter_status_id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM
```

The lang_from and lang_to fields will contain the language code to translate from and to, respectively. The lang_from field is, in a sense, redundant. When we are retrieving a translation of a status, we don't really need to know the language it was in, as we have a foreign key link to the twitter_statuses table, but we may need it in the future. The t_text_translation field will contain the actual translation, so it needs to be in utf8_unicode_ci.

Since localization will be involved somewhere, we see it as an advantage to have a language table that will help us bind different language areas together. Listing 6-5 shows the schema for this table, named languages.

Listing 6-5. *The languages Table Schema*

```
CREATE TABLE `languages` (
  `id` int(11) NOT NULL auto_increment,
  `lang_name` varchar(255) collate latin1_general_ci NOT NULL,
  `lang_code` varchar(10) collate latin1_general_ci NOT NULL,
  `google_lang_code` varchar(10) collate latin1_general_ci NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM;
```

The languages table is essentially a lookup table for language codes. The lang_name field is for the name of the language, such as Japanese. The lang_code field is the three-letter ISO 639-3 language code used by Cake. The google_lang_code field is for the two-letter language code used by Google.

■ **Note** After much digging, we're still not sure why Google's language code has mostly two letters. We think it's a combination of codes, like the two-letter ISO 639-1 language code combined with locale names.

Figure 6-3 shows the relationships between the tables. The google_lang_code field in the languages table is linked to the lang_from and lang_to fields in the twitter_translations table. So we have a translation from one language to another language based on the google_lang_code entry. In the twitter_requests table, each request gives us many statuses, which are recorded in the twitter_statuses table, and each status has many different translations, which are recorded in the twitter_translations table.

twitter_statuses	
PK	**id**
FK1	twitter_request_id t_created_at t_id t_text t_source t_truncated t_in_reply_to_status_id t_in_reply_to_user_id t_favorited t_user_id t_user_name t_user_screen_name t_user_location t_user_description t_user_profile_image_url t_user_url t_user_protected t_user_followers_count

Figure 6-3. *Database schema for all the Twitter Twister tables*

Cake Models

We now have enough information to build our model classes. As shown in the database schema (Figure 6-3), one Twitter request generates many Twitter statuses. This relationship is shown in the `TwitterRequest` model using the `$hasMany` variable, as shown in Listing 6-6.

Listing 6-6. *The TwitterRequest Model (/app/models/twitter_request.php)*

```
<?php

class TwitterRequest extends AppModel {
    // Mainly for PHP4 users
    var $name = 'TwitterRequest';

    // Each journey has many locations and also many tags.
    var $hasMany = array(
        'TwitterStatus' => array(
        'className' => 'TwitterStatus',
        'order' => 'TwitterStatus.t_created_at'
        )
 );
}

?>
```

The twitter_statuses table sits between the twitter_translations table and the twitter_requests table. One twitter_statuses record has many translations in the twitter_translations table. To complete the has-many relationship between the TwitterRequest and TwitterStatus models, we need to add the belongsTo relationship to the TwitterStatus Cake model, shown in Listing 6-7.

Listing 6-7. *The TwitterStatus Model (/app/models/twitter_status.php)*

```
<?php

class TwitterStatus extends AppModel
{
    // Mainly for PHP4 users
    var $name = 'TwitterStatus';

    // Which db table to use
    var $useTable = 'twitter_statuses';

    var $hasMany = array(
        'TwitterTranslation' => array(
                                 'className' => 'TwitterTranslation',
                                 'foreignKey' => 'twitter_status_id' )
        );

    var $belongsTo = array(
        'TwitterRequest' => array(
                            'className' => 'TwitterRequest',
                            'foreignKey' => 'twitter_request_id' )
        );
}

?>
```

The TwitterTranslation model, shown in Listing 6-8, is quite simple, in that one single translation belongs to one Twitter status.

Listing 6-8. *The TwitterTranslation Model (/app/models/twitter_translation.php)*

```
<?php

class TwitterTranslation extends AppModel
{
    // Mainly for PHP4 users
    var $name = 'TwitterTranslation';
```

```
    // Which db table to use
    var $useTable = 'twitter_translations';

    var $belongsTo = array(
        'TwitterStatus' => array(
                            'className' => 'TwitterStatus',
                            'foreignKey' => 'twitter_status_id' )
        );
}

?>
```

Note that we'll be adding some methods to the `TwitterStatus` and `TwitterRequest` models later in the chapter (see Listings 6-20 and 6-22).

Oddly, the `Language` model is the most complex out of the lot, as shown in Listing 6-9. Some of the code will make more sense when we explain caching later on.

Listing 6-9. *The Language Model (/app/models/language.php)*

```
<?php

class Language extends AppModel
{
    // Mainly for PHP4 users
    var $name = 'Language';

    /*
    * Remove cache after database has been changed
    */
    function afterSave() {

        Cache::delete( 'getLang' );
    }

    /*
    * Get all the Twitter requests
    */
    function getLang() {

        $result = array();

        $allLang = $this->find( 'all' );

        foreach ( $allLang as $lang ) {

            $google_lang_code = $lang[ "Language" ][ "google_lang_code" ];
            $lang_name = $lang[ "Language" ][ "lang_name" ];
```

```
            $result[ $google_lang_code ] = $lang_name;
        }

        return $result;
    }
}

?>
```

The `Language` model includes two methods: `getLang` and `afterSave`. `getLang` simply fetches all the Twitter requests using the `find` method. There is nothing surprising about the method, but bear in mind that the data it generates will be cached.

The `afterSave` method is a Cake model callback. The code within the method is executed whenever some data is saved into the database. In our case, we want to delete the cache that holds the data for the language drop-down list, so we can have an up-to-date list of languages.

Caution The `afterSave` method is called only if you execute the `save` model method. If you use the `query` method to execute raw SQL statements, `afterSave` won't get executed.

Internationalization and Localization

Internationalization (abbreviated as i18n) is the process of developing software so it is portable between different cultures without any change to its internal coding. *Localization* (abbreviated as l10n) is the process of adapting that software to any specific culture by using a predefined set of parameters called *locales*, normally stored in text files. In our case, a culture is defined by a number of parameters: its language, number format, date/time format, and currency.

Adding or changing the locale in Cake is quite easy. Note that in our application, i18n and l10n will be handling only the static text of the site—such as tag lines, error messages, and so on—so they are quite separate from the job of the Google Translator.

To get started, we first add the default language that we will use. In the `/app/config/core.php` file, we add the following line:

```
Configure::write( 'Config.language', "eng" )
```

Tip In a real application, you would probably want to override `Config.language` depending on where the user is connecting from. You can detect from a subdomain, the user agent header `HTTP_ACCEPT_LANGUAGE`, or any of the IP geolocation web services. You can add this in the `beforeFilter` so each action uses the correct language to display the view.

Next, we need to set up the locale files. For our application, we will have only two languages available: English and Japanese. We will create the two files as shown here:

```
/app/locale/eng/LC_MESSAGES/default.po
/app/locale/jpn/LC_MESSAGES/default.po
```

The cultural locale parameters are divided into categories. The `LC_MESSAGES` category basically handles text messages. Other categories exist, like `LC_TIME` to handle date and time formats, but we won't be using those in this chapter.

Cake's i18n and l10n modules will use the content within the `.po` files to map between languages.

The folder name of the language uses the three-letter ISO 639-3 language code. The same three-letter code is also used in the `Config.language` setting. If you want to expand the number of languages, either look in the Cake file `/cake/libs/l10n.php` or go to the official standards body web site `http://www.sil.org/iso639-3/`.

The `.po` portable object files are human-readable and editable. There were originally developed as part of the GNU `gettext` utilities for language translation.

Listings 6-10 and 6-11 show our two `.po` files.

Listing 6-10. *The English .po File*

```
msgid  "app_tag_line"
msgstr "Found in Translation!"
```

Listing 6-11. *The Japanese. po File*

```
msgid  "app_tag_line"
msgstr "混乱! 翻訳! 笑い!"
```

It's pretty obvious to see that `app_tag_line` will be used as the handle for the translations.

Now, to display the localized content, we use the global convenience function `__();` for example, `<?php __( "app_tag_line" ) ?>`. This will echo the `msgstr` text, which corresponds to the language that we have set in `Config.language`.

In the section "The Controllers" later in this chapter, we will go through how we have actually implemented i18n and l10n in our application.

Tip In a production environment where there will be a lot of `.po` files, editing them should ideally be done with a `.po` file editor, such as Poedit. Also, as `.po` files may contain multibyte characters, remember to save them in UTF-8 format.

Web Services

In order to provide instant gratification for our application when people are away from their PCs, we will give users the option to view our application as an RSS feed. They will be able to easily see translated statuses when they are using a handheld device.

Adding different output formats for other devices to consume is a piece of cake! Or so we thought. . . .

We wrote a few different methods to achieve what we wanted. Starting with our first method, we saw that in the Cake documentation and some forums, the standard way to output RSS or any other format is to attach the format name you want at the end of the action as a file extension, like this:

```
/TwitterStatus/index.rss
```

In our case, we'll add the RSS feed to the `index` action. To get this to work, we need to tell the `Router` class about our new extension. We do this by adding the following line in `/app/config/routes.php`:

```
Router::parseExtensions( 'rss' );
```

Following this, we include the RSS helper in our `app_controller.php` file so we don't need to manually deal with any of the nitty-gritty RSS XML tags or headers.

Next, we need to have a specific layout for our RSS output. Under `/app/views/layout/rss`, we create a file called `default.ctp`. This is the layout that will surround the actual RSS data. As shown in Listing 6-12, we start off by adding the RSS header using the RSS helper. Next, we set up some basic channel data. And finally we echo the RSS document by passing `$rss->document` the `$content_for_layout`, which contains the list of statuses.

Listing 6-12. *The RSS Layout File (/app/views/layout/rss/default.ctp)*

```
<?php
    echo $rss->header();

    if (!isset($channel)) {
        $channel = array();
    }

    if (!isset($channel['title'])) {
        $channel['title'] = $title_for_layout;
    }

    echo $rss->document($rss->channel(  array(),
                                        $channel,
                                        $content_for_layout );
?>
```

Before we proceed further, let's recap. We want to view the same data in different formats, while keeping the coding for the action unchanged. It will be the view that will change depending on the type of request. Now, having completed the layout part, we need to write the new RSS view. Under the controller view folder `/app/views/twitter_status/`, we need

to create another folder called `rss`. In this folder, we create a file with the same name as the action, `index.ctp`. Listing 6-13 shows this file.

Listing 6-13. *The RSS View (/app/views/twitter_status/rss/index.ctp)*

```
<?php

    function rssTransform( $current_status ) {

        return array('title' => $current_status['v_t_user_name'],
                     'link' => 'http://twitter.com/'.$current_status[➥
"v_t_user_screen_name" ],
                     'guid' => 'http://twitter.com/'.$current_status[➥
"v_t_user_screen_name" ],
                     'description' => $current_status['v_t_text'],
                     'pubDate' => $current_status['v_t_created_at'],
                     );

    }

    echo $rss->items( $statuses, 'rssTransform' );
?>
```

In Listing 6-13, the method `$rss->items` takes the `$statuses` array and loops through each status item through the function `rssTransform`, which sets up the RSS XML data tags—like `title`, `link`, and so on—in a form that the `items` method can use.

Finally, we need to add the built-in `RequestHandler` component, so it can pick out the correct view:

```
var $components = array( 'RequestHandler' );
```

And that is pretty much the standard way of adding an RSS feed to your data. When you specify the `.rss` extension, the class router parses the extension. If you don't tell Cake about the extension and you make an `index.rss` request, you will probably get an error like this:

```
The action index.rss is not defined in controller TwittertwisterController.
```

Once Cake knows about the extension through the `Router` class, the `RequestHandler` takes over and renders the view that corresponds to the extension.

We haven't quite finished with web services just yet. As we hinted early on, we ran into some trouble with our RSS feed. The problem starts when we add parameters to our URL action requests. For example, if want to view an archive Twitter request in another language, this is the URL we could use:

```
/TwitterStatus/index/lang:ja/id:1/
```

It's fine as it is, but if we want to view it in XML format, the URL would need to look like this (note the `.rss` extension):

```
/TwitterStatus/index/lang:ja/id:1/.rss
```

or even like this:

```
/TwitterStatus/index/lang:ja/id:1.rss
```

Basically, the `Router` class just parses out the extension at the end of the URL. We feel these are just too awkward looking. We want our URLs to look something like this instead:

```
/TwitterStatus/index/lang:ja/id:1/rss
```

Basically, we just add `/xml` instead of `.rss`.

To overcome this problem, we created two alternative methods. The first method is a bit hackish, while the second one works better.

The first method is quite easy. Using the `RequestHandler` component, we just add the following code snippet in the `beforeFilter` method in the `app_controller.php`:

```
// hackish!
if ( preg_match( '/\/rss$/', Router::url() ) ) {
    $this->RequestHandler->ext = 'rss';
}
```

`Router:url()`returns the requested URL, such as `/TwitterStatus/index/lang:ja/id:1/rss`. `preg_match` then matches for the ending `/rss`. If a match is found, we artificially set the extension variable `ext` to `rss`. We feel quite uneasy with this method, as we shouldn't really be setting class variables manually like this. But, hey, it works, and one alternative solution is better than none.

The second method we used to handle the `/rss` extension is easier and more obvious than the first. We just write a `mod_rewrite` rule in our root `.htaccess` file, as follows:

```
<IfModule mod_rewrite.c>
   RewriteEngine on
   RewriteRule    ^$ app/webroot/    [L]
   RewriteRule (.*)/rss$ app/webroot/$1/.rss [L]
   RewriteRule    (.*) app/webroot/$1 [L]
</IfModule>
```

The new rule is shown in bold. It matches anything ending with `/rss` and rewrites it in the `.rss` version. So if a user types

```
/TwitterStatus/index/lang:ja/id:1/rss
```

it gets translated to

```
/TwitterStatus/index/lang:ja/id:1/.rss
```

Using this `mod_rewrite` rule and the standard method to add web services, we feel this combination works well.

Caching

Our seemingly simple application has gotten a lot more complicated with the need to cache results. To start caching, we need to turn on Cake caching. This is done in the controller `/app/config/core.php`:

```
Configure::write('Cache.check', true);
```

We want to use the Cache helper. We'll just include it in `app_controller.php`, so all controllers in the future can have access to it. In our `app_controller.php` file, we simply add the following line:

```
var $helpers = array( 'Cache', 'Form', 'Html', 'Rss' );
```

Caching Views

Caching is a broad term. In our application, we're doing different types of caching, one of which is caching the views using Cake. This is where the output is saved, by default to `/app/tmp/cache/views`.

To cache a view, the setting is made in the action that corresponds to the view. In the controller, there are two different ways you can do this. One way is to set the `cacheAction` member variable, as in this example:

```
$cacheAction = array( 'TwitterRequest/view, 60 );
```

This caches the `viewArchive` action for 60 seconds.

We can also cache an action within the action function itself, like this:

```
$this->cacheAction = array( 'duration' => 60 );
```

This caches the current action for 60 seconds.

Caching Models

Cake goes through a lot of processing to build the object-relational mapping part of the MVC structure. To save on the execution time, you can cache the model data by setting the `persistModel` controller variable to `true`:

```
var $persistModel = true;
```

The cached model data is stored in the `/app/tmp/cache/persistent` folder.

■ **Caution** If you change or add database fields, you must delete the cached data models. Be careful of cached data during your development. Sometimes you forget about your caching and then wonder why your changes are not showing up. Check your browser session cache and caching in Cake.

Caching Twitter and Google Translations

Once we get data back from Twitter and Google, that data is fairly permanent. The data we get is saved to the database in the three tables we have created. This is unlike some web caching, where data can be occasionally purged or destroyed.

In our application, the caching works in two levels. When a user requests the index page, Cake first checks to see whether the view is held in cache. If its not, the `index` action is called.

This then fetches the most recent statuses from the database. This data is essentially saved or cached in the database by the background cron process, which fetches the actual status messages from Twitter.

Caching and the Application Layout

Caching can bring up a lot of unforeseen problems within the application view. There are two main points to be aware of:

- We need to cache only some parts of a view. To achieve this, we use the Cake `<cake:nocache>This is not cached</cake:nocache>` tag to wrap content that we don't want to cache.
- Remember that no actions are called in the controller.

When we specify in the controller that we want to cache an action view, Cake will also cache the application layout. As such, we can't cache any dynamic content that changes within a session, whether in the action view or application layout.

As shown in the application layout in Listing 6-14, we don't cache our tag line, so we wrap it around the `<cake:nocache>` tag to say that it is not cached.

Listing 6-14. *Header HTML Code with the nocache Tag (in /app/views/layouts/base.ctp)*

```
<div class="header_wrapper">
    <h1>Twitter Twister</h1>
    <h2><i><cake:nocache><?__("app_tag_line")?></cake:nocache></i></h2>
</div>
```

Changing Languages

Since the HTML language selection drop-down list is displayed on every page within the application layout, we have pushed it out as a Cake element, as shown in Listing 6-15.

Listing 6-15. *Language Selection Form (/app/views/elements/lang_drop.ctp)*

```
<?php

    echo $form->create( 'Twittertwister',
                        array(  'url' => '/Twittertwister/changeLanguage',
                                'class' => 'lang_change_form'
                                ) );
?>

<cake:nocache>

<?php

    ClassRegistry::addObject( 'view', $this );
```

```
    echo $form->select( "Twittertwister.lang",
                        $session->read( "getLang" ),
                        $session->read( "userLang" ),
                        null,
                        true
                        );
?>

</cake:nocache>

<?php

    echo $form->submit( 'Change', array( 'label' => false,
                                         'div' => false
                                            ) );

    echo $form->end();
?>
```

The important part here is the `<cake:nocache>` tag. We don't want to cache the HTML drop-down list, because if a user changes the language, the newly selected item in the language should be selected and not a cached selection. However, we do want to cache the language data that was generated by the SQL find operation from the `languages` table list.

Some readers may notice this odd line of code:

```
ClassRegistry::addObject( 'view', $this );
```

When a view is rendered from the cache, the current view object isn't registered in Cake's global object register. As a result, some helpers that depend on the view object, such as the form helper, fail. Because of this, we need to manually register the current view object.

This highlights the complexity of caching. And in Cake, when an item is cached, no method in the controller is called. Therefore, we need to rely on session data as a conduit for any dynamic data that we need to pass to the view. As highlighted in the code, when we form the HTML `select` tag, we populate it with data that we saved into the Cake session. The `getLang` session variable holds the language data from the `languages` table, and `userLang` holds the current language that the user has selected.

Changing Locales

As no methods are called in the controller when data is cached, we need to set the new locale in the top part of the base layout file `/app/views/layouts/base.ctp`, as shown in Listing 6-16. When a user selects a new language, the cached page is presented while noncached content within the cached content is still being executed.

■ **Caution** Adding caching to our application was complex and time-consuming, mainly because we were also using other Cake features such as locales. If you are going to add caching, try to add it early on in your development. This will make your life easier when your application grows.

Listing 6-16. *Setting Locale in the Base View (/app/views/layouts/base.ctp)*

```
// ...
<cake:nocache>
<?php

    if ( $session->read( "userLocale" ) ) {
        Configure::write( 'Config.language', $session->read( "userLocale" ) );
    }

?>
</cake:nocache>
// ...
```

The Controllers

For our application, three controllers are used: `TwittertwisterController`, `TwitterRequestController`, and `TwitterStatusController`. Let's look at how each of these controllers works.

The Twittertwister Controller

The `TwittertwisterController` class file is shown in Listing 6-17. This is a base controller for the application. At present, it holds one action called `changeLanguage`, which changes the viewing language of the application.

Listing 6-17. *The TwittertwisterController Class (app/controllers/twittertwister_controller.php)*

```
 1:    <?php
 2:
 3:    class TwittertwisterController extends AppController {
 4:
 5:        var $name = 'Twittertwister';
 6:
 7:        var $uses = array();
 8:
 9:       /*
10:        * User specifically changed language
11:        */
12:        function changeLanguage() {
13:
14:            if ( $this->data[ "Twittertwister" ][ "language" ] ) {
15:                $this->Session->write(
16:                            "userLang",
17:                            $this->data[ "Twittertwister" ][ "language" ] );
18:            }
19:
```

```
20:             // We must also change the locale language
21:             $this->__changeSessionLocale(
22:                             $this->data[ "Twittertwister" ][ "language" ] );
23:
24:             $url = Router::parse( $this->referer() );
25:
26:             $url_str = '/'.$url[ 'controller' ].'/'.$url[ 'action' ].'/';
27:
28:             if ( isset( $url[ 'named' ][ 'id' ] ) ) {
29:                 $url_str .= 'id:'.$url[ 'named' ][ 'id' ].'/';
30:             }
31:
32:             // Set language parameter for action in calling page
33:             $url_str .= 'lang:'.$this->data['Twittertwister']['language'].'/';
34:
35:             // Redirect back to calling page
36:             $this->redirect( $url_str );
37:         }
38:
39:         /*
40:         * Change the locale
41:         */
42:         function __changeSessionLocale( $google_lang_code ) {
43:
44:             $lang = $this->Language->findByGoogleLangCode( $google_lang_code );
45:
46:             if ( $lang ) {
47:                 $this->Session->write(  "userLocale",
48:                                         $lang[ 'Language' ][ 'lang_code' ] );
49:             }
50:         }
51:     }
52:
53:     ?>
```

This `changeLanguage` action is used by the language-selection form. Starting from line 12, this action takes the language code that the user has selected and changes the status viewing language on line 14 and the locale on line 21. From line 24 onwards, we redirect the users back to the page where they selected the language change, where they would view the same page they have selected in the new language.

The private method `__changeSessionLocale` on line 42 is used to change the web site locale. The language code used by Google was incompatible with the ISO 639-3 three-letter language code used by Cake's l10n. As a result, we needed to do some language code translation. We simply make a query to the `languages` table and get the three-letter code that corresponds with Google's two-letter code, and then save this into the session. We do this using Cake's model method `findBy[field_name]( [field_value] )`. Attach the field name in camel case as a suffix to the `findBy` keyword. In our case, the field name is `google_lang_code`, so the method name will be `findByGoogleLangCode`. We then supply the method with the value of the field as an argument.

The TwitterRequest Controller

The `TwitterRequestController` class is responsible for fetching Twitter statuses from the Twitter public timeline. The controller file `app/controllers/twitter_request_controller.php` is shown in Listing 6-18.

Listing 6-18. *The TwitterRequestController Class (app/controllers/twitter_request_controller.php)*

```
<?php

class TwitterRequestController extends AppController {

    var $cacheAction = array( 'TwitterRequest/view' => '60' );

    /*
    * View entire archive
    */
function view() {

        // Get all Twitter requests
        $allRequests = $this->TwitterRequest->find( 'all',
                                            array(  null,
                                                    null,
                                                    'recursive' => -1
                                                              ) );

        $this->set( 'twitterRequests', $allRequests );
    }

    /*
    * This should be called by cron every few seconds
    */
    function getTwitterRequests() {

        $this->layout = 'blank';

        $this->makeTwitterRequest();
    }

    /*
    * This call the Twitter public timeline
    */
    function makeTwitterRequest() {

```

```
37:             // Save the request header
38:             $this->TwitterRequest->saveRequest();
39:
40:             // Set up and execute the socket call
41:             $url = 'http://www.twitter.com/statuses/public_timeline.xml';
42:
43:             App::import( 'HttpSocket' );
44:             $http = new HttpSocket();
45:             $request = array( 'uri' => $url );
46:             $body = $http->request($request);
47:
48:             // Now save into db
49:             $this->TwitterStatus->saveStatuses( $body,
50:                                                 $this->TwitterRequest->id );
51:         }
52:
53:     }
54:
55:     ?>
```

On line 5, we set `$cacheAction` to cache the `viewArchive` action to 60 seconds. We will explain this at the end of this section.

The controller contains two main actions: `view` and `getTwitterRequests`. The `view` action on line 10 lists all the Twitter requests in the database. As we are just listing the Twitter requests, we don't need the other tables that link to it, so we set `recursive` to `-1`. Remember that we have cached this action at the start of the controller to 60 seconds, so we are not regenerating the same query for different users. One obvious shortcoming with this action is the number of calls that will be returned. As we are making Twitter requests every 60 seconds, it would quickly generate a lot of requests. In fact, it probably won't take long before it would take more than 60 seconds to list all the status requests.

Line 25 starts with the `getTwitterRequests` action. This is essentially an action that we use during development to fetch statuses; however, it can still be used as the URL for a cron job entry. This action makes public timeline calls to Twitter requests every 60 seconds. The view in Listing 6-19 is even simpler. (If you use this in a cron entry, comment out the meta refresh, as that should be the job for cron.)

Listing 6-19. *The getTwitterRequests View (app/views/twitter_rquest/get_twitter_requests.ctp)*

```
<meta http-equiv="refresh" content="60">
```

The `makeTwitterRequest` action shown on line 35 of Listing 6-18 carries out the task of requesting the public timeline from Twitter. We start on line 38 by calling the `saveRequest` method in the `TwitterRequest` model, which is shown in Listing 6-20. On line 14 of this listing, the `saveRequest` method simply saves a record into the `twitter_requests` table by using the model's `save` method on line 20.

Listing 6-20. *The TwitterRequest Model with the saveRequest Method (app/models/twitter_request.ctp)*

```
 1:    <?php
 2:
 3:    class TwitterRequest extends AppModel
 4:    {
 5:        // Mainly for PHP4 users
 6:        var $name = 'TwitterRequest';
 7:
 8:        // Each journey has many locations and also many tags.
 9:        var $hasMany = array(
10:                        'TwitterStatus' => array(
11:                            'className' => 'TwitterStatus',
12:                            'order'     => 'TwitterStatus.t_created_at' ) );
13:
14:        function saveRequest() {
15:
16:            // Each request must be saved in the twitter_requests table
17:            $reqData = array();
18:            $reqData[ 'request_time' ] = date( 'Y-m-d H:i:s', mktime() );
19:            $this->create( $reqData );
20:            $this->save();
21:        }
22:    }
23:
24:    ?>
```

Let's now go back to the `makeTwitterRequest` action in `TwitterRequestController` (Listing 6-18). Having now saved a request, we make a call to the Twitter public timeline using Cake's `HttpSocket` class. We fetch the statuses and save the results into the `$body` variable on line 46. The data returned will be in XML format. You can request the return format to be `json`, `rss`, or `atom` by appending those format strings as the extension instead of `xml`.

Finally, on line 49 of the controller, we save the status return message into the `twitter_statuses` table using the `TwitterStatus` model method `saveStatuses`. We will talk more about this method in the next section.

The TwitterStatus Controller

The `TwitterStatusController` class is the largest of the three controllers. It's responsible for displaying either statuses specified by the user or the most recent statuses that were fetched within the last minute. Listing 6-21 lists the controller with the actions omitted. As the controller is quite large, we will show each action in turn.

Listing 6-21. *The TwitterStatusController Class (app/controllers/twitter_status_controller.php)*

```
 1:    <?php
 2:
 3:    class TwitterStatusController extends AppController {
 4:
 5:        // View any particular Twitter request
 6:        var $currentTwitterReqId = '';
 7:
 8:        // actions ...
 9:
10:    }
11:
12:    ?>
```

On line 6, the `$currentTwitterReqId` variable holds the current Twitter request, if any. If a user has requested the home page, this would be empty, and the most recent statuses will be returned.

Before we talk about the controller's actions, let us first talk about the model. As shown in Listing 6-22, the model has only one method called `saveStatuses`. As we mentioned earlier, this method is used by the `makeTwitterRequest` action in the `TwitterRequestController`. It saves the statuses returned from a Twitter request.

Listing 6-22. *The TwitterStatus Model with the saveStatuses Method (app/models/twitter_status.ctp)*

```
 1:    <?php
 2:
 3:    class TwitterStatus extends AppModel
 4:    {
 5:        // Mainly for PHP4 users
 6:        var $name = 'TwitterStatus';
 7:
 8:        // Which db table to use
 9:        var $useTable = 'twitter_statuses';
10:
11:        var $hasMany = array( 'TwitterTranslation' => array(
12:                                  'className' => 'TwitterTranslation',
13:                                  'foreignKey' => 'twitter_status_id' ) );
14:
15:        var $belongsTo = array( 'TwitterRequest' => array(
16:                                  'className' => 'TwitterRequest',
17:                                  'foreignKey' => 'twitter_request_id' ) );
18:
19:        /*
20:        * Save statuses into database
21:        */
22:        function saveStatuses( $statuses, $twitterReqId ) {
23:
```

```
24:             // Check we have a Twitter request id
25:             if ( !is_numeric( $twitterReqId ) ) {
26:                 return;
27:             }
28:
29:             $xml = new SimpleXMLElement( $statuses );
30:
31:             foreach ($xml->status as $status) {
32:
33:                 $statusData = array();
34:
35:                 $statusData[ "twitter_request_id" ] = $twitterReqId;
36:
37:                 $statusData[ "t_created_at" ] = $status->created_at;
38:                 $statusData[ "t_id" ] = $status->id;
39:                 $statusData[ "t_text" ] = $status->text;
40:                 $statusData[ "t_source" ] = $status->source;
41:                 $statusData[ "t_truncated" ] = $status->truncated;
42:                 $statusData[ "t_in_reply_to_status_id" ] =
43:                                         $status->in_reply_to_status_id;
44:                 $statusData[ "t_in_reply_to_user_id" ] =
45:                                         $status->in_reply_to_user_id;
46:                 $statusData[ "t_favorited" ] = $status->favorited;
47:
48:                 $statusData[ "t_user_id" ] = $status->user->id;
49:                 $statusData[ "t_user_name" ] = $status->user->name;
50:                 $statusData[ "t_user_screen_name" ] =
51:                                         $status->user->screen_name;
52:                 $statusData[ "t_user_location" ] = $status->user->location;
53:                 $statusData[ "t_user_description" ] =
54:                                         $status->user->description;
55:                 $statusData[ "t_user_profile_image_url" ] =
56:                                     $status->user->profile_image_url;
57:                 $statusData[ "t_user_url" ] = $status->user->url;
58:                 $statusData[ "t_user_protected" ] = $status->user->protected;
59:                 $statusData[ "t_user_followers_count" ] =
60:                                         $status->user->followers_count;
61:
62:                 $this->create( $statusData );
63:                 $this->save();
64:             }
65:         }
66:
67:     }
68:
69:     ?>
```

The `saveStatuses` method starts on line 22. We take the statuses and parse the XML using PHP's `SimpleXMLElement` class. You can, of course, accomplish the same thing using Cake. Listings 6-23 and 6-24 show how to get the `created_at` field using Cake's XML classes.

Listing 6-23. *Using Cake's XML Class, Longer Version*

```
App::import( 'Xml' );
$xml_1 = new Xml( $statuses );

$node_1 = $xml_1->children( 'statuses' );
$node_2 = $node_1[0]->children( 'status' );
$node_3 = $node_2[0]->children( 'created_at' );
$created_at = $node_3[0]->children[0]->value;
```

Listing 6-24. *Using Cake's XML Class, Shorter Version*

```
App::import( 'Xml' );
$xml_1 = new Xml( $statuses );

$node_1 = $xml_1->child(0)->child(0)->children( 'created_at' );
$node_2 = $node_1[0]->child(0)->value;
```

Now we'll look at each of the actions in the `TwitterStatusController`:

The index() Action

The `index` action is shown in Listing 6-25.

Listing 6-25. *TwitterStatusController index Action*

```
 1:        function index( $twitterReqId = '', $transLang = '' ) {
 2:
 3:            // Override current Twitter request
 4:            if ( $twitterReqId ) {
 5:                $this->currentTwitterReqId = $twitterReqId;
 6:            }
 7:
 8:            if ( isset( $this->passedArgs['id'] ) ) {
 9:                $this->currentTwitterReqId = $this->passedArgs['id'];
10:            }
11:
12:            // If there is no id for the current request, then we assume
13:            // most recent statuses so we only cache for 60 secs
14:            if ( $this->currentTwitterReqId ) {
15:                // Cache it for a year
16:                $this->cacheAction = array( 'duration' => 31536000 );
17:            }
```

```
18:             else {
19:                 $this->cacheAction = array( 'duration' => 60 );
20:             }
21:
22:             // Override current language view
23:             if ( $transLang ) {
24:                 $this->currentLang = $transLang;
25:             }
26:
27:             if ( isset( $this->passedArgs['lang'] ) ) {
28:                 $this->currentLang = $this->passedArgs['lang'];
29:             }
30:
31:             $this->__displayTwitters();
32:         }
```

On lines 4 and 8, we can override which Twitter request we want to view, via either the pretty URL method or the named parameter method. Both of the following examples would work:

```
twittertwister/index/1
```

or

```
twittertwister/index/id:1/
```

Following this on line 14, if the user has requested to view a particular Twitter request, we will cache it for a whole year. We wanted to cache it indefinitely, but we couldn't find an easy way except by hacking the Cake library files, which we definitely didn't want to do.

If a user hasn't requested a particular ID, then by default, the most recent statuses would be displayed. In this case, we cache it for 60 seconds, as new statuses are fetched from the Twitter servers every 60 seconds as well.

Next, we do the same check for the language selection. Once those two checks are complete, we display the Twitter statuses on line 31.

On line 27, even if a user has selected to view statuses in a language via the language drop-down list, he can still override the viewing language by passing a different language code via the URL, which overrides the `$currentLang` variable. For example, a Japanese user can still view a post in English by using a URL with a `lang` parameter in it set to `en`.

The __displayTwitters() Action

The private action `__displayTwitters`, shown in Listing 6-26, is the core of the application.

Listing 6-26. *TwitterStatusController __displayTwitters Action*

```
    function __displayTwitters() {

            // We check whether a particular Twitter request was requested or
            // we just fetch the recent one
            $conditions = array();

            if ( is_numeric( $this->currentTwitterReqId ) ) {
                $conditions[ 'TwitterRequest.id' ] = ➥
$this->currentTwitterReqId;
            }
            else {
                $conditions = array ();
                $conditions[] = array(
                    "TwitterStatus.t_created_at <" => date(
                                                'Y-m-d H:i:00',
                                                strtotime( "-1 minute" )
                                                ) );
                $conditions[] = array(
                    "TwitterStatus.t_created_at >" => date(
                                                'Y-m-d H:i:00',
                                                strtotime( "-2 minute" )
                                                ) );
            }

            // Check to see if user has selected to view the statuses in
            // any particular language
            if ( $this->currentLang ) {

                $twitTransTable = array(
                    'TwitterTranslation' => array(
                                    'className' => 'TwitterTranslation',
                                    'foreignKey' => 'twitter_status_id',
                                    'conditions' => ➥
"TwitterTranslation.lang_to = '{$this->currentLang}'" ) );

                $this->TwitterStatus->bindModel(
                                    array( 'hasMany' => $twitTransTable ) );
            }
            else {
                // No point getting translation, so lets unbind it
                $this->TwitterStatus->unbindModel(
                   array( 'hasMany' => array( 'TwitterTranslation' ) ) );
            }

```

```
43:             $min_statuses = $this->TwitterStatus->find( 'all',
44:                         array(  'conditions' => $conditions,
45:                                 null,
46:                                 'order' => 'TwitterStatus.t_created_at DESC'
47:                                      ) );
48:
49:             // OK, now we got the status for the minute in question.
50:             // We check whether the language translation exists
51:             // in those statuses
52:             $statusesTrans = $this->__statusTranslate( $min_statuses );
53:
54:             // The view has no knowledge of what language
55:             // to display something
56:             $this->set( 'statuses', $statusesTrans );
57:         }
```

This method fetches Twitter statuses according to a number of query conditions, starting on line 5. We need to know what time period of statuses we want. Do we want archived statuses or the most recent statuses? When we are retrieving the most recent 20 statuses, we set it between 1 and 2 minutes behind the current time, so we will always get a full minute.

To make the query more efficient, if no language was selected, there is no need to try to fetch any associated translations, which is by default what would happen—see line 26 onwards. Therefore, we unbind any associations using the `unbindModel` method when no language was chosen. Once we have the statuses, on line 52, we make a call to `_statusTranslate`, which will translate any statuses if there are any. Finally, we set the view variable `statuses` so it's available to be displayed in any view.

The __statusTranslate() Action

The `__statusTranslate` private action, shown in Listing 6-27, is a supporting action to the `__displayTwitters` action. It helps us to translate statuses transparently, independent of the language in which the status is originally.

Listing 6-27. *TwitterStatusController __statusTranslate Action*

```
 1:    function __statusTranslate( $statuses ) {
 2:
 3:        $result = array();
 4:
 5:        for( $idx=0; $idx<sizeof( $statuses ); $idx++ ) {
 6:
 7:            // Original language text
 8:            $t_user_name = $statuses[$idx][ "TwitterStatus" ][ "t_user_name" ];
 9:            $t_text = $statuses[$idx][ "TwitterStatus" ][ "t_text" ];
10:            $t_user_url = $statuses[$idx][ "TwitterStatus" ][ "t_user_url" ];
11:            $t_user_profile_image_url = ➥
$statuses[$idx][ "TwitterStatus" ][ "t_user_profile_image_url" ];
```

```
12:             $t_user_location =
13:                     $statuses[$idx][ "TwitterStatus" ][ "t_user_location" ];
14:             $t_user_screen_name =
15:                 $statuses[$idx][ "TwitterStatus" ][ "t_user_screen_name" ];
16:             $t_created_at = ➥
$statuses[$idx][ "TwitterStatus" ][ "t_created_at" ];
17:
18:             // We only check if there is a destination language specified
19:             // Check if there is a translation from the original to the
20:             // destination; if there is, we override, original text
21:             if ( $this->currentLang ) {
22:
23:                 if ( isset( $statuses[$idx][ "TwitterTranslation" ] ) ) {
24:
25:                     // Note we only translate status
26:
27:                     $trans_result = $this->__getStatusTranslation(
28:                                                 $statuses[$idx] );
29:
30:                     if ( isset( $trans_result[ "t_text" ] ) ) {
31:                         $t_text = $trans_result[ "t_text" ];
32:                     }
33:                 }
34:             }
35:
36:             $current_result = array();
37:
38:             $current_result[ "v_t_user_name" ] = $t_user_name;
39:             $current_result[ "v_t_text" ] = $t_text;
40:             $current_result[ "v_t_user_url" ] = $t_user_url;
41:             $current_result[ "v_t_user_profile_image_url" ] =
42:                                         $t_user_profile_image_url;
43:             $current_result[ "v_t_user_location" ] = $t_user_location;
44:             $current_result[ "v_t_user_screen_name" ] = $t_user_screen_name;
45:             $current_result[ "v_t_created_at" ] = $t_created_at;
46:
47:             $result[] = $current_result;
48:         }
49:
50:         return $result;
51:     }
```

Basically, if there is a translation found for the status, we replace the original status with that translation. This happens on line 21. If a translation is not found, we will just display the status in its original language.

Additionally, we have formed a new result array to return on line 36. In that array, we prefix each key with v_ to make it clear that the array is to be used in the view.

Now that we have covered both `__displayTwitters` and `__statusTranslate`, the following view (`/app/views/twitter_status/index.ctp`), which displays the statuses, should make sense.

```
<?php

    foreach ( $statuses as $current_status ) {

        echo '<div class="status_rec">';
            echo '<h3><a href="http://twitter.com/'.$current_status[➥
"v_t_user_screen_name" ].'" target="_blank">'.$current_status[➥
"v_t_user_name" ].'</a></h3>';
            echo '<hr align="left" noshade="" size=1 width="100%">';

            if ( $current_status[ "v_t_user_url" ] ) {
                echo '<a href="'.$current_status[ "v_t_user_url" ].'">➥
<img class="profile_img" src="'.$current_status[➥
"v_t_user_profile_image_url" ].'" align="left"></a>';
            }
            else {
                echo '<img class="profile_img" src="'.$current_status[➥
"v_t_user_profile_image_url" ].'" align="left">';
            }

            echo $current_status[ "v_t_text" ].'<br />';

            if ( $current_status[ "v_t_user_location" ] ) {
               echo '<b>From:</b> '.$current_status[ "v_t_user_location" ];
            }

        echo '</div>';
    }

?>
```

Here, we basically loop through the statuses and display the fields. Figure 6-4 shows an example of the translated statuses page.

Figure 6-4. *A translated statuses page*

The __getStatusTranslation() Action

The `__getStatusTranslation` action, shown in Listing 6-28, complements the previous two actions.

Listing 6-28. *TwitterStatusController __getStatusTranslation Action*

```
 1:    function __getStatusTranslation( $statuses ) {
 2:
 3:        $result = array();
 4:
 5:        $transTo = $this->currentLang;
 6:
 7:        // 1. Check t_text translation. We always use the first one
 8:        if ➥
( isset( $statuses[ "TwitterTranslation" ][0][ "t_text_translation" ] ) ) {
 9:
10:            // Yes there is a translation. Let's use that
11:            $result[ "t_text" ] =
12:                $statuses[ "TwitterTranslation" ][0][ "t_text_translation" ];
13:        }
14:        else {
15:            $sourceLang = '';
16:
17:            // No there is no translation. Let's get one
18:
19:            // Start with t_text
20:            $t_text = $statuses[ "TwitterStatus" ][ "t_text" ];
21:            $google_trans_result =
22:                                $this->__translateText( $t_text, $transTo );
23:
24:            $result[ "t_text" ] = $google_trans_result[ "translation" ];
25:            $sourceLang = $google_trans_result[ "source_lang" ];
26:
27:            $this->__saveTrans( $sourceLang,
28:                                $transTo,
29:                                $result,
30:                                $statuses[ "TwitterStatus" ][ "id" ] );
31:        }
32:
33:        return $result;
34:    }
```

On line 8, if a translation already exists, we won't translate it. If there isn't a translation, we make the request to Google to translate the status, from line 14 onwards. If a translation is found, we always take the first translation, as we will explain shortly, in the description of the `_saveTrans` action.

The __translateText() Action

The private action in Listing 6-29 uses Cake's `HttpSocket` class methods to make translation requests to Google. Sometimes, the call may fail. In that situation, we still return the status minus the translation. We throttle the translation requests with a sleep period, as that helps the reliability of any immediate future requests.

Listing 6-29. *TwitterStatusController__translateText Action*

```
 1:    function __translateText( $transText, $destLang ) {
 2:
 3:        $result = array(    "translation" => "",
 4:                            "source_lang" => "",
 5:                            "response_status" => "" );
 6:
 7:        if ( empty( $transText ) ) { return ""; }
 8:
 9:        $params = array();
10:    $params[ "v" ] = "1.0";
11:    $params[ "q" ] = $transText;
12:    $params[ "langpair" ] = "|".$destLang;
13:    $params[ "key" ] = "!!!!!! Insert Your Google API Key Here !!!!!!";
14:
15:    $paramStr = $this->__constructURL( $params );
16:
17:        $url = ➥
"http://ajax.googleapis.com/ajax/services/language/translate?".$paramStr;
18:
19:        App::import( 'HttpSocket' );
20:        $http = new HttpSocket();
21:        $request = array(
22:            'uri' => $url,
23:            'header' => array(
24:            'Referer' => 'http://'.env('SERVER_NAME')
25:            )
26:        );
27:        $body = $http->request($request);
28:
29:        // Now, process the JSON string
30:        $json = json_decode( $body );
31:
32:        if ( isset( $json->responseStatus ) ) {
33:
34:            // Translation was good
35:            if ( $json->responseStatus == "200" ) {
36:                $result[ "translation" ] = $json->responseData->translatedText;
37:                $result[ "source_lang" ] = ➥
$json->responseData->detectedSourceLanguage;
38:                $result[ "response_status" ] = "200";
39:            }
40:            else {
41:                // We just fill with original
42:                $result[ "translation" ] = $transText;
43:                $result[ "source_lang" ] = "UNKNOWN";
44:                $result[ "response_status" ] = $json->responseStatus;
```

```
45:             }
46:         }
47:
48:         // We always wait for a bit before next action, 0.5 sec
49:         usleep( 500000 );
50:
51:         return $result;
52:     }
```

As this action is used by `__getStatusTransation`, which in turn saves it into the database, we may occasionally see the value `UNKNOWN` in the `twitter_translations` table. In this chapter, we won't be going into the reasons why some text doesn't get translated. Our goal is to translate some piece of text using Google. If the translation fails for some reason, we still return the data in the same format, as if it were translated. This way, we have a log of the failure of the translation, and the user still gets to see the original text, which we think is better than seeing no status text or an empty translation display.

We start the action off on line 7. If no text is given, we just return an empty string. If a string is given, we form the code to carry out the `HttpSocket` from line 9 to line 27. We then decode the returned result on line 30 onwards. On line 13, remember to use your own Google API key.

The __saveTrans() Action

There isn't much to the `__saveTrans` action, shown in Listing 6-30.

Listing 6-30. *TwitterStatusController__saveTrans Action*

```
 1:    function __saveTrans( $sourceLang, $destLang, $transResult, ➥
$twitterStatusId ) {
 2:
 3:        // We first check whether the translation exists for the
 4:        // "twitter_status_id" and "lang_to". Even if it does, we may
 5:        // still have 2 or more of the same entries in the database.
 6:
 7:        $conditions = array (
 8:                    "TwitterTranslation.twitter_status_id" => $twitterStatusId,
 9:                    "TwitterTranslation.lang_to" => $destLang
10:                                );
11:
12:        $transExist = $this->TwitterTranslation->find(
13:                            'first', array(  'conditions' => $conditions ) );
14:
15:
16:        if ( empty( $transExist ) ) {
17:
18:            $statusTrans = array();
19:            $statusTrans[ 'lang_from' ] = $sourceLang;
20:            $statusTrans[ 'lang_to' ] = $destLang;
```

```
21:             $statusTrans[ 't_text_translation' ] = $transResult[ "t_text" ];
22:             $statusTrans[ 'twitter_status_id' ] = $twitterStatusId;
23:
24:             $this->TwitterTranslation->create( $statusTrans );
25:             $this->TwitterTranslation->save();
26:         }
27:     }
```

On line 12, we check if there are two or more of the same translation. If not, we enter the translation into the `twitter_translations` table on lines 24 and 25. Why would there be two or more of the same translation? Well, since we are not translating the statuses in a background process, any user who first brings up the statuses in a specific language would fire up the translation parts of the controller. As we are not using SQL transaction locks, there could easily be the situation where two or more users are requesting the same statuses at roughly the same time. Using transaction locks would hold up the rendering of the page. We don't feel they are necessary, as we can always write some housekeeping function to clear any duplicate entries. This is better than delaying the translation of the page.

The AppController

Along with our three controllers, the global `AppController` also plays a part in our application. Listing 6-31 shows the contents of the `AppController` file `app/app_controller.php`.

Listing 6-31. *The Application Base Controller (app/app_controller.php)*

```
 1:    <?php
 2:
 3:    class AppController extends Controller {
 4:
 5:        // Default page title
 6:        var $pageTitle = 'Chapter 6 - ➥
Mashing Twitter with the Google Translator';
 7:
 8:        // The view helpers that we'll use globally
 9:        var $helpers = array( 'Cache', 'Form', 'Html', 'Rss' );
10:
11:        // Components that we'll often use
12:        var $components = array( 'Session', 'RequestHandler' );
13:
14:        // The default language to view the statuses
15:        var $currentLang = '';
16:
17:        var $uses = array( 'TwitterRequest', 'TwitterStatus',
18:                           'TwitterTranslation', 'Language' );
19:
20:        // We cache the models
21:        var $persistModel = true;
22:
```

```
23:         function beforeFilter() {
24:
25:             $cache_get_lang = Cache::read( 'getLang' );
26:
27:             if ( empty( $cache_get_lang ) ) {
28:
29:                 // This is a site wide function
30:                 Cache::write( 'getLang', $this->Language->getLang(), 0 );
31:             }
32:
33:             $session_get_lang = $this->Session->read( "getLang" );
34:
35:             if ( empty( $session_get_lang ) ) {
36:                 $this->Session->write( "getLang", Cache::read( 'getLang' ) );
37:             }
38:
39:             // Hackish!
40:             /*
41:             if ( preg_match( '/\/rss$/', Router::url() ) ) {
42:                 $this->RequestHandler->ext = 'rss';
43:             }*/
44:
45:             $this->__langChoice();
46:         }
47:
48:         function __langChoice() {
49:
50:             // If user has selected default viewing language
51:             if ( $this->Session->read( "userLang" ) ) {
52:                 $this->currentLang = $this->Session->read( "userLang" );
53:             }
54:         }
55:
56:     }
57:     ?>
```

Starting on line 9, we employ a number of view helpers. The cache helper helps us to cache the output view. The form helper is needed for the language selection drop-down list. The HTML helper is used for various HTML tags. And last, the RSS helper deals with the output needed for RSS requests.

We have also used a number of components. The `Session` component is used to store the language the user has selected and also the data generated from the language drop-down list. `RequestHandler` automatically handles user RSS web service requests by picking the correct layout and view. The `RequestHandler` will use the RSS layout under the `layout` folder. In the view, it will use the `index.ctp` in the `/app/views/twittertwister/rss` folder. Note, however, that beginning on line 40, we have used the alternative hack to get our `/rss` URL working. As we explained earlier, we want to have an RSS URL feed that ends with `/rss`. This method tricks the `RequestHandler` into thinking that the request has an `rss` extension.

Continuing our cache theme, on line 21, we are also caching the model using the `persistModel` variable. This will help to speed up the application. The code within the `beforeFilter` just caches the data from the language drop-down list. We have commented some code at the end of the method.

The `beforeFilter` finishes with a call to the `__langChoice` private method. If a user has previously selected a language via the top navigation drop-down list, this method makes sure that all the other controllers are aware of the chosen language. We assign the session `userLang` value to the global `currentLang` value.

Summary

In this chapter, we have successfully mashed two popular online applications. We have also covered a number of Cake topics, including caching. Its worth highlighting that *caching* is quite a loose term, with different meanings in different contexts, but the end result is always to save on some resource by not repeating the same action twice.

Additionally, we added i18n and l10n. Note that we will come back to this topic in Chapter 9, where we will add different languages to data that is stored in a database.

Since we have consumed two different web services, it seems only right that we also offer one as well. Adding the RSS web service feature was quite straightforward, except for our requirement that the URLs should end with `/xml`.

There are quite a number of features you could add to this application. The following are some suggestions:

- The `view` action could be split into years/months as one page, then days, then hours. Thus, you would only list up to 60 items in any one page. You could then have URLs like `TwitterRequest/view/2008/08/03/14/00`.
- You could further the cron background process and translate all archives into all languages gradually over a period of time. This would make the archive available via links instead of a drop-down list, so it would be search engine optimization–friendly.
- This is a big one: add the ability for users to write messages in their own language and post it to Twitter in the recipient's own language.

CHAPTER 7

Unit Testing and Web Testing

As you may have inferred, *unit testing* is the practice of testing individual units of code, to check that the application code works as expected. To help developers with their testing, CakePHP 1.2 includes integrated unit testing features. In this chapter, we'll show you how to use these features. We'll start by looking at a methodology that can provide some insight into the goals of unit testing.

Getting Programming Done

David Allen's productivity book, *Getting Things Done: The Art of Stress-Free Productivity* (Penguin, 2002), has attracted a lot of interest in the past few years. The methodology, abbreviated GTD, has become especially popular among the tech crowd. It's the primary subject of Merlin Mann's 43folders.com web site and many other blogs.

The GTD methodology might be summarized with these principles:

Define what done looks like: Visualize what your end goal is. Envision what it will take to get there. Allow your mind to brainstorm how to get yourself there.

Define what doing looks like: Decide what the next physical action is that will move you closer to your goal. What can you do to move closer to completion, based on your current priorities, resources, and context?

Unit testing helps you *define what done looks like* for your code. Unit testing gives you a specific goal to aim for as you code. It requires mental exertion up-front, as you decide specifically what your code should do, and then it "signals" to you in clear terms when you've reached your goal.

Over the life of your project, unit testing also provides one other GTD-like benefit: it helps you "get it all out of your head." The GTD methodology encourages you to write down all of your "open loops"—things, large and small, that you want to do or change—to free your mind. As you free your mind of trying to remember everything that's unfinished in your life, you'll have more mental energy for creativity.

Similarly, you may have noticed that when you refactor your code, especially in a dynamic, loosely typed language like PHP, you need to keep a lot in your head. You may wonder whether writing new code will break your existing code if you forget something. For example, have you ever had to rename a variable that was strewn throughout your code? Have you ever had to change all your `<h2>` elements to `<h3>` elements? If so, then you know the

feeling. It's worrisome to refactor your code and wonder if it will still work afterwards. Unit testing provides an alert system and a security net to inform you when you've broken something and to catch you if you make a mistake. By freeing your mind from worrying whether you'll break something as you move forward, unit testing allows you to focus on the more creative aspects of your work.

Now that you have an idea about the benefits of unit testing, let's see how it works with a Cake application.

Our Case Study: An App Like In/Out

In April 2008, Jason Fried announced that his company, 37signals, uses a simple internal application called In/Out for communicating among team members (see `http://www.37signals.com/svn/posts/976`). In/Out shows what each team member is doing and what each has accomplished. Here, we'll build a similar application as our case study. We'll call it Accomplishments, and it will allow team members to create a log of tasks and projects they have accomplished. It won't have a real authentication system or security. For this example, we'll assume that it's being deployed on an intranet or otherwise secured.

Creating the Application

We'll start with a fresh installation of CakePHP 1.2, which includes integrated unit testing features, though we still need to install the unit testing framework. Figure 7-1 shows the green messages that indicate Cake is ready to use. We've set up caching, a MySQL database, and a custom security salt, so the development environment is ready to use.

For the application, we'll create an `accomplishments` table as shown in Listing 7-1.

Listing 7-1. *The accomplishments Table Schema*

```
1: CREATE TABLE `accomplishments` (
2:   `id` int(11) NOT NULL auto_increment,
3:   `team_member` varchar(30) NOT NULL,
4:   `description` varchar(140) NOT NULL,
5:   `created` datetime default NULL,
6:   `modified` datetime default NULL,
7:   PRIMARY KEY  (`id`)
8: );
```

The table includes fields for a primary key, the team member's name (this table isn't normalized), a description of up to 140 characters (Twitter style), and created and modified dates.

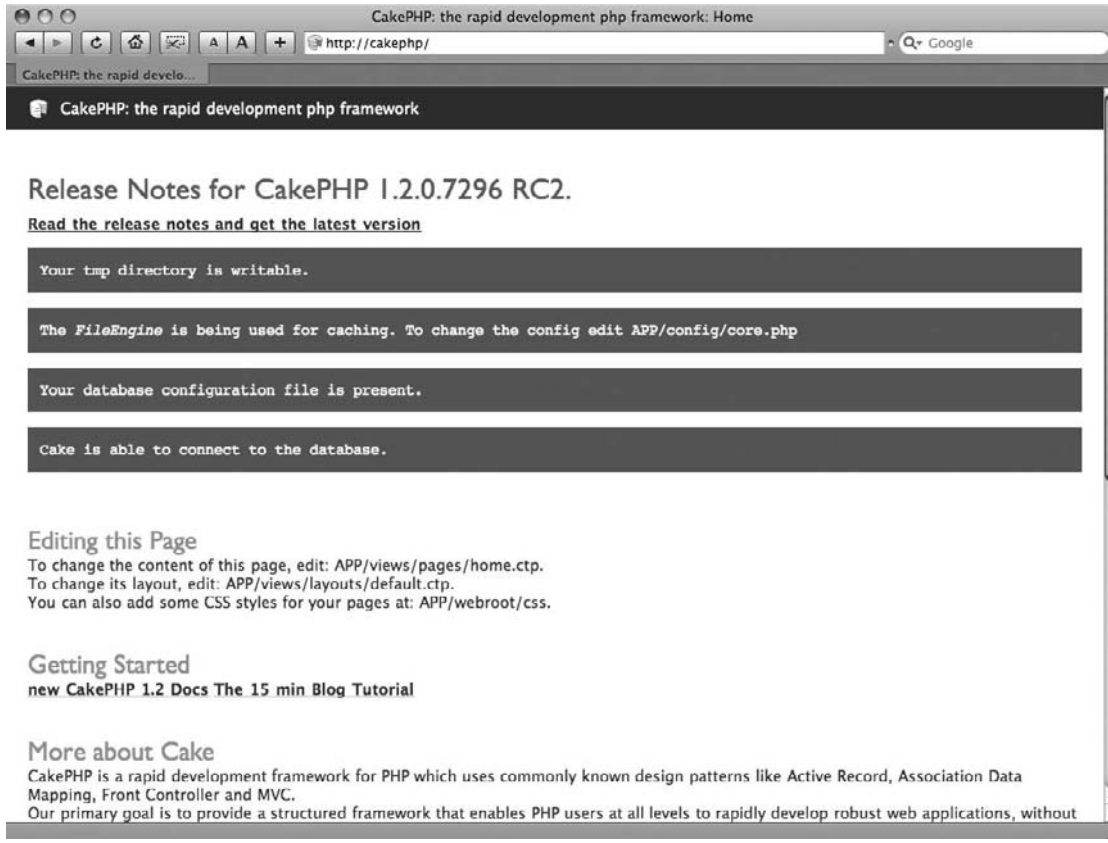

Figure 7-1. *A fresh installation of CakePHP 1.2*

A simple model will connect the application to our database. The `Accomplishment` model is shown in Listing 7-2.

Listing 7-2. *The Accomplishment Model (app/models/accomplishment.php)*

```
1: <?php
2:
3: class Accomplishment extends AppModel {
4:    var $name = 'Accomplishment';
5:
6: }
```

Next, we define an accomplishments controller, as shown in Listing 7-3.

Listing 7-3. *The AccomplishmentsController (app/controllers/accomplishment_controller.php)*

```
 1: <?php
 2:
 3: App::import('Core', 'Sanitize');
 4:
 5: class AccomplishmentsController extends AppController {
 6:    var $name = 'Accomplishments';
 7:    var $helpers = array('Time');
 8:
 9:    function index() {
10:       if (empty($this->params['url']['team_member'])) {
11:          $this->redirect(array('action' => 'login'), 302);
12:       }
13:
14:       $this->set('my_accomplishments', $this->Accomplishment->find('all',
15:          array('conditions' => array("Accomplishment.team_member" => ➥
$this->Accomplishment->data['Accomplishment']['team_member']),
16:                'order' => array('Accomplishment.created DESC'))));
17:
18:       $this->set('other_accomplishments', $this->Accomplishment->find('all',
19:          array('conditions' => array('not' => array( ➥
"Accomplishment.team_member" => $this-> ➥
Accomplishment->data['Accomplishment']['team_member'])),
20:                'order' => array('Accomplishment.created DESC'))));
21:    }
22:
23:    function login() {
24:    }
25:
26:    function add() {
27:       if (!empty($this->data)) {
28:          $this->Accomplishment->save($this->data);
29:       }
30:       $this->redirect("/accomplishments/?team_member=" . ➥
$this->data['Accomplishment']['team_member'], 302);
31:    }
32: }
```

We load the Sanitize plugin in line 3 and the time helper in line 7, both of which will be used later. For simplicity, we'll store the user's name in the URL as a `GET` variable. In lines 10 through 12, we check that the `GET` variable `team_member` is not empty. If it is empty, the user is redirected to a login page.

With `team_member` in hand, the controller calls the model to get a list of accomplishments by the user (lines 14 through 16). The model is also called for a list of the other team members' accomplishments (lines 18 through 20).

Line 23 defines a `login` action. Line 26 defines an `add` action that will allow us to insert new accomplishments into our database.

The login view, shown in Listing 7-4, prompts for a username and submits it as a URL parameter (a GET variable). Both of our views use Cake's default layout and styling.

Listing 7-4. *The Login View (app/views/accomplishments/login.ctp)*

```
1: <?= $form->create('Accomplishment', array('type' => 'get', ➥
'action' => 'index')) ?>
2:
3:    <?= $form->input('team_member', array('label' => 'Please enter your name')) ?>
4:
5:    <?= $form->submit('Login') ?>
6:
7: </form>
```

In Listing 7-5, we define a view for our index page. This is what the user will see after logging in.

Listing 7-5. *The Main View (app/views/accomplishments/index.ctp)*

```
 1: <h2>Team Accomplishments</h2>
 2: <p>What have you done?</p>
 3: <?= $form->create('Accomplishment') ?>
 4: <?= $form->input('description') ?>
 5: <?= $form->hidden('team_member', array('value' => ➥
$this->params['url']['team_member'])) ?>
 6: <?= $form->submit('Post Accomplishment') ?>
 7: </form>
 8:
 9: <h2>My Accomplishments</h2>
10:    <?php foreach($my_accomplishments as $accomplishment) : ?>
11:    <p><strong><?= Sanitize::html($accomplishment['Accomplishment'] ➥
['description']) ?></strong></p>
12:    <p><small><?= $time->niceShort($accomplishment['Accomplishment'] ➥
['created']) ?></small></p>
13:    <br/>
14:    <?php endforeach; ?>
15:
16: <h2>Others' Accomplishments</h2>
17:    <?php foreach($other_accomplishments as $accomplishment) : ?>
18:    <p><em><?= Sanitize::html($accomplishment['Accomplishment'] ➥
['team_member']) ?></em>: <strong><?= Sanitize::html( ➥
$accomplishment['Accomplishment'] ['description']) ➥
 ?></strong></p>
19:    <p><small><?= $time->niceShort($accomplishment['Accomplishment'] ➥
['created']) ?></small></p>
20:    <br/>
21:    <?php endforeach; ?>
```

In lines 3 through 7, we provide a form for entering new accomplishments. We then display a list of accomplishments by the user, in lines 9 through 14, and by the user's team members, in lines 16 through 21. You'll notice we're using the `Sanitize::html` method (lines 11 and 18) to prevent cross-site scripting (XSS) attacks. This is a good practice whenever you're echoing previously entered text back to the user.

We now have a working, admittedly simplistic, application. We can log in (see Figure 7-2) and begin using our application (see Figure 7-3).

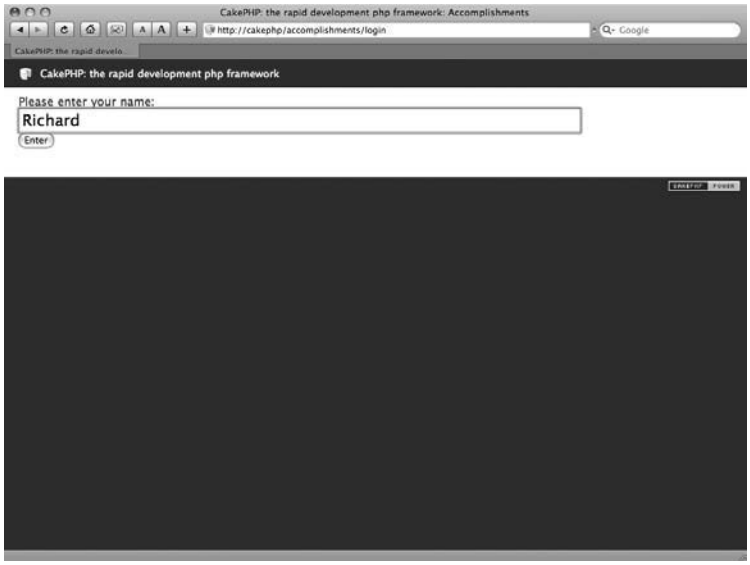

Figure 7-2. *Logging in to our new Accomplishments application*

Figure 7-3. *Posting a completed accomplishment to our application*

EDITING YOUR HOSTS FILE

When working on a development machine, we find it convenient to create a `hosts` entry, or "mock domain," for each web site we're developing. This allows us to access the web site at a convenient address like `http://cakephp/`. You'll see this mock address used throughout this chapter.

You can create a similar mock domain by editing your `hosts` file, as follows:

- On Linux or a Mac, find and open the file `/etc/hosts`. On Windows XP, open the file `C:\WINDOWS\system32\drivers\etc\hosts`.
- At the bottom of the file, add a new entry, with the IP address `127.0.0.1` on the left, and the hostname `cakephp` (or whatever hostname you want) on the right, separated by space.

Assuming you already have your web server installed and running on your local machine, you should now be able to access your web site in your browser at the address `http://cakephp/`.

If you have multiple development web sites hosted on your local machine, you'll need to create a `VirtualHost` entry for each one in your Apache configuration file.

The rest of this chapter will assume you've created a host entry like our `http://cakephp/` entry.

Adding Username Validation

With our simple application now running, let's suppose we want to add validation code to ensure that only alphabetic letters are acceptable for the username. We don't want to allow any numbers, symbols, or spaces in usernames. Listing 7-6 shows the validation code added to our `Accomplishment` model.

Listing 7-6. *Adding Username Validation to the Model (in app/models/accomplishment.php)*

```
 1: <?php
 2:
 3: class Accomplishment extends AppModel {
 4:    var $name = 'Accomplishment';
 5:
 6:    var $validate = array(
 7:       'team_member' => array(
 8:          'rule' => array('validUsername'),
 9:          'message' => "Invalid Username!"
10:       ),
11:    );
12:
13:    function validUsername($data) {
14:       return (preg_match('/^[A-Za-z]+$/', $data['team_member']));
15:    }
16: }
```

We could use Cake's built-in validation rules, but for the sake of discussion, we define our own custom method for validating the `team_member` field, in lines 6 through 11. This custom method, `validUsername()`, is defined in lines 13 through 15. It uses a regular expression to ensure

that only uppercase and lowercase letters are allowed. In the real world, this method might connect to a centralized company directory web service to look up and validate the username.

Next, we'll modify our controller to do the validating, as shown in Listing 7-7 (additions shown in bold).

Listing 7-7. *The AccomplishmentsController Updated to Validate the team_member Name (app/controllers/accomplishment_controller.php)*

```
 1: <?php
 2:
 3: App::import('Core', 'Sanitize');
 4:
 5: class AccomplishmentsController extends AppController {
 6:    var $name = 'Accomplishments';
 7:    var $helpers = array('Time');
 8:
 9:    function index() {
10:       if (empty($this->params['url']['team_member'])) {
11:          $this->redirect(array('action' => 'login'), 302);
12:       }
13:
14:       $this->Accomplishment->set(array('Accomplishment' => ➥
array('team_member' => $this->params['url']['team_member'])));
15:
16:       if (!$this->Accomplishment->validates()) {
17:          $this->Session->setFlash('Invalid username: ' . Sanitize::html( ➥
$this->Accomplishment->data['Accomplishment']['team_member']));
18:          $this->redirect(array('action' => 'login'), 302);
19:       }
20:
21:       $this->set('my_accomplishments', $this->Accomplishment->find('all',
22:          array('conditions' => array("Accomplishment.team_member" => ➥
$this->Accomplishment->data['Accomplishment']['team_member']),
23:                'order' => array('Accomplishment.created DESC'))));
24:
25:       $this->set('other_accomplishments', $this->Accomplishment->find('all',
26:          array('conditions' => array('not' => array( ➥
"Accomplishment.team_member" => ➥
$this->Accomplishment->data['Accomplishment']['team_member']')),
27:                'order' => array('Accomplishment.created DESC'))));
28:    }
29:
30:    function login() {
31:    }
32:
33:    function add() {
34:       if (!empty($this->data)) {
35:          $this->Accomplishment->save($this->data);
36:       }
```

```
37:       $this->redirect("/accomplishments/?team_member=" . ➥
$this->data['Accomplishment']['team_member'], 302);
38:    }
39: }
```

In line 14, we load the `team_member` variable into the model. In line 16, we make sure the model validates. If the model does not validate, we know it must be an invalid username (because that's the only thing we've changed), so we set an error message and redirect to the login page (lines 17 and 18). We are performing the validation explicitly in the controller, as opposed to implicitly when the model attempts to save to the database, because we need a valid username before we can display accomplishments.

After adding username validation code, what if we discover that our code is too strict? What if we want to allow usernames with numeric digits? And, as a further complication, what if we don't want numeric digits at the beginning of the username? Suppose we want to accept a username like `jared11114`, but not `14031brittany`. This could quickly become complicated as we try to create an algorithm that will accept the usernames we want to accept and reject the usernames we want to reject. This is where unit testing comes in.

Using Cake's Unit Testing Framework

Begin by visiting `http://cakephp/test.php`, the location of Cake's testing interface. You should see the error message shown in Figure 7-4, indicating that SimpleTest is not yet installed.

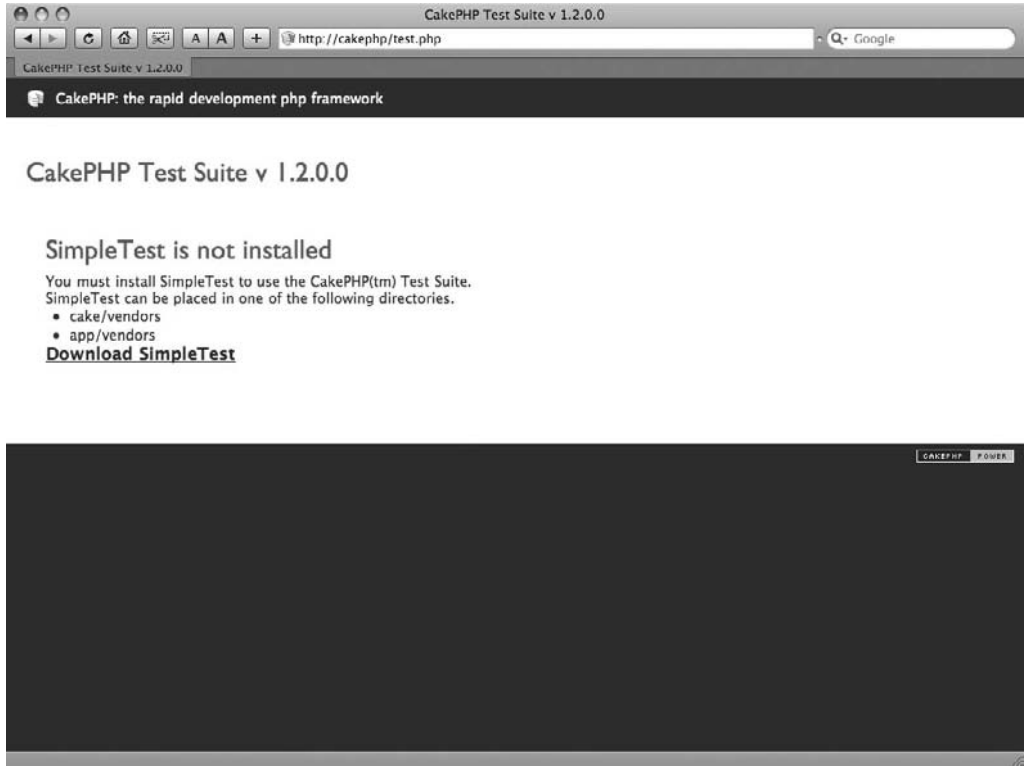

Figure 7-4. *You'll see this message if Cake's test interface, SimpleTest, is not yet installed.*

Installing SimpleTest

SimpleTest is the third-party unit testing library (created by Marcus Baker, a developer in London) that powers Cake's unit testing module. SimpleTest can be obtained from `http://www.simpletest.org`. As shown in Figure 7-5, the SimpleTest web site also offers a manual, a mailing list, and links to several articles and tutorials.

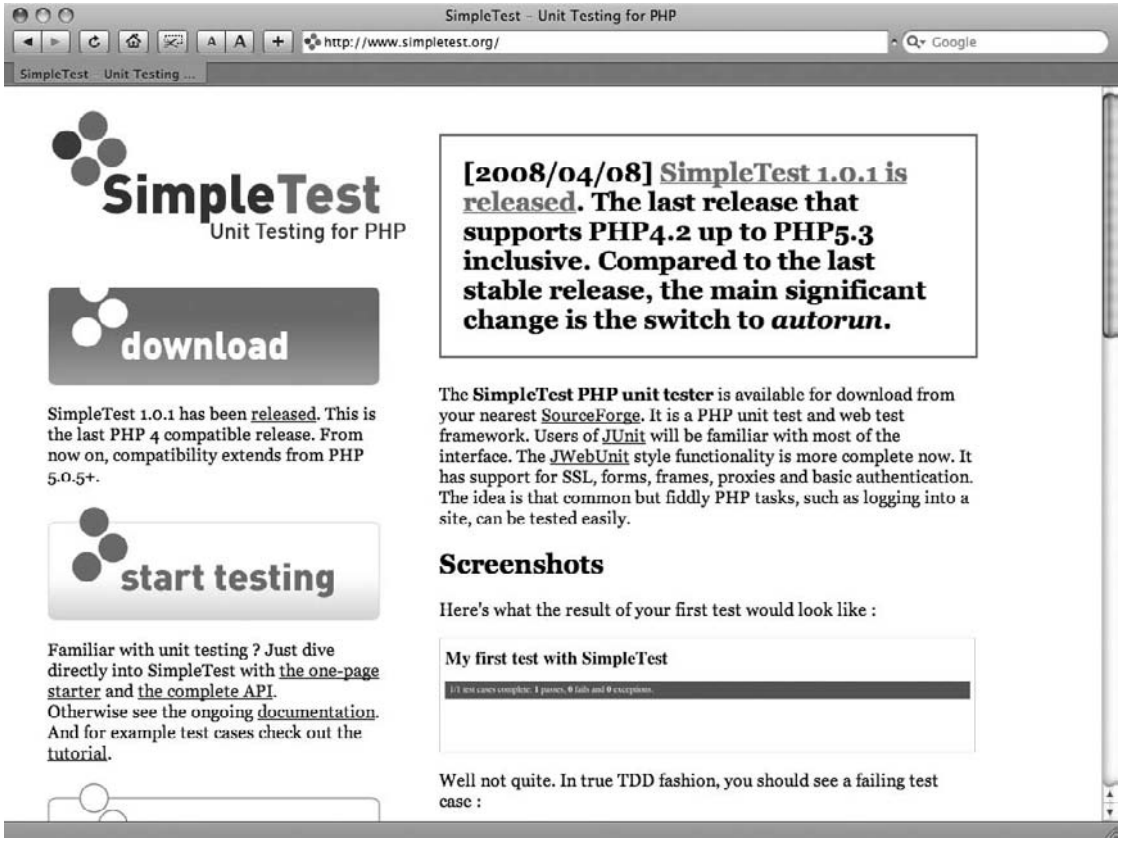

Figure 7-5. *The SimpleTest web site (http://www.simpletest.org)*

After downloading and uncompressing SimpleTest, installation is simple. Just copy the `simpletest` folder to either your `vendors` folder or your `app/vendors` folder, depending on whether you want the SimpleTest library to be available to all your applications or to only this application. In our development environment, this decision will make little difference. We'll flip a coin and copy `simpletest` to the `vendors` folder.

You'll also need to make sure the `debug` variable in `app/config/core.php` is greater than 0. Now when you revisit `http://cakephp/test.php`, you'll see a list of test groups and test cases, as shown in Figure 7-6. This means that SimpleTest has been successfully installed.

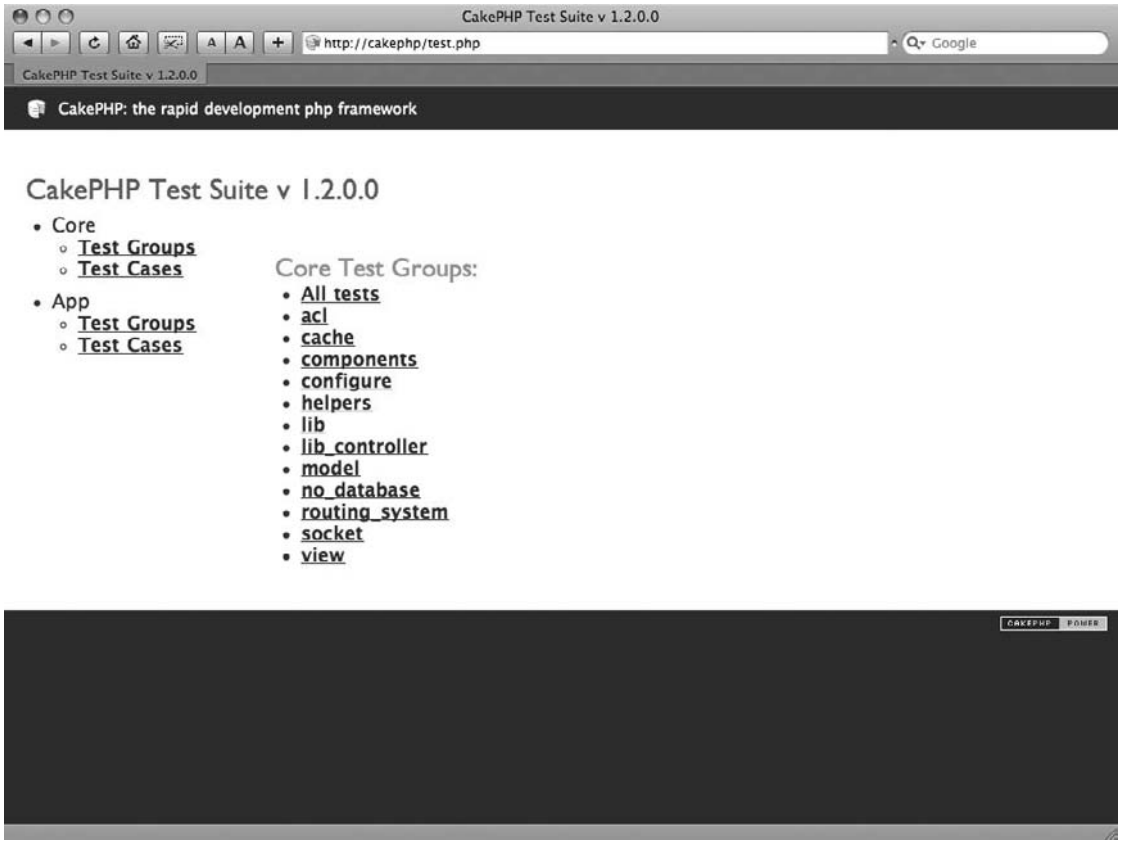

Figure 7-6. *SimpleTest is now successfully installed.*

You'll notice a list of several test groups and a longer list of test cases. These are the tests that come with every installation of Cake. They are designed to ensure the correct operation of the core Cake code. If you click the `acl` group, you may get a screen like the one shown in Figure 7-7, showing that 146 tests passed and 0 tests failed. In general terms, this means that Cake's ACL code is working correctly, according to the 146 criteria the Cake developers used to judge its correctness. This may not mean a lot to you now, but let's give the Cake developers a virtual pat on the back for creating code that passes their own tests, and we'll move along to create tests for our own code.

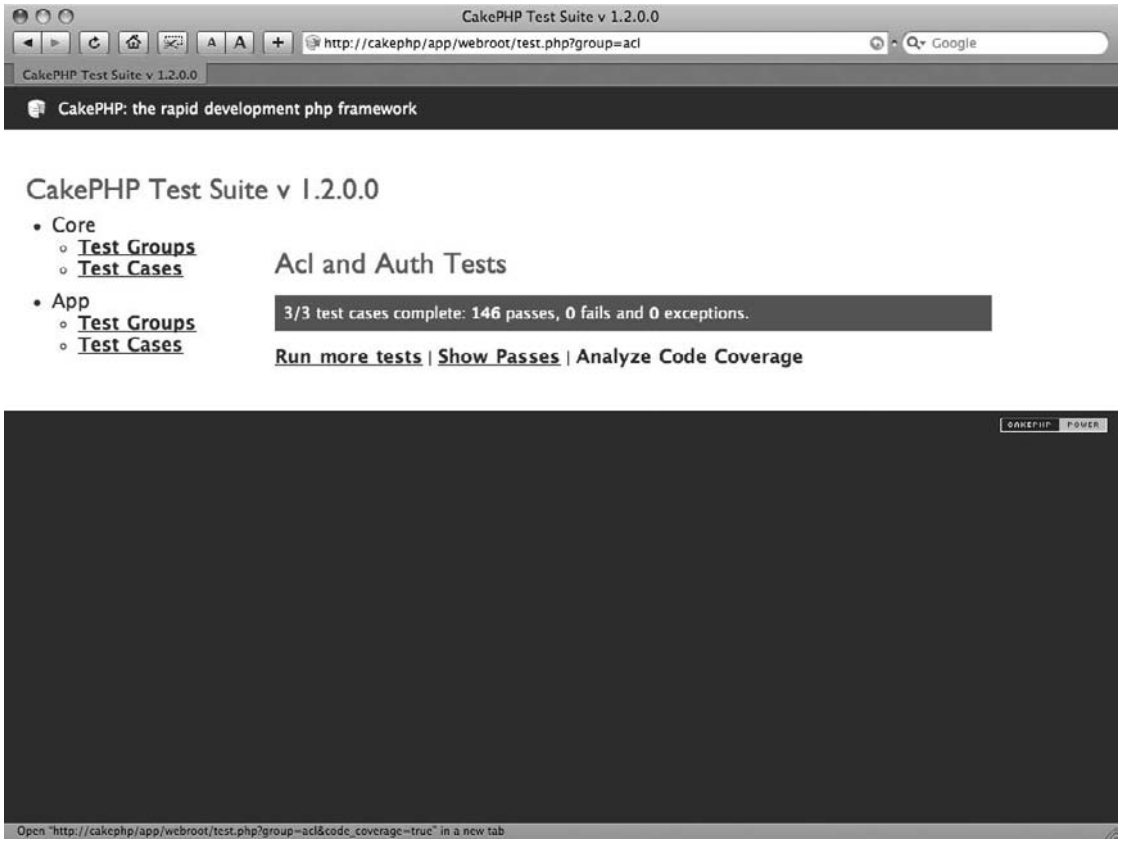

Figure 7-7. *Cake's built-in Acl and Auth tests successfully passed.*

Creating Your Own Unit Tests

The problem at hand is that we want to ensure that certain usernames are allowed in our application and certain usernames are not allowed. For example, `adam`, `MATT`, `Sean`, and `BH44` are allowed; `97devin` and `Stukdog!` are not. We want to ensure, in a systematic way, that our code accepts the good usernames and rejects the bad ones. To ensure this, we'll create unit tests.

Creating the Test File

We'll create a new test file at `app/tests/cases/models/accomplishment.test.php`. Every test file should end with the double extension `.test.php`, not just `.php`. The file name before the extension can be whatever you want. For convenience and as a matter of convention, we'll use the same name as our model. Listing 7-8 shows the `accomplishment.test.php` file.

Listing 7-8. *Our Test Module (app/tests/cases/models/accomplishment.test.php)*

```
 1: <?php
 2:
 3: App::import('Model', 'Accomplishment');
 4:
 5: class AccomplishmentTest extends Accomplishment {
 6:    var $name = 'AccomplishmentTest';
 7:    var $useDbConfig = 'test';
 8: }
 9:
10: class AccomplishmentTestCase extends CakeTestCase {
11:    var $fixtures = array( 'app.accomplishment_test' );
12:
13:    function testValidUsername() {
14:       $this->AccomplishmentTest =& new AccomplishmentTest();
15:
16:       $this->assertTrue($this->AccomplishmentTest->validUsername( ➥
array('team_member' => 'adam')));
17:       $this->assertTrue($this->AccomplishmentTest->validUsername( ➥
array('team_member' => 'MATT')));
18:       $this->assertTrue($this->AccomplishmentTest->validUsername( ➥
array('team_member' => 'Sean')));
19:       $this->assertTrue($this->AccomplishmentTest->validUsername( ➥
array('team_member' => 'BH44')));
20:       $this->assertFalse($this->AccomplishmentTest->validUsername( ➥
array('team_member' => '97devin')));
21:       $this->assertFalse($this->AccomplishmentTest->validUsername( ➥
array('team_member' => 'stukdog!')));
22:    }
23: }
```

In `accomplishment.test.php`, we define a new class called `AccomplishmentTest`, which extends the `Accomplishment` model class (lines 5 through 8). Similarly, your test classes should follow the same naming convention of `<model class name>Test` (although this is not required) and should extend your model's class.

Notice that we need to explicitly import our model (line 3). The Cake testing module doesn't make the automagic assumptions that Cake does elsewhere, to avoid muddying the testing environment. The test class also defines a new database configuration (line 7).

Next, we define a class called `AccomplishmentTestCase`, which extends `CakeTestCase` (line 10). Your test classes should follow the same naming pattern and extend `CakeTestCase`.

In the `AccomplishmentTestCase` class, we define a method called `testValidUsername`. Each method in a test case class should begin with `test` (as in line 13). The method `testValidUsername` begins by instantiating an object of the class `AccomplishmentTest` (line 14). Then six assertions are performed against the `AccomplishmentTest` object (lines 16 through 21). These assertions ensure that our `validUsername` method is behaving as expected. For example, if we pass in `adam`, `MATT`, `Sean`, or `BH44`, we expect `validUsername` to

return `true`, so we use the `assertTrue` method to ensure it (lines 16 through 19). On the other hand, if we pass in `97devin` or `Stukdog!`, we expect `validUsername` to return `false`, so we use the `assertFalse` method to ensure it (lines 20 and 21).

Creating a Test Fixture

To test a data model, as we are doing here, we need to create a test fixture. A fixture is a set of data used for testing purposes. It's called a *fixture* because it remains fixed across all tests, like the control group in a scientific experiment. Your fixture should use test data that approximates the breadth and variedness of your real data.

Test fixtures are created using PHP code that looks like SQL code. You define a multidimensional array to represent the database fields, and another array to represent the database records. The advantage of defining test data in PHP code, as opposed to in your database, is that it is less likely to be changed arbitrarily from the outside. It can also be versioned in a code-versioning system such as Subversion. Listing 7-9 shows a hypothetical test fixture, which would go in `app/tests/fixtures/accomplishment_test_fixture.php`, but we're not going to use this one in our example.

Listing 7-9. *A Hypothetical Test Fixture*

```
 1: <?php
 2:
 3: class AccomplishmentTestFixture extends CakeTestFixture {
 4:    var $name = 'AccomplishmentTest';
 5:
 6:    var $fields = array(
 7:       'id' => array('type' => 'integer', 'key' => 'primary'),
 8:       'team_member' => array('type' => 'string', 'length' => 30, 'null' ➥
=> 'false'),
 9:       'description' => array('type' => 'string', 'length' => 140, 'null' ➥
=> 'false'),
10:       'created' => 'datetime',
11:       'modified' => 'datetime'
12:       );
13:
14:    var $records = array(
15:       array('id' => 1, 'team_member' => 'Richard', 'description' => ➥
 'Wrote new unit tests for application.', 'created' => ➥
'2008-09-19 10:00:00', 'modified' => '2008-09-19 10:00:00'),
16:       array('id' => 2, 'team_member' => 'Steve', 'description' => ➥
'Prosecuted criminals.', 'created' => '2008-09-19 11:00:00', ➥
'modified' => '2008-09-19 11:00:00'),
17:       array('id' => 3, 'team_member' => 'David', 'description' => ➥
'Saved lives.', 'created' => '2008-09-19 12:00:00', 'modified' => ➥
'2008-09-19 12:00:00')
18:       );
19: }
```

Our particular unit tests don't need any data. We're simply testing the functionality of our validation method. So let's just punt and use a simpler test fixture, as shown in Listing 7-10.

Listing 7-10. *The Simpler Fixture Definition for This Example (app/tests/fixtures/accomplishment_test_fixture.php)*

```
1: <?php
2:
3: class AccomplishmentTestFixture extends CakeTestFixture {
4:    var $name = 'AccomplishmentTest';
5:    var $import = 'Accomplishment';
6:
7: }
```

Instead of defining data using PHP code, we'll simply indicate that we want the test fixture to import data from our production database table, `Accomplishment` (line 5), which we won't use anyway.

The test fixture uses the `$test` database connection defined in `app/config/database.php` if it is available. If not, it uses your production database connection, prepending the test tables with `test_` to avoid overwriting your existing tables.

Running the Tests

Now that the tests are written and our fixture is set up, we can run our unit tests. Visit `http://cakephp/test.php` and then click Test Cases. You should see a test case entitled models / Accomplishment, as shown in Figure 7-8.

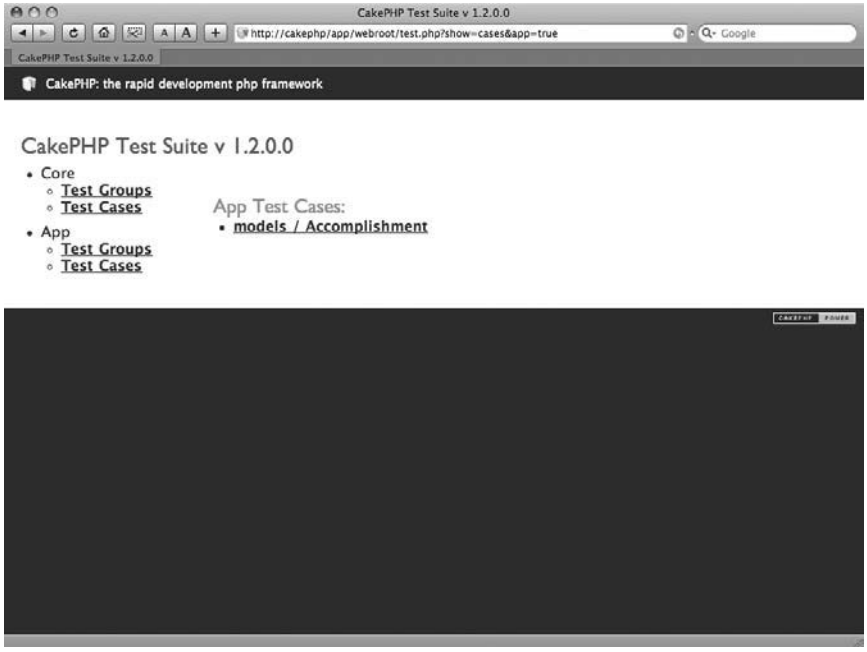

Figure 7-8. *Our test case is now listed in the testing interface.*

Now, the moment of truth! Click the models / Accomplishment link to see how our tests do. Figure 7-9 shows the result.

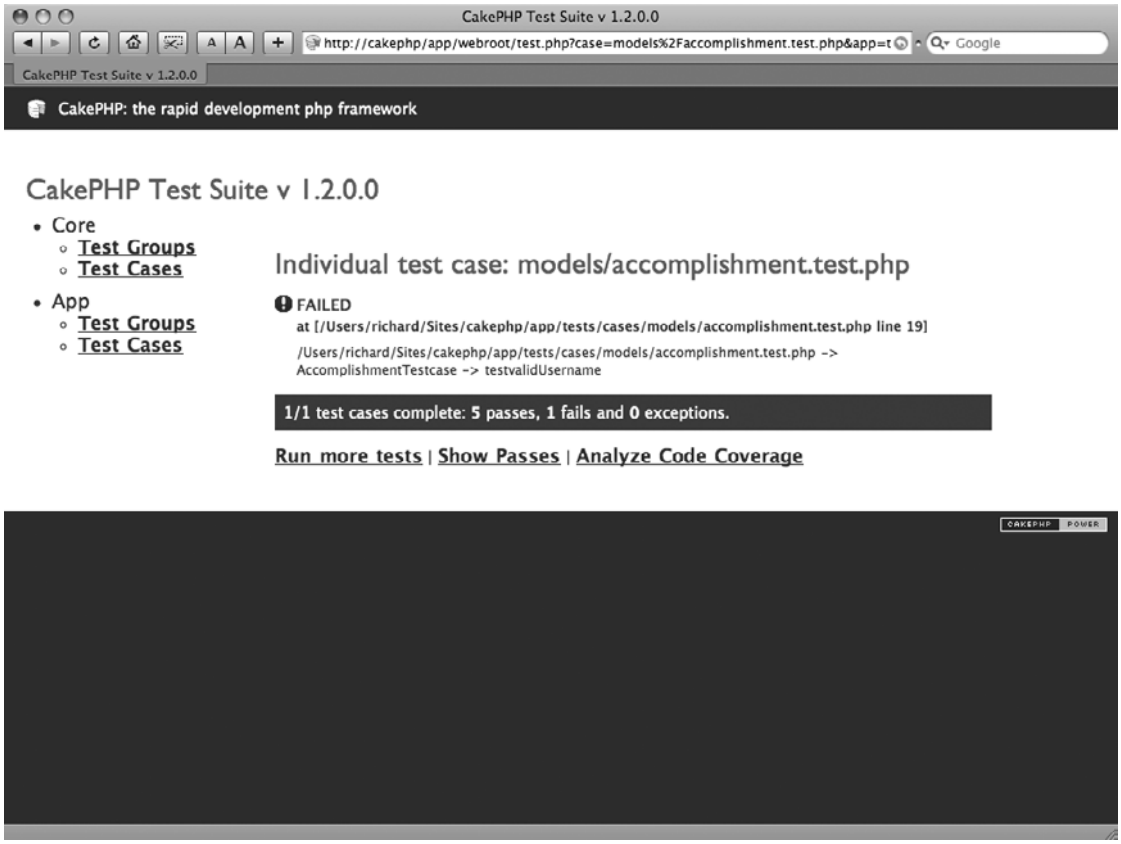

Figure 7-9. *Running our first test case*

You will see a lot of red! We failed our test, or more specifically, we failed one of the six tests, and it says the failure was in line 19 of `accomplishment.test.php`. Looking back at the code (Listing 7-8), we're not surprised that the test for the username `BH44` failed, because we haven't yet coded `validUsername` to accept numeric digits.

Go to `app/models/accomplishment.php` and change the regular expression in line 14 from `/^[A-Za-z]+$/` to `/^[A-Za-z0-9]+$/` to allow digits in the username. Listing 7-11 shows this change.

Listing 7-11. *Modified Regular Expression to Accept Numbers in the Username (in app/models/accomplishment.php)*

```
1: <?php
2:
3: class Accomplishment extends AppModel {
4:    var $name = 'Accomplishment';
5:
```

```
 6:    var $validate = array(
 7:       'team_member' => array(
 8:          'rule' => array('validUsername'),
 9:          'message' => "Invalid Username!"
10:       ),
11:    );
12:
13:    function validUsername($data) {
14:       return (preg_match('/^[A-Za-z0-9]+$/', $data['team_member']));
15:    }
16: }
```

With our new `validUsername` method, let's rerun our tests and see what we get. Refresh your browser.

As shown in Figure 7-10, once more, we see a lot of red, meaning our tests have failed again. But notice the failure is on a different line—line 20 instead of line 19. Our test for `BH44` passed, but `97devin` did not. You may have expected this, since our new regular expression wasn't sophisticated enough to reject usernames that begin with numeric digits, and our test case `97devin` slipped through.

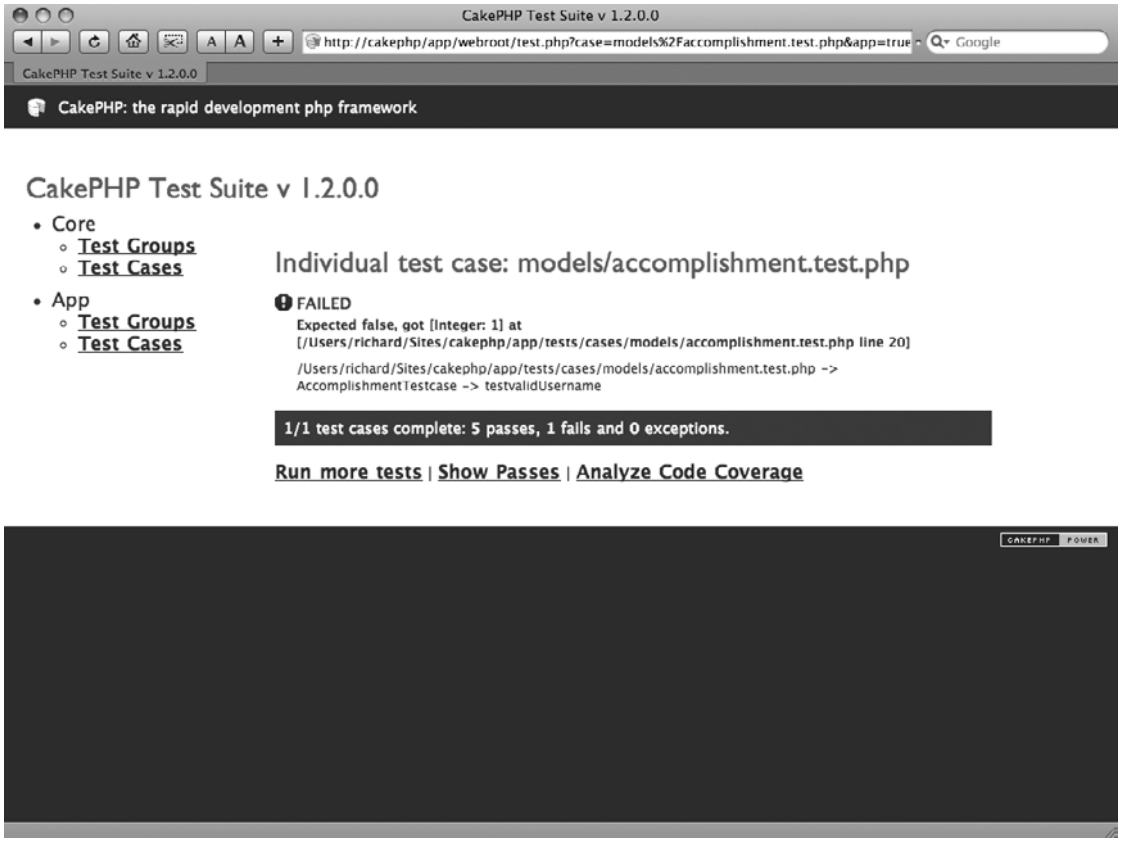

Figure 7-10. *Rerunning our test case to see if we've made progress*

Let's tweak our `validUsername()` method one more time and see if we can get it to behave as expected. We'll change the regular expression from `/^[A-Za-z0-9]+$/` to `/^[A-Za-z][A-Za-z0-9]+$/`, as shown in Listing 7-12. This requires the first character of the username to be a letter, while any subsequent character can be a letter or a number. This also implies that our username must be at least two characters long, which is fine.

Listing 7-12. *Modified Regular Expression to Reject Usernames That Begin with a Number (in apps/models/accomplishment.php)*

```
 1: <?php
 2:
 3: class Accomplishment extends AppModel {
 4:    var $name = 'Accomplishment';
 5:
 6:    var $validate = array(
 7:       'team_member' => array(
 8:          'rule' => array('validUsername'),
 9:          'message' => "Invalid Username!"
10:       ),
11:    );
12:
13:    function validUsername($data) {
14:       return (preg_match('/^[A-Za-z][A-Za-z0-9]+$/', $data['team_member']));
15:    }
16: }
```

Refresh your browser again. Good news! Now all of the tests pass. We're rewarded with a green bar stating "6 passes, 0 fails and 0 exceptions," as shown in Figure 7-11.

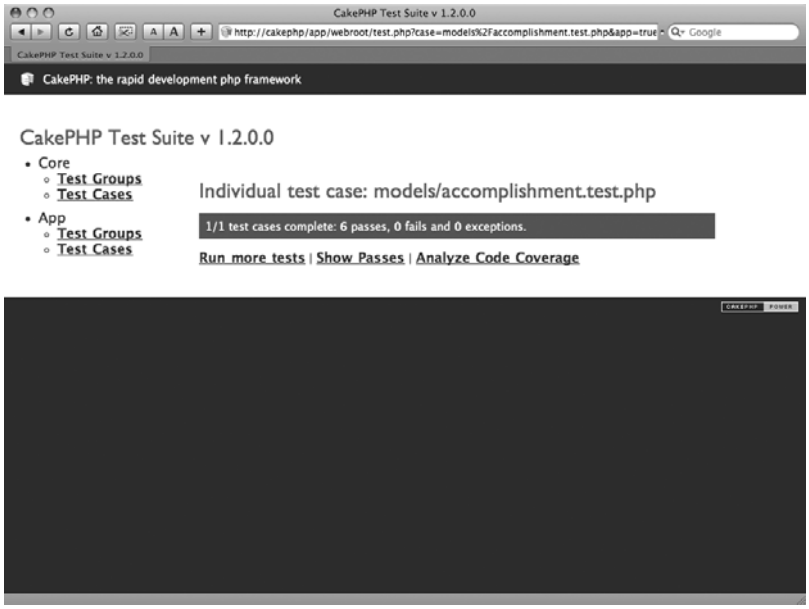

Figure 7-11. *Our tests now pass.*

Unit testing gives us confidence that our `validUsername` method now functions exactly as it should.

Using Assert Methods

Our unit tests used only two of about a dozen available `assert*` methods. Table 7-1 lists the `assert*` methods that you can use in your unit testing.

Table 7-1. *SimpleTest Assert Methods*

Method	Description
`assertClone( &$x, &$y [, $message = '%s'], $x, $y )`	Asserts that `$x` and `$y` should be clones of each other, meaning they are identical but not the same object
`assertEqual( $x, $y [, $message = '%s'] )`	Asserts that `$x` should be equal to `$y`
`assertFalse( $x [, $message = '%s'] )`	Asserts that `$x` should be false
`assertIdentical( $x, $y [, $message = '%s'] )`	Asserts that `$x` and `$y` should be identical, meaning they have the same value and are of the same type
`assertIsA( $object, $class [, $message = '%s'] )`	Asserts that `$object` should be of class `$class`
`assertNoPattern( $regex, $subject [, $message = '%s'] )`	Asserts that no portion of `$subject` should match the regular expression `$regex`
`assertNotA( $object, $class [, $message = '%s'] )`	Asserts that `$object` should not be of the class `$class`
`assertNotEqual( $x, $y [, $message = '%s'] )`	Asserts that `$x` should not be equal to `$y`
`assertNotIdentical( $x, $y [, $message = '%s'] )`	Asserts that `$x` should not be identical to `$y`, meaning it should either have a different value and/or be of a different type
`assertNotNull( $x [, $message = '%s'] )`	Asserts that `$x` should not be null
`assertNull( $x [, $message = '%s'] )`	Asserts that `$x` should be null
`assertOutsideMargin( $x, $y, $margin [, $message = '%s'] )`	Asserts that `$x` and `$y` should differ by more than `$margin`
`assertPattern($regex, $subject [, $message = '%s'] )`	Asserts that `$subject` should match the regular expression `$regex`
`assertReference( &$x, &$y [, $message = '%s'], $x, $y )`	Asserts that `$x` and `$y` should both refer to (be pointers to) the same object
`assertTrue( $x [, $message = false] )`	Asserts that `$x` should be true
`assertWithinMargin( $x, $y, $margin [, $message = '%s'] )`	Asserts that `$x` and `$y` should be within `$margin` of each other

Each `assert` method allows you to define an optional `$message` to be displayed when the assertion fails. You can optionally use the placeholder `%s` in your message to display the default error details.

Testing the Entire MVC System

The unit tests we wrote in the preceding example were for testing our model. But unit tests can also be written for testing controllers, views, plugins, and components.

The process for testing controllers, plugins, and components is similar to the process of testing models. Testing views, on the other hand, is a different beast, and we'll cover that in the next section.

Web Testing

The SimpleTest library that powers Cake's testing framework has a very cool web testing feature. It can simulate a user's navigation through a web site, allowing you to test your views and make sure your web site is working properly "from the outside." You might also call this a form of integration testing, since it can test the fitness and accuracy of the entire site, not just the individual units.

Creating Web Tests

Let's add a few web tests to our application, in `app/tests/cases/models/accomplishment.test.php`, as shown in Listing 7-13.

Listing 7-13. *The Class AccomplishmentWebTestCase (in app/tests/cases/models/accomplishment.test.php)*

```
24:
25: class AccomplishmentWebTestCase extends CakeWebTestCase {
26:    function testLoginGoodUsername() {
27:       $this->get('http://cakephp/accomplishments/');
28:       $this->setField('team_member', 'Richard');
29:       $this->click('Login');
30:       $this->assertText('Team Accomplishments');
31:       $this->assertText('My Accomplishments');
32:       $this->assertText("Others' Accomplishments");
33:    }
34:    function testLoginBadUsername() {
35:       $this->get('http://cakephp/accomplishments/');
36:       $this->setField('team_member', '97devin');
37:       $this->click('Login');
38:       $this->assertText('Invalid username');
39:    }
40: }
```

We define a class `AccomplishmentWebTestCase`, which extends `CakeWebTestCase`. The class contains two methods: `testLoginGoodUsername` will ensure that when we enter a valid username, we can log in, and `testLoginBadUsername()` will ensure that when we enter an invalid username, we are denied access.

You can probably guess what the code in Listing 7-13 does. In `testLoginGoodUsername`, we simulate a user going to the web site `http://cakephp/accomplishments/`, entering `Richard` into the `team_member` field, and clicking the Login button. We then assert that the resulting page contains three strings of text: `Team Accomplishments`, `My Accomplishments`, and `Others' Accomplishments`. The test could be more complex, but the presence of these three strings is likely to be a good indicator of whether the login was successful.

In `testLoginBadUsername`, we repeat the same simulation, this time entering the username `97devin`, which we know to be unacceptable. We then test that the resulting page contains the text `Invalid username`, which we expect when the user is unable to log in.

Let's run our tests again, by visiting the test page `http://cakephp/test.php`, to see the results. As shown in Figure 7-12, we find that ten tests passed successfully, which include our six unit tests plus the four additional web tests we've just created.

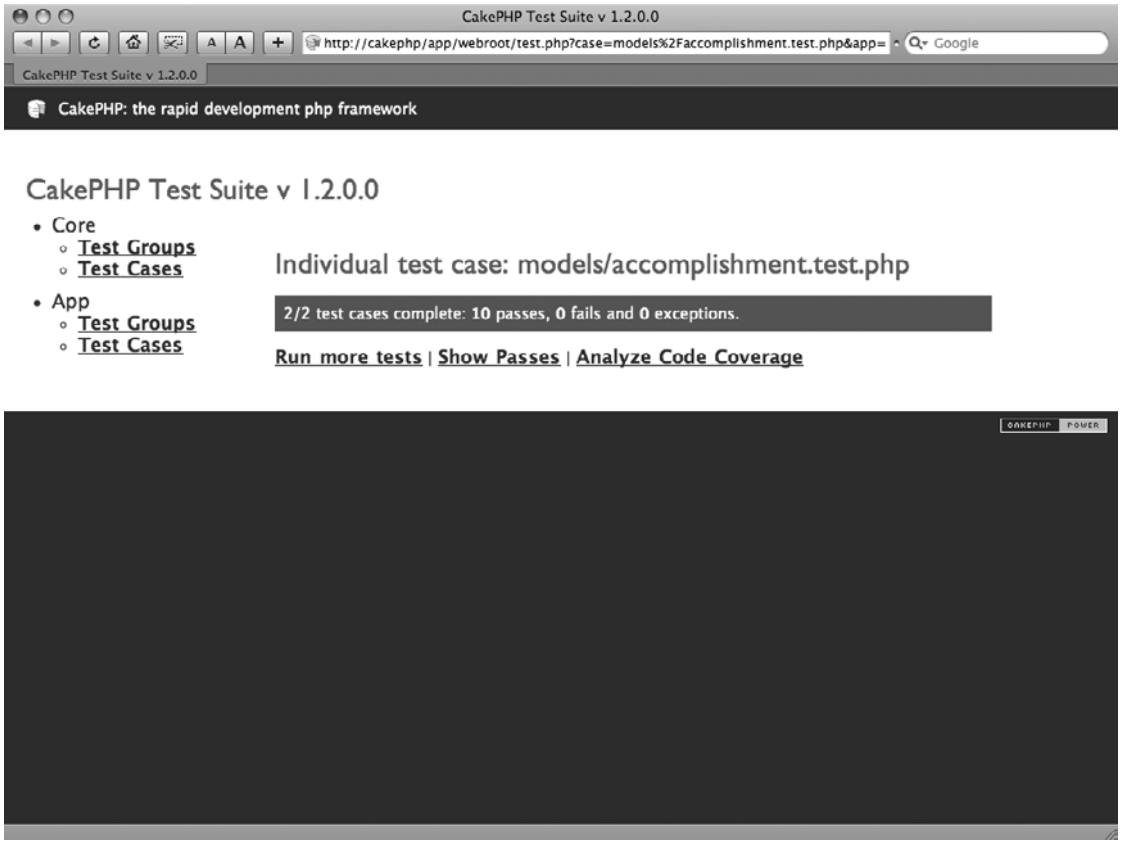

Figure 7-12. *Test results page after running our four web tests*

Web tests can be a powerful tool for ensuring that your web application is working correctly, in a holistic sense. With web tests, you can imitate the following user actions:

- Clicking links, clicking images, and submitting forms
- Clicking the back and forward buttons, and refreshing the page
- Sending and checking headers
- Sending, ignoring, and checking cookies
- Logging in using HTTP authentication
- Using a proxy

For a full list of available web testing methods, visit `http://www.simpletest.org/api/SimpleTest/WebTester/WebTestCase.html`.

Web Testing Any Application

One very cool aspect of the web testing framework, which may already be apparent to you, is that you can test any feature of any web site, not just Cake web sites or web sites you own. For example, imagine setting up tests to ensure that your legacy corporate web site is up and running, that the password protection is working, and that private areas of your site are not publicly accessible. You could set up a cron job to run the tests regularly and e-mail you when there are abnormalities.

Listing 7-14 shows a hypothetical test for ensuring that the Wikipedia article on CakePHP mentions unit testing. As of this writing, unit testing is mentioned as a feature of CakePHP in the Wikipedia article. If this reference were ever removed, the following test would fail.

Listing 7-14. *A Hypothetical Test Case for Testing an External Web Site*

```
40:    function testWikipediaArticle() {
41:       $this->addHeader("User-Agent: CakePHP/SimpleTest");
42:       $this->get('http://en.wikipedia.org/wiki/Cakephp');
43:       $this->assertText('Unit testing');
44:    }
```

■ **Caution** If you use Cake's web testing features to test web sites that you don't own, you'll want to make sure you comply with their terms of usage.

Test-Driven Development

Test-driven development is a form of agile development that requires unit tests to be written before writing application code. Writing the unit tests before the actual code requires developers to think carefully about what the application should do. It also requires thinking about development with an eye to modularity; each unit should do something distinct and uncoupled from other units. In fact, someone else on your team could be the one to write the unit tests, while you write the code, or vice versa. The ensuing communication and coordination would be healthy.

Your lines of test code could easily outnumber your application code lines if you were to test every aspect of your application. This, of course, is not practical, nor is it smart business practice. You might consider writing tests only for the most important units of your code. What code might potentially cause the most damage if it were left to chance? Write unit tests for that code first. Also, as bugs are reported and fixed, write unit tests that ensure those bugs will not resurface in the future. This sort of testing is called *regression testing*, because it prevents your code from regressing to a prior, bug-laden state.

Imagine if your home appliances were wired directly into the electrical system without using plugs and sockets. Moving a lamp to a new room would be a huge pain. You would need to cut the wire and resplice it. It's much more convenient to use plugs and sockets. This also allows a separation of concerns; that is, if a lamp stops working, you can plug it into a different socket and see if it's the lamp or the socket that failed. Likewise, when your code is modular and decoupled, you can easily move it, use it elsewhere, and test it. Unit testing encourages this sort of modularity.

Summary

In this chapter, we have accomplished the following:

- Created a simple web application like the 37signals In/Out application
- Installed the SimpleTest unit testing framework
- Demonstrated how to run the built-in unit tests for Cake
- Created custom unit tests
- Modified the application code to pass unit tests
- Created web tests and run them against a web application
- Demonstrated how web tests can be used to test external web sites
- Explained when and why you should do unit testing

Unit testing can be a powerful practice for ensuring that your application code works as you expect. It can catch errors before you release new code, and save you time and energy by freeing your mind of concern. Cheers to your newfound freedom!

CHAPTER 8

A Cake Control Panel

Let us first say that there is no such feature as a Cake Control Panel in CakePHP, which is why we are creating one in this chapter.

Once you have developed several web sites, you will start to spot common functions that clients ask for time and again. Here are just some of the features clients typically request:

- Ability to update the content themselves
- Ability to have different categories of users, and allow only certain users to use particular features of the site; for example, allow only registered users to access special promotions
- Ability to change the web site configuration themselves
- Ability to view application statistics, such as number of users, last login, and so on

If your clients haven't asked for these functions yet, you can always turn them off temporarily, or maybe increase your income by selling them these extra cool features!

Developing a control panel for yourself makes a lot of sense. The same control panel can be used for different sites. You could say it's another form of the DRY principle.

Security is a basic functionality to have in any web site and also one of the most complex. If you're going to add a back-end administration area, or a control panel as we are calling it, this will be one of the functions that will be developed first.

In this chapter, we're just going to start the development of the control panel by writing a web-based front end that will allow a user to manage user security. Feel free to use the code in this chapter as a base for your own projects. We'll use several different Cake features, including the authentication (`Auth`) and access control list (`Acl`) components, and the Acl behavior, which also uses the Tree behavior.

Application Requirements

The security management function we are building in the control panel will have the following functions:

- Ability to add, edit, delete, and view user
- Ability to add, edit, delete, and view different user groups, such as administrators and ordinary users
- Ability to deny or allow a user or group access to controller actions

The Authentication and ACL Components

We'll start by talking a little about both the `Auth` and `Acl` components, and then we'll show you how we have combined them. We will warn you that although Cake's authentication component is easy to use, the access control list component is not that straightforward. However, using it is still much easier than writing the access-control code from scratch.

The Authentication Component

The reason for the `Auth` component is simple. It controls the login process and further controls access to controller actions. You can allow or deny access to certain actions using the component's `allow` and `deny` methods, but not on a per-user basis.

What the `Auth` component doesn't do is provide more granular access control between any one user and controller actions. This is where Cake's `Acl` component comes in.

To get started with the `Auth` component, you will first need a `users` table. The one for this chapter's application is shown in Listing 8-1.

Listing 8-1. *The users Table Schema*

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL auto_increment,
  `username` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  `group_id` int(11) NOT NULL,
  PRIMARY KEY  (`id`)
)
```

The first three fields—`id`, `username`, and `password`—are mandatory for the `Auth` component. The `password` field is a hashed field. We have included the `group_id` field because we'll be adding user groups as well, so we can categorize users into groups.

The Access Control List Component

Cake's `Acl` component is a generic way to control the access rights between one entity and another entity. An entity can represent almost anything you want, but usually it will be users, user groups, and controller actions. For example, you can allow every developer access to a controller action, except for the new junior developer who just joined the team.

The security permissions between entities can be stored either in a text file in `app/config/acl.ini.php` or in a database. Here, we're just going to cover database storage, so we need to create the tables to store the permissions.

One way to create the database tables required by the `Acl` component is to use the `cake` command to do this automatically. To take this approach, in your command-line environment, go to your `app` folder and run the following command:

```
$    cake acl initdb
```

This will create three database tables:

- acos: This table holds the access control objects, which are the entities to be controlled, such as actions.
- aros: This table stores the access request objects, which are the entities that request access, such as users or groups.
- aros_acos: This is the link table between the acos and aros tables. An acos object (for example, a controller action) can be accessed by many aros objects (such as users). Conversely, an aros object (such as a user) can have access to many acos objects (such as controller actions).

If you want to create the tables manually, you can do that using the SQL statements in Listings 8-2, 8-3, and 8-4.

Listing 8-2. *The acos Acl Table Schema*

```
CREATE TABLE `acos` (
  `id` int(10) NOT NULL auto_increment,
  `parent_id` int(10) default NULL,
  `model` varchar(255) default NULL,
  `foreign_key` int(10) default NULL,
  `alias` varchar(255) default NULL,
  `lft` int(10) default NULL,
  `rght` int(10) default NULL,
  PRIMARY KEY (`id`)
)
```

Listing 8-3. *The aros Table Schema*

```
CREATE TABLE `aros` (
  `id` int(10) NOT NULL auto_increment,
  `parent_id` int(10) default NULL,
  `model` varchar(255) default NULL,
  `foreign_key` int(10) default NULL,
  `alias` varchar(255) default NULL,
  `lft` int(10) default NULL,
  `rght` int(10) default NULL,
  PRIMARY KEY (`id`)
)
```

Listing 8-4. *The aros_acos Table Schema*

```
CREATE TABLE `aros_acos` (
  `id` int(10) NOT NULL auto_increment,
  `aro_id` int(10) NOT NULL,
  `aco_id` int(10) NOT NULL,
  `_create` varchar(2) NOT NULL default '0',
  `_read` varchar(2) NOT NULL default '0',
  `_update` varchar(2) NOT NULL default '0',
  `_delete` varchar(2) NOT NULL default '0',
  PRIMARY KEY (`id`)
)
```

The `acos` and `aros` records are stored using Cake's own Tree behavior, which uses a binary tree data structure to store hierarchical data. One node (a record in our case) can have two children (records) below it, sometimes called the *left branch* and *right branch*. In turn, any one record can have a parent. Tables 8-1, 8-2, and 8-3 describe the important fields in each of these tables.

Table 8-1. *The acos Table Columns*

Column	Description
`parent_id`	ID of the parent `acos` record
`model`	Model name of the entity
`foreign_key`	Foreign key to the model record ID
`alias`	Unique name to identify the record
`lft`	ID of the left branch of the record
`rght`	ID of the right branch of the record

Table 8-2. *The aros Table Columns*

Column	Description
`parent_id`	ID of the parent `aros` record
`model`	Model name of the entity, which is normally `User` or `Group`
`foreign_key`	Foreign key to the model record ID, which is normally the ID of the user or group record
`alias`	Unique name to identify the record
`lft`	ID of the left branch of the record
`rght`	ID of the right branch of the record

Table 8-3. *The aros_acos Table Columns*

Column	Description
aro_id	Foreign key to the aros record ID
aco_id	Foreign key to the acos record ID
_create	Controls access to the add action—either 1 or 0
_read	Controls access to the index or view action—either 1 or 0
_update	Controls access to the edit action—either 1 or 0
_delete	Controls access to the remove action—either 1 or 0

Component Setup

To start using the Auth and Acl components, we need to declare their use. This is done in the global AppController file, app/app_controller.php, which is shown in Listing 8-5.

Listing 8-5. *Auth and Acl Components in the AppController Class (app/app_controller.php)*

```
 1:    class AppController extends Controller {
 2:
 3:        var $pageTitle = 'Chapter 8 - The Cake Control Panel';
 4:
 5:        var $components = array( 'Auth', 'Acl' );
 6:
 7:        function beforeFilter() {
 8:
 9:            $this->Auth->authorize = 'actions';
10:
11:            $this->Auth->authError = '  You do not have permission to access
12:                                        the page you just selected.';
13:
14:            $this->Auth->loginRedirect = array( 'controller' => ➥
'ControlPanel',
15:                                                 'action' => 'index' );
16:        }
17:
18:    }
```

We include the two components on line 5. There is also a beforeFilter, which is called before any other action. In it, we specify some Auth settings. The authorize parameter on line 9 is set to actions. This parameter controls how a user is authorized. Setting it to actions means we want the Acl component to authenticate the user for us automatically based on the entries in the users table. The authorize parameter can also be set to controller, crud, array, or object, as shown in Table 8-4.

Table 8-4. *Settings for the authorize Parameter in the Auth Component*

Values	Description
`controller`	Defines an `isAuthorized` action in your controller; the user is authenticated depending on whether the function returns `true` or `false`
`actions`	Uses the `Acl`'s `check` method to authenticate users
`crud`	Also uses the `Acl`'s `check` method to authenticate users
`array('model'=> 'name')`	Defines an `isAuthorize` (note no `d` at the end) action in another model
`object`	Uses the `isAuthorized` method in your own object to authenticate the user

The `authError` parameter on line 11 of Listing 8-5 is what gets displayed when a user does not have permission to view a controller action. The `loginRedirect` parameter on line 14 specifies the URL to redirect the user to once a user has logged in.

Control Panel Application Controllers

We will be using five different controllers in our application, as shown in Table 8-5.

Table 8-5. *The Controllers Used in Our Control Panel Application*

Controller	Description
`ActionsController`	Manages which of our controller actions will be security-managed
`GroupsController`	Manages user groups
`UsersController`	Manages users
`ControlPanelController`	Base controller for our application
`WidgetsController`	Sample controller to demonstrate the access control list security

The first three controllers are essential for the working of the application; the last two are mainly used to demonstrate our application. Now we'll look at each controller, starting with the base control panel controller.

The Control Panel Controller

The `ControlPanelController`, shown in Listing 8-6, is the base class for the application. It contains the first public page of the application.

Listing 8-6. *Control Panel Controller (app/controllers/control_panel_controller.php)*

```
 1:    <?php
 2:
 3:    class ControlPanelController extends AppController {
 4:
 5:    var $name = 'ControlPanel';
 6:    var $helpers = array('Html', 'Form');
 7:
 8:    var $uses = array( 'User' );
 9:
10:        function beforeFilter() {
11:
12:            // Public access actions
13:            $this->Auth->allow( 'welcome' );
14:        }
15:
16:        // Public welcome page of control panel
17:        function welcome() {
18:
19:            // Check if the temporary user exists
20:            $tmpUser = $this->User->findByUsername( 'temp' );
21:
22:            if ( empty( $tmpUser ) ) {
23:
24:                $this->User->create();
25:                $this->User->save( array(   'username' => 'temp',
26:                                            'password' => ➥
Security::hash( 'temp', null, true ) ) );
27:            }
28:        }
29:
30:        // Page when logged in
31:    function index() {
32:
33:    }
34:
35:    }
36:    ?>
```

On line 13, we explicitly allow access to the `welcome` action. If you use an asterisk symbol, all actions will be accessible. Within the `welcome` action itself, we set up a temporary user, since we're starting off without any entries in the database tables. From line 20 onward, if we can't find a user called `temp`, we create a user called `temp` with the password `temp`. The welcome screen of the control panel is shown in Figure 8-1.

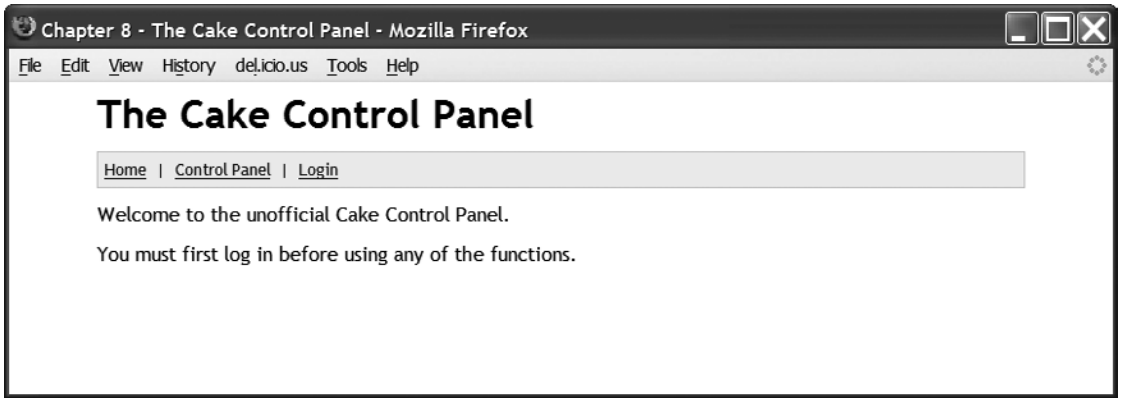

Figure 8-1. *The control panel welcome screen*

The Actions Controller

The actions controller is used to manage the access control objects—essentially, the controller actions that need security management. We can add or delete which functions within the whole application a user can or cannot access. This controller is shown in Listing 8-7.

Listing 8-7. *Actions Controller (app/controllers/actions_controller.php)*

```
<?php
class ActionsController extends AppController {

    var $name = 'Actions';
    var $helpers = array('Html', 'Form');

    function beforeFilter() {

        // We need to temporarily allow access during the setup
        $this->Auth->allow( 'index' );
    }

    // Add and remove controller actions for Aco
    function index() {

        if (!empty($this->data)) {

            $this->__processActions();
        }

        $this->__listActions();
    }

```

```
24:         // Add or delete actions
25:         function __processActions() {
26:
27:             $securityAccess = $this->data['Actions']['SecurityAccess'];
28:
29:             $inflect = new Inflector();
30:
31:             foreach ( $securityAccess as $name_pair_key => $access_selection ) {
32:
33:                 $name_pair = explode( "__", $name_pair_key );
34:
35:                 $controller = $inflect->singularize( $name_pair[0] );
36:                 $action = $name_pair[1];
37:
38:                 if ( $access_selection == 'delete' ) {
39:
40:                     $aco = new Aco();
41:
42:                     $aco_record = $aco->find( array(
43:                                         "Aco.model" => $controller,
44:                                         "Aco.alias" => $action ) );
45:
46:                     if ( !empty( $aco_record ) ) {
47:
48:                         $delete_id = $aco_record['Aco']['id'];
49:                         $this->Action->Aco->Delete( $delete_id );
50:                     }
51:                 }
52:                 elseif ( $access_selection == 'include' ) {
53:
54:                     $parent_id = '0';
55:
56:                     // Find the parent. If no parent, we create one
57:                     $aco_parent = new Aco();
58:                     $aco_parent_record = $aco_parent->find(
59:                                         array( "Aco.model" => $controller,
60:                                                "Aco.alias" => $name_pair[0] ) );
61:
62:                     if ( empty( $aco_parent_record ) ) {
63:
64:                         $aco_parent = new Aco();
65:
66:                         $aco_parent->create();
67:                         $aco_parent->save( array(   'model' => $controller,
68:                                                     'foreign_key' => '',
69:                                                     'alias' => $name_pair[0],
70:                                                     'parent_id'        => ''
71:                                                     ) );
72:
```

```
 73:                         $parent_id = $aco_parent->id;
 74:                     }
 75:                     else {
 76:
 77:                         $parent_id =  $aco_parent_record['Aco']['id'];
 78:                     }
 79:
 80:                     // Now let's create the aco record itself
 81:                     $aco = new Aco();
 82:
 83:                     $aco->create();
 84:                     $aco->save( array(  'model' => $controller,
 85:                                         'foreign_key' => '',
 86:                                         'alias' => $action,
 87:                                         'parent_id'        => $parent_id
 88:                                         ) );
 89:                 }
 90:             }
 91:         }
 92:
 93:         function __listActions() {
 94:
 95:             // Get all the actions in the controllers
 96:
 97:             $actions = array();
 98:
 99:             App::import( 'File', 'Folder' );
100:
101:             $folder = new Folder( APP.'controllers/' );
102:             $folders = $folder->find();
103:
104:             foreach( $folders as $file  ) {
105:
106:                 if ( is_file( APP.'controllers/'.$file ) ) {
107:
108:                     $file = new File( APP.'controllers/'.$file );
109:                     $file_contents = $file->read();
110:                     $file->close();
111:
112:                     // Get the controller name
113:                     $class_pattern = '/class [a-zA-Z0-9]*Controller ➥
extends AppController/';
114:                     preg_match($class_pattern, $file_contents, $matches);
115:                     $class_name_1 = str_replace( 'class ', '', $matches[0] );
116:                     $class_name = str_replace(
117:                                 'Controller extends AppController',
118:                                 '', $class_name_1 );
119:
```

```
120:                    // Get the action names
121:                    $pattern = '/function [a-zA-Z0-9]*\(/';
122:                    preg_match_all($pattern, $file_contents, $matches);
123:
124:                    // Now gather action details together
125:                    $action_group = array();
126:
127:                    $inflect = new Inflector();
128:                    $class_name_sing = $inflect->singularize( $class_name );
129:
130:                    $action_group[ 'name' ] = $class_name;
131:                    $action_group[ 'name_singular' ] = $class_name_sing;
132:                    $action_group[ 'actions' ] = $matches[0];
133:
134:                    $actions[] = $action_group;
135:                }
136:            }
137:
138:            $this->set( 'actions', $actions );
139:
140:            // Get the full list of Aco records
141:            $aco = new Aco();
142:
143:            $aco_list = $aco->find('all');
144:
145:            $result = array();
146:
147:            $inflect = new Inflector();
148:
149:            foreach ( $aco_list as $current_aco ) {
150:
151:                $key_0 = $current_aco['Aco']['model'];
152:                $key_1 = $current_aco['Aco']['alias'];
153:
154:                $result[ $key_0.'__'.$key_1 ] = $current_aco;
155:            }
156:
157:            $this->set( 'aco_list', $result );
158:        }
159:    }
160:    ?>
```

The only action in this controller is the `index` action. Figure 8-2 shows the page produced by this listing.

The Cake Control Panel

Home | Control Panel | Logout

Users | Groups | Actions | Widgets

Actions	
beforeFilter	Add: ☐
index	Add: ☐
ControlPanel	
beforeFilter	Add: ☐
welcome	Add: ☐
index	Add: ☐
dashBoard	Add: ☐
Groups	
beforeFilter	Add: ☐
add	Add: ☐
edit	Add: ☐
security	Add: ☐
index	Add: ☐
view	Add: ☐
delete	Add: ☐
Users	
beforeFilter	Add: ☐
login	Add: ☐
logout	Add: ☐
edit	Add: ☐
security	Add: ☐
index	Add: ☐
view	Add: ☐
add	Add: ☐
delete	Add: ☐
Widgets	
someAction	Add: ☐

Submit

Figure 8-2. *The index action in the actions controller*

In the Cake `Auth` component, if you set the `authorize` parameter to `crud`, Cake assumes you have all the CRUD (create, read, update, and delete) actions in your controller: `index`, `add`, `edit`, `delete`, and `view`. Cake will security-manage these actions for you using the fields `_create`, `_read`, `_update`, and `_delete` in the `aros_acos` table. However, when you set the parameter to `actions`, as we have for this application, you can security-manage any actions in your controller. Essentially, the `actions` value is more generic than the `crud` value in terms of the `Auth` security.

When the `authorize` parameter is set to `actions`, the `acos` table needs to be in a specific format, structured as follows:

```
Controller A
        Action 1
        Action 2
        Action 3
        . . .
Controller B
        Action 1
        Action 2
        Action 3
        . . .
```

There are only two levels. The top level is always occupied by the controllers, and the second levels form the controller actions that belong to the controller.

As an example, suppose we select only the last action in the control panel (Figure 8-2), by checking the someAction check box under Widgets and clicking the Submit button. This will generate the entries in the `acos` table shown in Figure 8-3.

←T→			id	parent_id	model	foreign_key	alias	lft	rght
☐	✎	✕	1	0	Widget	0	Widgets	1	4
☐	✎	✕	2	1	Widget	0	someAction	2	3

Figure 8-3. *The acos table entries after choosing the someAction action*

Now that you've seen how the actions controller works, the code in Listing 8-7 will be easier to understand. On line 10, we temporarily allow access to the `index` action; otherwise, we wouldn't be able to use it.

The `index` action is fairly simple. It has two private methods: `__listActions` and `__processActions`. The `__listActions` action lists the actions available. The code for that starts on line 93. We use Cake's `File` and `Folder` convenience classes to get all the PHP files in the controller folder, and carry out a `preg_match` operation on the content of the files, matching controller names and action names. The following is a sample of the HTML code for the check boxes in Figure 8-2:

```
<input type="checkbox"
       name="data[Actions][SecurityAccess][Widgets__someAction]"
       label=""
       div=""
       value="include"
       id="ActionsSecurityAccessWidgetsSomeAction" />
```

The name attribute in this HTML code gives you an idea of the format of the data that will be returned to the action's __processActions method via the index action.

The __processActions method is responsible for adding and deleting entries in the acos table. Again, assuming we select the last entry in Figure 8-2 (someAction), the controller data will contain the following entry:

```
Array
(
    [Actions] => Array
        (
            [SecurityAccess] => Array
                (
                    [Actions__beforeFilter] => 0
                    [Actions__index] => 0
                    [ControlPanel__beforeFilter] => 0
                    [ControlPanel__welcome] => 0
                    [ControlPanel__index] => 0
                    [ControlPanel__dashBoard] => 0
                    [Groups__beforeFilter] => 0
                    [Groups__add] => 0
                    [Groups__edit] => 0
                    [Groups__security] => 0
                    [Groups__index] => 0
                    [Groups__view] => 0
                    [Groups__delete] => 0
                    [Users__beforeFilter] => 0
                    [Users__login] => 0
                    [Users__logout] => 0
                    [Users__edit] => 0
                    [Users__security] => 0
                    [Users__index] => 0
                    [Users__view] => 0
                    [Users__add] => 0
                    [Users__delete] => 0
                    [Widgets__someAction] => include
                )

        )
)
```

Once the form has been submitted, the function __processActions on line 25 kicks in. We start on line 31 by looping through the whole action list in the data variable. Each entry is composed of a name/value pair in the following format:

```
[controller]__[action] => [security_value]
```

In our example, the value is as follows:

```
[Widgets__someAction] => include
```

This tells us that the user wants to add the `someAction` action in the widgets controller into the `acos` table, so the user can allow or deny access to certain users or groups.

On line 38, if the user has selected to delete the `acos` entry, we simply find the `acos` entry and delete it using the model's `delete` method. If the user has decided to add the entry, we first check whether the controller parent record exists. If it doesn't, we create one. If it does exist, we need the ID for the action entry. Both entries are shown in Figure 8-3.

From line 141 onward, we fetch all the `acos` entries for the view so it can display an Add or Delete check box. If an entry doesn't exist, we display an Add check box. If an entry already exists, we display a Delete check box, as shown in Figure 8-4.

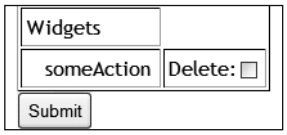

Figure 8-4. *The Delete check box option when the acos entry already exists*

To summarize, we now have one action called `someAction` in the widgets controller, which we can security-manage. If we didn't add that entry in the `acos` table, Cake's `Auth` component would always deny access to it, unless explicitly overridden by `Auth`'s `allow` method.

The Groups Controller

The groups controller is an important element in our application. In Cake's `Acl` component, we can control groups of users simply by changing the security on the group itself, rather than changing the security on every individual in that group. This is due to the fact that we can layer groups in a hierarchical fashion. For example, we can have a root administrator, and below that role, a department administrator, followed by department users.

In the groups controller, we have the ability to list groups, add groups, edit groups, delete groups, and control the security access of groups.

We start the development of the controller by using the `bake` command to bake the code for the controller, model, and views. The code listing for the `GroupsController` class is shown in Listing 8-8.

Listing 8-8. *Groups Controller (app/controllers/groups_controller.php)*

```
<?php
class GroupsController extends AppController {

    var $name = 'Groups';
    var $helpers = array('Html', 'Form');

    function beforeFilter() {

        // We need to temporarily allow access during the setup
        $this->Auth->allow( 'add', 'index' );
    }

```

```
13:         function add() {
14:
15:             // For saving group
16:            if (!empty($this->data)) {
17:                 $this->Group->create();
18:                 if ($this->Group->save($this->data)) {
19:                     $this->Session->setFlash(➥
__('The Group has been saved', true));
20:                     $this->redirect(array('action'=>'index'));
21:                 } else {
22:                     $this->Session->setFlash(__('The Group could not be
23:                                         saved. Please, try again.', true));
24:                 }
25:             }
26:
27:             // For the group parent listing
28:             $groups = $this->Group->find('list');
29:             $this->set( 'parents', $groups );
30:         }
31:
32:         function edit($id = null) {
33:
34:             if (!$id && empty($this->data)) {
35:                 $this->Session->setFlash(__('Invalid Group', true));
36:                 $this->redirect(array('action'=>'index'));
37:             }
38:             if (!empty($this->data)) {
39:                 if ($this->Group->save($this->data)) {
40:                     $this->Session->setFlash(➥
__('The Group has been saved', true));
41:                     $this->redirect(array('action'=>'index'));
42:                 } else {
43:                     $this->Session->setFlash(__('The Group could not be saved.
44:                                                 Please, try again.', true));
45:                 }
46:             }
47:             if (empty($this->data)) {
48:                 $this->data = $this->Group->read(null, $id);
49:             }
50:
51:             // For the parent group
52:             $groups = $this->Group->find('list');
53:             $this->set( 'parents', $groups );
54:         }
55:
56:         function security($id) {
57:
```

```
 58:            if (!empty($this->data)) {
 59:
 60:                // Let's get the Aro, i.e. the group
 61:                $aro_foreign_key = $this->data['Group']['id'];
 62:
 63:                $aro = new Aro();
 64:                $aro_record = $aro->findByAlias( 'Group:'.$aro_foreign_key );
 65:
 66:                $aro_alias = $aro_record['Aro']['alias'];
 67:                $aco_of_aro = $aro_record['Aco'];
 68:
 69:                // Let's run through the security selection
 70:                $sec_access = $this->data['Group']['SecurityAccess'];
 71:
 72:                $aco = new Aco();
 73:                $inflect = new Inflector();
 74:
 75:                foreach ( $sec_access as $aco_id => $access_type ) {
 76:
 77:                    $aco_record = $aco->findById( $aco_id );
 78:
 79:                    $model_plural = ➥
$inflect->pluralize( $aco_record['Aco']['model'] );
 80:
 81:                    if ( $access_type == 'allow' ) {
 82:                        $this->Acl->allow(  $aro_alias,
 83:                  $model_plural.'/'.$aco_record[ 'Aco' ][ 'alias' ], '*');
 84:                    }
 85:                    elseif ( $access_type == 'deny' ) {
 86:                        $this->Acl->deny(   $aro_alias,
 87:                  $model_plural.'/'.$aco_record[ 'Aco' ][ 'alias' ], '*');
 88:                    }
 89:                }
 90:            }
 91:
 92:            // Let's gather the aco selections available
 93:            $aco = new Aco();
 94:
 95:            // List the whole tree
 96:            $aco_tree = $aco->generateTreeList();
 97:
 98:            // Now get the details of the Aco records
 99:            $aco_records = $aco->find('all');
100:
101:            $this->set( compact( 'aco_tree', 'aco_records' ) );
102:
```

```
103:            $this->set( 'current_alias', ➥
$this->Group->name.':'.$this->Group->id );
104:
105:            if (empty($this->data)) {
106:                $this->data = $this->Group->read(null, $id);
107:            }
108:        }
109:
110:        // The following was baked
111:
112:        function index() {
113:            $this->Group->recursive = 0;
114:            $this->set('groups', $this->paginate());
115:        }
116:
117:        function view($id = null) {
118:            if (!$id) {
119:                $this->Session->setFlash(__('Invalid Group.', true));
120:                $this->redirect(array('action'=>'index'));
121:            }
122:            $this->set('group', $this->Group->read(null, $id));
123:        }
124:
125:        function delete($id = null) {
126:            if (!$id) {
127:                $this->Session->setFlash(__('Invalid id for Group', true));
128:                $this->redirect(array('action'=>'index'));
129:            }
130:            if ($this->Group->del($id)) {
131:                $this->Session->setFlash(__('Group deleted', true));
132:                $this->redirect(array('action'=>'index'));
133:            }
134:        }
135:
136:    }
137:    ?>
```

When we first ran the application, we found ourselves in a catch-22 scenario: we couldn't log in because there were no users, but we had to log in to create a user. Of course, you can manually insert entries into the database using a tool like phpMyAdmin, but we wanted to automate processes as much as possible. So initially, we allow access to actions without the user needing to log in. This is done on line 10 in Listing 8-8.

The `edit` action beginning on line 32 is mostly based on baked code, with the exception of lines 52 and 53, which generate the parent drop-down list for the current group. This `edit` action follows the same pattern as the `add` action beginning on line 13, except on line 48, where we need to read the record into the `data` controller variable to be used in the edit form.

Adding a Group

We need to add some more lines of code into the model in order to get Cake's `Acl` component to work. Listing 8-9 shows the `Group` model.

Listing 8-9. *Group Model Class (app/models/group.php)*

```
<?php
class Group extends AppModel {

    var $name = 'Group';

    var $actsAs = array('Acl'=>'requester');

    var $hasMany = array(
            'User' => array( 'className' => 'User',
                             'foreignKey' => 'group_id',
                             'dependent' => false,
                             'conditions' => '',
                             'fields' => '',
                             'order' => '',
                             'limit' => '',
                             'offset' => '',
                             'exclusive' => '',
                             'finderQuery' => '',
                             'counterQuery' => ''
            )
    );

    var $validate = array( 'title' => VALID_NOT_EMPTY );

      function afterSave($created) {

          if ( $created ) {

              // It's a creation

            $id = $this->getLastInsertID();

            $aro = new Aro();

            $aro->updateAll(    array('alias'=>'\'Group:'.$id.'\''),
                                array(  'Aro.model'=>'Group',
                                        'Aro.foreign_key'=>$id)
                                );
        }
        else {

```

```
42:                 // It's an edit; we have to update the tree
43:                 $data = $this->read();
44:                 $parent_id = $data['Group']['parent_id'];
45:
46:                 $aro = new Aro();
47:
48:                 $aro_record = $aro->findByForeignKey( $this->id );
49:                 $parent_record = $aro->findByForeignKey( $parent_id );
50:
51:                 if ( empty( $aro_record ) ) {
52:
53:                     // Orphaned child
54:                     $this->Aro->save( array(
55:                         'model' => $this->name,
56:                         'foreign_key' => $this->id,
57:                         'alias' => $this->name.':'.$this->id,
58:                         'parent_id'        => $parent_record['Aro']['id']
59:                     ) );
60:                 }
61:                 else {
62:
63:                     // Just moving nodes
64:                     $this->Aro->save( array(
65:                         'model' => $this->name,
66:                         'foreign_key' => $this->id,
67:                         'alias' => $this->name.':'.$this->id,
68:                         'parent_id'        => $parent_record['Aro']['id'],
69:                         'id'            => $aro_record['Aro']['id']
70:                     ) );
71:                 }
72:             }
73:
74:             return true;
75:         }
76:
77:         function parentNode(){
78:
79:             // This should be the alias of the parent $model::$id
80:             $data = $this->read();
81:
82:             // This needs to be unique
83:             return 'Group:'.$data['Group']['parent_id'];
84:         }
85:     }
86:     ?>
```

On line 6 in Listing 8-9, we add the Acl behavior, providing it with the `requester` parameter value so it knows the `Group` model is an `Aro` entity. This behavior automatically deals with the `acos` and `aros` entries. However, it doesn't quite do everything we need.

The Acl behavior automatically adds an entry into the `aros` table when we create a group by using the model's `save` method. Unfortunately, we need to update the entry with some additional details in order to get the `Acl` component to work properly, as follows:

- After a new `aros` record has been created via the `save` method, we need to update the `alias` field because the `Acl` component sometimes uses this field as a unique field when fetching nodes. The format of the `alias` field will be `Group:[group_id]`, as shown in line 35 in Listing 8-9.
- When an existing record is being edited and saved, we need to manually update the `parent_id` field in the `aros` table as well.

We also need to include the `parentNode` method when using the Acl behavior. It needs to return the `alias` value of the parent `aros` record. Each group can have only another group entity as a parent, so the parent `alias` value will always be in the format `Group:[group_id]`.

Line 25 in Listing 8-9 starts the `afterSave` operation. If it's a new record, we update the `alias` field with the unique string format `Group:[user_id]`. If it's an edit, we face a special scenario unique to the groups controller: if the parent was previously deleted, the children may have parent IDs pointing to nonexistent records; that is, they are orphaned. The Acl behavior does not automatically reassign orphaned children, so we must do that ourselves. The code between lines 40 and 71 in Listing 8-9 takes care of that scenario. Within that section of code, we also deal with the case of simply changing the parent; see line 64. It's worth noting that the Tree behavior will automatically adjust the other groups accordingly only when we are moving parents.

Group Security

The other important action in the groups controller is the `security` action. To explain this action, it's better to start from the group listing, as shown in the example in Figure 8-5.

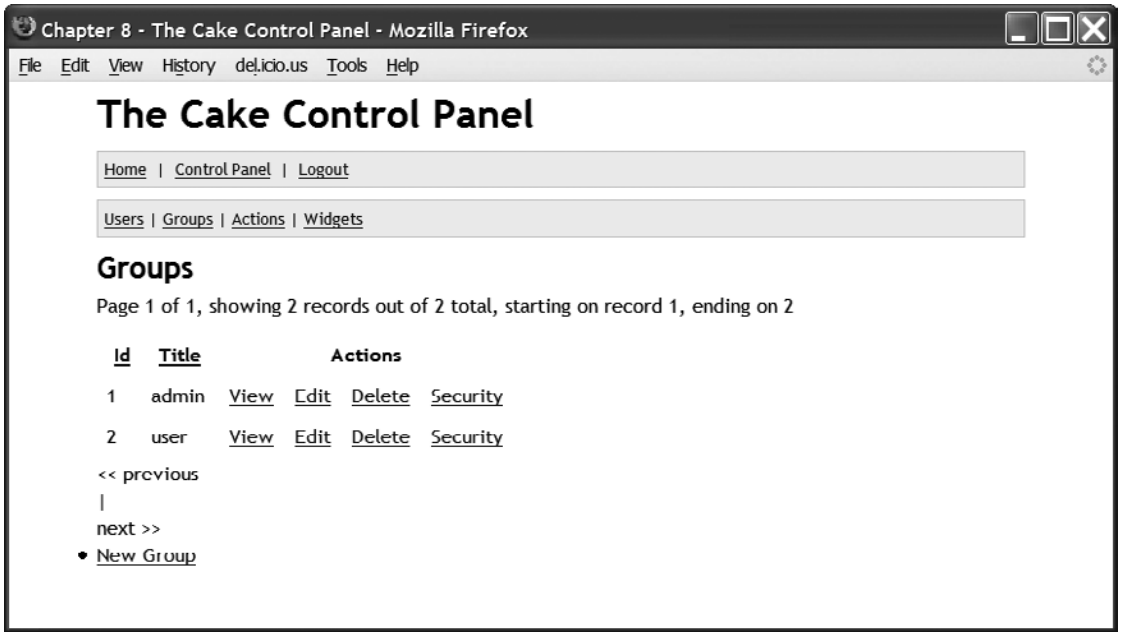

Figure 8-5. *The index action in the GroupController*

In Figure 8-5, you can see a Security link next to a group record. We simply added the new link into the baked view of the `index` action. When users click the Security link, they will be able to control the security settings relating to the group. Figure 8-6 shows the security page.

Figure 8-6. *The security action in the GroupController*

When you first bring up the security page, it will list all the access control objects that you can security-manage. In our case, it will list only the `someAction` action. This is handled from line 92 onward in Listing 8-8.

The `Aco` class has a method called `generateTreeList`, which generates the hierarchy of the tree. We also need the entire `acos` entry in order to know which actions have been granted and which ones have been denied.

Once a user has selected Allow or Deny, we process the entry from line 58 onward of Listing 8-8. It's similar to adding actions in the actions controller. We loop through the selection. If a user has selected Allow, we use this command:

```
$this->Acl->allow( [aro_alias], '[model_plaural]/[aco_alias]', '*');
```

If a user has selected Deny, we use this command:

```
$this->Acl->deny( [aro_alias], '[model_plaural]/[aco_alias]', '*');
```

In both the `allow` and `deny` methods, we are specifying a security relationship between the `aros` record and the `acos` record. The * is the standard value to use if the `authorize` parameter is set to `actions`. If it were set to `crud`, then you could specify which CRUD action to security-manage. But with the `actions` setting (sometimes called *actions mode*), the CRUD database columns must be all 1 values, all 0 values, or –1 values.

In Listing 8-8, the `index`, `view`, and `delete` actions are shown for the sake of completeness. Note that when we use the `delete` method to delete a group record, the Acl behavior also deletes the `aros` record and rearranges the tree accordingly.

The Users Controller

The users controller is essential for managing users. We will have the ability to list users, add users, edit users, delete users, and control the security access of users.

As with the groups controller, we get Cake to help us out by using the `bake` command to bake the code for the controller, model, and views. The code listing for the `UsersController` is shown in Listing 8-10.

Listing 8-10. *Users Controller (app/controllers/users_controller.php)*

```
<?php
class UsersController extends AppController {

    var $name = 'Users';
    var $helpers = array('Html', 'Form');

    function beforeFilter() {

        // We need to temporarily allow access during the setup
        $this->Auth->allow( 'index', 'security',
                            'add', 'edit', 'delete');
    }

    function login() {

    }

    function logout() {

        return $this->redirect( $this->Auth->logout() );
    }

    function edit($id = null) {

        if (!$id && empty($this->data)) {
            $this->Session->setFlash(__('Invalid User', true));
            $this->redirect(array('action'=>'index'));
        }
        if (!empty($this->data)) {

            // If no password is supplied, we don't change it
            if ( trim( $this->data['User']['password'] ) == ➥
Security::hash( '', null, true) ) {
                unset( $this->data['User']['password'] );
            }

```

```
36:                 if ($this->User->save($this->data)) {
37:                     $this->Session->setFlash( ➥
__('The User has been saved', true));
38:                     $this->redirect(array('action'=>'index'));
39:                 } else {
40:                     $this->Session->setFlash(__('The User could not be saved.
41:                                                 Please, try again.', true));
42:                 }
43:             }
44:             if (empty($this->data)) {
45:                 $this->data = $this->User->read(null, $id);
46:
47:                 // We set the password to nothing
48:                 // Passwords are only changed if you enter something
49:                 // since it's one way only!
50:                 $this->data['User']['password'] = '';
51:             }
52:
53:             // For the parent group
54:             $groups = $this->User->Group->find('list');
55:             $this->set( 'groups', $groups );
56:         }
57:
58:         function security($id) {
59:
60:             if (!empty($this->data)) {
61:
62:                 // Let's get the Aro, i.e., the group
63:                 $aro_foreign_key = $this->data['User']['id'];
64:
65:                 $aro = new Aro();
66:                 $aro_record = $aro->findByAlias( 'User:'.$aro_foreign_key );
67:
68:                 $aro_alias = $aro_record['Aro']['alias'];
69:                 $aco_of_aro = $aro_record['Aco'];
70:
71:                 // Let's run through the security selection
72:                 $sec_access = $this->data['User']['SecurityAccess'];
73:
74:                 $aco = new Aco();
75:                 $inflect = new Inflector();
76:
77:                 foreach ( $sec_access as $aco_id => $access_type ) {
78:
79:                     $aco_record = $aco->findById( $aco_id );
80:
```

```
 81:                    $model_plural = ➥
$inflect->pluralize( $aco_record['Aco']['model'] );
 82:
 83:                    if ( $access_type == 'allow' ) {
 84:                        $this->Acl->allow( $aro_alias, ➥
$model_plural.'/'.$aco_record[ 'Aco' ][ 'alias' ], '*');
 85:                    }
 86:                    elseif ( $access_type == 'deny' ) {
 87:                        $this->Acl->deny( $aro_alias, ➥
$model_plural.'/'.$aco_record[ 'Aco' ][ 'alias' ], '*');
 88:                    }
 89:                }
 90:            }
 91:
 92:            // Let's gather the aco selections available
 93:            $aco = new Aco();
 94:
 95:            // List the whole tree
 96:            $aco_tree = $aco->generateTreeList();
 97:
 98:            // Now get the details of the Aco records
 99:            $aco_records = $aco->find('all');
100:
101:            $this->set( compact( 'aco_tree', 'aco_records' ) );
102:
103:            $this->set( 'current_alias', ➥
$this->User->name.':'.$this->User->id );
104:
105:            if (empty($this->data)) {
106:                $this->data = $this->User->read(null, $id);
107:            }
108:        }
109:
110:        // The following are baked
111:
112:        function index() {
113:            $this->User->recursive = 0;
114:            $this->set('users', $this->paginate());
115:        }
116:
117:        function view($id = null) {
118:            if (!$id) {
119:                $this->Session->setFlash(__('Invalid User.', true));
120:                $this->redirect(array('action'=>'index'));
121:            }
122:            $this->set('user', $this->User->read(null, $id));
123:        }
124:
```

```
125:        function add() {
126:            if (!empty($this->data)) {
127:                $this->User->create();
128:                if ($this->User->save($this->data)) {
129:                    $this->Session->setFlash(➥
__('The User has been saved', true));
130:                    $this->redirect(array('action'=>'index'));
131:                } else {
132:                    $this->Session->setFlash(➥
__('The User could not be saved. Please, try again.', true));
133:                }
134:            }
135:            $groups = $this->User->Group->find('list');
136:            $this->set(compact('groups'));
137:        }
138:
139:        function delete($id = null) {
140:            if (!$id) {
141:                $this->Session->setFlash(__('Invalid id for User', true));
142:                //$this->redirect(array('action'=>'index'));
143:            }
144:            if ($this->User->del($id)) {
145:                $this->Session->setFlash(__('User deleted', true));
146:                $this->redirect(array('action'=>'index'));
147:            }
148:        }
149:
150:    }
151:    ?>
```

Much of the code in Listing 8-10 follows the same pattern as the groups controller. On line 10, we temporarily allow access to some actions during the initial setup of the users.

The `login` and `logout` actions, on lines 14 and 18, respectively, are just stub functions that are needed in order for the `Auth` component to automatically log the user in and out.

The `edit` action on line 23 is mostly based on baked code. There are two sections worth noting. When a user edits user details, the `password` field is set to blank on line 50. If no password has been supplied in the form, we don't change the password; we simply check whether it is empty, on line 32. Passwords are hashed, so we never have the plain text of the original password. Therefore, we need to manually empty the `password` field; otherwise, Cake will fill it with the hashed password string. When we update the user details, and the user has provided a new password, the `Auth` component will automatically hash the password provided.

User Security

The `security` action is best explained by starting from the user listing (`index` action), shown in Figure 8-7, and the security form, shown in Figure 8-8.

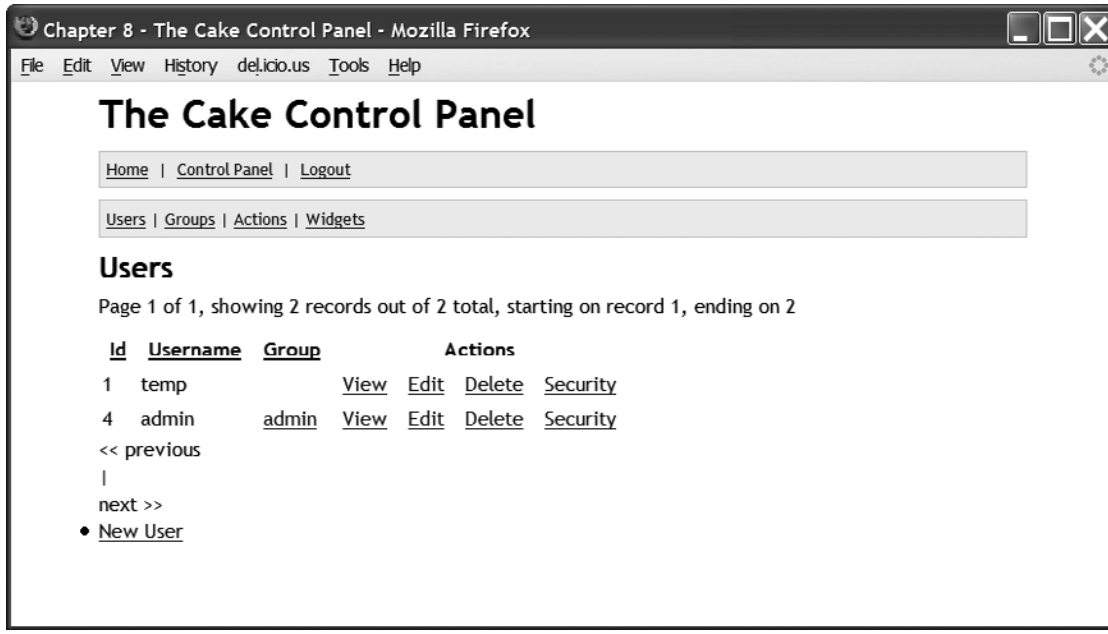

Figure 8-7. *The index action in the UserController*

Figure 8-8. *The security action in the UserController*

The security action in the users controller, on line 58 of Listing 8-10, is similar to the security action in the groups controller. We display all the acos records, which we can security-manage. We allow the administrator to set whether the user should have access or not by selecting Allow or Deny. When we submit the form, the code beginning on line 60 in Listing 8-10 takes over. We loop over the selection, and depending on whether the user selected Allow or Deny, we use the Acl component's allow or deny method.

In Listing 8-10, the index, view, add, and delete methods are shown for the sake of completeness. As with the groups controller's delete action, when we delete a user record, the action also deletes the aros record and rearranges the tree accordingly.

Adding a User

The User model code for adding a user is similar in structure to that in the Group model. This code is shown in Listing 8-11.

Listing 8-11. *User Model (app/models/user.php)*

```
 1:    <?php
 2:    class User extends AppModel {
 3:
 4:        var $name = 'User';
 5:
 6:        var $actsAs = array('Acl'=>'requester');
 7:
 8:        var $belongsTo = array(
 9:                'Group' => array('className' => 'Group',
10:                                 'foreignKey' => 'group_id',
11:                                 'conditions' => '',
12:                                 'fields' => '',
13:                                 'order' => ''
14:                )
15:        );
16:
17:          function afterSave($created) {
18:
19:              if($created) {
20:
21:                  // It's a creation
22:
23:                $id = $this->getLastInsertID();
24:
25:                $aro = new Aro();
26:
27:                $aro->updateAll( array('alias'=>'\'User:'.$id.'\''),
28:                                 array(  'Aro.model'=>'User',
29:                                         'Aro.foreign_key'=>$id)
30:                                  );
31:            }
```

```
32:             else {
33:
34:                 // It's an edit; we have to update the tree
35:                 $data = $this->read();
36:                 $parent_id = $data['User']['group_id'];
37:
38:                 $aro = new Aro();
39:
40:                 $aro_record = $aro->findByAlias( $this->name.':'.$this->id );
41:                 $parent_record = $aro->findByAlias( 'Group:'.$parent_id );
42:
43:                 if ( !empty( $aro_record ) ) {
44:
45:                     $parent_id = '0';
46:
47:                     if ( !empty( $parent_record ) ) {
48:                         $parent_id = $parent_record['Aro']['id'];
49:                     }
50:
51:                     // Just changing parents
52:                     $this->Aro->save( array(
53:                         'parent_id'        => $parent_id,
54:                         'id'            => $aro_record['Aro']['id']
55:                     ) );
56:                 }
57:             }
58:
59:             return true;
60:         }
61:
62:         function parentNode(){
63:
64:             // This should be the alias of the parent $model::$id
65:             $data = $this->read();
66:
67:             // This needs to be unique
68:             return 'Group:'.$data['User']['group_id'];
69:         }
70:
71:     }
72:     ?>
```

After a `save` operation, whether it is creating a new `aros` record or editing an existing one, we still need to update some of the `aros` record ourselves.

Line 19 starts the `afterSave` operation. If it's a new record, we update the `alias` field with the unique string format `User:[user_id]`.

Because of the hierarchical nature of `aros` records, we also need an action called `parentNode`. As in the `Group` model, this must return the unique `alias` value of the group, in the format `Group:[group_id]`.

Testing the Control Panel

Now that we have all the code and tables in place, we can run a little test to check whether our access control list is working properly. Starting from a blank database, follow these steps:

1. Make sure all the database tables are blank. See the Records column in Figure 8-9.

	Table	Action	Records
☐	acos		0
☐	aros		0
☐	aros_acos		0
☐	groups		0
☐	users		0
	5 table(s)	Sum	0

Figure 8-9. *Blank database tables*

2. Bring up the `ControlPanel/welcome` page (see Figure 8-1). This will automatically create the `temp` user entry in the `users` table and the `aros` table, as shown in Figures 8-10 and 8-11.

←T→			id	username	password	group_id
☐			1	temp	055cd666a7840de8ae29d4477585480b08d1559c	0

Figure 8-10. *The temp user in the users table*

←T→			id	parent_id	model	foreign_key	alias	lft	rght
☐			1	0	User	1	User:1	1	2

Figure 8-11. *The temp user in the aros table*

3. Log in with the `temp` user name and `temp` password.

Next, we start adding groups. Remember that we have temporarily allowed access to the `add` and `index` actions set in the `beforeFilter` method of the groups controller (see Listing 8-8, line 10). Remove this when you have all the groups and users in place.

4. Add a group called `admin` using the interface. Figure 8-12 shows the two entries in the `aros` table: one for `temp` and one for the `admin` group that you just added.

←T→			id	parent_id	model	foreign_key	alias	lft	rght
☐	✎	✕	1	0	User	1	User:1	1	2
☐	✎	✕	2	0	Group	1	Group:1	3	4

Figure 8-12. *The admin group entry in the aros table*

5. Add another group called `user`. Figure 8-13 shows the `aros` table entries.

←T→			id	parent_id	model	foreign_key	alias	lft	rght
☐	✎	✕	1	0	User	1	User:1	1	2
☐	✎	✕	2	0	Group	1	Group:1	3	4
☐	✎	✕	3	0	Group	2	Group:2	5	6

Figure 8-13. *The user group entry in the aros table*

Remember that we have temporarily allowed access to the `add` and `index` actions set in the `beforeFilter` method of the groups controller (see Listing 8-8, line 10). Remove this when you have all the groups and users in place.

6. Add the "real" `admin` user, using `admin` for the username, password, and group, as shown in Figure 8-14. Figure 8-15 shows the `aros` table entry.

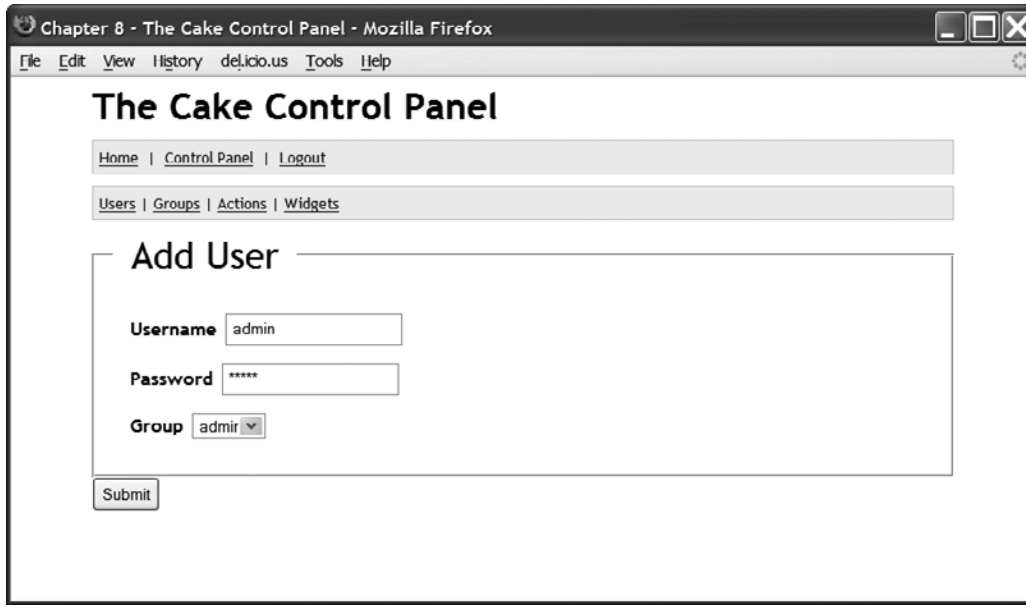

Figure 8-14. *Adding an admin user*

←T→			id	parent_id	model	foreign_key	alias	lft	rght
☐	✎	✕	1	0	User	1	User:1	1	2
☐	✎	✕	2	0	Group	1	Group:1	3	6
☐	✎	✕	3	0	Group	2	Group:2	7	8
☐	✎	✕	4	2	User	2	User:2	4	5

Figure 8-15. *The admin user entry in the aros table*

7. Log out, and then log back in using the `admin` username and password.
8. Delete the `temp` user using the Delete link.
9. Click the Actions link on the top menu, select the someAction action in the form, as shown in Figure 8-16, and then submit it.

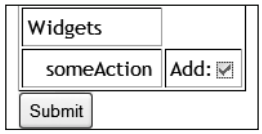

Figure 8-16. *Selecting the someAction action*

10. Click the Groups link, and select the Security link for the admin group.
11. Check the Allow selection, as shown in Figure 8-17, and submit the form.

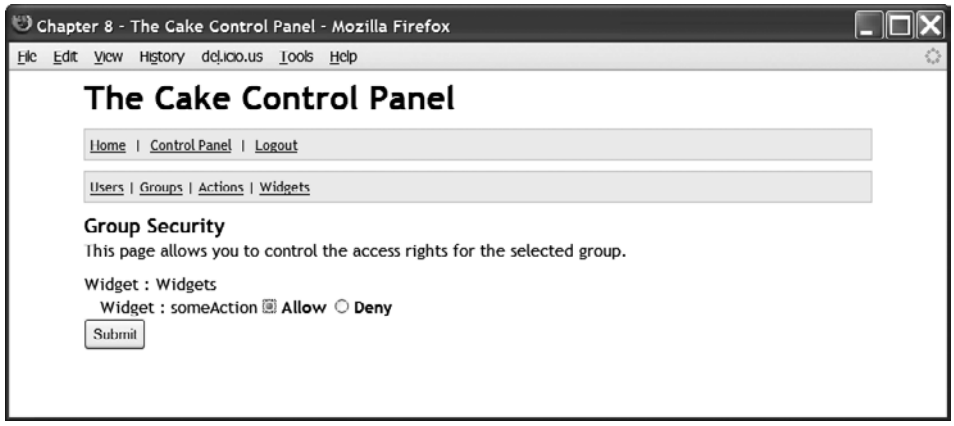

Figure 8-17. *Select the Allow check box.*

12. Click the Widgets link on the top menu. You should now have access to that page, as shown in Figure 8-18.

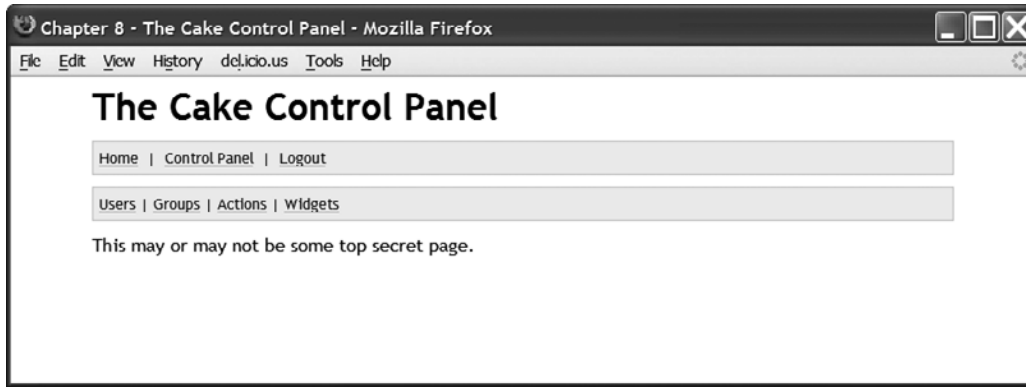

Figure 8-18. *Access allowed*

13. Return to the Security link for the `admin` group, and this time select Deny. Then submit the form.
14. Click the Widgets link again. Now you should be denied access, as shown in Figure 8-19.

Figure 8-19. *Access denied*

By controlling the group, you indirectly control the users who belong to that group. However, you can also control individual security if necessary.

Summary

In this chapter, we used the idea of a control panel as a starting point for developing basic functions common to web applications. We began that process by using Cake's `Acl` component to develop a web front end to manage user and group security.

As we said, Cake's access control list is one of its most complex elements. Much of the complexity is due to its implicit link to other elements in Cake. Using the access control list functionality involves the `Acl` and `Auth` components. You also need the Acl behavior, which in turn uses the Tree behavior. On top of that, you still need to carry out some operations yourself. Furthermore, the `Acl` component behaves differently depending on the value of the `authorize` parameter. But despite this complexity, it's one of the most flexible security systems you can use. Once the basic system is up and running, it's much easier to change and adapt your access control list for other scenarios.

There are many other functions you can add to the access control list and to the control panel as a whole. Here are some ideas:

- Add a dashboard feature in the `ControlPanelController` class to list useful statistics about the application. For example, if the control panel is used for an e-commerce application, the dashboard could list the number of products sold, number of new users, and so on.
- Improve on the access control list. You may want a feature where you can drill down from the actions listings in the top menu and find out which `aro` entry has access or no access to that particular action.
- At the moment, if you delete an action from the `acos` list, security permissions on that list will no longer exist. It would be useful to have a feature that told the user which `aros` entities currently have permissions set on the action before deleting it.
- Once you have created the essential groups and users and also removed the temporary `allow` method in the controllers, these groups and users should remain permanent; no user should be able to delete them. You could modify the application so that the `delete` method would not affect these special groups and users.

CHAPTER 9

Translating Stories

In this chapter, we'll be writing a news story application in which the news stories will be available in other languages. There will also be an admin area, which will give translators the ability to translate stories from a base language (in our case, English) to another language. To protect the admin area, we will build a simple authentication system.

In Chapter 6, we covered Cake's internationalization features and demonstrated using `.po` files to display different languages. Using `.po` files is adequate for static text. However, if the data is stored in the database, you need to employ Cake's built-in Translate behavior. You'll see how that works in this chapter.

Application Structure

To start off, we need two database tables: `stories` and `users`. Naturally, `stories` will be used to store the stories, while `users` will be the admin users who will enter stories in different languages. There will also be two controllers to go with the two tables: `StoriesController` in `/app/controllers/stories_controller.php` and `UsersController` in `/app/controllers/users_controller.php`. The `stories` table schema is shown in Listing 9-1. We will discuss the `users` table later in the chapter, in the "Logging In" section.

Listing 9-1. *The stories Table Schema*

```
CREATE TABLE `stories` (
  `id` int(11) NOT NULL auto_increment,
  `title` varchar(255) NOT NULL,
  `body` mediumtext NOT NULL,
  PRIMARY KEY  (`id`)
);
```

The two fields we need to be concerned with are `title`, which holds the story title, and `body`, which holds the actual story itself.

To use the Translate behavior, we need an extra table named `i18n`, which will be used to store the translation. The schema is shown in Listing 9-2.

Listing 9-2. *The i18n Table Schema*

```
CREATE TABLE `i18n` (
  `id` int(11) NOT NULL auto_increment,
  `locale` varchar(6) NOT NULL,
  `model` varchar(255) NOT NULL,
  `foreign_key` int(10) NOT NULL,
  `field` varchar(255) NOT NULL,
  `content` mediumtext,
  PRIMARY KEY  (`id`)
);
```

This table has the following fields:

- `locale`: This is the language locale code. We need to decide which language the translation refers to. At present, unlike for language codes, there are no ISO locale standard codes; different software uses slightly different codes. In Cake, you can see the locale codes in the file `/cake/libs/l10n.php`. For further reading, `http://www.openi18n.org` is a good place to start.
- `model`: Since there could be many different tables using the Translate behavior, we need this field to identify the correct table via the model. In our case, our model will be `Story`.
- `foreign_key`: We need the `id` field to identify which record the translation relates to.
- `field`: This is the name of the field in the database table that needs translation. In our case, it will be either `title` or `story`.
- `content`: This contains the translation text itself.

Using this `i18n` table, we map one field in one record in our `stories` table to different translations. It's a one-to-many association.

The Translate Behavior

The Translate behavior provides the model class with a number of functions that assist with language translations in our own tables. It sits between the database and the controllers, transparently dealing with language translations between the two areas.

To use the Translate behavior, we start with a simple statement we insert into our `story.php` model file:

```
var $actsAs = array( 'Translate' => array( 'title', 'body' ) );
```

■ **Note** You can add other behaviors to the model by appending the behavior name to the `actsAs` array variable. In our example, if we were going to add the Containable behavior, the `actsAs` variable would look like this: `array( 'Translate' => array( 'title', 'body' ), 'Containable' );`.

In this line, we bring in the Translate behavior, and tell it that we want the `title` and `body` fields in our `stories` table to be managed by the Translate behavior. When we say "managed," we mean that any model database function relating to the fields we specified should be transparently handled when there is a translation available. For example, when we use a `find( 'all' )` command, it will pick out the correct translation content relating to the language locale we specify.

Stories

Our application centers around the news stories to be translated. We need to add stories and translate them. Users will be able to view the stories either in their original language or a translated version. We will also add functionality to allow administrators to manage the stories.

■ **Note** To simplify our application, we have used Cake's `bake` command to generate most of the code relating to the controller and views. Additionally, we have created an admin section during the baking, so human translators can translate the stories from English into other languages.

Baking Cake

The `bake` command is a command-line tool that will generate the model, controller, and view files based on the database you created. For this chapter's application, we will use Cake's `bake` command to generate the Cake files relating to the administration of the stories. We're going to jump ahead a little by showing you the output of our `bake` session, in Listing 9-3.

Listing 9-3. *Output of the bake Session*

```
C:\chapter_9>cake\console\cake bake


Welcome to CakePHP v1.2.0.7296 RC2 Console
---------------------------------------------------------------
App : app
Path: C:/chapter_9/app
---------------------------------------------------------------
Interactive Bake Shell
---------------------------------------------------------------
[D]atabase Configuration
[M]odel
[V]iew
[C]ontroller
[P]roject
[Q]uit
What would you like to Bake? (D/M/V/C/P/Q)
> M
```

```
 19:    ---------------------------------------------------------------
 20:    Bake Model
 21:    Path: C:/chapter_9\app\models\
 22:    ---------------------------------------------------------------
 23:    Possible Models based on your current database:
 24:    1. I18n
 25:    2. Story
 26:    3. User
 27:    Enter a number from the list above, type in the name of another model, ➥
or 'q' to exit
 28:    [q] > 2
 29:    Would you like to supply validation criteria for the ➥
fields in your model? (y/n)
 30:    [y] > n
 31:    Would you like to define model associations ➥
(hasMany, hasOne, belongsTo, etc.)? (y/n)
 32:    [y] > n
 33:
 34:    ---------------------------------------------------------------
 35:    The following Model will be created:
 36:    ---------------------------------------------------------------
 37:    Name:       Story
 38:    Associations:
 39:    ---------------------------------------------------------------
 40:    Look okay? (y/n)
 41:    [y] > y
 42:
 43:    Baking model class for Story...
 44:
 45:    Creating file C:/chapter_9\app\models\story.php
 46:    Wrote C:/chapter_9\app\models\story.php
 47:    Cake test suite not installed.  Do you want to bake ➥
unit test files anyway? (y/n)
 48:    [y] > n
 49:    ---------------------------------------------------------------
 50:    Interactive Bake Shell
 51:    ---------------------------------------------------------------
 52:    [D]atabase Configuration
 53:    [M]odel
 54:    [V]iew
 55:    [C]ontroller
 56:    [P]roject
 57:    [Q]uit
 58:    What would you like to Bake? (D/M/V/C/P/Q)
 59:    > C
 60:    ---------------------------------------------------------------
 61:    Bake Controller
 62:    Path: C:/chapter_9\app\controllers\
```

```
 63:    ---------------------------------------------------------------
 64:    Possible Controllers based on your current database:
 65:    1. I18ns
 66:    2. Stories
 67:    3. Users
 68:    Enter a number from the list above, type in the name ➥
of another controller, or 'q' to exit
 69:    [q] > 2
 70:    ---------------------------------------------------------------
 71:    Baking StoriesController
 72:    ---------------------------------------------------------------
 73:    Would you like to build your controller interactively? (y/n)
 74:    [y] > y
 75:    Would you like to use scaffolding? (y/n)
 76:    [n] > n
 77:    Would you like to include some basic class methods ➥
(index(), add(), view(), edit())? (y/n)
 78:    [n] > y
 79:    Would you like to create the methods for admin routing? (y/n)
 80:    [n] > y
 81:    Would you like this controller to use other helpers ➥
besides HtmlHelper and FormHelper? (y/n)
 82:    [n] > n
 83:    Would you like this controller to use any components? (y/n)
 84:    [n] > n
 85:    Would you like to use Sessions? (y/n)
 86:    [y] > n
 87:
 88:    ---------------------------------------------------------------
 89:    The following controller will be created:
 90:    ---------------------------------------------------------------
 91:    Controller Name:  Stories
 92:    ---------------------------------------------------------------
 93:    Look okay? (y/n)
 94:    [y] > y
 95:
 96:    Creating file C:/chapter_9\app\controllers\stories_controller.php
 97:    Wrote C:/chapter_9\app\controllers\stories_controller.php
 98:    Cake test suite not installed.  Do you want to bake ➥
unit test files anyway? (y/n)
 99:    [y] > n
100:    ---------------------------------------------------------------
101:    Interactive Bake Shell
102:    ---------------------------------------------------------------
103:    [D]atabase Configuration
104:    [M]odel
105:    [V]iew
106:    [C]ontroller
```

```
    [P]roject
    [Q]uit
    What would you like to Bake? (D/M/V/C/P/Q)
    > V
    ---------------------------------------------------------------
    Bake View
    Path: C:/chapter_9\app\views\
    ---------------------------------------------------------------
    Possible Controllers based on your current database:
    1. I18ns
    2. Stories
    3. Users
    Enter a number from the list above, type in the name of ➥
another controller, or 'q' to exit
    [q] > 2
    Would you like to create some scaffolded views ➥
(index, add, view, edit) for this controller?
    NOTE: Before doing so, you'll need to create your ➥
controller and model classes (including associated models). (y/n)
    [n] > y
    Would you like to create the views for admin routing? (y/n)
    [y] > y

    Creating file C:/chapter_9\app\views\stories\index.ctp
    Wrote C:/chapter_9\app\views\stories\index.ctp

    Creating file C:/chapter_9\app\views\stories\view.ctp
    Wrote C:/chapter_9\app\views\stories\view.ctp

    Creating file C:/chapter_9\app\views\stories\add.ctp
    Wrote C:/chapter_9\app\views\stories\add.ctp

    Creating file C:/chapter_9\app\views\stories\edit.ctp
    Wrote C:/chapter_9\app\views\stories\edit.ctp

    Creating file C:/chapter_9\app\views\stories\admin_index.ctp
    Wrote C:/chapter_9\app\views\stories\admin_index.ctp

    Creating file C:/chapter_9\app\views\stories\admin_view.ctp
    Wrote C:/chapter_9\app\views\stories\admin_view.ctp

    Creating file C:/chapter_9\app\views\stories\admin_add.ctp
    Wrote C:/chapter_9\app\views\stories\admin_add.ctp

    Creating file C:/chapter_9\app\views\stories\admin_edit.ctp
    Wrote C:/chapter_9\app\views\stories\admin_edit.ctp
    ---------------------------------------------------------------

```

```
152:    View Scaffolding Complete.
153:
154:    ---------------------------------------------------------------
155:    Interactive Bake Shell
156:    ---------------------------------------------------------------
157:    [D]atabase Configuration
158:    [M]odel
159:    [V]iew
160:    [C]ontroller
161:    [P]roject
162:    [Q]uit
163:    What would you like to Bake? (D/M/V/C/P/Q)
164:    >
```

As you can see, the `bake` command first asks what we would like to bake. We actually need to bake the story model first, before the controller or view. Line 46 confirms that the model has been created.

Next, we need to create the story controller, starting on line 69. The two most important questions are on lines 77 and 79: whether we want some basic class methods and admin routing. We say yes to both questions. The `bake` command will create the code for those actions for us.

We finish off by creating the views for the stories, beginning on line 109. On lines 121 and 124, we are asked about creating scaffolding views and the views for admin routing, and we say yes.

The code we have baked is shown at the end of this chapter.

Adding Stories

Our application will start with an empty `i18n` table and an empty `stories` table. So let's start at the beginning with adding a story. Listing 9-4 shows the `admin_add` action in `/app/controllers/stories_controller.php`, which adds a story.

Listing 9-4. *Adding a Story (in /app/controllers/stories_controller.php)*

```
 1:    function admin_add() {
 2:
 3:        $this->_setI18nByLocale( 'en_us' );
 4:
 5:        if (!empty($this->data)) {
 6:            $this->Story->create();
 7:            if ($this->Story->save($this->data)) {
 8:                $this->Session->setFlash(__( 'Story saved.', true ) );
 9:                $this->redirect( array( 'controller' => 'Stories', ➥
'action' => 'index' ) );
10:            } else {
11:            }
12:        }
13:    }
```

In the `admin_add` action, the important method is the `setI10nByLocale` call on line 3. It's one of our own private methods, which is also housed within the `StoriesController`. Listing 9-5 shows this method.

Listing 9-5. *The setI10nByLocale Method (in /app/controllers/stories_controller.php)*

```
 1:    function _setI10nByLocale( $current_locale = null ) {
 2:
 3:        // Default locale is en_us
 4:        $locale = "en_us";
 5:
 6:        // Decide on session
 7:        if ( $session_locale = $this->Session->read( 'locale' ) ) {
 8:            $locale = $session_locale;
 9:        }
10:
11:        // User can override locale
12:        if ( $current_locale != '' ) {
13:            $locale = $current_locale;
14:        }
15:
16:        $this->Story->locale = $locale;
17:    }
```

The `_setI10nByLocale` method decides which locale, and thus which language, we'll be using in the Translate behavior. We start off with some default settings on line 4, and then we ask whether the locale has already been decided beforehand by checking the session on line 7. Line 16 sets the locale for the model.

In the `admin_add` action (Listing 9-4), we start off by setting the locale for the model so that the Translate behavior knows which locale we'll dealing with. The section of code after that is simply baked using Cake's `bake` command, which saves a record in the `Story` model. (See the "Baked Code" section at the end of this chapter for the actual baked version of the `admin_add` action.)

Now we enter three sample stories into our application. When the `save` command in the `Story` mode is called, we actually get a total of three different records for each story. We get the entry in the `stories` table, which we expect, but we also get two entries in the `i18n` table. The records for the three stories are shown in Figures 9-1 and 9-2.

←T→			id	title	body
☐	✎	✕	1	Manned Mars Mission	After an epic 2 year voyage, our astronauts are a ...
☐	✎	✕	2	The Large Hydron Collider	Scientists working on the world's largest atom sma...
☐	✎	✕	3	Mind to Machine Programming	Neuro engineers have developed a new technique to ...

Figure 9-1. *The stories table records*

←T→			id	locale	model	foreign_key	field	content
☐	✎	✕	1	en_us	Story	1	title	Manned Mars Mission
☐	✎	✕	2	en_us	Story	1	body	After an epic 2 year voyage, our astronauts are a ...
☐	✎	✕	3	en_us	Story	2	title	The Large Hydron Collider
☐	✎	✕	4	en_us	Story	2	body	Scientists working on the world's largest atom sma...
☐	✎	✕	5	en_us	Story	3	title	Mind to Machine Programming
☐	✎	✕	6	en_us	Story	3	body	Neuro engineers have developed a new technique to ...

Figure 9-2. *The i18n table records for the title and body*

As it stands, the records in the `i18n` table just look like unnecessary duplication. However, the Translate behavior will show its usefulness when we start inserting other languages, as you'll see a little later in the "Translating Stories" section.

Administering Stories

Most CakePHP sites have a back-end administrative area. It would be convenient to have all actions come under one specific folder so administration can be more easily managed. There is a helpful feature in Cake that helps us with this. In the `config/core.php` file, we uncomment the following line of code:

```
Configure::write('Routing.admin', 'admin');
```

With this line uncommented, Cake will map all actions in the format `admin_[`*`action_name`*`]()` to the URL `/admin/[`*`controller_name`*`]/[`*`action_name`*`]`. For example, the `admin_index` action in the `StoriesController` would be accessed via `/admin/Stories/index`. However, Cake doesn't automatically password-protect those admin actions, so we will need to do that ourselves, as discussed in the "User Authentication" section.

Translating Stories

The first task for translating stories is to list them. We will list only the stories that need translating. This makes it slightly easier for the human translator to pick out which stories need translating. This is handled by using the `admin_toTrans` method, as shown in Listing 9-6.

Listing 9-6. *Listing Stories to Translate (in /app/controllers/stories_controller.php)*

```
 1:    function admin_toTrans() {
 2:
 3:        $this->set('trans_lang', '');
 4:        $language = '';
 5:
 6:        // Once user has picked the language, here we list the stories
 7:        // that need translating
 8:        if ( !empty( $this->params['url']['language'] ) ) {
 9:            $language = $this->params['url']['language'];
10:        }
```

```
11:         elseif ( isset( $this->passedArgs[ 'language' ] ) ) {
12:             $language = $this->passedArgs[ 'language' ];
13:         }
14:
15:         if ( $language ) {
16:
17:             // The language we're using to translate the stories in
18:             $this->Story->transLanguage = $language;
19:             $this->set('trans_lang', $language);
20:
21:             // Get the language name, e.g., German
22:             $cat = $this->getLang( $language );
23:             $this->set('language', $cat[ 'language' ] );
24:
25:             // Which pagination method we're using
26:             $this->Story->usePaginate = 'paginateTranslation';
27:             $this->set('stories', $this->paginate());
28:         }
29:     }
```

In Listing 9-6, we start by initializing the variables on lines 3 and 4. Then we decide how the language code is picked out from the URL in the `if` statement on line 8. Once we know which language the user wants, we start retrieving the results using the controller `paginate` method on line 27.

Cake's `paginate` method turned out to be unsuitable for our needs. It lacks the functionality that returns only the stories that have not been translated in the language we chose—which is understandable, as that functionality is quite unique to our application. We had to override Cake's `paginate` method and write our own. We will go into the pagination a bit later, in the "Translation Pagination" section.

The story listing is shown in Figure 9-3. As you can see, each story has two links: View and Translate. Assuming we have a Japanese translator, she can view the story in English by clicking the View link. She can then click the Translate link to go to the story editing screen, as shown in Figure 9-4. A sample Translate URL link looks like this:

```
http://localhost/chapter_9/admin/stories/edit/2/trans_lang:jpn
```

Chapter 9 - Translating Stories - Mozilla Firefox

File Edit View History del.icio.us Tools Help

Future News

Read tomorrow's news today!

Home | Logout | List Stories | New Story | Translate | [en | ja | de]

Find stories which have not been translated to the language selected.

Translate Stories to Language

Language: Japanese Submit

Stories to Translate into Japanese

Page 1 of 2, showing 2 records out of 3 total, starting on record 1, ending on 2

Id	Title	Body	Actions
1	Manned Mars Mission	After an epic 2 year voyage, our astronauts are a week away from landing on the red planet.	View | Translate
2	The Large Hydron Collider	Scientists working on the world's largest atom smasher have finally cracked the secret to dark matter, but its not what they thought it was.	View | Translate

<< previous | 1 | 2 next >>

Figure 9-3. *Stories that need translating*

Chapter 9 - Translating Stories - Mozilla Firefox

File Edit View History del.icio.us Tools Help

Future News

Read tomorrow's news today!

Home | Logout | List Stories | New Story | Translate | [en | ja | de]

Translating Story to Japanese

Title Manned Mars Mission

Body

```
After an epic 2 year voyage, our
astronauts are a week away from
landing on the red planet.
```

Submit

Figure 9-4. *Story editing in the translation phase*

For this form, we have simply used the baked `admin_edit.ctp` view. However, we have added a hidden tag named `trans_lang`, which holds the language code in which we want the story to be saved.

After the Japanese translator has translated the story and saved it, the Translate behavior creates two extra records: one for the title in Japanese and another for the body in Japanese. Figures 9-5 and 9-6 show how our `i18n` table looks now. You can see how our translation functions are taking place. We have a main story stored in `stories` and the language versions of those stories held in `i18n`. Notice that our story in the `stories` table is now in Japanese as well. This is a side effect of the Translate behavior; however, it will display the correct version when the specified locale is used.

←T→			id	title	body
☐			1	ãƒ³ãƒ„ã‚¢ã‚¯ã‚»ã‚·	ã®ãª æ‹¡ãªãƒž ã«ã«ã‚ˆã‚‹ ãƒžãƒ«ãƒãƒ¡ ã‚ªãƒ–ã...
☐			2	The Large Hydron Collider	Scientists working on the world's largest atom sma...
☐			3	Mind to Machine Programming	Neuro engineers have developed a new technique to ...

Figure 9-5. *The stories table after our first translation*

←T→			id	locale	model	foreign_key	field	content
☐			1	en_us	Story	1	title	Manned Mars Mission
☐			2	en_us	Story	1	body	After an epic 2 year voyage, our astronauts are a ...
☐			3	en_us	Story	2	title	The Large Hydron Collider
☐			4	en_us	Story	2	body	Scientists working on the world's largest atom sma...
☐			5	en_us	Story	3	title	Mind to Machine Programming
☐			6	en_us	Story	3	body	Neuro engineers have developed a new technique to ...
☐			7	jpn	Story	1	title	ãƒ³ãƒ„ã‚¢ã‚¯ã‚»ã‚·
☐			8	jpn	Story	1	body	ã®ãª æ‹¡ãªãƒž ã«ã«ã‚ˆã‚‹ ãƒžãƒ«ãƒãƒ¡ ã‚ªãƒ–ã...

Figure 9-6. *The i18n table after our first translation*

Viewing Stories

When a user visits the home page, we list the stories in the language that is automatically selected by the Translate behavior. On the top navigation bar, we inserted three links that allow us to view the stories if the translated stories are available. The code for these links is shown in Listing 9-7. They belong in the `default.ctp` layout (`/app/views/layouts/default.ctp`).

Listing 9-7. *Links to Change the Locale (/app/views/layouts/default.ctp)*

```
<?php
    echo $html->link( 'en', '/Stories/changeLocale/locale:en_us' );
    echo ' | ';
    echo $html->link( 'ja', '/Stories/changeLocale/locale:jpn' );
    echo ' | ';
    echo $html->link( 'de', '/Stories/changeLocale/locale:deu' );
?>
```

As you can see in Listing 9-7, the Cake `link` methods all point to the `changeLocale` action in the `StoriesController`. The named variable `locale` is picked up via two methods in the controller: `beforeFilter` and `changeLocale` itself, both shown in Listing 9-8.

Listing 9-8. *StoriesController Methods to Change the Locale (in /app/controllers/stories_controller.php)*

```
function beforeFilter() {

    // The locale can be set by any action
    if ( isset( $this->passedArgs[ 'locale' ] ) ) {
        $this->Session->write( 'locale', $this->passedArgs[ 'locale' ] );
    }

    parent::beforeFilter();
}

function changeLocale() {

    // The locale session var is actually changed in the App beforeFilter

    // Redirect back to calling page
    $this->redirect( $this->referer() );
}
```

In the `beforeFilter` method in Listing 9-8, we set the locale in the session so it can be used by other methods. The `changeLocale` method simply redirects users back to the page where they clicked the link. We have used the `beforeFilter` method because we may want to change the locale later on via other links.

Now when we click the ja link on the top navigation, we just get the Japanese translated stories, as shown in Figure 9-7.

Figure 9-7. *Viewing Japanese stories*

Deleting Stories

Deleting stories is quite a simple affair. When the user clicks the Delete link relating to a story, the story record in the `stories` table and the associated records in the `i18n` table are deleted. This includes all the translations as well. The `delete` action code is shown in Listing 9-9.

Listing 9-9. *The delete Action (/app/controllers/stories_controllers.php)*

```
function admin_delete($id = null) {

    if (!$id) {
        $this->Session->setFlash(__( 'Invalid Story.', true ) );
    }

    if ($this->Story->del($id)) {
        $this->Session->setFlash(__( 'Story deleted.', true ) );
    }

    $this->redirect( array( 'controller' => 'Stories', 'action' => 'index' ) );
}
```

Listing Stories

Listing of the stories is a basic function. Users need to view stories that interest them. The people who manage and translate the stories also need to list the stories in the database.

We have written two story listing versions. One version will list the stories in the specified locale for public viewing, and the other will list the stories in the admin section.

The controller action for the public story listing is shown in Listing 9-10.

Listing 9-10. *Listing Stories for Public Viewing (/app/controllers/stories_controller.php)*

```
1:    function viewAllStories() {
2:
3:            $this->_setI1OnByLocale();
4:
5:            $this->Story->recursive = 0;
6:            $this->set('stories', $this->paginate());
7:    }
```

Line 3 sets the locale. Since we want only the stories, we turn off any fetching of associative model data in line 5. Line 6 is the workhorse of the action; it fetches the stories using the standard `paginate` method.

The corresponding view that goes with the `viewAllStories` action is shown in Listing 9-11.

We start off with line 4, which returns pagination information. Lines 8 to 12 simply loop through the stories and display the title and the body in full. We complete the view by providing pagination previous and next links, starting on line 18.

Listing 9-11. *Listing the Stories (in app/views/stories/view_all_stories.ctp)*

```
 1:    <div>
 2:
 3:        <?php
 4:            echo $paginator->counter( array( 'format' => __('Page %page% of ➥
%pages%, showing %current% records out of %count% total, starting on record ➥
%start%, ending on %end%', true ) ));
 5:        ?>
 6:
 7:        <?php
 8:            foreach ($stories as $story) {
 9:
10:                echo '<h2>'.$story['Story']['title'].'</h2>';
11:                echo '<p>'.$story['Story']['body'].'</p>';
12:            }
13:        ?>
14:
15:    </div>
16:
17:    <div class="paging">
18:    <?php echo $paginator->prev('<< '.__('previous', true), array(), ➥
null, array('class'=>'disabled'));?>
19:     |     <?php echo $paginator->numbers();?>
20:    <?php echo $paginator->next(__('next', true).' >>', array(), ➥
null, array('class'=>'disabled'));?>
21:    </div>
```

A typical listing of the stories is shown in Figure 9-8. Just to illustrate Cake's pagination function, we have specified two stories per page.

Figure 9-8. *Listing stories in the default language*

The listing of the stories in the admin section is very similar to the public listing. In the admin section, we list all the stories in the base language (English, in our case). Additionally, we adopt a traditional table format listing, with each record occupying a single record. There will also be an Actions column, which will contain View, Edit, and Delete links. The code for this controller action is shown in Listing 9-12.

Listing 9-12. *Admin Story Listing (in app/controllers/stories_controller.php)*

```
1:    function admin_index() {
2:
3:        $this->_setI1OnByLocale( 'en_us' );
4:
5:        $this->Story->recursive = 0;
6:        $this->set('stories', $this->paginate());
7:    }
```

The difference between this listing and the public listing (Listing 9-10) is in the setting of the locale on line 3. In the admin version, we specifically set the locale to the US English.

The corresponding view is quite straightforward and is composed of two parts. The first part is the view file itself (`/app/views/stories/admin_index.ctp`). This file just contains the following lines:

```
<?php
    echo $this->element( 'admin_list_stories' );
?>
```

The element that it points to is shown in Listing 9-13.

Listing 9-13. *The Admin Story Listing Element (/app/views/elements/admin_list_stories.ctp)*

```
 1:    <div>
 2:
 3:        <h2><?php __('Stories');?></h2>
 4:        <p>
 5:            <?php
 6:                echo $paginator->counter( array('format' => ➥
__('Page %page% of %pages%, showing %current% records ➥
out of %count% total, starting on record %start%, ending on %end%', ➥
true)));
 7:            ?>
 8:        </p>
 9:
10:        <table class="list_stories" cellpadding="0" cellspacing="0">
11:
12:            <tr>
13:                <th><?php echo $paginator->sort('id');?></th>
14:                <th><?php echo $paginator->sort('title');?></th>
15:                <th><?php echo $paginator->sort('body');?></th>
16:                <th class="actions"><?php __('Actions');?></th>
17:            </tr>
18:
```

```
19:             <?php
20:                 $i = 0;
21:                 foreach ($stories as $story):
22:                     $class = null;
23:                     if ($i++ % 2 == 0) {
24:                         $class = ' class="altrow"';
25:                     }
26:             ?>
27:                 <tr<?php echo $class;?>>
28:
29:                     <td>
30:                         <?php echo $story['Story']['id']; ?>
31:                     </td>
32:
33:                     <td>
34:                         <?php echo $story['Story']['title']; ?>
35:                     </td>
36:
37:                     <td>
38:                         <?php echo $story['Story']['body']; ?>
39:                     </td>
40:
41:                     <td class="actions">
42:                         <?php echo $html->link( ➥
__('View', true), array( 'admin' => '', 'action' => 'view', ➥
$story['Story']['id'] ), array( 'target' => '_blank' ) ); ?>
43:                         <?php echo ' | '; ?>
44:                         <?php echo $html->link(__('Edit', true), ➥
array('action'=>'edit', $story['Story']['id'])); ?>
45:                         <?php echo ' | '; ?>
46:                         <?php echo $html->link( ➥
__('Delete', true), array('action'=>'delete', $story['Story']['id']), ➥
null, sprintf(__('Are you sure you want to delete # %s?', true), ➥
$story['Story']['id'])); ?>
47:                     </td>
48:
49:                 </tr>
50:
51:             <?php endforeach; ?>
52:         </table>
53:
54:     </div>
55:
56:     <div class="paging">
57:             <?php echo $paginator->prev('<< '.__('previous', true), ➥
array(), null, array('class'=>'disabled'));?>
58:         |       <?php echo $paginator->numbers();?>
59:             <?php echo $paginator->next(__('next', true).' >>', ➥
array(), null, array('class'=>'disabled'));?>
60:     </div>
```

We start with some pagination information on line 6. This is followed by the listing of the stories in a tabular format on line 10. Cake's paginator helper comes into play again, starting from line 13, with links to sort the data columns. We start the looping of the stories on line 21. The view finishes with previous and next links. An example of this view is shown in Figure 9-9.

Figure 9-9. *The admin story listing*

Translation Pagination

Earlier, we mentioned that the pagination of the stories to be translated wasn't that straightforward. We had to override Cake's `paginator` controller method and create our custom `paginator` method. The entire `story.php` model, which is mainly composed of code relating to the pagination, is shown in Listing 9-14.

Listing 9-14. *The Story Model (/app/models/story.php)*

```
 1:    <?php
 2:    class Story extends AppModel {
 3:
 4:        var $name = 'Story';
 5:
 6:        var $actsAs = array( 'Translate' => array( 'title', 'body' ) );
 7:
 8:        var $usePaginate = 'paginateStandard';
 9:
10:        var $transLanguage = '';
11:
```

```
12:         function paginate(  $conditions, $fields, $order, $limit,
13:                             $page = 1, $recursive = null) {
14:
15:             switch( $this->usePaginate ) {
16:
17:                 case 'paginateStandard':
18:                     return $this->paginateStandard( $conditions,
19:                                                     $fields,
20:                                                     $order,
21:                                                     $limit,
22:                                                     $page,
23:                                                     $recursive );
24:
25:                 case 'paginateTranslation':
26:                     return $this->paginateTranslation(  $conditions,
27:                                                         $fields,
28:                                                         $order,
29:                                                         $limit,
30:                                                         $page,
31:                                                         $recursive );
32:             }
33:         }
34:
35:         function paginateCount($conditions = null, $recursive = 0) {
36:
37:             switch( $this->usePaginate ) {
38:
39:                 case 'paginateStandard':
40:                     return $this->paginateCountStandard(    $conditions,
41:                                                             $recursive );
42:
43:                 case 'paginateTranslation':
44:                     return $this->paginateCountTranslation( $conditions,
45:                                                             $recursive );
46:             }
47:         }
48:
49:         /* The following are the different pagination functions */
50:
51:         // Method 1
52:         function paginateStandard(  $conditions, $fields, $order,
53:                                     $limit, $page = 1,
54:                                     $recursive = null ) {
55:
56:             $recursive = -1;
57:
58:             return $this->find( 'all', array(   'conditions' => $conditions,
59:                                                 'fields' => $fields,
```

```
                                            'order' => $order,
                                            'limit' => $limit,
                                            'page' => $page,
                                            'recursive' => $recursive
                                            ) );
        }

        function paginateCountStandard($conditions = null, $recursive = 0) {

            $recursive = -1;

            return $this->find( 'count', array( 'conditions' => $conditions,
                                            'recursive' => $recursive
                                            ) );
        }

        // Method 2
        function paginateTranslation(   $conditions, $fields, $order,
                                        $limit, $page = 1,
                                        $recursive = null ) {

            $locale = $this->getLocale( $this->transLanguage );
            $offset = $limit*($page-1);

            return $this->query( "  select * from stories as Story
                                    where
                                    Story.id not in
                                    (
                                        select foreign_key from i18n
                                        where
                                        locale = '$locale'
                                    )
                                    limit $offset, $limit
                                    " );
        }

        function paginateCountTranslation( $conditions = null,
                                           $recursive = 0) {

            $locale = $this->getLocale( $this->transLanguage );

            $results = $this->query( "  select id from stories as Story
                                        where
                                        Story.id not in
```

```
103:                                        (
104:                                            select foreign_key from i18n
105:                                            where
106:                                            locale = '$locale'
107:                                        )
108:                                        " );
109:
110:            return count($results);
111:        }
112:
113:        function getLocale( $lang ) {
114:
115:            App::import('i18n');
116:            $I18n =& I18n::getInstance();
117:            $langCode = $I18n->l10n->map( $lang );
118:            $cat = $I18n->l10n->catalog( $langCode );
119:
120:            if ( isset( $cat[ 'locale' ] ) ) {
121:                return $cat[ 'locale' ];
122:            }
123:
124:            return '';
125:        }
126:    }
127:    ?>
```

Table 9-1 lists the methods in the model.

Table 9-1. *Story Model Class Methods*

Method	Purpose
`paginate`	Cake's pagination method, which we override
`paginateCount`	Another method we must override so Cake can get the correct page count
`paginateStandard`	Same as Cake's `paginate` method
`paginateCountStandard`	Used by the standard pagination method
`paginateTranslation`	The pagination method used by the translation listing
`paginateCountTranslation`	Used by the translation listing to count the number of records

Although we override Cake's `paginate` method, we still make use of Cake's pagination helper. The `$usePaginate` variable on line 8 indicates which pagination method to use: the standard one or the translation pagination. If it's ordinary pagination of results, such as from a `find` command, we basically paginate the results in the same way as how Cake would do it. This seems like duplication, but part of the problem is that once we override the `paginate` method, there's no easy way of going back temporarily to use Cake's built-in `paginate` method.

When $usePaginate is set to paginateTranslation, the paginateTranslation method is called. Here, we create a manual SQL command to fetch all the stories that we need to translate. The paginateCountTranslation method is used by Cake so it can provide us with the correct number of total records.

One interesting method in the model is getLocale. This gets the locale information from the i18n object using the language. When developing any i18n/l10n application, you should have a good grasp of the language codes and locales. We had a hard time picking whether to use language codes or locales in the various actions. The following section will give you a better understanding of the Translate behavior in relation to locales and language codes.

Locale and Language Selection

Setting the locale via the model is the main technique for telling the Translate behavior which locale to use. However, there are other techniques, all of which will give you an understanding of how the behavior works. If we do not set the locale anywhere, the Translate behavior uses en_us as the default locale, which is set in the class itself.

Setting Locale by Browser

If you do not specify a locale, the behavior uses the browser's HTTP_ACCEPT_LANGUAGE header to work out which language to use. Cake automatically maps this out in the key value of the $__l10nCatalog variable (see /cake/libs/l10n.php). If HTTP_ACCEPT_LANGUAGE is en, for example, the en value in the $__l10nCatalog variable is used. Listing 9-15 shows an example.

Listing 9-15. *A Sample Listing in the $__l10nCatalog Array*

```
var $__l10nCatalog = array(

    'en' => array(  'language' => 'English',
                    'locale' => 'eng',
                    'localeFallback' => 'eng',
                    'charset' =>
                    'utf-8'),

    'en-gb' => array(   'language' => 'English (British)',
                        'locale' => 'en_gb',
                        'localeFallback' => 'eng',
                        'charset' => 'utf-8'),

    'en-us' => array(   'language' => 'English (United States)',
                        'locale' => 'en_us',
                        'localeFallback' => 'eng',
                        'charset' => 'utf-8'),

    'ja' => array(  'language' => 'Japanese',
                    'locale' => 'jpn',
                    'localeFallback' => 'jpn',
                    'charset' => 'utf-8')
);
```

In this method, we use the DEFAULT_LANGUAGE constant to override the HTTP_ACCEPT_LANGUAGE header; however, this method works only in a specific situation. If $_SERVER['HTTP_ACCEPT_LANGUAGE'] is set and Cake has found a match, the DEFAULT_LANGUAGE value will have no effect. You cannot override the browser HTTP_ACCEPT_LANGUAGE value. However, if Cake cannot find a match, the DEFAULT_LANGUAGE language will be used. This constant uses the three-letter ISO 639-3 language code standard.

■ **Note** If you are interested in the format of HTTP_ACCEPT_LANGUAGE, see section 14.4 Accept-Language of the HTTP standard (http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html).

Setting Locale by Language Code

You can set the locale manually via the two-letter language code. This is done as follows in the l10n class, which is used by the Translate behavior.

```
App::import('i18n');
$I18n =& I18n::getInstance();
$I18n->l10n->get( 'ja' );
```

Here, we set the locale to Japanese. To set the locale, we first import the i18n class file in /cake/libs/i18n.php. Next, we get the instance of that class. Calling the get method in /cake/libs/l10n.php sets the locale.

Setting Locale by Hand

To give you an idea of the additional ways for setting the locale, here's an alternate, albeit not recommended, method: if you manually set the HTTP_ACCEPT_LANGUAGE header, the get method in /cake/libs/l10n.php will pick that as the language to use. Here is an example:

```
$_SERVER[ 'HTTP_ACCEPT_LANGUAGE' ] = 'en-gb';
App::import('Core','I18n');
$I18n =& I18n::getInstance();
$I18n->l10n->get();
```

User Authentication

In the baking of our application, we were asked whether we also wanted admin features as well. We answered yes, and the bake command created the actions and the corresponding views for us. Four admin actions were created: admin_view, which lists stories with links to edit and delete in each record; and admin_add, admin_edit, and admin_delete, which add, edit, and delete stories, respectively. However, the baking doesn't add user functionality or authentication. We're happy to say that adding user authentication is a straightforward process.

In Cake, you can easily use the `Auth` component to add user authentication, as we have shown in the previous chapter. In this chapter, we're going to show you how authentication can be added manually.

For authentication, we create a user session variable, which tells us whether the user is logged in. Using the `beforeFilter` command, we check whether the user is accessing any of the admin features by checking the URL for the `admin` keyword. If that exists, we then check whether the user session variable has been set. If it has, we do nothing and just let execution continue on to the action. If the user session variable has not been set, we redirect the user to the login screen. This checking is placed in the `/app/app_controller.php` file, as shown in Listing 9-16, so it is called automatically on every action.

Listing 9-16. *Admin Authentication (in /app/app_controller.php)*

```
function checkAdminSession() {

    if (!$this->Session->check( 'User' ) ) {

        // Set flash message and redirect
        $this->Session->setFlash( __( 'Please login first.', true ) );
        $this->redirect('/users/login/');
    }
}


function beforeFilter() {

    if( isset( $this->params[ 'admin' ] ) ) {
        $this->checkAdminSession();
    }
}
```

Logging In

Before we talk about the `login` action, we need to look at the `users` table. This table's schema is shown in Listing 9-17.

Listing 9-17. *The users Table Schema*

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL auto_increment,
  `name` varchar(255) NOT NULL,
  `username` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  PRIMARY KEY  (`id`)
)
```

As we are just creating a simple authentication system, the User model is almost empty.

```
<?php
class User extends AppModel {
    var $name = 'User';
}
?>
```

Next, we create the login action in the UserController, as shown in Listing 9-18.

Listing 9-18. *The login Action (/app/controllers/user_controller.php)*

```
<?php

class UsersController extends AppController {

    var $name = 'Users';

    var $helpers = array('Html', 'Form');

    function login() {

        if ( !empty( $this->data ) ) {

            $user = $this->User->findByUsername( $this->data['User']['username'] );

            if( $user ) {

                if( $user['User']['password'] == md5( ➥
$this->data['User']['password'] ) ) {

                    $this->Session->write( 'User', $user );

                    $this->Session->setFlash( 'Hello! '.$user['User']['name'] );

                    $this->redirect('/');

                }
                else {
                    $this->set( 'error', 'Login Failed!' );
                }
            }
        }
    }

}

?>
```

When a user logs in, the `login` action in the `UserController` is called. This action is wholly used for the process of attempting to login the user. The corresponding login view simply holds the login form.

In the `login` action, we first check whether any data is being submitted. Then we try to locate the user's name. If one is found, we check the password—using MD5 hashing, naturally. If the password is found, we assign the result to the user session variable, set a flash welcome message, and redirect the user back to the home page.

The layout in `/app/views/layout/default.ctp` contains some user session-related code. These are the links that are displayed when a user logs in:

```
<?php
    if ( $session->check( 'User' ) ) {

        echo $html->link( __( 'Logout', true ), '/Users/logout' );

        echo ' | ';

        echo $html->link( __( 'List Stories', true ), '/admin/Stories/index' );

        echo ' | ';

        echo $html->link( __( 'New Story', true ), '/admin/Stories/add' );

        echo ' | ';

        echo $html->link( __( 'Translate', true ), '/admin/Stories/toTrans' );
    }
    else {

        echo $html->link( 'Login', '/Users/login' );
    }
?>
```

Logging Out

Logging a user out of the system is probably the easiest action in the whole application. We simply delete the user session variable, set a flash message, and then redirect the user back to the home page. The logout action is created in the `UsersController`. No view is needed, as we are redirecting the user. It looks like this:

```
function logout() {

    $this->Session->delete('User');

    $this->Session->setFlash( 'Log out OK. Please come back soon!' );

    $this->redirect('/');
}
```

Baked Code

Listings 9-19 through 9-28 show the code that was automatically generated using Cake's `bake` command.

Listing 9-19. *The Story Model Class (/app/models/story.php)*

```
<?php
class Story extends AppModel {

    var $name = 'Story';

}
?>
```

Listing 9-20. *The StoriesController Class (/app/controllers/stories_controller.php)*

```
<?php
class StoriesController extends AppController {

    var $name = 'Stories';
    var $helpers = array('Html', 'Form');

    function index() {
        $this->Story->recursive = 0;
        $this->set('stories', $this->paginate());
    }

    function view($id = null) {
        if (!$id) {
            $this->flash(    __('Invalid Story', true),
                            array('action'=>'index'));
        }
        $this->set('story', $this->Story->read(null, $id));
    }

    function add() {
        if (!empty($this->data)) {
            $this->Story->create();
            if ($this->Story->save($this->data)) {
                $this->flash(   __('Story saved.', true),
                                array('action'=>'index'));
            } else {
            }
        }
    }
```

```
    function edit($id = null) {
        if (!$id && empty($this->data)) {
            $this->flash(    __('Invalid Story', true),
                            array('action'=>'index'));
        }
        if (!empty($this->data)) {
            if ($this->Story->save($this->data)) {
                $this->flash(   __('The Story has been saved.', true),
                                array('action'=>'index'));
            } else {
            }
        }
        if (empty($this->data)) {
            $this->data = $this->Story->read(null, $id);
        }
    }

    function delete($id = null) {
        if (!$id) {
            $this->flash(__('Invalid Story', true),
                        array('action'=>'index'));
        }
        if ($this->Story->del($id)) {
            $this->flash(__('Story deleted', true),
                        array('action'=>'index'));
        }
    }


    function admin_index() {
        $this->Story->recursive = 0;
        $this->set('stories', $this->paginate());
    }

    function admin_view($id = null) {
        if (!$id) {
            $this->flash(__('Invalid Story', true),
                        array('action'=>'index'));
        }
        $this->set('story', $this->Story->read(null, $id));
    }
```

```
    function admin_add() {
        if (!empty($this->data)) {
            $this->Story->create();
            if ($this->Story->save($this->data)) {
                $this->flash(__('Story saved.', true),
                            array('action'=>'index'));
            } else {
            }
        }
    }

    function admin_edit($id = null) {
        if (!$id && empty($this->data)) {
            $this->flash(__('Invalid Story', true),
                        array('action'=>'index'));
        }
        if (!empty($this->data)) {
            if ($this->Story->save($this->data)) {
                $this->flash(__('The Story has been saved.', true),
                            array('action'=>'index'));
            } else {
            }
        }
        if (empty($this->data)) {
            $this->data = $this->Story->read(null, $id);
        }
    }

    function admin_delete($id = null) {
        if (!$id) {
            $this->flash(__('Invalid Story', true),
                        array('action'=>'index'));
        }
        if ($this->Story->del($id)) {
            $this->flash(__('Story deleted', true),
                        array('action'=>'index'));
        }
    }

}
?>
```

Listing 9-21. *The add Action View File (/app/views/stories/add.ctp)*

```
<div class="stories form">
<?php echo $form->create('Story');?>
    <fieldset>
         <legend><?php __('Add Story');?></legend>
    <?php
        echo $form->input('title');
        echo $form->input('body');
    ?>
    </fieldset>
<?php echo $form->end('Submit');?>
</div>
<div class="actions">
    <ul>
        <li><?php echo $html->link(__('List Stories', true), array ➥
('action'=>'index'));?></li>
    </ul>
</div>
```

Listing 9-22. *The admin_add Action View File (/app/views/stories/admin_add.ctp)*

```
<div class="stories form">
<?php echo $form->create('Story');?>
    <fieldset>
         <legend><?php __('Add Story');?></legend>
    <?php
        echo $form->input('title');
        echo $form->input('body');
    ?>
    </fieldset>
<?php echo $form->end('Submit');?>
</div>
<div class="actions">
    <ul>
        <li><?php echo $html->link(__('List Stories', true), array ➥
('action'=>'index'));?></li>
    </ul>
</div>
```

Listing 9-23. *The admin_edit Action View File (/app/views/stories/admin_edit.ctp)*

```
<div class="stories form">
<?php echo $form->create('Story');?>
    <fieldset>
         <legend><?php __('Edit Story');?></legend>
    <?php
        echo $form->input('id');
        echo $form->input('title');
        echo $form->input('body');
    ?>
    </fieldset>
<?php echo $form->end('Submit');?>
</div>
<div class="actions">
    <ul>
        <li><?php echo $html->link(__('Delete', true), array('action'=> ➥
'delete', $form->value('Story.id')), null, sprintf(__('Are you sure you want to ➥
delete # %s?', true), $form->value('Story.id'))); ?></li>
        <li><?php echo $html->link(__('List Stories', true), array ➥
('action'=>'index'));?></li>
    </ul>
</div>
```

Listing 9-24. *The admin_index Action View File (/app/views/stories/admin_index.ctp)*

```
<div class="stories index">
<h2><?php __('Stories');?></h2>
<p>
<?php
echo $paginator->counter(array(
'format' => __('Page %page% of %pages%, showing %current% records out of
        %count% total, starting on record %start%, ending on %end%', true)
));
?></p>
<table cellpadding="0" cellspacing="0">
<tr>
    <th><?php echo $paginator->sort('id');?></th>
    <th><?php echo $paginator->sort('title');?></th>
    <th><?php echo $paginator->sort('body');?></th>
    <th class="actions"><?php __('Actions');?></th>
</tr>
```

```
<?php
$i = 0;
foreach ($stories as $story):
    $class = null;
    if ($i++ % 2 == 0) {
        $class = ' class="altrow"';
    }
?>
    <tr<?php echo $class;?>>
        <td>
            <?php echo $story['Story']['id']; ?>
        </td>
        <td>
            <?php echo $story['Story']['title']; ?>
        </td>
        <td>
            <?php echo $story['Story']['body']; ?>
        </td>
        <td class="actions">
            <?php echo $html->link(__('View', true),
                    array('action'=>'view', $story['Story']['id'])); ?>
            <?php echo $html->link(__('Edit', true), array('action'=>'edit',
                    $story['Story']['id'])); ?>
            <?php echo $html->link(__('Delete', true),
                    array('action'=>'delete', $story['Story']['id']), null,
                    sprintf(__('Are you sure you want to delete # %s?', true),
                                    $story['Story']['id'])); ?>
        </td>
    </tr>
<?php endforeach; ?>
</table>
</div>
<div class="paging">
    <?php echo $paginator->prev('<< '.__('previous', true), array(),
                                        null, array('class'=>'disabled'));?>
 |     <?php echo $paginator->numbers();?>
    <?php echo $paginator->next(__('next', true).' >>', array(), null,
                                            array('class'=>'disabled'));?>
</div>
<div class="actions">
    <ul>
        <li><?php echo $html->link(__('New Story', true),
                                        array('action'=>'add')); ?></li>
    </ul>
</div>
```

Listing 9-25. *The admin_view Action View File (/app/views/stories/admin_view.ctp)*

```
<div class="stories view">
<h2><?php  __('Story');?></h2>
    <dl><?php $i = 0; $class = ' class="altrow"';?>
        <dt<?php if ($i % 2 == 0) echo $class;?>><?php __('Id'); ?></dt>
        <dd<?php if ($i++ % 2 == 0) echo $class;?>>
            <?php echo $story['Story']['id']; ?>
             
        </dd>
        <dt<?php if ($i % 2 == 0) echo $class;?>><?php __('Title'); ?></dt>
        <dd<?php if ($i++ % 2 == 0) echo $class;?>>
            <?php echo $story['Story']['title']; ?>
             
        </dd>
        <dt<?php if ($i % 2 == 0) echo $class;?>><?php __('Body'); ?></dt>
        <dd<?php if ($i++ % 2 == 0) echo $class;?>>
            <?php echo $story['Story']['body']; ?>
             
        </dd>
    </dl>
</div>
<div class="actions">
    <ul>
        <li><?php echo $html->link(__('Edit Story', true),
                array('action'=>'edit', $story['Story']['id'])); ?> </li>
        <li><?php echo $html->link(__('Delete Story', true),
                array('action'=>'delete', $story['Story']['id']),
                null, sprintf(__('Are you sure you want to delete # %s?', true),
                $story['Story']['id'])); ?> </li>
        <li><?php echo $html->link(__('List Stories', true),
                                    array('action'=>'index')); ?> </li>
        <li><?php echo $html->link(__('New Story', true),
                                        array('action'=>'add')); ?> </li>
    </ul>
</div>
```

Listing 9-26. *The edit Action View File (/app/views/stories/edit.ctp)*

```
<div class="stories form">
<?php echo $form->create('Story');?>
    <fieldset>
         <legend><?php __('Edit Story');?></legend>
    <?php
        echo $form->input('id');
        echo $form->input('title');
        echo $form->input('body');
    ?>
    </fieldset>
```

```
<?php echo $form->end('Submit');?>
</div>
<div class="actions">
    <ul>
        <li><?php echo $html->link(__('Delete', true),
                array('action'=>'delete', $form->value('Story.id')),
                null, sprintf(__('Are you sure you want to delete # %s?', true),
                $form->value('Story.id'))); ?></li>
        <li><?php echo $html->link(__('List Stories', true),
                        array('action'=>'index'));?></li>
    </ul>
</div>
```

Listing 9-27. *The index Action View File (/app/views/stories/index.ctp)*

```
<div class="stories index">
<h2><?php __('Stories');?></h2>
<p>
<?php
echo $paginator->counter(array(
'format' => __('Page %page% of %pages%, showing %current% records out of
            %count% total, starting on record %start%, ending on %end%', true)
));
?></p>
<table cellpadding="0" cellspacing="0">
<tr>
    <th><?php echo $paginator->sort('id');?></th>
    <th><?php echo $paginator->sort('title');?></th>
    <th><?php echo $paginator->sort('body');?></th>
    <th class="actions"><?php __('Actions');?></th>
</tr>
<?php
$i = 0;
foreach ($stories as $story):
    $class = null;
    if ($i++ % 2 == 0) {
        $class = ' class="altrow"';
    }
?>
    <tr<?php echo $class;?>>
        <td>
            <?php echo $story['Story']['id']; ?>
        </td>
        <td>
            <?php echo $story['Story']['title']; ?>
        </td>
```

```
        <td>
            <?php echo $story['Story']['body']; ?>
        </td>
        <td class="actions">
            <?php echo $html->link(__('View', true),
                    array('action'=>'view', $story['Story']['id'])); ?>
            <?php echo $html->link(__('Edit', true),
                    array('action'=>'edit', $story['Story']['id'])); ?>
            <?php echo $html->link(__('Delete', true),
                array('action'=>'delete', $story['Story']['id']),
                null, sprintf(__('Are you sure you want to delete # %s?', true),
                $story['Story']['id'])); ?>
        </td>
    </tr>
<?php endforeach; ?>
</table>
</div>
<div class="paging">
    <?php echo $paginator->prev('<< '.__('previous', true), array(), null, array ➥
('class'=>'disabled'));?>
 |     <?php echo $paginator->numbers();?>
    <?php echo $paginator->next(__('next', true).' >>', array(), null, array ➥
('class'=>'disabled'));?>
</div>
<div class="actions">
    <ul>
        <li><?php echo $html->link(__('New Story', true), array ➥
('action'=>'add')); ?></li>
    </ul>
</div>
```

Listing 9-28. *The view Action View File (/app/views/stories/view.ctp)*

```
<div class="stories view">
<h2><?php  __('Story');?></h2>
    <dl><?php $i = 0; $class = ' class="altrow"';?>
        <dt><?php if ($i % 2 == 0) echo $class;?>><?php __('Id'); ?></dt>
        <dd><?php if ($i++ % 2 == 0) echo $class;?>>
            <?php echo $story['Story']['id']; ?>
             
        </dd>
        <dt><?php if ($i % 2 == 0) echo $class;?>><?php __('Title'); ?></dt>
        <dd><?php if ($i++ % 2 == 0) echo $class;?>>
            <?php echo $story['Story']['title']; ?>
             
        </dd>
```

```
        <dt<?php if ($i % 2 == 0) echo $class;?>><?php __('Body'); ?></dt>
        <dd<?php if ($i++ % 2 == 0) echo $class;?>>
            <?php echo $story['Story']['body']; ?>
             
        </dd>
    </dl>
</div>
<div class="actions">
    <ul>
        <li><?php echo $html->link(__('Edit Story', true),
                array('action'=>'edit', $story['Story']['id'])); ?> </li>
        <li><?php echo $html->link(__('Delete Story', true),
            array('action'=>'delete', $story['Story']['id']),
            null, sprintf(__('Are you sure you want to delete # %s?', true),
            $story['Story']['id'])); ?> </li>
        <li><?php echo $html->link(__('List Stories', true),
                array('action'=>'index')); ?> </li>
        <li><?php echo $html->link(__('New Story', true),
                                array('action'=>'add')); ?> </li>
    </ul>
</div>
```

Summary

In this chapter, we introduced Cake's Translate behavior. Using Cake's `i18n` table, we were able to store the translations of stories held in a database, rather than in a static HTML file.

Next, we created some admin actions using Cake's `bake` command, which generated actions with the `admin_` prefix. Using those actions, a real human translator will have the ability to translate stories from one language to another.

Within the `Story` model, we wrote our own pagination code so the appropriate stories will be paginated depending on the selected language.

Finally, to secure the admin actions, we rolled our own authentication code.

CHAPTER 10

Adding Automagic Fields

In many other MVC frameworks, certain database fields have special significance when a user accesses the model data. These fields are often called *magic fields*, because the underlying controller automatically works out which value it should contain. Cake also has such magic fields.

In addition to using the built-in magic fields, you can create custom magic fields to suit your needs. In this chapter, we'll create three new magic fields. But before we begin, let's look at the built-in magic fields.

Cake's Built-in Magic Fields

When you use Cake's built-in magic fields, you don't need to write any code in order for Cake to automatically update them. Once the field is present in the database, Cake will detect its presence and update the value automatically. Table 10-1 lists the magic database fields in Cake.

Table 10-1. *Cake's Magic Fields*

Field	Type	Description
`id`	`int`, `bigint`, or `varchar`	This is the default field name for the primary key of the table. If it is defined as an `int` field with `auto_increment`, it will automatically generate a numeric primary key. However, if the field is something like a `varchar(36)`, Cake will generate and manage a UUID for this field.
`name`	`varchar` or other text type	Cake will use this field in various circumstances, mainly for the Scaffolding, List, and Tree behaviors. For example, when you create an HTML drop-down list using the `select` method in the form helper, it will automatically use the `name` field as the display string in the drop-down list.
`title`		This is an alias for the `name` field.
`created`	`datetime`	When a new record is added, this field contains a timestamp of when the record was created.
`modified`	`datetime`	Similar to the `created` field, when a record is changed, this field contains a timestamp of when it was modified.
`updated`		This is an alias for the `modified` field.
`[singular model name]_id`	Matches type in associated table	This is the foreign key reference used in model associations. If you do not manually specify the foreign key field name, this is what it would use.

There are other special fields that relate only to particular behaviors. For example, in the Tree behavior, the following three fields are used:

- `parent` stores the ID of the parent field.
- `lft` stores the ID of the left-branch record of the tree structure.
- `rght` stores the ID of the right-branch record of the tree structure.

Writing a Custom Behavior

Magic fields are managed by the model layer through resources called *behaviors*. Behaviors make it possible to perform automagic methods on database fields as the model runs its various data-handling methods, such as `afterFind` or `beforeSave`.

For instance, upon saving a new record to the table, the model passes all of its data through to any attached behavior classes. This gives the attached behaviors the opportunity to intercept the save process and perform any business logic based on or relating to the intercepted data. This may involve cross-table or cross-database updates based on what is being saved. In other words, an automagic update occurs in fields that are not necessarily part of the record being saved.

So, before we get to creating custom magic fields, we need to talk about how to create your own custom behavior.

First, you create a PHP file to be stored in the folder `/app/models/behaviors`. This file contains the code that will drive the behavior. The file name must be in lowercase, and multiple words are separated with underscores. The behavior class name must also be in camel case, with the word `Behavior` appended, and it must extend the class `ModelBehavior`. Here's how to start a custom behavior class:

```
<?php
class CustomBehavior extends ModelBehavior {

}
?>
```

The next step is to include the behavior in the model itself. This is done by including the `$actsAs` variable in the model, as follows:

```
var $actsAs = array( 'Custom' => array( 'parameter', ➥
'another_parameter') );
```

A Cake behavior helps you by transparently performing some operation in the background. When the model methods `delete`, `save`, and `find` are used, it will trigger other functions (commonly called *callback functions*) in your behavior class.

Here are the other functions of the model's `delete` method in a behavior and the sequence in which they are called:

1. Model behaviors' `beforeDelete()` method
2. Model's `delete()` method
3. Model behaviors' `afterDelete()` method

So, when a model's `delete` method is called, starting from step 1, the `beforeDelete` method in all the behaviors that are attached to the model is called before the record is deleted. In step 2, the record is deleted. And finally, in step 3, the `afterDelete` method in all the behaviors that are attached to the model is called.

Here are the other functions of the model's `save` method in a behavior and their order:

1. Model behaviors' `beforeValidation()` method
2. Model behaviors' `beforeSave()` method
3. Model's `save()` method
4. Model behaviors' `afterSave()` method

And the following are the other functions of the model's `find` method in a behavior and their order:

1. Model behaviors' `beforeFind()` method
2. Model's `find()` method
3. Model behaviors' `afterFind()` method

Within your behavior class, there is also a method called `setup`. This method is called when the parent model is instantiated. Its syntax is as follows:

```
function setup(&$model, $config = array()) {
}
```

The model is passed by reference, which means you can directly manipulate the values in the model. The `$config` variable contains the values you passed to the behavior in the model.

Tip Sometimes, you may need to carry out your own SQL queries within a model rather than using the model's standard CRUD methods. The callbacks would not work as intended, since they would not know the kind of query you have made. However, you can still make use of the behavior by using the `setup` method as an entry point into the behavior, since that is called whenever the model is invoked in any way. This is not an ideal solution, since you would need to manually call the methods within the model behavior yourself; however, it does work.

Building Custom Magic Fields

Now that you know how to create a custom behavior, we can start building some magic fields. First, we need to create our behavior, starting with the behavior file itself, which we'll call `MagicFieldsPlus`. The behavior file is created as `app/models/behaviors/magic_fields_plus.php`. The skeleton of the file is shown in Listing 10-1. We'll be adding callback methods to the class. Note that in our case, the `setup` method is empty. You can easily add parameters that you can use to alter the way the behavior works.

Listing 10-1. *The MagicFieldsPlus Cake Behavior (app/models/behaviors/magic_fields_plus.php)*

```
<?php

class MagicFieldsPlusBehavior extends ModelBehavior {

    function setup( &$model, $config = array() ) {

    }

    // Callback methods here ...
}
?>
```

To avoid potential field name clashes, we'll append the prefix `m_` to the names of all our magic fields.

Access Data Field

We're going to start with something simple. This magic field will increment by one whenever it is accessed. We will consider only the use of the model's `find` method as an "access." Using the `save` method will not be counted. (Of course, you can easily change this behavior.)

To get this field working, we need to have a field called `m_accessed`, which will be updated during the `afterFind` callback. Since the `afterFind` command is a generic method, we're not going to put the code that will actually do the incrementing in that method. Instead, we'll create a separate method and call it from the `beforeFind` method. This two-step process is shown in Listing 10-2.

Listing 10-2. *Methods for the Access Data Field*

```
 1:    function afterFind(&$model, $results, $primary) {
 2:        $this->m_accessed_magic( $model, $results, $primary );
 3:        return $results;
 4:    }
 5:
 6:    function m_accessed_magic( &$model, $results, $primary ) {
7:        if ( $model->hasField( 'm_accessed' ) ) {
8:            foreach( $results as $record ) {
9:                $record[ $model->name ][ 'm_accessed' ]++;
10:                $model->save( $record );
11:            }
12:        }
13:    }
```

Starting from the `afterFind` method, this simply calls the `m_accessed_magic` method on line 2. Within the `m_accessed_magic` method, we first check whether the field name `m_accessed` exists. If it does, we loop through the results, incrementing the `m_accessed` field by one in each case.

Note This access data field can be quite useful for data-intelligence gathering. However, it's obviously not that efficient, as it needs to save each record in turn. As an exercise, you can change the code to save the access data out to an external flat file and update the `m_accessed` field at a later date.

Record Order Data Field

Sometimes, you may just want to order the fields numerically by default. For example, in a shop's products listing, the shop owner may want to place the best-selling items at the top.

Our next magic field applies to find operations. We want the results returned to be ordered numerically. We will add a new field called `m_record_order`. We will specify the ordering in the `beforeFind` callback. Similar to how we constructed the access data field, the record order data field will be composed of two parts within the behavior, as shown in Listing 10-3.

Listing 10-3. *Methods for the Record Order Data Field*

```
 1:    function beforeFind( &$model, $query ) {
 2:        return $this->m_record_order_magic( $model, $query );
 3:    }
 4:
 5:    function m_record_order_magic( &$model, $query ) {
 6:        if ( $model->hasField( 'm_record_order' ) ) {
 7:            $query[ 'order' ] = 'm_record_order DESC';
 8:        }
 9:        return $query;
10:    }
```

The method `m_record_order_magic` is called from the `beforeFind` callback. On line 6, we check whether the field exists. If it does, we specify a particular ordering in the `query` array on line 7.

The format of the `order` key is the same as the `order` attribute in the model's `find` method, so you can order it according to your own needs. For example, if you have another field called `product_group_id`, you can order it according to `product_group_id` and then by `m_record_order`. So, you can use any of the following statements for the `order` value:

- `product_group_id, m_record_order DESC`
- `product_group_id DESC, m_record_order ASC`
- `array( 'product_group_id', 'm_record_order DESC' )`
- `array( 'product_group_id DESC', 'm_record_order ASC' )`

If you do not want to hard-code the ordering direction, you can specify it in the `$actsAs` behavior variable. There are any number of array formats that you can use to pass the `direction` parameter from the `$actAs` variable into the behavior. In the example in Listing 10-4, we're using the magic field name as the key, with the value containing an associative array of key/value pairs that the magic field can use.

Listing 10-4. *Passing Configuration Parameters to the Behavior*

```
var $actsAs = array( 'MagicFieldsPlus' => ➥
array( "m_record_order" => array( "direction" => "ASC" ) ) );
```

Once we have set up the configuration in the model file, we need to catch the configuration values in the `setup` method in the behavior itself. This is shown in Listing 10-5.

Listing 10-5. *Setting the Default Values in the Behavior (/app/models/ behaviors/magic_fields_plus.php)*

```
1:    var $magicFieldParams = array();
2:
3:    function setup( &$model, $config = array() ) {
4:        $this->magicFieldParams = am( $this->magicFieldParams, $config );
5:    }
```

The `$config` variable on line 3 will now contain the configuration values as passed to it in the model's `$actAs` variable. We've also added a `$magicFieldParams` variable, which we'll use to store the configuration values. The contents of the `$config` variable are as follows:

```
Array
(
    [m_record_order] => Array
        (
            [direction] => ASC
        )

)
```

Using Cake's convenience method `am` for merging arrays, we merge the class member variable `$magicFieldParams` with the `$config` variable.

Now using our configuration parameters for the sorting direction, we have a new `m_record_order_magic` method, which is shown in Listing 10-6. It is similar to the one in Listing 10-3, except that instead of hard-coding our sorting direction, we're picking it up from the value that was passed to us from the model's `$actAs` variable.

Listing 10-6. *Magic Method Using Configuration Values*

```
function m_record_order_magic( &$model, $query ) {
    if ( $model->hasField( 'm_record_order' ) ) {
        $direction = 'DESC';
        if ( isset( $this->magicFieldParams[ 'm_record_order' ] ➥
[ 'direction' ] ) ) {
            $direction = $this->magicFieldParams[ 'm_record_order' ] ➥
[ 'direction' ];
        }
        $query[ 'order' ] = 'm_record_order '.$direction;
    }
    return $query;
}
```

Locking Data Field

Our third magic field is related to optimistic locking. It is more complicated than the previous two. First, let's consider how optimistic locking works.

When two or more processes or users are accessing the same record, who should update first? This question arises quite frequently. In a relational database, the problem falls under the heading of *concurrency control*.

For example, in ticket-booking systems, you sometimes have a time limit for your booking. If you fail to purchase the ticket within the time frame, you probably must start again, and by then, another user may have gotten there first. The time limit demonstrates the use of optimistic locking. When you read the record, you mark it with a value. You take this value with you within the web session. Now when you come to updating the record again, you check whether the value that you are still holding matches the marked record in the database. If it's different, another user may be trying to update the same record at the same time.

You can develop your own ways to mark a record. For example, you might have an application where registered users can override nonregistered users when booking a ticket. Here are some standard methods:

- *Use a modified date as the handle.* You can still list and view the record, but during updates, you will use the modified date as the comparison.
- *Use a unique ID field as the handle.* You write a unique ID to a field when you access the record. If, during an update, it is different from the one you already have in the session, you don't perform the update.
- *Use an access time field as the handle.* Each process will check the access time. If it's more than a certain set limit, you can access the record and attempt to write new changes.

You can also use another concurrency control called *pessimistic locking*. This is where you completely lock the record for certain actions like `select` or `update`, until the process that owns the lock releases it. But in a stateless web environment, this is obviously not practical. Just imagine the scenario where a user is looking at the prices for football game tickets. Using pessimistic locking, we will prevent all other users from looking at those prices, until the user who has the lock releases the lock. But what indicates that the user is releasing the lock? When he closes the browser window? When he navigates to another page? What happens if he just minimizes that particular page and decides to get a drink before making a purchase? As you can see, pessimistic locking is just not workable here.

Now let's go over how we have implemented optimistic locking in our behavior. We simply use a unique ID to mark our records using a field called `m_lock`. Our magic field will be used only when a user edits a single record and then attempts to update it. The process is split into two stages:

- After a find operation, we update the record with the unique ID.
- When a user attempts to save the record, we check the ID against the one stored in the database; if it's different, we reject the change.

Listing 10-7 shows how we have carried out the first stage in the process.

Listing 10-7. *Optimistic Locking with a Model Behavior*

```
 1:    function afterFind(&$model, $results, $primary) {
 2:        $this->m_lock_magic( $model, $results, $primary );
 3:        return $results;
 4:    }
 5:
 6:    function m_lock_magic( &$model, &$results, $primary ) {
 7:        if ( $model->hasField( 'm_lock' ) ) {
 8:            if ( sizeof( $results ) == 1 ) {
 9:                $uuid = String::uuid();
10:                // Results that we will present to the user
11:                $results[0][ $model->name ][ 'm_lock' ] = $uuid;
12:            }
13:            $tableName = $model->table;
14:            $id = $model->id;
15:            $model->query( "update $tableName set m_lock = ➥
'".$uuid."' where id = '".$id."'" );
16:            // The current model data, maybe used in a form
17:            $model->data[$model->name][ 'm_lock' ] = $uuid;
18:        }
19:    }
```

On line 6, we call our `m_lock_magic` method, which will do the work for the first stage. First, we check whether the field `m_lock` exists, on line 7. Next, we proceed only if there's just one record. Basically, we assume it's an edit for now. If it is, we create a unique ID using Cake's `uuid()` method.

Next, we update the `m_lock` field by running a SQL `UPDATE` statement. We are using a manual update because using the `save` command would cause a loop by calling our magic field behavior. Alternatively, you can use the behavior's `disable()` and `enable()` methods to temporarily disconnect the behavior from the model, but we just wanted to point out that query commands have no effect on behaviors.

The `$results` array now contains the unique ID, which it can use as a hidden field in a form.

In the second stage of the optimistic locking, we carry out the validation. This is where we compare the ID that's being used on a form and the ID that's in the database. This code is shown in Listing 10-8.

Listing 10-8. *Validating an Optimistic Locking Data Field*

```
 1:    function beforeValidate(&$model) {
 2:        // First find the record
 3:        if ( isset( $model->data[$model->name][ 'id' ] ) ) {
 4:            $id = $model->data[$model->name][ 'id' ];
 5:            $table = $model->table;
 6:            $currentRecord = $model->find('all', ➥
array('conditions'=>array('id'=>$id)));
```

```
7:            if ( !empty( $currentRecord ) ) {
8:                if ( isset( $model->data[$model->name]['m_lock'] ) ) {
9:                    if ( $model->data[$model->name]['m_lock'] != ➥
$currentRecord[0][ $table ][ 'm_lock' ] ) {
10:                       $model->validationErrors[ 'm_lock' ] = ' ➥
Update conflict, another user has already updated the record. Please ➥
list and edit the record again.';
11:                       return false;
12:                   }
13:               }
14:           }
15:       }
16:       return true;
17:   }
```

On line 3, we first check whether we are editing an existing record. If so, we fetch the record from the database using the model's `find` method. Once a record is found, we check that record against the one in the model data array. If it is different, we manually set the `validationErrors` error array with an error message and then return `false`, which in turn will prevent the `save` command from going forward.

Summary

Magic fields are database fields that have special meanings within a Cake model. In this chapter, we added new, custom magic data fields, which involved building custom Cake behaviors.

This chapter also highlights a particular point. There's a school of thought in MVC that recommends that developers write fat models and skinny controllers. Any data manipulation, such as finding or saving records, should be done in the model. Controller actions should be skinny managers with surrounding support from components, models, and behaviors. As you've seen in this chapter, by using behaviors, you can really cut down on the amount of logic that is performed by the controller.

Some of our magic fields are essentially metadata fields. You might want to move all of them out to a separate metatable. Additionally, you could quickly make the following improvements:

- Add an `m_security` magic field. This can store a `user_group_id` type field. If a user is not within this group, access will be denied to that user.
- Add an `m_display` magic field. Use this field to turn the display of the record on or off in the view. For example, you may have an article that you want to display only occasionally, such as once every few weeks.
- Most developers probably will not use all the magic fields at the same time, so it seems logical to separate them into separate behaviors.
- Add magic fields that are specific to certain applications. For example, in an e-commerce application, you might have a `number_of_sales` field in a product table. This would allow you to place orders based on the best-selling products.

CHAPTER 11

Cake Tags

In this chapter, we're going to develop our own HTML-based tags to display two Yahoo! maps. The idea for our application stems from our strong opinion on an important aspect in web development: avoiding mixing presentation markup and logic. We'll start off this chapter by addressing that point. Our discussion isn't limited to Cake but covers many other languages and frameworks, including Ruby on Rails, Extensible Stylesheet Language Transformations (XSLT), and the Smarty template engine.

Content and Data Separation

The basic premise is that no programming logic should appear in any presentation files, namely templates. To see what we mean, take a look at the semi-pseudo PHP code in Listing 11-1.

Listing 11-1. *Web Programming in the 1990s*

```
<body>

<?php
    $conn = mysql_connect( "localhost", "mysql_user", "mysql_password" );

    $sql = "SELECT patient_name, date_of_birth, address
            FROM patients
            WHERE gender = 'male'";

    $result = mysql_query($sql);

    echo '<table>';

    while ($row = mysql_fetch_assoc($result)) {
        echo '<tr>';
            echo '<td>'.$row["patient_name "].'</td>';
            echo '<td>'.$row["date_of_birth "].'</td>';
            echo '<td>'.$row["address "].'</td>';
        echo '</tr>';
    }

```

```
22:         echo '</table>';
23:     ?>
24:
25:     </body>
```

Here, we create a MySQL connection on line 4. Then we execute a query on line 10. Next, we loop through the results, mixing HTML markup with the database data to create a particular view.

This code suffers from a number of problems. Suppose that you wanted to change the output format—for example, instead of displaying the data in a `table` tag, you wanted it to appear in an RSS format. In this case, you would need to add another layer of logic between the data and the display. You could do that within the code itself with an `if` statement, as in this example:

```
if ( $_GET[ 'format' ] == 'RSS' ) {
    // Output in RSS format
}
else {
    // Just output in table tag
}
```

Using an `if` statement is OK, but you will run into other problems. For example, when more logic is added, where do you put it? The code will start to get more and more proprietary. No one will understand your structure unless you write good documentation to go with it, and even when you do, other developers may not agree with you or your structure. You also may find that you have not taken other scenarios into account.

You could also create another separate file, as in Listing 11-1, that specifically deals with RSS output. So instead of outputting HTML, you output XML in the RSS format.

The code in Listing 11-1 can be improved by encapsulating some of the code in functions, as shown in Listing 11-2.

Listing 11-2. *Using PHP Includes*

```
 1:     <body>
 2:
 3:     <?php
 4:
 5:         makeDatabaseConnection();
 6:
 7:         $sql = "SELECT patient_name, date_of_birth, address
 8:                 FROM patients
 9:                 WHERE gender = 'male'";
10:
11:         $result = mysql_query($sql);
12:
13:         if ( $_GET[ "format" ] == "XML" ) {
14:             displayXML( $result );
15:         }
```

```
        else {
            displayHTML( $result );
        }
    ?>

    </body>
```

This is better, and using an MVC pattern helps, but developers can still develop proprietary designs within the controller. And with such proprietary designs, documentation should be written, but sadly, often it is not.

Note As a brief diversion, there's an entertaining take on the separation of data and content on `http://www.youtube.com/watch?v=6gmP4nk0EOE`.

We jump ahead now to the use of MVC in the web development process. In basic terms, the controller delegates incoming requests and funnels them to the appropriate end point, which is normally an action in a class. The model handles the data, and the view is responsible for the return format to the recipient. However, the solution still has a problem: in too many cases, we see developers putting more and more logic into the view. Plus, it's often difficult to distinguish whether a piece of code relates to the application or to the view—for example, when there is some specific data or code that's just used for a particular format. And with time pressures, it's often easier to place it in the view.

Our Cake tags approach is one small step toward addressing the problem. Following from the previous examples, Listing 11-3 shows how Cake tags would do it.

Listing 11-3. *Using Cake Tags*

```
<body>
    <ct plugin="DatabaseAccess" controller="DbRecords" ➥
action="listRecords" table="patients" gender="male"/>
</body>
```

Our idea in using tags isn't new, of course. ColdFusion has used it since the 1990s, and Java has JSP tags. For example, in ColdFusion, you can send an e-mail message using the following tag:

```
<cfmail to="another@example.com" from=another2@example.com ➥
subject = "Hello Friend!">How are things?</cfmail>
```

View Template

The best place to start explaining how we wrote our Cake tags is from the view. In this application, our Cake tags will be available to use in any view. We're going to create a sample page in `/app/views/pages/home.ctp`, Cake's default home page.

A Cake tag is essentially an XML wrapper for Cake plugins, where the output from the plugin replaces the Cake tag. We could have used any number of ways of interfacing the tag with Cake, but plugins seem generic enough. Additionally, we envisage some point in the future where the Cake tags can be used as a clean interface between third-party code (Cake plugins) and Cake itself. We can even go one step further and imagine a framework where we can add Cake plugins in a visual development environment and specify the attribute values using a form-based user interface.

Now back to our application. In it, we will have a Yahoo! Maps plugin. This plugin will display a geographical map location, which we specify, much the same as with Google Maps. How the Yahoo! Maps plugin is written will be explained a little later, in the "Cake Plugins" section. From our view in Listing 11-4, using our tag method, we will make two requests to display the Yahoo! maps via our Cake tags on lines 3 and 7.

Listing 11-4. *The Home Page (/app/views/pages/home.ctp)*

```
1:    <h3>My Vacation Destinations</h3>
2:
3:    <ct plugin="Yahoomaps" controller="Maps" action="display"  ➥
latitude="48.856925" longitude="2.341210" />
4:
5:    <br />
6:
7:    <ct plugin="Yahoomaps" controller="Maps" action="display" ➥
location="San Francisco" />
8:
9:    <br />
```

From this listing, you can see that our Cake tags are named `ct` (for Cake tag, of course). We'll be using standard XML syntax format for the tags.

A Cake tag is formed with a single XML tag, with no tag end. The attributes are used to pass key value parameters to the Cake plugin. Table 11-1 describes these attributes.

Table 11-1. *Cake Tag Attributes*

Attribute	Description
`plugin`	The name of the plugin that we are calling
`controller`	The controller within the plugin that we are calling
`action`	The action within the controller that we are calling
Others	Passed into the plugin as named parameters, which can be accessed via Cake's `passedArgs` controller variable

If you ran the code in Listing 11-4 as it is, without overriding Cake's view, it will just display "My Vacation Destinations" in the `h3` tag format, which is not very useful.

Our goal is to display two Yahoo! maps, one after the other. In order for that to happen, we must override Cake's view class so we can parse our Cake tags, as described next.

Cake View Class Extension

Using your own view is quite simple, as shown in Listing 11-5.

Listing 11-5. *Overriding Cake's View*

```
 1:    <?php
 2:
 3:    App::import( 'View', 'CakeTags', array( 'file' => 'cake_tags.php' ) );
 4:
 5:    class AppController extends Controller {
 6:
 7:        var $pageTitle = 'Chapter 11 - Cake tags';
 8:
 9:        var $view = 'CakeTags';
10:    }
11:    ?>
```

First, in the global controller `/app/app_controller.php`, we import the view using Cake's `App::import` command (line 3 in Listing 11-5). Next, we set the name of our view class in the controller to our own on line 9.

■ **Note** If you set the Controller's `autoRender` variable to `false`, no view is rendered, regardless of whether the view was overridden or not.

Next, we create our new view class, as shown in Listing 11-6.

Listing 11-6. *Our New View Class (/app/views/cake_tags.php)*

```
 1:    <?php
 2:
 3:    class CakeTagsView extends View {
 4:
 5:        function render($action, $layout, $file) {
 6:
 7:            $result = parent::render($action, $layout, $file);
 8:
 9:            $result = $this->_renderCt( $result );
10:
11:            return $result;
12:        }
13:
14:        function _renderCt( $output ) {
15:
```

```
16:             $result = $output;
17:             $match = 1;
18:             $offset = 0;
19:
20:             while ( $match ) {
21:
22:                 preg_match("/<ct ([\w]+)[^>]*\/>/", $result, ➥
$match, PREG_OFFSET_CAPTURE, $offset );
23:
24:                 if ( $match ) {
25:
26:                     $plugin = $controller = $action = $params = '';
27:
28:                     $tag = $match[0][0];
29:                     $offset = $match[0][1];
30:
31:                     $xml = new SimpleXMLElement( $tag );
32:
33:                     foreach( $xml->attributes() as $attr => $value ) {
34:
35:                         switch ($attr) {
36:
37:                             case 'plugin':
38:                                 $plugin = $value;
39:                                 break;
40:
41:                             case 'controller':
42:                                 $controller = $value;
43:                                 break;
44:
45:                             case 'action':
46:                                 $action = $value;
47:                                 break;
48:
49:                             default:
50:                                 $params .= $attr.':'.$value.'/';
51:                                 break;
52:                         }
53:                     }
54:
55:                     if ( $controller && $action ) {
56:
57:                         $tagResult = $this->requestAction( ➥
'/'.$plugin.'/'.$controller.'/'.$action.'/'.$params, array( 'return' ) );
58:
59:                         $result = str_replace( $tag, $tagResult, $result );
60:                     }
61:                 }
```

```
62:                 else {
63:                     break;
64:                 }
65:             }
66:
67:             return $result;
68:         }
69:     }
70:
71:     ?>
```

Our view class name must be in the format [*YourViewName*]`View` and it must extend Cake's `View` class. The next step is to override the `render` method. In line 7, we first render the real view using Cake's `render` command. On line 9, we intercept the output and parse the `ct` tags. This is done using our `_renderCt` method, starting from line 14. (If you were to comment out line 9, it would be as if you had not changed any of Cake's original view output.)

In the `_renderCt` method, we take the output of the view after Cake has rendered it, parse any `ct` tags we find, and call the relevant plugin as specified in the `ct` tag attributes. We continuously loop through the output and parse and replace the `ct` tags until there are no more `ct` tags to process. The `ct` tags are simply matched with the regular expression in the following line of code:

```
/<ct ([\w]+)[^>]*\/>/
```

Once a match is found, we attempt to fill the four main variables: `$plugin`, `$controller`, `$action`, and `$params`. PHP's `SimpleXMLElement` class is used to extract all the attributes. Once the `$controller` and `$action` values are in place, we call the actions by using the `$this->requestAction` call. In line 59, we simply replace the whole `ct` tag with the output of the `$this->requestAction` call.

In line 57 of Listing 11-6, we make a request to a particular action in a Cake plugin. If you were to build your own Cake tag, it would need to be a Cake plugin, so line 57 could access that action in the URL format `/plugin/controller/action/params`.

Cake Plugins

Next, to create our Yahoo! Maps plugin, we first create a folder called `yahoomaps` in the `app/plugins` folder. Within that folder, we create several files and folders, which will resemble a Cake application folder structure, as follows:

```
Yahoomaps
    yahoomaps_app_controller.php
    yahoomaps_app_model.php
    controllers
        maps_controller.php
    models
    views
        maps
            display.ctp
```

The two main files we are concerned with are `maps_controller.php` and `display.ctp`. The `maps_controller.php` file is shown in Listing 11-7.

Listing 11-7. *Map Controller (maps_controller.php)*

```
<?php

class MapsController extends GoogleMapsAppController {

    var $uses = null;

    function display() {

       // do longitude
       $longitude = '-0.127144';

       if ( isset( $this->passedArgs['longitude'] ) ) {
           $longitude = $this->passedArgs['longitude'];
       }

       $this->set( 'longitude', $longitude );

       // do latitude
       $latitude = '51.506325';

       if ( isset( $this->passedArgs['latitude'] ) ) {
            $latitude = $this->passedArgs['latitude'];
       }

       $this->set( 'latitude', $latitude );

       // Location can also be specified
       // which overrides the long and lat values
       $this->set( 'location', '' );

       if ( isset( $this->passedArgs['location'] ) ) {
           $this->set( 'location', $this->passedArgs['location'] );
       }
    }
}

?>
```

To be honest, this example isn't terribly exciting. The `display` method simply acts as a proxy for passing values from the `ct` tag to the view. If no `longitude` or `latitude` values are passed, we use some default values, which at present are set to London. However, don't underestimate what you can actually do. Since you can essentially invoke any plugin, controller, or action, you can wrap any functionality behind a Cake tag.

The other two files, yahoomaps_app_controller.php and yahoomaps_app_model.php, are shown in Listings 11-8 and 11-9, respectively. These are the global controller and model files, similar to the app_controller.php and app_model.php files. In Listing 11-8, we need the JavaScript helper to display the Yahoo! Maps API JavaScript files.

Listing 11-8. *Yahoo! Maps Controller (yahoomaps_app_controller.php)*

```
<?php
    class YahooMapsAppController extends AppController {
        var $helpers = array( 'Javascript' );
    }
?>
```

Listing 11-9. *Yahoo! Maps Model (yahoomaps_app_model.php)*

```
<?php
    class YahooMapsAppModel extends AppModel {

    }
?>
```

You can achieve more complex operations in other scenarios. Here are some examples:

- Display the current shopping basket, with the total number of products and a total cost, as follows:

  ```
  <cf plugin="Ecommerce" controller="Basket" action="display" skin="minimalist" />
  ```

- Display the products available as a tree menu:

  ```
  <cf plugin="Ecommerce" controller="Products" action="listProductsTree" />
  ```

- Display the Twitter public timeline messages in 3D using a Flash 3D display engine called Papervision3D:

  ```
  <cf plugin="Twitter" controller="StatusMethods" action="public_timeline" ➥
  format="paperVision3D" />
  ```

Yahoo! Maps

Adding Yahoo! Maps is similar to adding Google Maps. We start by including Yahoo!'s JavaScript Map API. Listing 11-10 shows the view.

Listing 11-10. *Yahoo! Maps View (/app/plugins/yahoomaps/views/maps/display.ctp)*

```
 1:    <?= $javascript->link( 'http://api.maps.yahoo.com/ajaxymap➥
?v=3.8&appid=[you own api key]' ) ?>
 2:
 3:    <style type="text/css">
 4:    #map{
 5:      height: 75%;
 6:      width: 100%;
 7:    }
 8:    </style>
 9:
10:    <?php
11:        $uuid = String::uuid();
12:        echo '<div id="'.$uuid.'"></div>';
13:    ?>
14:
15:    <script type="text/javascript">
16:
17:        // Create a map object
18:        var map = new YMap(document.getElementById('<?php echo $uuid; ?>'));
19:
20:        // Zoom Control
21:        map.addZoomLong();
22:
23:        // Add map type control
24:        map.addTypeControl();
25:
26:        // Set map type to either of: YAHOO_MAP_SAT, ➥
YAHOO_MAP_HYB, YAHOO_MAP_REG
27:        map.setMapType(YAHOO_MAP_REG);
28:
29:        // Display the map centered on a geocoded location
30:        <?php
31:            if ( $location ) {
32:                echo "map.drawZoomAndCenter( '".$location."', 3 );";
33:            }
34:            else {
35:                echo "map.drawZoomAndCenter( ➥
new YGeoPoint( ".$latitude.", ".$longitude." ), 3 );";
36:                //echo "map.drawZoomAndCenter( ➥
new YGeoPoint( 16.773480, -97.747175 ), 3 );";
37:            }
38:        ?>
39:
40:    </script>
```

When including the API, you need to provide your application ID, which you can get from `http://www.developer.yahoo.com/maps/`.

Next, we create the `div` container, which will hold the map, on lines 10 through 13. Since this plugin may be included in the same view more than once, we create a unique ID for the `<div>` element using Cake's `uuid()` string method.

Just to recap, the output from Listing 11-10 will replace the `ct` tag that called for it. A sample output of the two Yahoo! maps is shown in Figure 11-1.

Figure 11-1. *Using Cake tags to display Yahoo! maps*

Our Yahoo! Maps example is quite simple. We can easily specify other parameters to pass to Yahoo! Maps. For example, to add a zoom level, our Cake tag would look like this:

```
<ct plugin="Yahoomaps" controller="Maps" action="display" ➥
location="San Francisco" zoom="2" />
```

On line 32 in Listing 11-10, the `drawZoomAndCenter` command would now look like this:

```
echo "map.drawZoomAndCenter( '".$location."', '".$zoom."' );";
```

And in the `display` action in Listing 11-10, we would need to capture the zoom value and pass it to the view with the following lines of code:

```
// Do zoom
$zoom = '3';

if ( isset( $this->passedArgs['zoom'] ) ) {
    $zoom = $this->passedArgs['zoom'];
}
$this->set( 'zoom', $zoom );
```

Summary

In this chapter, we have shown how you can override Cake's output. Additionally, we created a Cake plugin that displays Yahoo! maps.

We mentioned early on that the whole Cake tags idea is a small step toward solving the problem of content and data separation. In our view, this is a big problem, because so many modern applications need to talk to other applications. Good separation of these two layers is vital in web development. However, at some stage, you will inevitably need to mix them. The questions will be how much and where. One thing is certain: you should never put any presentational markup in the controller. With Cake, use Cake elements and helpers to reduce the size of your views.

CHAPTER 12

Dynamic Data Fields

In this chapter, we'll present a snippet of an e-commerce feature. We will supply enough code and explanation to allow you to use this feature in a real-life project. Our feature centers on product searching. We take a nontraditional approach to this feature, basing it on dynamic data fields. We'll start off by reviewing how product searches are usually conducted.

Traditional Product Searching

While shopping online, many e-commerce sites allow you to narrow down the product range by selecting specific attributes of interest. For example, you can narrow down the search to just a particular brand or a particular price range. Figures 12-1 and 12-2 show typical product filtering on Amazon and Kelkoo, respectively.

Figure 12-1. *Amazon product filtering*

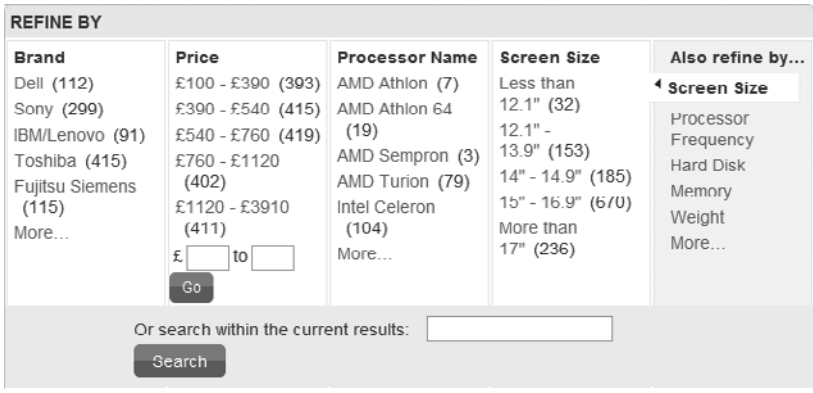

Figure 12-2. *Kelkoo product filtering*

Traditionally, to provide such a feature, you would put the products and their attributes in a single table, as shown in Listing 12-1.

Listing 12-1. *A Typical Products Table*

```
CREATE TABLE `products` (
  `id` int(11) NOT NULL auto_increment,
  `title` varchar(255) NOT NULL,
  `price` float NOT NULL,
  `stock_qty` int(11) NOT NULL,
  `color` varchar(255) NOT NULL,
  `size` varchar(255) NOT NULL,
  PRIMARY KEY  (`id`)
);
```

You would then create a form that contained hard-coded fields that users could select. These fields would be passed to a standard SQL `SELECT` statement via a `POST` action, and the filtered results would be returned.

The traditional method works well if the products are mostly the same or you have only a few hundred products. However, if you want to build a more flexible system and be able to handle a much larger product base with attributes that vary widely, you need a different approach.

The Dynamic Data Approach

Our method turns the traditional products table on its side. For example, in the traditional table, the price field occupies one column on its own, with the prices running down the table, as shown in Figure 12-3.

←T→			id	title	product_brand	product_price
☐	✎	✕	1	Panasonic AB-12XY34	Panasonic	600
☐	✎	✕	2	Panasonic AB-78XY89	Panasonic	5000

Figure 12-3. *A table with a traditional price field*

In our approach, we have the price field as a data type in itself, and create a separate record for each price based on that price field data type, as shown in Figure 12-4. Our price field now is essentially dynamic data. We can add and delete it as if it were ordinary data. Taking this concept further, we can dynamically create table data fields.

←T→			id	title	field_type_group_id	product_field_group_id
☐	✎	✕	1	Product Brand	4	1
☐	✎	✕	2	Product Price	1	1

Figure 12-4. *The price field as a database record*

From an e-commerce standpoint, we can now create thousands or millions of products with varying properties. Each property or attribute would be a record in itself.

Considerations for Using the Dynamic Data Approach

Since we're moving away from the standard way of creating relational tables, we need to consider the implications of taking this approach. There are both advantages and disadvantages.

Our approach has the following disadvantages:

- Filtering data using the standard `WHERE` clause conditions will not work in some circumstances. For example, the simple SQL statement `select * from products where price > 10 and color = 'red'` will not return the desired results, because there are no `price` and `color` fields. We could still achieve the same results using other means, but it would involve more code and more SQL statements.
- Programming other e-commerce features becomes much more complicated. For example, a simple SQL statement like `select * from products` would not return the results we want. To get this data, we would need to make further SQL queries.
- Code maintenance becomes more difficult. Developers are familiar with the traditional way in which SQL statements interact with code.

The advantages of the dynamic data approach are as follows:

- You can create dynamic tables and data fields based on the attributes of products. Indeed, you can also create dynamic forms (although that isn't our goal in this chapter's example).
- You can create automatic validation since you are controlling the data type of the field. (We won't cover that feature in this chapter, but you can easily add it.)
- What wasn't practically possible previously using standard SQL methods is now possible—namely, mixed data sets.

Our dynamic data field technique is recommended only if the ability to have dynamic fields is a core feature of your application. If you have several hundred brands or products, it's still much better to create separate tables as and when needed, such as `products_shoes`, `products_televisions`, `products_hats`, and so on (although, honestly, that solution isn't that attractive either when the number of tables increases).

The Product Database Design

Most of our tables will be metatables—tables that hold data that describes other data. For example, the integer value 42 may numerically represent anything, but if we attach an attribute called `price` to it, then 42 numerically represents price. The attribute is the metadata. As we've said, creating dynamic data fields is a complicated business. To achieve our aim, we have created nine tables, as shown in Figure 12-5.

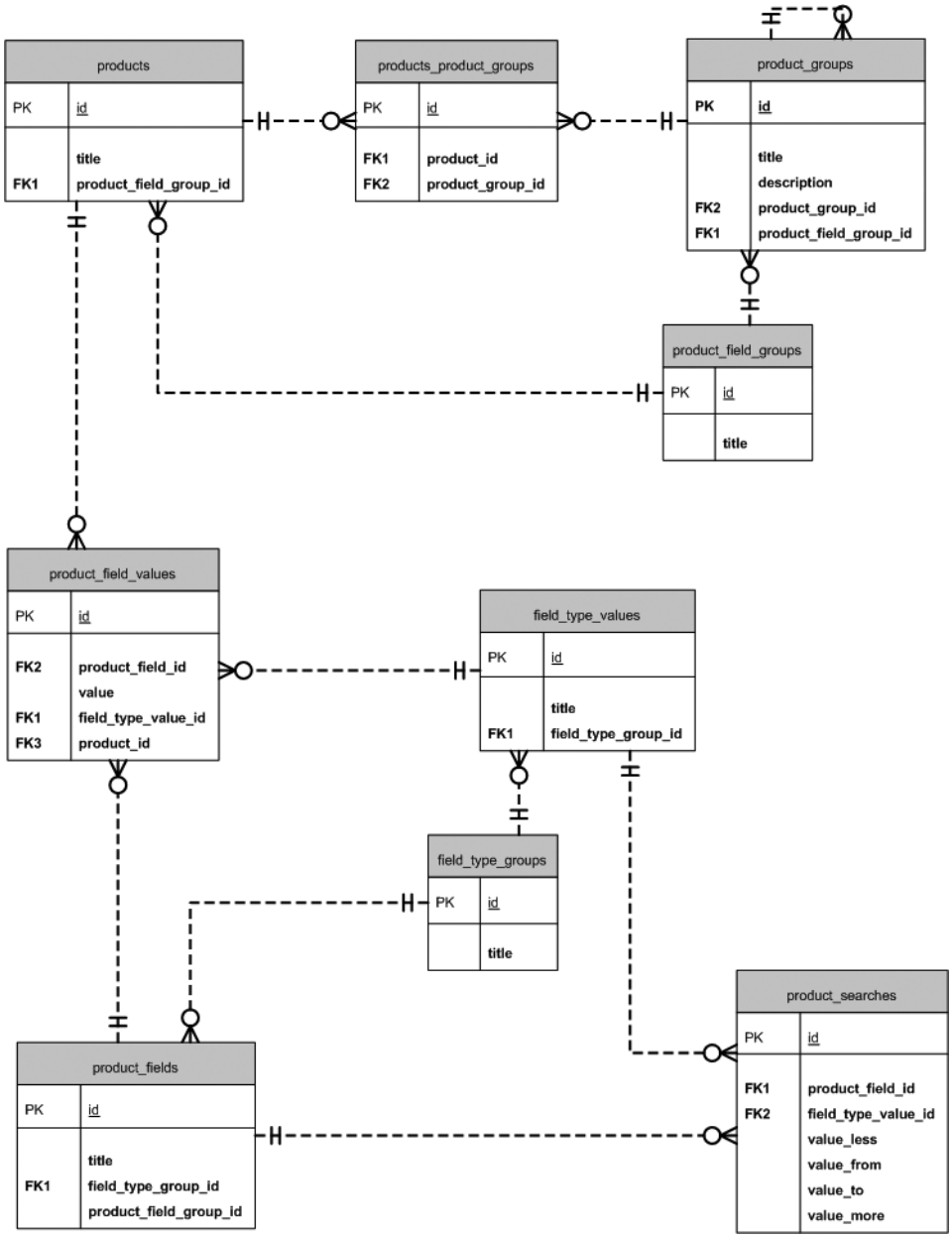

Figure 12-5. *Dynamic data fields schema*

We will now describe the type of data each table will contain and show some examples. This will give you enough information to allow you to adapt the code to suit your own needs.

The field_type_groups Table

The `field_type_groups` table is used to group the different data types, as shown in the example in Figure 12-6. For example, a price field is a `decimal` field; a free-form text field, such as the description of a product, is a `varchar` type; and a drop-down field of brand names is an enumerated type. Table 12-1 describes the two fields in this table.

←T→			id	title
☐	✎	✕	1	Decimal Field
☐	✎	✕	3	Varchar Field
☐	✎	✕	4	Television Brands

Figure 12-6. *Sample data in the field_type_groups table*

Table 12-1. *The field_type_groups Table Fields*

Field	Description
`id`	Unique primary key field
`title`	Name of the field type

The field_type_values Table

The `field_type_values` table holds the actual data types, as shown in the example in Figure 12-7. It's used with the `field_type_groups` table. It exists mainly for the benefit of enumerated types. Each field in this table is described in Table 12-2.

←T→			id	title	field_type_group_id
☐	✎	✕	3	Panasonic	4
☐	✎	✕	4	Samsung	4
☐	✎	✕	5	[DECIMAL]	1
☐	✎	✕	6	[VARCHAR]	3

Figure 12-7. *Sample data in the field_type_values table*

Table 12-2. *The field_type_values Table Fields*

Field	Description
`id`	Unique primary key field
`title`	Name of the data type
`field_type_group_id`	Foreign key to `field_type_groups`

The products Table

Each product will have one entry in the `products` table, as shown in the example in Figure 12-8. All the other tables are ultimately used to support this single table. Table 12-3 shows the fields in the `products` table.

←T→			id	title	product_field_group_id
☐	✎	✕	1	Panasonic AB-12XY34	1
☐	✎	✕	2	Panasonic AB-78XY89	1

Figure 12-8. *Sample data in the products table*

Table 12-3. *The products Table Fields*

Field	Description
`id`	Unique primary key field
`title`	Name of the product
`product_field_group_id`	Foreign key to `product_field_groups`

The products_product_groups Table

The `products_product_groups` table links the `products` table and the `product_groups` table. The `products` and `product_groups` table have a many-to-many relationship: one product can belong to many product groups, and a product group can contain many products. In Cake, this association is called has and belongs to many (HABTM). For example, SD memory cards can be used in many devices, such as cameras and computers, and devices like cameras and computers can take different types of SD memory cards. An example of the `products_product_groups` table is shown in Figure 12-9. Each field is described in Table 12-4.

←T→			id	product_id	product_group_id
☐	✎	✕	1	1	1
☐	✎	✕	2	1	9
☐	✎	✕	3	2	1
☐	✎	✕	4	2	9

Figure 12-9. *Sample data in the products_product_groups table*

Table 12-4. *The products_product_groups Table Fields*

Field	Description
`id`	Unique primary key field
`product_id`	Foreign key to `products`
`product_group_id`	Foreign key to `product_groups`

The product_fields Table

The `product_fields` table holds the name of the data fields, as shown in the example in Figure 12-10. Table 12-5 describes the four fields in this table.

←T→			id	title	field_type_group_id	product_field_group_id
☐	✎	✕	1	Product Brand	4	1
☐	✎	✕	2	Product Price	1	1

Figure 12-10. *Sample data in the product_fields table*

Table 12-5. *The product_fields Table Fields*

Field	Description
`id`	Unique primary key field
`title`	Name of the data field
`field_type_group_id`	Foreign key to `field_type_groups`
`product_field_group_id`	Foreign key to `product_field_groups`

The product_field_groups Table

The `product_field_groups` table is essentially used to hold together the different data fields in the `product_fields` table, as shown in the example in Figure 12-11. Table 12-6 describes the fields in this table.

←T→			id	title
☐	✎	✕	1	Television Data

Figure 12-11. *Sample data in the product_field_groups table*

Table 12-6. *The product_field_groups Table Fields*

Field	Description
`id`	Unique primary key field
`title`	Name of the grouping of the field (like a table name)

The product_field_values Table

After the `products` table, `product_field_values` is the second most important table, as it holds the attribute data for each product. Figure 12-12 shows an example of the table. Each of its fields is described in Table 12-7.

←T→			id	product_field_id	value	field_type_value_id	product_id
☐	✎	✕	1	1	Panasonic	3	1
☐	✎	✕	2	2	600	5	1
☐	✎	✕	3	1	Panasonic	3	2
☐	✎	✕	4	2	5000	5	2

Figure 12-12. *Sample data in the product_field_values table*

Table 12-7. *The product_field_values Table Fields*

Field	Description
`id`	Unique primary key field
`product_field_id`	Foreign key to `product_fields`
`value`	Value of the data itself—probably the most important field
`field_type_value_id`	Foreign key to `field_type_values`
`product_id`	Foreign key to `products`

The product_groups Table

The `product_groups` table is one found in many other applications. It simply groups the products and puts the groups in a hierarchical structure. The `product_group_id` field points back to itself and is used to record the hierarchy structure. Figure 12-13 shows an example of this table. Each field in the `product_groups` table is described in Table 12-8.

←T→			id	title	description	product_group_id	product_field_group_id
☐	✎	✕	1	LCD Displays		9	1
☐	✎	✕	6	Home	All Products	0	0
☐	✎	✕	9	Televisions		8	0
☐	✎	✕	8	Electronics		6	0
☐	✎	✕	13	Computer Monitors		8	0

Figure 12-13. *Sample data in the product_groups table*

Table 12-8. *The product_groups Table Fields*

Field	Description
`id`	Unique primary key field
`title`	Name of the group of products
`description`	Description of the group
`product_group_id`	Foreign key to `product_groups`
`product_field_group_id`	Foreign key to `product_field_groups`

The product_searches Table

The `product_searches` table holds the search criteria for each product group, as shown in the example in Figure 12-14. Table 12-9 describes the fields in this table. The fields that begin with `value_` are used in price range search filtering. For example, if `value_from` is `100` and `value_to` is `500`, then one of the search fields would allow you to search for products with the price range between $100 and $500. You'll see the code that uses these fields later in this chapter, in Listing 12-4 starting on line 39.

←T→			id	product_field_id	field_type_value_id	value_less	value_from	value_to	value_more
☐	✎	✕	1	1	3				
☐	✎	✕	2	1	4				
☐	✎	✕	3	2	5		500	1000	
☐	✎	✕	6	2	5		1001	10000	

Figure 12-14. *Sample data in the product_searches table*

Table 12-9. *The product_searches Table Fields*

Field	Description
`id`	Unique primary key field
`product_field_id`	Foreign key to `product_fields`
`field_type_value_id`	Foreign key to `field_type_values`
`value_less`	Used for price range filtering; filter for values less than this limit
`value_from`	Used for price range filtering; filter for values from this limit
`value_to`	Used for price range filtering; filter for values to this limit
`value_more`	Used for price range filtering; filter for values more than this limit

Baking for This Application

Using Cake's `bake` command, we generated all the models for this application automatically. Even more important is that we also created the associations in the models automatically. To generate the associations, we named the foreign key using the naming convention `[foreign_table_singular]_id`.

Regarding the controllers, we have baked only the products controller. In it, we will create two actions: the `search` action for searching and the `addData` action for adding a product. Because of the architecture of our tables, we cannot simply use the automatically generated `add` action to add products. However, that `add` action will still help us to some extent, as you'll see when we discuss adding products later in this chapter.

We have also baked some views, but again, only for the products.

Building the Product Search Feature

One of the main objectives of our dynamic data field approach is to allow flexible product searches. Users can refine their search to a narrow set of desired attributes. For our search feature, we need to create the search form and code the search process.

Creating the Product Search Form

Listing 12-2 shows the products controller for our application.

Listing 12-2. *The Products Controller (/app/controllers/products_controller.php)*

```
<?php
class ProductsController extends AppController {

    var $name = 'Products';
    var $helpers = array( 'Html', 'Form');

    var $uses = array( 'Product', 'ProductSearch', 'ProductGroup',
                       'ProductFieldValue', 'ProductField',
                       'FieldTypeValues' );

    function search() {

        if ( isset( $this->params['form']['product_group_id'] ) ) {

            $searchResult = $this->Product->doSearch( ➥
$this->params['form'] );

            $this->set( 'search_results', $searchResult );
        }

        $this->_listProductSearch();
    }

    function _listProductSearch() {

        // At the moment, this is hard coded to a default value of 1
        // but you may want to change this to your product group id
        $productGroupId = '1';

        if ( isset( $this->passedArgs[ 'productGroupId' ] ) ) {
            $productGroupId = $this->passedArgs[ 'productGroupId' ];
        }

        // Get the end node group
        $productGroup = $this->ProductGroup->findById( $productGroupId );

```

```
 36:            $this->set( 'product_group', $productGroup );
 37:
 38:            $this->set( 'product_fields',
 39:                        $this->ProductField->searchFilters( $productGroup ) );
 40:        }
 41:
 42:        function addData() {
 43:
 44:            if (!empty($this->data)) {
 45:
 46:                $fieldData = $this->data[ 'Product' ]['data_fields'];
 47:
 48:                foreach( $fieldData as $fieldVal ) {
 49:
 50:                    if ( is_array( $fieldVal ) ) {
 51:                        list( $key, $value ) = each( $fieldVal );
 52:
 53:                        $fieldData2Split = explode( ",", $key );
 54:                        $product_field_id = $fieldData2Split[0];
 55:                        $field_type_value_id = $fieldData2Split[1];
 56:                    }
 57:                    else {
 58:                        $fieldData2Split = explode( ",", $fieldVal );
 59:                        $product_field_id = $fieldData2Split[0];
 60:                        $field_type_value_id = $fieldData2Split[1];
 61:                        $value = ➥
$this->getFieldTypeValue( $field_type_value_id );
 62:                    }
 63:
 64:                    $data = array();
 65:                    $data[ 'ProductFieldValue' ][ 'product_field_id' ] = ➥
$product_field_id;
 66:                    $data[ 'ProductFieldValue' ][ 'value' ] = $value;
 67:                    $data[ 'ProductFieldValue' ][ 'field_type_value_id' ] = ➥
$field_type_value_id;
 68:                    $data[ 'ProductFieldValue' ][ 'product_id' ] = ➥
$this->data[ 'Product' ]['product_id'];
 69:
 70:                    $this->ProductFieldValue->create( $data );
 71:                    $this->ProductFieldValue->save();
 72:                }
 73:            }
 74:
 75:            if ( isset( $this->passedArgs[ 'productFieldGroupId' ] ) ) {
 76:
 77:                $productFieldGroupId = $this-> ➥
passedArgs[ 'productFieldGroupId' ];
 78:
```

```
                // Let's find all the data fields
                $productFields = $this->ProductField-> ➥
findAllByProductFieldGroupId( $productFieldGroupId );

                // Next we need the values relating to the data fields
                foreach( $productFields as &$field ) {

                    $fieldTypeGroupId = $field[ 'FieldTypeGroup' ][ 'id' ];

                    $fieldTypeValues = $this-> ➥
FieldTypeValues->findAllByFieldTypeGroupId( $fieldTypeGroupId );

                    $field[ 'FieldTypeValues' ] = $fieldTypeValues;
                }

                $this->set( 'field_type_values', $productFields );
            }

            $productId = '';
            if ( isset( $this->passedArgs[ 'productId' ] ) ) {
                $productId = $this->passedArgs[ 'productId' ];
            }

            $this->set( 'product_id', $productId );

        }

        function getFieldTypeValue( $field_type_value_id ) {

            $result = '';

            $fieldTypeValues = $this->FieldTypeValues-> ➥
findById( $field_type_value_id );

            if ( !empty( $fieldTypeValues ) ) {

                if ( isset( $fieldTypeValues ➥
[ "FieldTypeValues" ][ "title" ] ) ) {
                    $result = $fieldTypeValues[ "FieldTypeValues" ][ "title" ];
                }
            }

            return $result;
        }
    }
    ?>
```

We have created a `search` action in the products controller to handle searches, on line 11. Line 20 carries out the task of listing the product search fields in the `_listProductSearch` method. On line 23, we find all the data fields for a particular product group. On line 27, we assume the default product group ID is `1`. If no `productGroupId` value has been specified, then the products with a `productGroupId` of `1` will be shown. In practice, this value would be passed to the function via the URL, maybe as a named parameter. For example, you may have a page with different product group types, and one URL might look like this:

```
http://localhost/chapter_12/Products/search/productGroupId:42/
```

This URL would then list all the products with a `productGroupId` value of `42`.

Once we retrieve the `productGroupId`, on line 30, we simply get details about the group for display in the view. Line 38 does the main work and fetches the product fields.

Finding the search fields is carried out by the `ProductField` model, shown in Listing 12-3. Remember that the search fields are not hard-coded; they are generated dynamically.

Listing 12-3. *ProductField Model (/app/models/product_field.php)*

```
 1:    <?php
 2:    class ProductField extends AppModel {
 3:
 4:        var $name = 'ProductField';
 5:
 6:        var $belongsTo = array(
 7:            'FieldTypeGroup' => array('className' => 'FieldTypeGroup',
 8:                                      'foreignKey' => 'field_type_group_id',
 9:                                      'conditions' => '',
10:                                      'fields' => '',
11:                                      'order' => ''
12:            ),
13:            'ProductFieldGroup' => array('className' => 'ProductFieldGroup',
14:                                      'foreignKey' => 'product_field_group_id',
15:                                      'conditions' => '',
16:                                      'fields' => '',
17:                                      'order' => ''
18:            )
19:        );
20:
21:        var $hasMany = array(
22:            'ProductFieldValue' => array('className' => 'ProductFieldValue',
23:                                      'foreignKey' => 'product_field_id',
24:                                      'dependent' => false,
25:                                      'conditions' => '',
26:                                      'fields' => '',
27:                                      'order' => '',
28:                                      'limit' => '',
29:                                      'offset' => '',
```

```
30:                                 'exclusive' => '',
31:                                 'finderQuery' => '',
32:                                 'counterQuery' => ''
33:             ),
34:             'ProductSearch' => array('className' => 'ProductSearch',
35:                                 'foreignKey' => 'product_field_id',
36:                                 'dependent' => false,
37:                                 'conditions' => '',
38:                                 'fields' => '',
39:                                 'order' => '',
40:                                 'limit' => '',
41:                                 'offset' => '',
42:                                 'exclusive' => '',
43:                                 'finderQuery' => '',
44:                                 'counterQuery' => ''
45:             )
46:         );
47:
48:         function searchFilters( $searchField ) {
49:
50:             // We get the fields relating to this product group
51:             $productFields = array();
52:
53:             if ( $searchField['ProductGroup'][ 'product_field_group_id' ] ) {
54:
55:                 $productFieldGroupId = ➥
$searchField['ProductGroup'][ 'product_field_group_id' ];
56:
57:                 $fields = ➥
$this->findAllByProductFieldGroupId( $productFieldGroupId );
58:
59:                 foreach( $fields as $product_field ) {
60:
61:                     $productFields[] = ➥
$product_field[ 'ProductField' ][ 'id' ];
62:                 }
63:             }
64:
65:             // Next, for each field, we get the search criteria, e.g., list of
66:             // brand names, price range, etc.
67:             $fieldRefine = array();
68:
69:             foreach( $fields as $productField ) {
70:
71:                 $productFieldId = $productField[ 'ProductField' ][ 'id' ];
72:
```

```
73:                 $fieldSelection = ➥
$this->ProductSearch->findAllByProductFieldId( $productFieldId );
74:
75:                 $fieldValues = array();
76:
77:                 foreach( $fieldSelection as $selection ) {
78:                     $fieldValues[] = $selection;
79:                 }
80:
81:                 $productField[ 'field_selections' ] = $fieldValues;
82:
83:                 $fieldRefine[] = $productField;
84:             }
85:
86:             return $fieldRefine;
87:         }
88:
89:     }
90:     ?>
```

At the beginning of the file, we have created some associations that correspond to the diagram shown earlier in Figure 12-5. The important code begins on line 48 with the `searchFilters` method. First, we get the fields relating to the product group, starting on line 53. Next, starting on line 69, we get the search criteria for each field. For example, if the field is about brand names, we need a list of all the brands. If it is a price field, we need to get the search range of the price field. Figure 12-15 shows an example of the form displaying these criteria.

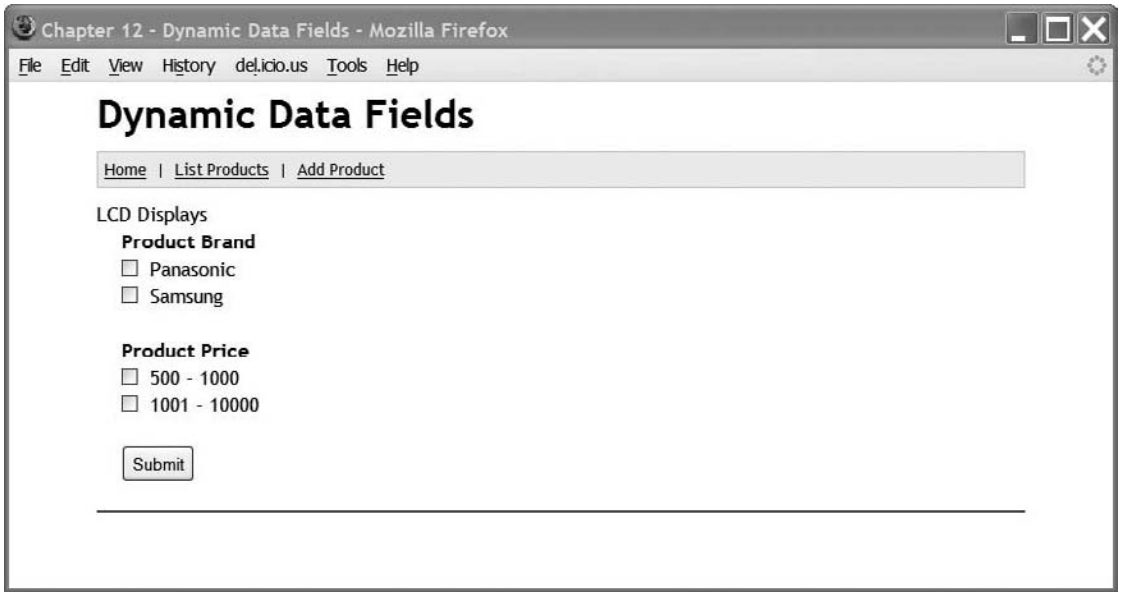

Figure 12-15. *The dynamic search form*

Let's now look at the view that corresponds to the search form, which is shown in Listing 12-4. The file is split into two sections: lines 1 through 105 list the search fields relating to the product group, and lines 107 through 140 list the products found.

Listing 12-4. *Search Action View (/app/views/products/index.ctp)*

```
  1:    <?php
  2:
  3:        echo $form->create( 'Product',
  4:                            array( 'url' => '/Products/search/' ) );
  5:
  6:    ?>
  7:
  8:    <?php echo $form->create('Product');?>
  9:
 10:    <?php
 11:
 12:        if ( isset( $product_group ) ) {
 13:
 14:            $product_group_id = $product_group['ProductGroup'][ 'id' ];
 15:
 16:            echo $form->hidden( 'product_group_id',
 17:                                    array(  'name' => 'product_group_id',
 18:                                            'value' => $product_group_id ) );
 19:        }
 20:    ?>
 21:
 22:    <?php
 23:
 24:        if ( isset( $product_group ) ) {
 25:
 26:            // List the product group
 27:            echo $product_group['ProductGroup'][ 'title' ];
 28:        }
 29:
 30:        echo '<br />';
 31:
 32:        echo '<div id="product_search">';
 33:
 34:            if ( isset( $product_group ) ) {
 35:
 36:                foreach( $product_fields as $product_field ) {
 37:
 38:                    echo ➥
'<b>'.$product_field[ 'ProductField' ][ 'title' ].'</b><br />';
 39:
 40:                    $field_selections = $product_field[ 'field_selections' ];
 41:
```

```
 42:                    foreach( $field_selections as $field_value ) {
 43:
 44:                        $title = $field_value[ 'FieldTypeValue' ][ 'title' ];
 45:                        $id = $field_value[ 'ProductSearch' ][ 'id' ];
 46:                        $product_field_id = ➥
$field_value[ 'ProductSearch' ][ 'product_field_id' ];
 47:
 48:                        if ( $title == '[DECIMAL]' ) {
 49:
 50:                            // We take it from the range
 51:                            $value_less = ➥
$field_value[ 'ProductSearch' ][ 'value_less' ];
 52:
 53:                            if ( $value_less ) {
 54:
 55:                                echo $form->checkbox( '', ➥
array( 'name' => 'field_selection['.$product_field_id.'][]', 'value' => $id ) );
 56:
 57:                                echo '<'.$value_less.'<br />';
 58:                            }
 59:
 60:                            $value_from = ➥
$field_value[ 'ProductSearch' ][ 'value_from' ];
 61:
 62:                            if ( $value_from ) {
 63:
 64:                                echo $form->checkbox( '', array( ➥
'name' => 'field_selection['.$product_field_id.'][]', 'value' => $id ) );
 65:
 66:                                echo $value_from;
 67:
 68:                                $value_to = ➥
$field_value[ 'ProductSearch' ][ 'value_to' ];
 69:
 70:                                if ( $value_to ) {
 71:
 72:                                    echo ' - '.$value_to.'<br />';
 73:                                }
 74:                            }
 75:
 76:                            $value_more = ➥
$field_value[ 'ProductSearch' ][ 'value_more' ];
 77:
 78:                            if ( $value_more ) {
 79:                                echo $form->checkbox( '', array( ➥
'name' => 'field_selection['.$product_field_id.'][]', 'value' => $id ) );
 80:
```

```
 81:                                 echo '>'.$value_more.'<br />';
 82:                             }
 83:                         }
 84:                         else {
 85:
 86:                             // For ordinary lists, e.g. brand names
 87:                             $id = $field_value[ 'ProductSearch' ][ 'id' ];
 88:
 89:                             echo $form->checkbox( '', array( ➥
'name' => 'field_selection['.$product_field_id.'][]', 'value' => $id ) );
 90:
 91:                             echo ➥
$field_value[ 'FieldTypeValue' ][ 'title' ].'<br />';
 92:                         }
 93:                     }
 94:
 95:                     echo '<br />';
 96:                 }
 97:
 98:                 echo $form->end('Submit');
 99:             }
100:
101:         echo '</div>';
102:
103:
104:
105:     ?>
106:
107:     <br />
108:     <hr width="100%" color="#555555">
109:     <br />
110:
111:     <?php
112:
113:         if ( isset( $search_results ) ) {
114:
115:             foreach( $search_results as $a_result ) {
116:
117:                 $product_fields = $a_result[ 'product_fields' ];
118:
119:                 if ( isset( $product_fields[0][ 'Product' ][ 'title' ] ) ) {
120:                     $product_title = $product_fields[0][ 'Product' ][ 'title' ];
121:                     echo '<p>'.$product_title.'</p>';
122:                 }
123:
124:                 foreach( $product_fields as $product_field ) {
125:
```

```
126:                    $field_title = $product_field[ 'ProductField' ][ 'title' ];
127:
128:                    switch ( $field_title ) {
129:
130:                        case 'Product Brand':
131:                            echo '<p><b>Brand:</b> '.$product_field ➥
[ 'ProductFieldValue' ][ 'value' ].'</p>';
132:                            break;
133:
134:                        case 'Product Price':
135:                            echo '<p><b>Price:</b> '.$product_field ➥
[ 'ProductFieldValue' ][ 'value' ].'</p>';
136:                            break;
137:                    }
138:                }
139:            }
140:        }
141:
142:    ?>
```

Lines 12 to 19 simply store the product group ID in a hidden field, so that when we perform the search, we know on which product group we're searching. The `if` statement on line 24 echoes the name of the product group—for example, Plasma TVs. On line 36, the big outer `foreach` loop goes through each field and displays the relevant filter input box and the label that goes with it.

Inside the big `foreach` loop on line 42, we have an `if` statement on line 48 that decides on the data type of the data field. If it's a decimal, we assume it's a price range filter. We then go through the `value_` variables to find what kind of range we should display. If it's not a decimal, we assume it's an enumerated list, like brand names or shoe sizes. We then go into the `else` block on line 84, where we echo a check box and the title of the field type.

Processing the Search

In the previous section, we showed you how the search form was created. We'll now explain how the search actually works. Referring to the products controller in Listing 12-2, on line 15, you can see that all the work to search for the products is done within the `Product` model. The code for the `Product` model is shown in Listing 12-5.

Listing 12-5. *Product Model (/app/models/product.php)*

```
1:    <?php
2:    class Product extends AppModel {
3:
4:        var $name = 'Product';
5:
6:        var $belongsTo = array(
7:                'ProductFieldGroup' => array('className' => 'ProductFieldGroup',
```

```
 8:                             'foreignKey' => 'product_field_group_id',
 9:                             'conditions' => '',
10:                             'fields' => '',
11:                             'order' => ''
12:                     )
13:         );
14:
15:         var $hasMany = array(
16:                     'ProductFieldValue' => array('className' => 'ProductFieldValue',
17:                             'foreignKey' => 'product_id',
18:                             'dependent' => false,
19:                             'conditions' => '',
20:                             'fields' => '',
21:                             'order' => '',
22:                             'limit' => '',
23:                             'offset' => '',
24:                             'exclusive' => '',
25:                             'finderQuery' => '',
26:                             'counterQuery' => ''
27:                     )
28:         );
29:
30:         var $hasAndBelongsToMany = array(
31:                     'ProductGroup' => array('className' => 'ProductGroup',
32:                             'joinTable' => 'products_product_groups',
33:                             'foreignKey' => 'product_id',
34:                             'associationForeignKey' => 'product_group_id',
35:                             'unique' => true,
36:                             'conditions' => '',
37:                             'fields' => '',
38:                             'order' => '',
39:                             'limit' => '',
40:                             'offset' => '',
41:                             'finderQuery' => '',
42:                             'deleteQuery' => '',
43:                             'insertQuery' => ''
44:                     )
45:         );
46:
47:         function doSearch( $formValues ) {
48:
49:             $result = array();
50:
51:             $productGroupId = $formValues[ 'product_group_id' ];
52:
53:             if ( isset( $formValues[ 'field_selection' ] ) ) {
54:
```

```
 55:                $result = $this->getFieldSelection(
 56:                                    $productGroupId,
 57:                                    $formValues[ 'field_selection' ] );
 58:
 59:                // We need to add the field information to each product
 60:                $result = $this->addFieldInformation( $result );
 61:            }
 62:
 63:            return $result;
 64:        }
 65:
 66:        function addFieldInformation( $products ) {
 67:
 68:            foreach( $products as &$current_product ) {
 69:
 70:                $productId = $current_product['id'];
 71:
 72:                $productFields = $this->ProductFieldValue-> ➥
findAllByProductId( $productId  );
 73:
 74:                $current_product[ 'product_fields' ] = $productFields;
 75:            }
 76:
 77:            return $products;
 78:        }
 79:
 80:        function getSearchCriteria( $fieldSelection ) {
 81:
 82:            $selectionFlat = array();
 83:
 84:            $idx = 0;
 85:
 86:            foreach( $fieldSelection as &$currentSelection ) {
 87:
 88:                for( $idx2=0; $idx2<sizeof( $currentSelection ); $idx2++ ) {
 89:
 90:                    $selectionValue = $currentSelection[$idx2];
 91:
 92:                    $sql = "    select * from product_searches
 93:                                where
 94:                                id = '$selectionValue'
 95:                                ";
 96:
 97:                    $searchs = $this->query($sql);
 98:
 99:                    $selectionFlat[$idx][$idx2] = $searchs[0];
100:                }
101:
```

```
102:                $idx++;
103:            }
104:
105:            return $selectionFlat;
106:        }
107:
108:        function getFieldSelection( $productGroupId, $fieldSelection ) {
109:
110:            $result = array();
111:
112:            // Get all products within the group
113:            $products = $this->ProductGroup->findById( $productGroupId );
114:
115:            $searchCriterias = $this->getSearchCriteria( $fieldSelection );
116:
117:            // Now walk through each product to filter out
118:            // according to user selection
119:
120:            foreach( $products[ 'Product' ] as $currentProduct ) {
121:
122:                $productId = $currentProduct[ 'id' ];
123:                $productFieldGroupId = $currentProduct ➥
[ 'product_field_group_id' ];
124:
125:                $sql = "    select * from product_fields
126:                            inner join product_field_values ➥
on product_field_values.product_field_id = product_fields.id
127:                            inner join field_type_values on ➥
field_type_values.id = product_field_values.field_type_value_id
128:                            where
129:                            product_fields.product_field_group_id = ➥
'".$productFieldGroupId."' and
130:                            product_field_values.product_id = '".$productId."'
131:                            ";
132:
133:                $productFields = $this->query($sql);
134:
135:                $topLevelMatches = 0;
136:
137:                // Walk through each field
138:                foreach( $productFields as $field ) {
139:
140:                    // Now match against the selection
141:                    // Top level groups must match with an "AND"
142:                    // e.g., brand, price, etc.
143:                    // While second level groups match with an "OR"
144:                    // e.g., Panasonic, Sony, etc.
145:
```

```
146:                    // Start with top level
147:                    for( $idx=0; $idx<sizeof( $searchCriterias ); $idx++ ) {
148:
149:                        $subLevels = $searchCriterias[$idx];
150:                        $subLevelMatching = 0;
151:
152:                        // Sub level
153:                        for( $idx2=0; $idx2<sizeof( $subLevels ); $idx2++ ) {
154:
155:                            if (    (   $subLevels[$idx2] ➥
['product_searches'][ 'product_field_id' ] ==
156:                                    (   $field ➥
[ 'product_field_values' ][ 'product_field_id' ] ) ) )
157:                            {
158:                                // User selected this field to filter
159:                                // Check if selection matches this field
160:
161:                                if ( ➥
$field[ 'field_type_values' ][ 'title' ] == '[DECIMAL]' ) {
162:
163:                                    $valueLess = ➥
$subLevels[$idx2]['product_searches'][ 'value_less' ];
164:
165:                                    if ( $valueLess ) {
166:                                        if ( $field ➥
[ 'product_field_values' ][ 'value' ] < $valueLess ) {
167:                                            $subLevelMatching = 1;
168:                                        }
169:                                    }
170:
171:                                    $valueFrom = $subLevels ➥
[$idx2]['product_searches'][ 'value_from' ];
172:                                    $valueTo = $subLevels ➥
[$idx2]['product_searches'][ 'value_to' ];
173:
174:                                    if ( ( $valueFrom ) && ( $valueTo ) ) {
175:                                        if (    ( $field ➥
[ 'product_field_values' ][ 'value' ] > $valueFrom ) &&
176:                                                ( $field ➥
[ 'product_field_values' ][ 'value' ] < $valueTo ) )
177:                                        {
178:                                            $subLevelMatching = 1;
179:                                        }
180:                                    }
181:
182:                                    $valueMore = $subLevels ➥
[$idx2]['product_searches'][ 'value_more' ];
183:
```

```
                                    if ( $valueMore ) {
                                        if ( $field ➥
[ 'product_field_values' ][ 'value' ] > $valueMore ) {
                                            $subLevelMatching = 1;
                                        }
                                    }
                                }
                                else {

                                    // If plain id selection, e.g., brand names,
etc.
                                    if (    $field ➥
[ 'product_field_values' ][ 'field_type_value_id' ] ==
                                            $subLevels ➥
[$idx2]['product_searches'][ 'field_type_value_id' ] ) {

                                        $subLevelMatching = 1;
                                    }
                                }
                            }
                        }

                        // Count how many
                        if ( $subLevelMatching ) {
                            $topLevelMatches++;
                        }
                    }
                }

                if ( $topLevelMatches == sizeof( $searchCriterias ) ) {
                    $result[] = $currentProduct;
                }
            }

            return $result;
        }

    }
    ?>
```

The search starts with `doSearch` on line 47. Starting on line 55, the `getFieldSelection` method gets the fields that were selected. As you may notice, we have used some raw SQL queries, which is sometimes necessary with such a complex system.

The `getFieldSelection` method also does the work of finding the products. On line 113, we first get all the products within the product group. Next, on line 115, we get the search criteria that the user selected. Then, beginning on line 120, we walk through each product and check it against the search criteria.

On line 125, we get all the data fields for the product. Then, on line 138, we take each field and try to match it against the search criteria values.

Let's consider the example shown earlier in Figure 12-15. If a user selects Panasonic and the price range 500–1000, that means she wants to find Panasonic Plasma TVs in the price range of $500 to $1,000. However, if the user also selects Samsung, that means she wants to find Panasonic or Samsung Plasma TVs in the price range of $500 to $1,000. Since we cannot use an ordinary `SELECT` statement with `AND` and `OR` operators in the `WHERE` clause, we need to write the code to perform the equivalent functionality of these two operators. This is done from line 147 to line 212.

Adding a Product

The process of adding a product is carried out in two stages: we add entries in the `products` and `products_product_groups` tables and then add the product data. (This could be combined into one step, but we'll leave that as an exercise for those who are interested.)

Creating Table Entries

First, we need to create an entry in the `products` table. This is quite simple, as we will use the action `/Products/add` that we baked, as mentioned earlier in the chapter. (We have not altered any of the baked code.) Figure 12-16 shows an example of the form, where we are adding a Panasonic AB-WOW product.

The important point regarding the `products` table is the relationship it has with the `product_groups` table. During the baking process, Cake generated the relevant associations for us. Because it knows about the associations, Cake will automatically carry out many operations for us, including the following:

- In the view of an `add` action, a multiselect field is automatically generated so users can select more than one entry in the product group, as shown in Figure 12-16.
- During the save process, entries for the other tables are automatically created. In Figure 12-16, we have selected two product groups. The model's `save` method will actually create three records: one for the `products` table and two for the `products_product_groups` link table.

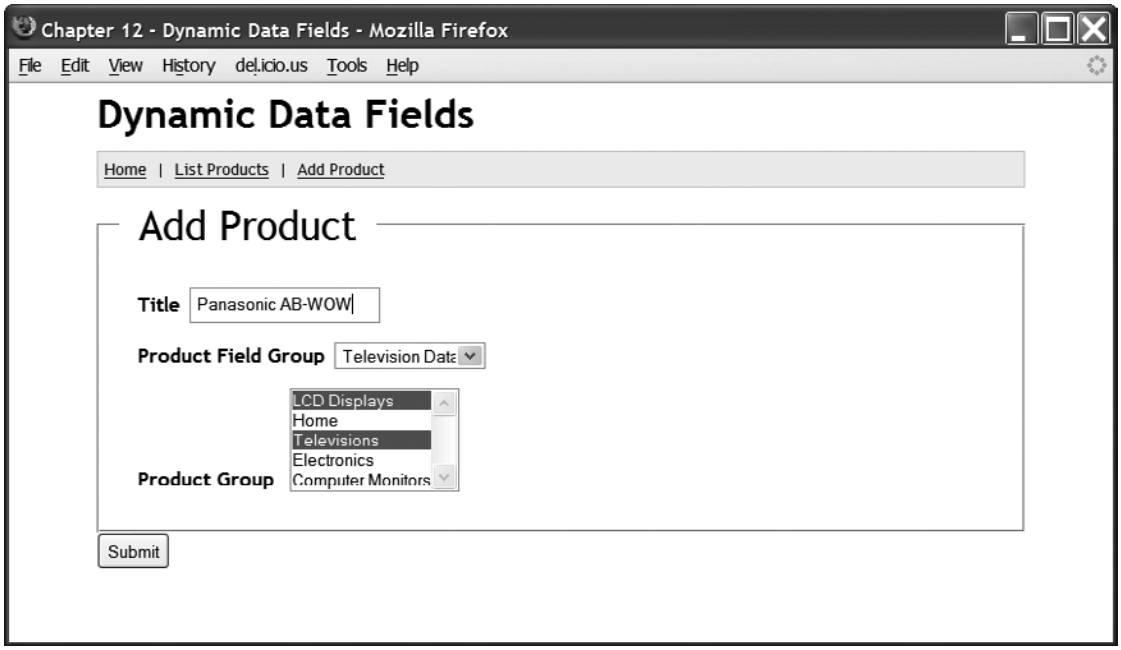

Figure 12-16. *The add product form*

Entering Product Data

Now we have created an entry in the `products` table for the new product, and we have also created two entries in the `products_product_groups` table. So, we know to which product group the product belongs. However, we still need to enter the actual data for the product, such as its price and specific attributes. For this task, we need to write specific code.

Figure 12-17 shows the form that lists the products. Most of what you see in Figure 12-17 is baked. The exception is the Add Data link in the middle of the Actions column. For each product, we need to add the data for the product attributes. For example, clicking the Add Data link for the Panasonic AB-WOW entry will take you to the form shown in Figure 12-18.

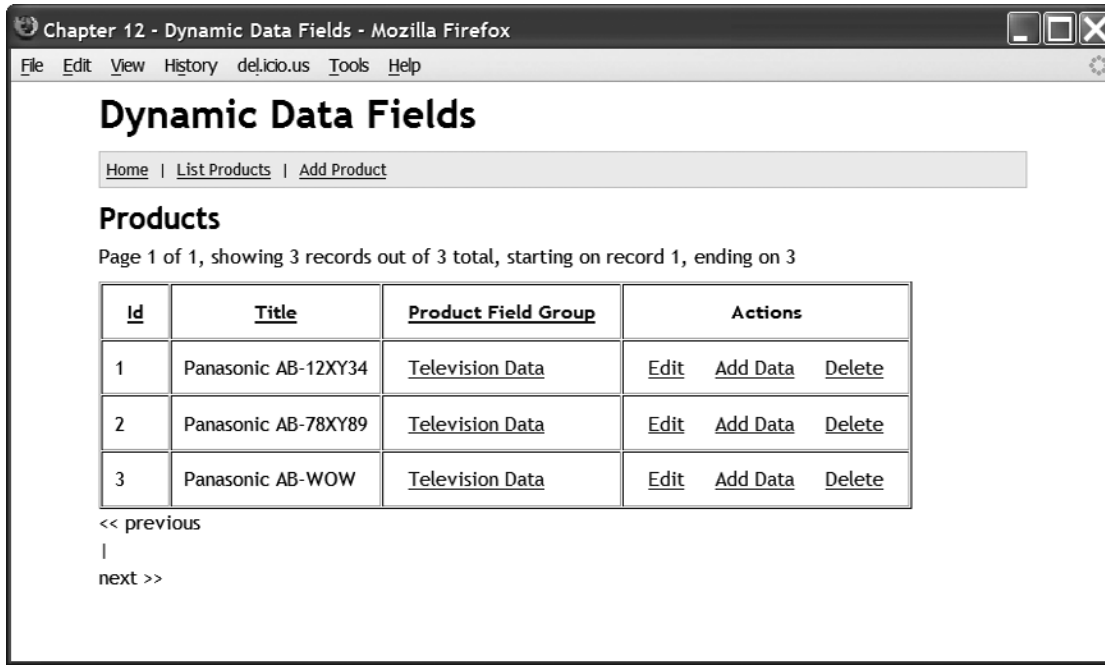

Figure 12-17. *Listing of the products*

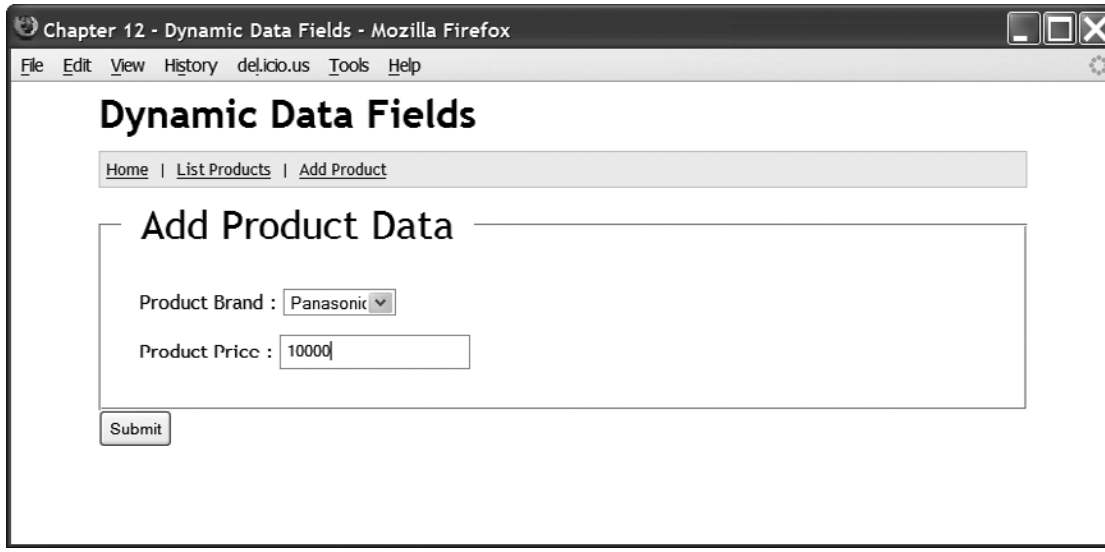

Figure 12-18. *Adding data for a product*

The Add Data link is in the following format:

```
/Products/addData/productId:6/productFieldGroupId:2
```

As you can see from the link, there is an `addData` action in the products controller (shown earlier in Listing 12-2). We pass into the action two named parameters: `productId` and `productFieldGroupId`. The code for the `addData` action is shown on line 42 in Listing 12-2.

When the product data is returned, it goes into the `if` statement on line 44 in Listing 12-2. To give you a better understanding of the code, the following is an example of the `$this->data` variable as used on line 44:

```
 1:    Array
 2:    (
 3:        [Product] => Array
 4:            (
 5:                [data_fields] => Array
 6:                    (
 7:                        [0] => 3,4
 8:                        [1] => Array
 9:                            (
10:                                [4,5] => 50
11:                            )
12:
13:                        [2] => Array
14:                            (
15:                                [5,5] => 8
16:                            )
17:
18:                    )
19:
20:                [product_id] => 6
21:            )
22:
23:    )
```

In this listing, the `Product` key has two elements: `data_fields` and `product_id`. We need the `product_id` so we know to which product the data is attached. The `data_fields` element contains the product data itself. The numeric order in the `data_fields` element is not important; the important part is the value. If the value is an array (see line 50 in Listing 12-2), then the key is composed of two values in the format `[ product_field_id, field_type_value_id ]`. Thus, we have all the information we need about the data: the data itself, the data type, and the field it is. If the value is not an array, then it's coming from a select list (in our example, the brand names). In this case, we will just have the `[ product_field_id, field_type_value_id ]` pair. It won't have a value, since we already know the value, as it is in the enumerated list. Remember that the data fields are not hard-coded, so they don't actually exist in the database as real data columns. As far as the database is concerned, these fields are just data.

The view that generated Figure 12-18 is shown in Listing 12-6. The form simply sends the data back to itself in the `addData` action. In the products controller in Listing 12-2, the data starts getting processed on line 44 onward.

Listing 12-6. *AddData View (/app/views/products/add_data.ctp)*

```
 1:    <div class="products form">
 2:    <?php echo $form->create('Product',array( ➥
'url' => '/Products/addData' ));?>
 3:        <fieldset>
 4:             <legend><?php __('Add Product Data');?></legend>
 5:        <?php
 6:
 7:            if ( isset( $field_type_values ) ) {
 8:
 9:                foreach( $field_type_values as $field ) {
10:
11:                    $fieldTypeValues = $field[ 'FieldTypeValues' ];
12:
13:                    $productFieldId = $field[ 'ProductField' ][ 'id' ];
14:                    $productFieldTitle = $field[ 'ProductField' ][ 'title' ];
15:                    $productFieldTitle_2 = low( $productFieldTitle );
16:                    $productFieldTitle_2 = r( " ",
17:                                                  "_",
18:                                                  $productFieldTitle_2 );
19:
20:                    echo '<div class="form_field">';
21:
22:                        echo $productFieldTitle.' : ';
23:
24:                        // Check type of data
25:                        if ( sizeof( $fieldTypeValues ) == 1 ) {
26:
27:                            // Get field type value id
28:                            $FieldTypeValueId = ➥
$field[ 'FieldTypeValues' ][0][ 'FieldTypeValues' ][ 'id' ];
29:
30:                            // It's a basic data type like int or a string
31:                            // and not a list of data items
32:
33:                            echo $form->input( '', array(   'label' => false,
34:                                                            'div' => false,
35:                                                            'name' => ➥
'data[Product][data_fields][][].$productFieldId.',.$FieldTypeValueId.']' ) );
36:                            echo '<br />';
37:                        }
38:                        else {
39:                            // We'll assume it's a list of data items
40:                            // Let's gather the data together
41:                            $listItems = array();
42:                            foreach( $fieldTypeValues as $fieldValue ) {
43:
```

```
                                $id = $fieldValue[ 'FieldTypeValues' ][ 'id' ];
                                $title = ➥
$fieldValue[ 'FieldTypeValues' ][ 'title' ];

                                $listItems[ $productFieldId.','.$id ] = $title;
                            }

                            echo $form->select( '',
                                                $listItems,
                                                null,
                                                array( ➥
'name' => 'data[Product][data_fields][]' ) );
                            echo '<br />';
                        }

                    echo '</div>';
                }
            }

        ?>
        </fieldset>
    <?php echo $form->hidden(   'product_id',
                                array( 'value' => $product_id ) ); ?>

    <?php echo $form->end('Submit');?>
    </div>
```

Summary

Now that we're at the end of the chapter, we hope you are still with us. As we said at the beginning of the chapter, our design is quite complex because we are essentially trying to do some of the job that the SQL engine usually does for us. Essentially, we have carried out some research and development work. And along the way, we've used Cake's has-and-belongs-to-many association, as well as raw queries using the `query` function. By stepping out of the norm, we can see how Cake responds to new ideas.

This chapter contains the skeleton of an application. You can take the code in numerous directions. For example, in the `product_groups` table, instead of using a single `product_group_id` field to represent a hierarchical tree structure, you could use the preorder tree traversal algorithm (see `http://www.sitepoint.com/article/hierarchical-data-database`). This ordering method allows you to fetch all products with certain attributes under a complete branch. For example, you can list all televisions between the price range of $500 and $1,000. You can achieve the same result using a single `product_group_id` field, but it would involve more code and more SQL statements.

Additionally, you might add a back-end administration area, where administrators can easily manage products, product groups, and search criteria.

CHAPTER 13

Captcha

Nowadays, most developers would not leave any web forms open to unlimited submission. Some safety measure must be put in place to prevent web forms from being used for spamming.

While some people focus on building creative web sites and extending the capability of the Internet, others spend considerable amount of time trying to crack and compromise the information on web sites. Spamming is a huge problem. A recent statistic from Sophos research revealed that during the first quarter of 2008, 96.5 percent of all e-mail was spam (`http://www.sophos.com/pressoffice/news/articles/2008/07/dirtydozjul08.html`)!

Spambots are everywhere. They simply engage in the process of filling out web forms as if they were customers. For example, a spambot could send thousands of spontaneous e-mail messages by filling out your contact form or your blog comment form, if that form is not protected.

You've probably encountered web sites where you're required to interpret some obfuscated characters and input them for validation when a web form is submitted. You will most often see this type of protection on high-profile sites like Google's Gmail and Yahoo! Mail. This fuzzy character output is called a *Captcha*.

In this chapter, we will briefly look at the various types of Captchas used as a means of protecting web forms. We'll then focus on the ASCII Art Captcha technique and implement an ASCII Art Captcha component.

Captcha Implementations

The term *Captcha* was coined in 2000 by Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. It is a shortened acronym for Completely Automated Public Turing Test to Tell Computers and Humans Apart. The purpose of a Captcha is to prevent automatic form submission by spambots or similar intrusion programs.

In 1950, the brain behind the advent of Captcha, Professor Alan Turing, wrote an article called "Imitation Game" to describe how machines can demonstrate intelligence similar to humans. In 1997, Alta Vista created an early spam-blocking measure. Now Captchas are common. There is even a company (reCAPTCHA, at `http://recaptcha.net/`) that offers Captcha security as a web service. Figure 13-1 shows an example of a reCAPTCHA Captcha box.

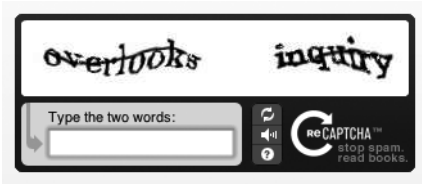

Figure 13-1. *A reCAPTCHA box*

Captcha Types

There are different types of Captchas and consequently many ways of implementing them. This presents us with many options when it comes to blocking spammers from using robots for submitting web forms. The following list describes some of the common implementations of Captcha designed to completely block automatic programs from web form submission.

Alphanumeric images: The most common Captcha implementation is the appearance of a random selection of distorted images made up of alphanumeric values. As humans, we can easily recognize distorted image characters, but spambots cannot. It's more challenging for spambots to crack the Captcha when the image characters are overlapping or distorted with lines across the characters.

Picture images: Another Captcha implementation uses a set of various images, such as animal images (a bird, a fish, an elephant, and so on) or furniture (table and chairs, for instance). You're expected to recognize the objects by their names and enter their names into an input box.

Audio: This technique involves embedding an audio (sound) to pronounce some words or random letters and digits. You're expected to type the words into an input box. Unfortunately, this requires some audio player, which not everyone has. Also, it makes things difficult for people with hearing problems.

Question/answer: A question-and-answer technique involves asking a user a question. If it's a difficult question, some potential users may be blocked from form submission.

Math problems: Using this technique, you are given a mathematical question (such as 56 minus 30), and you are expected to input the resulting value before you can submit the web form.

ASCII Art: This technique displays a set of fonts that are created from a combination of characters artistically designed to form gigantic versions of some keyboard characters (A–Z, 0–9, @, and so on). This set of fonts is called the ASCII Art characters.

The Captcha implementation presented in this chapter uses ASCII Art, so we'll take a closer look at that approach.

ASCII Art Captcha

Although most of the standard Captchas on the Internet display graphic characters, users sometimes find it difficult to decipher all the characters correctly, perhaps due to their screen resolution, sight, or both. Many people find it is easier to recognize ASCII Art characters compared to their graphic counterparts.

Note ASCII Art is the creation of images by using the strokes of the characters defined by the ASCII Standard as lines and shading. It is a mini-industry in itself. For more information, just search for "ASCII Art" on the Internet. You can start by going to `http://chris.com/ascii/` for some enlightening information.

As new ways of protecting web information are evolving, so are techniques for cracking that protection. Spammers are finding ways of decoding them, typically using optical character recognition (OCR) programs. Not many web forms use ASCII Art, so many deciphering spambot programs find it difficult to decode.

Another advantage ASCII Art Captchas have over graphics-based Captchas has to do with external libraries. Some of the graphics-based Captchas rely on PHP extensions (for example, the PHP GD library). The ASCII Art Captchas do not rely on any external library for their operation. They simply rely on your design of the characters.

One of the major characteristics of a Captcha is its look and feel. It includes the overlapping of characters and lines that touch the characters. To some extent, this characteristic is dependent on the spacing between characters and the fonts of the individual characters. One of the advantages of using ASCII characters is that you have total control over the look and feel of the characters.

ASCII Art Captchas offer a very strong way of thwarting spambots. First, the spambot would need to decipher which keyboard characters are used to compose a single individual character. A single font character could combine some hash (#) characters, some pipe (|) characters, a few ampersands (&), and so on. Additionally, spambots face the problems of determining the start and end of characters, actual borders or coordinates of each character, and the position of the random text, as well as distinguishing the background noise from the character.

In our ASCII Art Captcha implementation outlined in this chapter, our array of fonts will define all characters of the alphabet from A–Z and the numbers 0–9. With this array of fonts, you can modify the characters to suit your own preferences. Here is an example of a constructed font character:

```
#!#!#!#!#!#
    #!#
    #!#
    #!#
    #!#
    #!#
    #!#
```

This is clearly a letter *T*. It's constructed using a combination of the hash (#) and exclamation point (!) characters.

A Captcha Component

Now we will take you through implementing an ASCII Art Captcha. This technique will involve the creation of an array of fancy fonts (ASCII Art) using our own custom keyboard character combinations. Our Captcha characters will be randomly drawn from the array of the ASCII Art fonts, distorted, and finally displayed on a simple web form.

The Captcha program will be rolled into a component, so that it can be reused when required in other Cake applications. Table 13-1 describes the properties of the ASCII Art Captcha component.

Table 13-1. *The Properties of the Captcha Component*

Property	Description
`fonts`	Consists of ASCII Art characters. Each character is built by putting together some keyboard characters to form a whole character. You can create your own pattern of characters to replace these fonts. As the name implies, the individual characters of the fonts are determined from the standard alphanumeric values A–Z and 0–9.
`noiseChars`	Defines a set of characters that includes a dash (`-`) character stored as an array that is used as the background of ASCII Art characters.
`numberChar`	A numeric value that must be greater than zero. It's used to determine the total number of characters in the CAPTCHA string text.

Of course, you can add to the list of the properties—for example, include background coloring—to intensify security.

The ASCII Art Component Class

Our component class is named `AsciiArtsComponent`. Listing 13-1 shows the properties in this class.

Listing 13-1. *The Beginning of the AsciiArtsComponent Class (app/controllers/components/ascii_captcha.php)*

```
<?php
class AsciiArtsComponent extends Object {
    var $noiseChars = array( ' ', "=", "-", ":" );
    var $asciiFonts = array();
    var $numberChar = 6;
?>
```

The `AsciiArtsComponent` class contains three properties:

- `$noiseChars` is assigned an array of four ASCII characters: `(' ', "=", "-", ":")`. These characters are randomly used to distort the Captcha characters.
- `$asciiFonts` is initially assigned an empty array. This property will later contain a list of all the defined ASCII Art characters for A–Z and 0–9.
- `$numberChar` is assigned an integer `6`, which specifies the total number of characters that will constitute the Captcha.

Now that we've dealt with the component properties, let's look at the functionality that we'll implement to build the ASCII Art Captcha. This functionality includes generating random text from the ASCII character fonts, distorting the randomly generated text, and writing the text to the screen.

Next up in our component class is Cake's `startup` function, as shown in Listing 13-2.

Listing 13-2. *AsciiArtsComponent's Startup Method*

```
function startup(&$controller) {
    $this->data = $controller->data;
}
```

You should be familiar with the `startup` method by now. In this case, it gives the `AsciiArtsComponent` access to the properties of its parent controller, which is the `CaptchaController` object. For example, it enables you to access a form object data (`$this->data`) submitted to a controller within a component.

Next, the `initialize` method performs a similar task to the `beforeFilter` method of the controller. This method contains logic that must be run before any component functionality is run. We initialize the `$fonts` property with a set of ASCII Art characters. This `font` property contains an array of the ASCII Art characters that we created from different sets of characters. For example, Listing 13-3 shows the first two elements of the `$fonts` array.

Listing 13-3. *AsciiArtsComponent's Initialize Method*

```
function initialize() {
$fonts['A'] =
"  @@@
  @@-@@
 @@---@@
@@- ---@@
@@@@@@@@@
@@- - -@@
@@-----@@ ";

$fonts['B'] =
"@@@@@@@@
@@------@@
@@---- -@@
@@@@@@@@@
@@------@@
@@- ----@@
@@@@@@@@@";

// Other alphabet characters follow...

}
```

Listing 13-3 shows the created ASCII Art characters for the first two elements (A and B) of the `$fonts` array variable (`$fonts['A'],…..$fonts['Z'], $fonts['0'],…..$fonts['9']`). In total, the `$fonts` array contains 36 elements. We are going to randomly draw a number of characters from this array, based on the `$numberChar` property of the component.

Next in this class is the `getCaptcha` method, which returns a Captcha string, as shown in Listing 13-4.

Listing 13-4. *AsciiArtsComponent's getCaptcha() Method*

```
function getCaptcha() {
        $result = array();
        $captchaChars = array_rand( $this->asciiFonts, $this->numberChar );

        for ($idx = 0; $idx < sizeof( $captchaChars ); $idx++) {
           $capChar = $captchaChars[$idx];
           $result[ $capChar ] = $this->asciiFonts[ $capChar ];
        }

        $result = $this->addNoise( $result );

        return $result;
}
```

The first statement in this method initializes the `$result` variable with an empty array. This variable will eventually contain the final result of this function. Next, we use the PHP function `array_rand` to randomly select a list of characters from this function's first argument, `$this->asciiFonts`. The total number of characters selected is determined by the second argument, `$this->numberChar`, which has a default value of 6. In this case, a set of six randomly selected characters is stored in the `$captchaChars` array variable.

Next, we use a `for` loop to store each character in `$capChar` variable. That variable is used as a key to pull the corresponding ASCII Art characters from the `$asciiFonts` array, and then stored in the `$result` array variable. The `for` loop's maximum iteration is based on the number of elements in the `$captchaChars` array variable.

Finally, we use the `addNoise` method to distort the ASCII Art characters and return the final Captcha result to be used in the `CaptchaController` object.

The next method in our component class is `addNoise`, which takes an array of the randomly selected ASCII Art characters and returns a distorted version of the character, as shown in Listing 13-5.

Listing 13-5. *AsciiArtsComponent's addNoise() Method*

```
function addNoise( $captchaStrings ) {

        $result = array();

        foreach( $captchaStrings as $capChar => $ascii ) {

            for ($idx = 0; $idx < strlen( $ascii ); $idx++) {
```

```
                if ( $ascii[$idx] == ' ' ) {

                     $noiseChar = array_rand( $this->noiseChars, 1 );
                    $ascii[$idx] = $this->noiseChars[$noiseChar];
                }
            }

            $ascii = str_replace( chr(13), '  ', $ascii );

            $result[ $capChar ] = $ascii;
        }

        return $result;
  }
}
?>
```

In Listing 13-5, we first set the `$result` variable to an empty array. Next, we use the `foreach` loop to iterate the list of the ASCII Art characters stored in `$captchaStrings`, and then use an inner `for` loop to iterate each ASCII Art character and check whether a character is empty or a whitespace character. If a character is a whitespace character, the program randomly selects a replacement character from the `$noiseChars` array variable. The final `$ascii` character is stored and returned as the `$result` array variable.

Now that we are finished with the `AsciiArtsComponent` properties and methods, let's move to the controller that employs the services of this component to provide security against a spambot's submission of our sample form.

The Captcha Controller

Next, we will create the `CaptchaController` class that displays a form to test our Captcha. It starts as shown in Listing 13-6.

Listing 13-6. *The Beginning of the CaptchaController Class (app/controllers/captcha_controller.php)*

```
<?php
class CaptchaController extends AppController {
    var $name = 'Captcha';
    var $uses = array('Captcha');
    var $helpers = array( 'Form', 'Html',  'Session');
    var $components = array( 'Session', 'RequestHandler', 'AsciiArts');
```

In Listing 13-6, we declare the helpers and components to aid some of the functions implemented in the controller. Notice that we've included our `AsciiArts` component.

Next in the controller is the `beforeFilter` function, which is invoked before any of the controller functionality is called to set the heading information for our screen, as shown in Listing 13-7.

Listing 13-7. *CaptchaController's beforeFilter Method*

```
function beforeFilter() {
    $actionHeading = 'ASCII Arts CAPTCHA!';
    $actionSlogan = 'Please fill in ASCII Arts text';
    $this->set(compact('actionHeading','actionSlogan'));
}
```

Next, we will create the `index` function, which renders a simple HTML form to do a Captcha test, as shown in Listing 13-8.

Listing 13-8. *CaptchaController's index Function*

```
function index() {
    $captcha = $this->AsciiArts->getCaptcha();
    $string = implode("",array_keys($captcha));
    $this->Session->write('string', $string);
    $this->set(compact('captcha','string'));
}
```

The `index` method starts by setting the `$captcha` array variable with the list of ASCII Art characters retrieved by using the `getCaptcha` method of the `AsciiArtsComponents` object. Next, using the PHP `implode` function, we obtain all the keys of the elements of the `$captcha` array variable with an empty string; this is the ASCII value of the Captcha characters. We then store the keys in the `$string` variable written to a string `Session` variable. We will use the characters stored in a `Session` object to validate the input a user has entered on a web form. We then make the `$captcha` and `$string` variables available to the `app/views/captcha/index.ctp` view code, which is shown in Listing 13-9.

Listing 13-9. *The Captcha Test View (app/views/captcha/index.ctp)*

```
<fieldset>
    <legend> <?php __("$actionHeading");?> </legend>
    <?=$actionSlogan;?>
    <br />
    <?php

    e("<pre><table><tr>");
    foreach($captcha as $key => $val) {
        e('<td style="font-size: 10px;">'.$val.'</td>');
    }
    e("</tr></table><pre>");
        echo $form->create('Captcha', array('url'=>'/captcha/check'));
        echo $form->error( 'Captcha.text' );
        echo $form->input( 'Captcha.text', array( ➥
'id' => 'text', 'label' => 'Text:', 'size' => '50', ➥
'maxlength' => '255', 'error' => false ) );
        echo $form->end( array( 'label' => ' Submit ' ) );
    ?>
</fieldset>
```

In Listing 13-9, the page heading and slogan were rendered using the variables set in the `beforeFilter` method of the controller. Next, we use the `foreach` loop to display the ASCII Art characters stored in the `$captcha` array variable. Finally, we create a Captcha form containing an input text field to accept user input. The user is expected to correctly enter the ASCII Art character displayed above the input element. The view is shown in Figure 13-2.

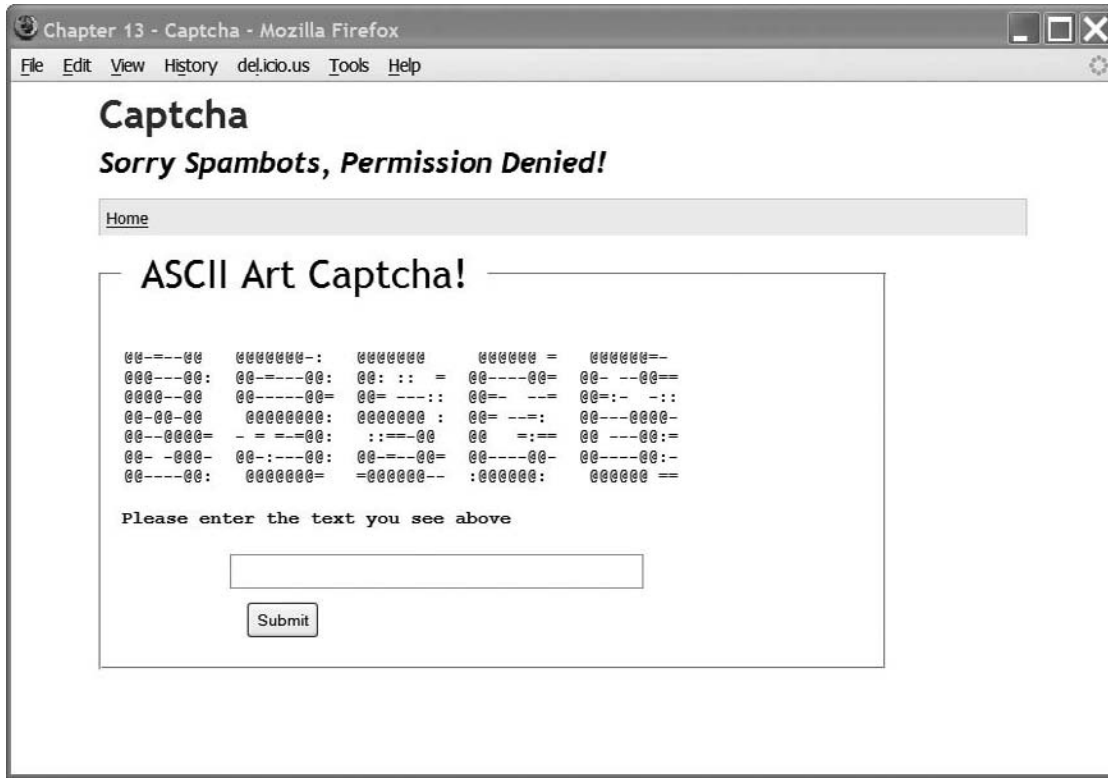

Figure 13-2. *The Captcha test form*

When a user clicks the Submit button, the form submission is handled by the `check` function declared in the component, as shown in Listing 13-10.

Listing 13-10. *The check() Function in the CaptchaController Class (app/controllers/captcha_controller.php)*

```
function check() {
    if (!empty($this->data['Captcha']['text'])) {
        if ($this->data['Captcha']['text'] == $this->Session->read('string')) {
            $this->Session->setFlash(
                __('<h1>You have entered the right characters</h1>', true));
            $this->redirect(array('action'=>'index'));
```

```
        } else {
            $this->Session->setFlash(
                __('You have entered the wrong characters. Please try again.', true));
            $this->redirect(array('action'=>'index'));
        }
    } else {
        $this->Session->setFlash(
                __('You need to enter the correct characters. ➥
Please try again.', true));
        $this->redirect(array('action'=>'index'));
    }
}
```

In the `check` function, we first check if the submitted data (`$this->data['Captcha']['text']`) is not empty. If it's empty, the appropriate error message is stored in the `Session` object, and the Captcha form is displayed with the preset error message. If it's not empty, we first check if the input value entered by the user is equal to the ASCII Art characters stored in the `Session` object. If they are not equal, the error message is set, using the `setFlash` method of the `Session` object. Otherwise, the success message is set for display in the Captcha view, as shown in Figure 13-3.

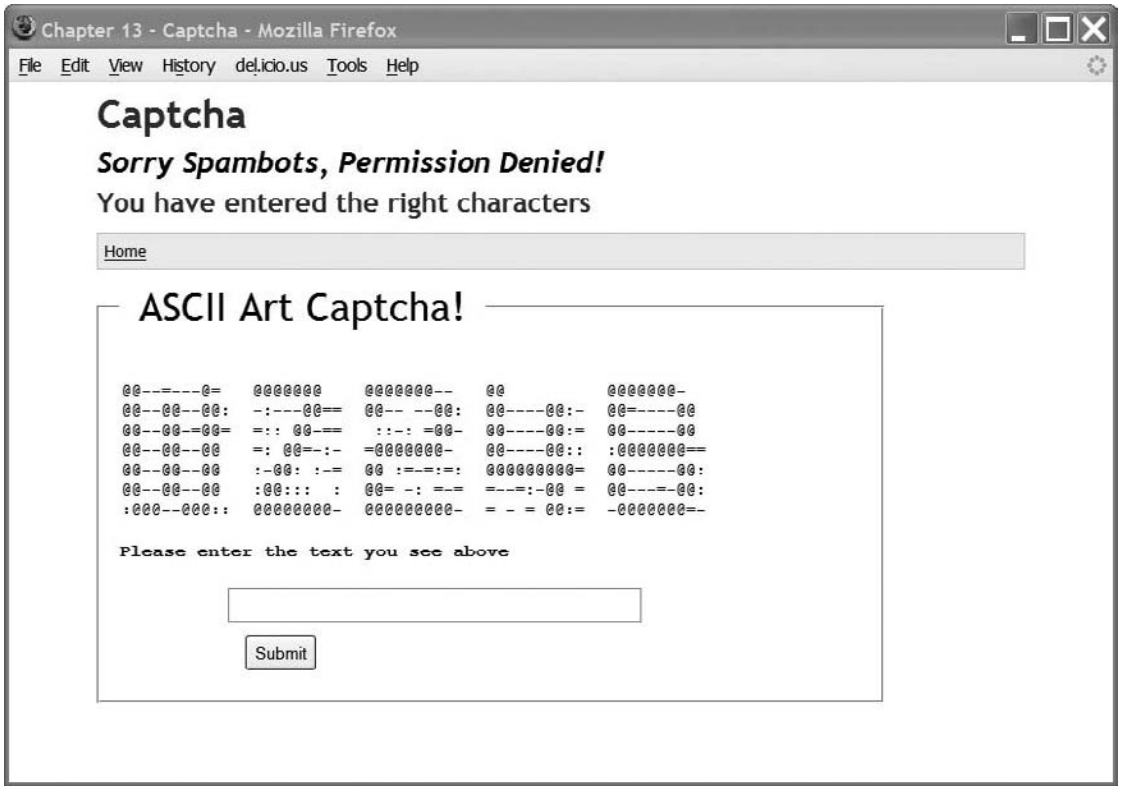

Figure 13-3. *The screen showing the success message*

Summary

In this chapter, we addressed the need to thwart spammers when collecting information using web forms. We concentrated on safeguarding form data against machine-code intruders such as spambots, who can act like humans and fill in web forms. We took a brief look at the various types of Captchas that can be implemented.

We chose to implement an ASCII Art Captcha. We created an ASCII Art component. This component contains properties and functionality that enable our Captcha controller to create a simple human test via a web form that randomly displays a set of ASCII Art characters on the screen. Finally, appropriate messages are displayed after a user has entered text.

There are many possible ways you can improve on our Captcha component. Here are some suggestions:

- Increase the security by replacing a random number of the `@` characters that are used as the character for the fonts with other ASCII characters. Replace only a few; replacing too many characters will render the ASCII Art representation unrecognizable.
- Increase the security even more by using JavaScript to output the font characters.
- Instead of using the fonts we have created, create your own fonts.

And do keep in mind that for every security measure, there will always be some countermeasure. Never underestimate the resolve of the enemy!

Index

A

access control, ActionsController and, 244–251
access data field, 310
Accomplishments (sample application), 214–221
  unit testing and, 221–232
  web testing and, 232–234
accomplishments (sample database table), 214
Acl (access control list) component, 19, 238, 264
  declaring, 241
  GroupsController and, 251
Aco class, 258
acos database table, 239, 249
ActionsController class, 242, 244–251
addCart() method, 65, 73
addData() method, 337, 356
Add Data view, 356
add_form() method, 160, 168
addListener()method, 134
add() method, 15, 16, 337
  Accomplishments sample application and, 216
  edit() method and, 254
  JourneysController and, 164
  online shop sample application and, 73
  PostsController and, 32, 36
  product search sample application and, 353
  retrieve() method and, 168
  UsersController and, 264
addNoise() method, 364
admin_ prefix, 306
admin_add() method, 277, 293
admin_delete() method, 284, 293
admin_edit() method, 293
admin_index() method, 279
admin_toTrans() method, 279
admin_view() method, 293
afterDelete() method, 308
afterFind() method, 309
  access data field and, 310
  behaviors and, 308
afterSave() method, 184, 257, 265, 309
Ajax
  _checkAjax() method and, 95, 99
  event-driven programming and, 134
  message forum sample web service and, 109, 113, 125
Ajax helper, 21
Allen, David, 213
allow() method, 258
  Acl component, 264
  Auth component, 238, 251
alphanumeric image Captchas, 360
am() method, 312
API (application programming interface), 86
app folder, 5
AppController class, 4, 18, 209
  components and, 20 241
  helpers and, 21
  plugins and, 23
app_controller.php file, 294, 325
App:import() method, 25
application controllers. *See* controllers
application layout, caching, 190
application programming interface (API), 86
AppModel class, 8, 23
app_model.php file, 325
aros_acos database table, 239
aros database table, 239, 266
arrays, utility for converting to objects, 19
ASCII Art character Captchas, 361–368
AsciiArtsComponent class, 362–365
$asciiFonts property, 362
assertFalse() method, 226
assert methods, 231
audio Captchas, 360
Auth (authentication) component, 238, 262, 294
  CRUD methods and, 249
  declaring, 241
authentication, 50, 293–296
authorize parameter, 241, 249, 258
automagic fields. *See* magic fields

B

bake command, 273–277, 297–306
  GroupsController and, 251
  ProductsController, 337
  UsersController and, 259
beforeDelete() method, 308

beforeFilter() method, 4, 283, 294
Auth/Acl components and, 241
CaptchaController and, 365
Google translation services and, 188, 211
message forum sample web service and, 96, 99
online shop sample application and, 64, 68–70
beforeFind() method, 309
access data field and, 310
record order data field and, 311
beforeMessage() method, 115
beforeRender() method, message forum sample web service and, 96, 99
beforeSave() method, 308, 309
beforeValidation() method, 309
behaviors, custom magic fields and, 308
blogging (sample application), 29–46
application structure for, 31
RSS feed for, 43
Blum, Manuel, 359
break helper, 22
buildCategories() method, 53, 56, 60
business logic, 7

C

caching
Twitter Twister sample application and, 188–192
when to add, 191
cake command, 238
cake folder, 5
cake tags, 320
CakePHP
control panel and, 237
core concepts/technologies used by, 2
directory structure of, 5
features of for web development, 1
framework for, downloading/installing, 5
fundamentals of, 1–27
naming conventions for, 6–8
vs. other frameworks, 2
calcTSP() method, 153
callback methods, 308
Captcha (Completely Automated Public Turing Test to Tell Computers and Humans Apart), 359–369
CaptchaController class, 363, 365
Cart model, 63–67
CartsController class, 69–75
carts (sample database table), 50, 51, 64
case study, In/Out communications application and, 214
categories, 185
CategoriesController class, 56
categories (sample database table), 50, 53
Category model, 53, 56
changeLanguage() method, 192
changeLocale() method, 283
_changeSessionLocale() method, 193
check() method, 367
_checkAjax() method, 95, 99
checkout() method, 76
class keyword, 8
cleanUp() method, 65, 73
code samples. *See* source code
Command design pattern, 94
Completely Automated Public Turing Test to Tell Computers and Humans Apart (Captcha), 359–369
components, 19, 232. *See also* Acl component; Auth component
concurrency control, 313
config/core.php file, 279
content and data separation, 317
control panel (sample application), 237–270
additional features for, 270
application controllers and, 242–269
Auth/Acl components and, 238–242
testing, 266
Controller class, 18
controllers, 4, 18
blogging sample application and, 32–45
components and, 20
control panel sample application and, 242–269
message forum sample web service and, 94–125
naming conventions for, 6
online shop sample application and, 56, 59–63, 67–82
testing, 232
Twitter Twister sample application and, 192–211
Yahoo! Maps sample plugin and, 324
ControlPanelController class, 242
controls, Google Maps and, 134
convert() method, 20
core defines, 7
create() method, 11
creating
behaviors, 308
magic fields, 309–315
models, 8
RSS, 43
tables, 9, 30
CRUD (create, read, update, and delete), 9–13, 53
ActionsController and, 249,
GroupController and, 258

D

DATABASE_CONFIG class, 3
database connection array, 3
databases, for blogging sample application, 29
data, separating from content, 317
data schemas, 139
data validation, 13
default page title, setting for an application, 18
delete() method, 42
  Auth component and, 251
  behaviors and, 308
  GroupsController and, 258
  UsersController and, 264
DELETE method, 87
deleting
  posts, 42
  tables, 9
deny() method, 238, 258, 264
departments (sample database table), 9
description tag, 174
directory structure, 5
disable() method, 40, 314
display() method, 20, 324, 328
_displayTwitters() method, 200
div tag, 327
docs folder, 5
documentation, for message forum sample web service, 125–128
Don't Repeat Yourself (DRY), 25
doSearch() method, 353
doUpdate() method, 66
downloads
  Cake framework, 5
  SimpleTest testing interface, 222
  text editors, 113
driving directions, travel log application and, 135
DRY (Don't Repeat Yourself), 25
dynamic data fields, 329–358
  advantages/disadvantages of, 331
  product database design and, 332–337

E

e() method, 125
e-commerce, 47–83
edit() method, 32, 38, 254, 262
Email component, 19
emptyBasket() method, 66
enable() method, 41, 314
encapsulation, 3
Event.observe method, 146
events, Google Maps and, 134
execute() method, 94
explode() method, 167

F

FCKeditor text editor, 113
feedback plugin (sample), 23
Fielding, Roy, 86
field_type_groups table, 333
field_type_values table, 333
File class, 249
files, reading content to screen and, 26
findBestJourney() method, 148, 151
findById() method, 73, 170
findBy() method, 97, 193
findLocation() method, 148
find() method, 8, 13, 20
  access data field and, 310
  behaviors and, 308
  language model and, 184
  locking data field and, 315
  retrieving journeys and, 168
Folder class, 249
fonts property, 362, 363
foreach loop, 347
form helper, 21
$form object, 11, 15
forum application. *See* message forum web service
Fried, Jason, 214

G

GClientGeocoder class, 133
GControl class, 134
GDirections class, 135
  fetching data and, 153
  plotting journeys via, 156–159
generateFields() method, 164
generateTreeList() method, 258
geocoding, 133
GET method, 86, 87, 173
getAll() method, 56, 71
getCaptcha() method, 364, 366
getCartContent() method, 66
getCart() method, 64, 73
getCategories() method, 53, 55
getChildCategories() method, 53, 55, 60
getDepartment() method, 11
getFieldSelection() method, 353
getJourneyDistance() method, 151, 153
getLang() method, 184
getLocale method, 292
getMessage() method, 109
getMiniCart() method, 74
_getNeighbours() method, 156
_getNextNeighbour() method, 155, 156
_getNextNeighbours() method, 155
getStatusTranslation() method, 205, 208
Getting Things Done (GTD) methodology, 213

getTwitterRequests() method, 195
GEvent class, 134
GGroundOverlay class, 135
global constants, 7
global functions, 7
GMarket class, 135
Google Ajax Language API (aka Google Translator), 175
Google Checkout button, online shop sample application and, 79
Google Maps, 131–136
Google translation services, using with Twitter, 173–211
  caching and, 189
  detecting languages and, 176
GPolygon class, 135
GPolyline class, 135
Group model class, 255
groups, 251–258, 266
GroupsController class, 242, 251–258
GTD (Getting Things Done) methodology, 213
GTileLayerOverlay class, 135
GUnload() method, 133

H

HABTM (has and belongs to many), 334
$hasMany property, 53
"Hello World," translating, 175
helpers, 21
home page
  for online shop sample application, 69
  for travel log sample application, 143
Hopper, Nicholas J., 359
HTML forms, blogging and, 29
HTML helper, 16, 21
HTTP protocol, 87
HttpSocket class, 175
  Google translation requests and, 206
  Twitter's public timeline, calling via, 174, 196

I

i18n (sample database table), 271, 277
if statement, 318
implode() method, 366
import() method, 26
index() method, 18, 94, 199
  ActionsController and, 249
  CaptchaController and, 366
  GroupsController and, 258
  group security and, 258
  message forum sample web service and, 95, 99
  PostsController and, 32, 43
  UsersController and, 264
inheritance, 4
initialize() method, 133, 363
In/Out communications application (case study), 214
insertBefore() method, 115
interfaces, 3
internationalization (i18n), 184
isCartEmpty() method, 64
ISO language codes, 180, 185, 193, 292

J

JavaScript, Google Maps and, 133
JavaScript helper, 21
JavaScript Object Notation (JSON), 157
journey model, 140
journeys, 160–172
  plotting, 156–159
  retrieving/editing, 168
  saving, 160–167
  viewing, 169–172
journeys table, 138
JourneysController class, 138, 160, 164
JSON (JavaScript Object Notation), 157
JSON web services, 88, 92

L

l10n (localization), 184, 293
_langChoice() method, 211
Langford, John, 359
Language model, 183
languages
  changing, Twitter Twister sample application and, 190, 192
  detecting, Twitter Twister sample application and, 176
  selecting, news story sample application and, 292
languages database table, 180
LC_MESSAGES category, 185
LC_TIME category, 185
left branch, 240
LIKE AGAINST operator, 123
link() method, 283
List behavior, magic fields and, 307
_listActions() method, 249
_listProductSearch() method, 341
lists() method, 59, 61
loadFromWaypoints() method, 156
locale codes, 272
locales, 184, 292
  changing, 191
  news story sample application and, 292
localization (l10n), 184, 293
location model, 140
locations
  driving directions and, 135
  finding, 148
  plotting journeys and, 156–159

locations table, 138
locking data field, 313
_locVisited() method, 156
logic, separating from presentation, 317
login() method, 216, 262, 294
logout() method, 262, 296

M

m_ prefix, for magic field names, 310
Magento e-commerce application, 47
magic fields, 307–315
  custom, 308, 309–315
  m_ prefix for, 310
MagicFieldsPlus behavior file, 309
makeTwitterRequest() method, 195
Mann, Merlin, 213
MapHelper class, 164
MarkerManager class, 134
_markVisited() method, 156
MATCH AGAINST operator, 123
mathematical question Captchas, 360
menu() method, 56
MessageController class, 96
message forum web service (sample application), 85–129
  additional features for, 129
  controller for, 94–125
  layout for, 91
  requirements for, 88–91
  structure of, 91
  writing documentation for, 125–128
messages (sample database table), 91
metatables, 332
MfController class, 95
MfFetchMessageController class, 97, 99–104
MfFetchMessage() method, 109, 126
MfFetchMessages() method, 126
MfFetchThreadsController class, 105–109
MfFetchThreads() method, 127
MfMessageFormController class, 110–115
MfMessageProcessController class, 100, 116–120
MfMessageProcess() method, 127
MfSearchProcessController class, 120–125
MfSearchProcess() method, 128
Model class, 8
Model-View-Controller. *See* MVC design pattern
ModelBehavior class, 308
models, 8–15
  blogging sample application and, 31
  caching, 189
  creating, 8
  message forum sample web service and, 96
  naming conventions for, 6
  news story sample application and, 272
  Twitter Twister sample application and, 181–184
  Yahoo! Maps sample plugin and, 325
Multibyte class, 175
MVC design pattern, 2, 8–19
  content/data separation and, 319
  controllers and, 315
  plugins and, 22
MySQL, search methods and, 123
MySQL database, sample applications and, 5, 29, 50–67

N

$name property, 11, 53, 70
naming conventions, 6–8
  for Cake tags, 320
  for magic fields, 310
  for test classes, 225
  for test files, 224
newline() method, 22
news story sample application, 271–306
  managing stories in, 273–288
  structure of, 271
next() method, 106
nocache tag, 190, 191
noiseChars property, 362
Norman, Donald, 137
numberChar property, 362

O

object-oriented programming (OOP), 3
objects, utility for converting arrays to, 19
onchange events, 146
onclick events, 146
online shop (sample application), 47–83
  flow of, 49
  Google Checkout button and, 79
  home page for, 69
  layout for, 47
  MySQL database for, 50–67
  orders and, 75
  PayPal Submit button and, 82
  shopping cart and, 63–75
  user requests and, 67–83
OOP (object-oriented programming), 3
optimistic locking, 313
Order model, 75
OrdersController class, 76–79
osCommerce e-commerce application, 47
overlays, 134

P

$pageTitle property, 67
page title, setting default for an application, 18
paginate() method, 106, 280, 284, 291
paginateCount() method, 291

paginateCountStandard() method, 291
paginateCountTranslation() method, 291
paginateStandard() method, 291
paginateTranslation() method, 291
pagination
  for message forum sample web service, 105–109
  for news story sample application, 288–292
paginator helper, 21
parentNode() method, 257
paths, 7
PayPal Submit button, online shop sample application and, 82
$pdid property, 67
permissions, 238
pessimistic locking, 313
PHP 4+, 3
picture image Captchas, 360
_plotBestJourney() method, 156
plugins, 22
  sample application, for Yahoo! Maps, 320, 323–328
  testing, 232
Poedit file editor, 185
polymorphism, 4
portable object (.po) files, 185, 271
POST method, 86, 87, 173
posts, blogging and, 30–45
  adding posts, 35
  deleting posts, 42
  listing posts, 32
  message forum sample web service and, 89
  publishing posts, 41
  unpublishing posts, 40
  updating posts, 38
PostsController class, 32, 36, 43
presentation, separating from logic, 317
prev() method, 106
print() method, 22
pr() method, 20
_processActions() method, 249
Product model, 59
  Cart model and, 63
  code for, 347–353
product search (sample application), 329–358
  adding products to, 353–358
  building, 338–353
  processing searches and, 347
product_field_groups table, 335
ProductField model, 341
product_fields table, 335
product_field_values table, 335
product_groups table, 336, 353
products (sample database table), 50, 53, 59
products available, displaying, 325
products table, 329–337, 334, 353
ProductsController class, 56, 59–63, 338, 347
product_searches table, 337
products_product_groups table, 334, 353
protocols, 86
public timeline, 174, 194, 325
PUT method, 87

Q

query() method, 13, 184

R

RAD (rapid application development), 2
ReadFile class, 26
read() method, 13, 52
reCAPTCHA, 359
record order data field, 311
records
  concurrency control and, 313
  retrieving, 9, 11–13
regression testing, 235
removeChild() method, 114
_renderCt() method, 323
render() method, 323
representational state transfer (REST), 86, 87, 173
requestAction() method, 58, 61, 71, 74
RequestHandler component, 19
  JSON requests and, 93
  web service requests and, 187, 210
request() method, 174
Request() method, 115
requests, in Twitter, 173, 177
  Cake models and, 181
  TwitterRequestController and, 194
resources for further reading
  ASCII Art, 361
  CakePHP vs. other frameworks, 2
  Google payment system, 82
  helpers, 21
  object inheritance, 4
  object-oriented programming, 4
  PayPal, 82
  PHP, 3
REST protocol, 86, 87, 173
result return formats, 88
retrieve() method, 168
retrieving records, 9, 11–13
right branch, 240
Router class
  JSON web services and, 93
  web services and, 186, 187
RSS feeds
  creating, 43
  Twitter Twister sample application and, 186–188
RSS helper, 43
RSS view file, 44

S

sample applications
  Accomplishments task completion, 214–221
  blogging, 29–46
  control panel, 237–270
  feedback plugin, 23
  message forum web service, 85–129
  news story, 271–306
  online shop, 47–83
  product search, 329–358
  travel log, 131–172
  Twitter Twister, 173, 180–211
  Yahoo! Maps plugin, 320, 323–328
sample code. *See* source code
Sanitize plugin, 216
save command, 278
save() method, 9, 184, 195
  behaviors and, 308
  product search sample application and, 353
saveJourney() method, 148
_save_locations() method, 167
saveMessage() method, 11
saveRequest() method, 195
saveStatuses() method, 196, 197
_save_tags() method, 167
_saveTrans() method, 206, 208
Scaffolding behavior, magic fields and, 307
ScreenController class, spillout.php script and, 26
screen_name tag, 174
scripts, 25
script tag, 107
SD memory cards, 334
Search Action view, 344–347
search feature, for products (sample application), 329–358
  adding products to, 353–358
  building, 338–353
  processing searches and, 347
search() method, 337, 341
searchFilters() method, 343
security, 237–270
  Acl/Auth components and, 238
  groups and, 251, 257
  permissions and, 238
security() method, 257, 263
SELECT statement, 97
Session component, 19, 210
session helper, 21
setCenter() method, 133
setFlash() method, 73
setI10nByLocale() method, 278
set() method, 15, 17, 62
setPageTitle() method, 68
setup() method, 309
$shop database connection array, 3
shopping basket, displaying, 325
show() method, 17, 18
$sid property, 67
SimpleTest unit testing interface, 221–232
  downloading/installing, 222
  web testing and, 232
SimpleXMLElement class, 199
SMTP protocol, 86
SOAP protocol, 86
someAction() method, 251, 258
source code, 2
  Accomplishments (task completion sample application), 215–233
  add_form() method, 161
  add() method, 16
  add view, 15
  App classes, 23
  AppController class, extending, 4
  application controllers, 242–247, 251–254, 259–262
  Auth/Acl components, 241
  authentication, news story sample application and, 294
  behaviors, 311
  blogging application, 30–45
  Captcha protection, 362–368
  components, 19
  content and data separation, 317
  controllers, 24, 94–125, 192–210
  DATABASE_CONFIG class, 3
  database tables, 238, 239
  data structures, 9
  Department class, 10
  GDirections class, 135
  global layout file, 141
  Google Maps example, 131
  Group model class, 255
  “Hello World,” translating, 175
  helpers, 22
  home page, for travel log sample application, 143
  journeys (travel log sample application), 139, 156, 161, 164, 168–172
  JSON layout file, 93
  JSON strings, 157
  Language model, 183
  locales, 191
  locations (travel log sample application), 139, 148
  magic fields, 310–315
  MagicFieldsPlus behavior file, 309
  MapHelper class, 164
  message forum sample web service, 89–125
  Model class, 8
  news story sample application, 271–306
  nocache tag, 190

online shop sample application and, 50–82
optimistic locking, 313
plugin for Yahoo! Maps, 320–328
plugins, 23–25
product search feature, 338–358
records, retrieving, 11–13
RSS feeds, 186
RSS view file, displaying contents of, 44
saveJourney() method, 157
show() method, 16
spillout.php script, 26
tables, creating, 9, 30
tags, 140, 167, 319
traveling salesman algorithm, 150–156
TravelMapprManager class, 146
Twitter Twister sample application, 174, 178–180, 185, 190
TwitterRequest model, 181
TwitterStatus model, 182
TwitterTranslation model, 182
User model class, 264
validation rules, 14
web site test, 234
spam, 359
spambots, 359–361
spillout.php script, 26
startAgain() method, 148
startup() method, 20, 363
statuses, in Twitter, 173
20 most recent, 174, 178
Cake models and, 181
TwitterRequestController and, 194
TwitterStatusController and, 196–209
_statusTranslate() method, 202
storeLocation() method, 149
StoriesController class, 271, 277–297, 283
stories (sample database table), 271, 277, 284

T

tables
creating, 9, 30
deleting, 9
naming conventions for, 7
updating, 9
table tag, 318
tag model, 140
tags, 317–328
attributes and, 320
saving, 167
tags table, 138
test.php file extension, 224
test classes, naming conventions for, 225
test-driven development, 235
test files, naming conventions for, 224
testing, 213–235
components, 232
controllers, 232
plugins, 232
regression testing and, 235
unit testing and, 221–232
web testing and, 232–234
testLoginBadUsername() method, 233
testValidUsername() method, 225
text helper, 21
text tag, 174
textarea tag, 113
threads, message forum sample web service and, 89
threads (sample database table), 91
time helper, 21, 216
TinyMCE text editor, 113
Translate behavior, 272, 278, 282, 292
translate command, 176
_translateText() method, 206
traveling salesman algorithm, 150–156
traveling salesman problem, 131
travel log (sample) application, 131–172
interface for, 141–145
journeys and, 160–172
requirements for, 136
structure of, 138
TravelMapprManager class, 146–159
Tree behavior, 240, 307, 308
Turing, Alan, 359
tweets. *See* statuses, in Twitter
Twitter, 173–211
caching and, 189
protocols and, 87
public timeline of, 174, 325
Twitter API and, 173–175
Twitter Twister (sample application), 173
additional features for, 211
database tables for, 178
requirements/structure of, 176–181
TwitterRequestController class, 194
TwitterRequest model, 181
twitter_requests database table, 178, 182
TwitterStatusController class, 196–209
twitter_statuses database table, 178, 179, 182
TwitterStatus model, 182
TwitterTranslation model, 182
twitter_translations database table, 179, 182
TwittertwisterController class, 180–211

U

unbindModel() method, 202
unit testing, 221–232
assert methods and, 231
benefits of, 213
Cake testing interface for, 221–232
reasons for using, 235
regression testing and, 235
running tests and, 227
test-driven development and, 235

updateCart() method, 65, 73
updateForm() method, 114
Updater() method, 115
updating
  posts, blogging and, 38
  tables, 9
  Twitter and. *See* statuses, in Twitter
UserController class, authentication and, 295, 296
user interfaces (UIs), for travel log sample application, 141–145
user journey, for online shop sample application, 49
User model class, 264
username validation, 219–233
user requests, online shop sample application and, 67–83
users
  authenticating, 293–296
  managing, 259–266
users (sample database table)
  control panel sample application and, 238, 266
  news story sample application and, 271, 294
UsersController class, 242, 259–266, 271
$uses property, 19, 56, 67
$useTable property, 7, 11
utilities
  FCKeditor text editor, 113
  Poedit file editor, 185
  TinyMCE text editor, 113
uuid() method, 314, 327

V

validate property, 76
_validation() method, 99
validation rules, 14
validationErrors error array, 315
validUsername() method, 219, 225, 229
vendors folder, 25
View class, 321–323
view() method, 195
  GroupsController and, 258
  ProductsController and, 59, 62
  UsersController and, 264
viewAllStories() method, 284
viewArchive() method, 195
viewing journeys, 169–172
views, 15
  naming conventions for, 7
  overriding, 320
_visitNextCity() method, 156
von Ahn, Luis, 359

W

web development
  features of CakePHP for, 1
  test-driven, 235
  RAD and, 2
web forms, 11
  Captcha protection for, 359–369
  views and, 15
web root configurable paths, 7
web service requests, 88, 90
web services, 85
  JSON, 92
  message forum sample application and, 85–129
  Twitter Twister sample application and, 186–188
web sites, testing external, 234
web testing, 232–234
welcome() method, 243
WidgetsController class, 242
Winer, David, 86
write method, 52

X

XHR (XMLHttpRequest), 87, 134
XML class, 199
XML helper, 21
XMLHttpRequest (XHR), 87, 134

Y

Yahoo! Maps plugin (sample application), 320, 323–328

Z

zoom level, adding, 327