

# The Yii Book



Developing Web Applications  
Using the Yii PHP Framework



**Larry Ullman**

---

*The Yii Book* by Larry Ullman

Self-published

Find this book on the Web at [yii.larryullman.com](http://yii.larryullman.com).

Revision: 1.0

Copyright © 2015 by Larry Ullman

Technical Reviewer: Qiang Xue

Technical Reviewer: Alexander Makarov

Cover design very kindly provided by Paul Wilcox.

### **Notice of Rights**

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

### **Notice of Liability**

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of the book, the author shall not 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 instructions contained in this book or by the computer software and hardware products described in it.

### **Trademarks**

MySQL is a registered trademark of Oracle in the United States and in other countries. Macintosh and Mac OS X are registered trademarks of Apple, Inc. Microsoft and Windows are registered trademarks of Microsoft Corp. Other product names used in this book may be trademarks of their own respective owners. Images of Web sites in this book are copyrighted by the original holders and are used with their kind permission. This book is not officially endorsed by nor affiliated with any of the above companies.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the author was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN-13:

ISBN-10:

---

## This book is dedicated to:

Qiang Xue, creator of the Yii framework; Alexander Makarov, and the whole Yii development team; and to the entire Yii community. Thanks to you all for making, embracing, and supporting such an excellent Web development tool.

# Contents

<b>Introduction</b>	<b>1</b>
Why Frameworks? . . . . .	1
Why Yii? . . . . .	3
What You'll Need . . . . .	6
About This Book . . . . .	8
Getting Help . . . . .	11
<b>1 FUNDAMENTAL CONCEPTS</b>	<b>13</b>
Object-Oriented Programming . . . . .	13
The MVC Approach . . . . .	19
Using a Web Server . . . . .	25
Command Line Tools . . . . .	26
<b>2 STARTING A NEW APPLICATION</b>	<b>31</b>
Downloading Yii . . . . .	31
Testing the Requirements . . . . .	32
Installing the Framework . . . . .	34
Building the Site Shell . . . . .	34
Testing the Site Shell . . . . .	36
<b>3 A MANUAL FOR YOUR YII SITE</b>	<b>37</b>
The Site's Folders . . . . .	37
Referencing Files and Directories . . . . .	39
Yii Conventions . . . . .	40
How Yii Handles a Page Request . . . . .	41

<b>4 INITIAL CUSTOMIZATIONS AND CODE GENERATIONS</b>	<b>46</b>
Enabling Debug Mode . . . . .	46
Moving the Protected Folder . . . . .	47
Basic Configurations . . . . .	48
Developing Your Site . . . . .	60
Generating Code with Gii . . . . .	69
<b>5 WORKING WITH MODELS</b>	<b>77</b>
The Model Classes . . . . .	77
Establishing Rules . . . . .	79
Changing Labels . . . . .	96
Watching for Model Events . . . . .	99
Relating Models . . . . .	103
<b>6 WORKING WITH VIEWS</b>	<b>107</b>
The View Structure . . . . .	107
Where Views are Referenced . . . . .	108
Layouts and Views . . . . .	109
Editing View Files . . . . .	112
Working with Layouts . . . . .	121
Alternative Content Presentation . . . . .	126
<b>7 WORKING WITH CONTROLLERS</b>	<b>131</b>
Controller Basics . . . . .	131
Revisiting Views . . . . .	134
Making Use of Models . . . . .	135
Handling Forms . . . . .	141
Basic Access Control . . . . .	142
Understanding Routes . . . . .	147
Tapping Into Filters . . . . .	153
Showing Static Pages . . . . .	155
Exceptions . . . . .	157

<b>8 WORKING WITH DATABASES</b>	<b>162</b>
Debugging Database Operations . . . . .	162
Database Options . . . . .	164
Using Active Record . . . . .	166
Using Query Builder . . . . .	180
Using Database Access Objects . . . . .	185
Choosing an Interface Option . . . . .	188
Common Challenges . . . . .	190
<b>9 WORKING WITH FORMS</b>	<b>193</b>
Understanding Forms and MVC . . . . .	193
Creating Forms without Models . . . . .	195
Using CHtml . . . . .	196
Using “Active” Methods . . . . .	197
Using CActiveForm . . . . .	198
Using Form Builder . . . . .	200
Common Form Needs . . . . .	206
<b>10 MAINTAINING STATE</b>	<b>226</b>
Cookies . . . . .	226
Sessions . . . . .	230
<b>11 USER AUTHENTICATION AND AUTHORIZATION</b>	<b>235</b>
Fundamentals of Authentication . . . . .	235
Authentication Options . . . . .	246
The UserIdentity State . . . . .	252
Authorization . . . . .	255
Working with Flash Messages . . . . .	270
<b>12 WORKING WITH WIDGETS</b>	<b>273</b>
Using Widgets . . . . .	273
Basic Yii Widgets . . . . .	276
Presenting Data . . . . .	282
The jQuery UI Widgets . . . . .	304

<b>13 USING EXTENSIONS</b>	<b>311</b>
The Basics of Extensions . . . . .	311
The bootstrap Extension . . . . .	314
The giix Extension . . . . .	319
Validator Extensions . . . . .	322
Auto-Setting Timestamps . . . . .	324
Using a WYSIWYG Editor . . . . .	325
<b>14 JAVASCRIPT AND JQUERY</b>	<b>329</b>
What You Must Know . . . . .	329
Adding JavaScript to a Page . . . . .	330
Using JavaScript with CActiveForm . . . . .	335
Implementing Ajax . . . . .	339
Common Needs . . . . .	351
<b>15 INTERNATIONALIZATION</b>	<b>357</b>
What is i18n? . . . . .	357
Setting the Locale . . . . .	359
Detecting the User's Locale . . . . .	360
Providing Language-Appropriate Text . . . . .	362
Formatting Dates . . . . .	369
Representing Time Zones . . . . .	371
Formatting Numbers . . . . .	372
i18n and Your Models . . . . .	374
<b>16 LEAVING THE BROWSER</b>	<b>376</b>
Writing a Proxy Script . . . . .	376
Creating Web Services . . . . .	379
Creating a Console Application . . . . .	393

<b>17 IMPROVING PERFORMANCE</b>	<b>400</b>
Testing Performance . . . . .	400
Changing the Bootstrap . . . . .	408
Changing the Configuration . . . . .	409
Caching . . . . .	409
Improving Database Performance . . . . .	420
Using Multiple Caching Mechanisms . . . . .	426
Browser Improvements . . . . .	427
Creating a Plan . . . . .	432
<b>18 ADVANCED DATABASE ISSUES</b>	<b>434</b>
Database Migration . . . . .	434
Calling Stored Procedures . . . . .	441
Using Complex Keys . . . . .	442
Using Complex Relationships . . . . .	446
<b>19 EXTENDING YII</b>	<b>450</b>
Fundamental Concepts . . . . .	450
More Examples . . . . .	470
Working with Modules . . . . .	479
Deploying Extensions . . . . .	506
<b>20 WORKING WITH THIRD-PARTY LIBRARIES</b>	<b>507</b>
Installation . . . . .	507
Accessing Library Classes . . . . .	508
Working with Composer . . . . .	514
Using Symfony . . . . .	518
Using Swift Mailer . . . . .	521
Using Elasticsearch . . . . .	524
<b>21 TESTING YOUR APPLICATIONS</b>	<b>547</b>
Test Directories . . . . .	548
Using PHPUnit . . . . .	548
Using Selenium . . . . .	559
Checking Coverage . . . . .	565

<b>22 CREATING A CMS</b>	<b>567</b>
Project Goals . . . . .	568
Creating the Database . . . . .	569
Getting Started . . . . .	570
Editing the Models . . . . .	571
Creating Login Functionality . . . . .	573
Creating Pages . . . . .	575
Making Comments . . . . .	577
Converting to Twitter Bootstrap . . . . .	582
Adding WYSIWYG . . . . .	584
Creating a Posts by Month Widget . . . . .	586
Applying ElasticSearch . . . . .	590
Tidying Up . . . . .	595
Things to Possibly Add . . . . .	600
<b>23 MAKING AN E-COMMERCE SITE</b>	<b>602</b>
Project Goals . . . . .	602
Creating the Database . . . . .	603
Getting Started . . . . .	607
Editing the Models . . . . .	607
Creating the Home Page . . . . .	608
Viewing a Book . . . . .	610
Writing a Utilities Class . . . . .	611
Creating Carts . . . . .	612
Adding the Payment Module . . . . .	617
Creating Customers . . . . .	619
Recording Orders . . . . .	621
Downloading Books . . . . .	624
Things to Possibly Add . . . . .	625
Selling Physical Goods . . . . .	626

<b>24 SHIPPING YOUR PROJECT</b>	<b>628</b>
The Framework Installation . . . . .	628
Setting Permissions . . . . .	629
Transferring Assets . . . . .	630
Moving the Application Directory . . . . .	630
Taking the Bootstrap File Live . . . . .	631
Temporary Debugging . . . . .	631
Using Multiple Configuration Files . . . . .	632
Better Logging . . . . .	634
Things to Do . . . . .	638

# Introduction

This is the 24th book that I've written, and of the many things I've learned in that time, a reliable fact is this: readers rarely read the introduction. Still, I put a fair amount of time into the introduction and would ask you to spend the five minutes required to read it.

In this particular introduction, I provide the arguments for (and against) frameworks, and the [Yii framework](#) specifically. I also explain what knowledge and technical requirements are expected of you, the dear reader. And if that was not enough, the introduction concludes by providing you with resources you can use to seek help when you need it.

So: five minutes of your time for all that. Okay, maybe 8 minutes. How about you give it a go?

## Why Frameworks?

Simply put, a framework is an established library of code meant to expedite software development. Writing everything from scratch on every project is impractical; code reuse is faster, more reliable, and possibly more secure.

Many developers eventually create a lightweight framework of their own, even if that's just a handful of commonly used functions. True frameworks such as Yii are just the release of a complete set of tools that a smart and hardworking person has been kind enough to make public. Even if you don't buy the arguments for using a framework in its own right, it's safe to say that the ability to use a framework, whether that means a few pieces of your own reusable code or a full-fledged framework such as Yii, is to be expected for any regular programmer today.

## Why You Should Use a Framework

The most obvious argument for using a framework is that you'll be able to develop projects much, much faster than if you didn't use a framework. But there are other arguments, and those are more critical.

As already stated, framework-based projects should also be both more reliable and secure than one coded by hand. Both qualities come from the fact that framework code will inevitably be far more thoroughly tested than anything you create. By using a framework, with established code and best practices, you're starting on a more stable, secure, and tested foundation than your own code would provide (in theory).

Similarly, a framework is likely to impose a quality of documentation that you might not take the time to implement otherwise. The same can go for other professional features, such as logging and error reporting. These are features that a good framework includes but that you may not get around to doing, or doing properly, despite your best intentions.

Still, the *faster development* argument continues to get the most attention. If you are like, well, almost everyone, your time is both limited and valuable. Being able to complete a project in one-third the time means you can do three times the work, and make three times the money. In theory.

You can also make more money when you know a framework because it improves your marketability. Framework adoption is almost a must for team projects, as frameworks impose a common development approach and coding standard. For that reason, most companies hiring new Web developers will expect you to know at least one framework.

In my mind, the best argument for using a framework is this: so that you can always choose the right tool for the job. Not to be cliché, but I firmly believe that one of the goals of life is to keep learning, to keep improving yourself, no matter what your occupation or station. As Web developers in particular, you must continue to learn, to expand your skill set, to acquire new tools, or else you'll be left behind. Picking up a framework is a very practical choice for your own betterment. In fact, I would recommend that *you actually learn more than one framework*. By doing so, you can find the right framework for you and better understand the frameworks you know (just as I understood English grammar much better only after learning French).

## Why You Shouldn't Use a Framework

If frameworks are so great, then why isn't everyone using a framework for every project? First, and most obviously, frameworks require extra time to learn. The fifth project you create using a framework may only take one-third the time it would have taken to create the site from scratch, but the first project will take at least the same amount of time as if you had written it from scratch, if not much longer. Particularly if you're in a rush to get a project done, the extra hours needed to learn a framework will not seem like time well spent. Again, eventually frameworks provide for much faster development, but it will take you a little while to get there.

Second, frameworks will normally do about 80% of the work really easily, but that last 20% (the part that truly differentiates this project from all the others) can be a

real challenge. This hurdle is also easier to overcome the better you know a framework, but implementing more custom, less common Web tasks using a framework can really put you through your paces.

Third, from the standpoint of running a Web site or application, frameworks can be terribly inefficient. For example, to load a single record from a database, a framework may require three queries instead of just the one used by a conventional, non-framework site. As database queries are one of the most expensive operations in terms of server resources, three times the queries is a ghastly thought. And framework-based sites will require a lot more memory, as more objects and other resources are constantly being created and used.

*{NOTE}* Frameworks greatly improve your development time at a cost of the site's performance.

That being said, there are many ways to improve a site's performance, and not so many ways to give yourself back hours in the day. More importantly, a good framework like Yii has built-in tools to mitigate the performance compromises being made. In fact, through such tools, it's entirely possible that a framework-based site could be *more* efficient than the one you would have written from scratch.

Fourth, when a site is based upon a framework, you are expected to update the site's copy of the framework's files (but not the site code itself) as maintenance and security releases come out. This is true whenever you use third-party code. (Although, on the other hand, this does mean that other people are out there finding, and solving, potential security holes, which won't happen with your own code.)

## How You Use a Framework

Once you've decided to give framework-based programming a try, the next question is: How? First, you must have a solid understanding of how to develop *without* using a framework. Frameworks expedite development, but they only do so by changing the way you perform common tasks. If you don't understand basic user interactions in conventional Web pages, for example, then switching to using a framework will be that much more bewildering.

And second, *you should give in to the framework*. All frameworks have their own conventions: how things are to be done. Attempting to fight those conventions will be a frustrating, losing battle. Do your best to accept the way that the framework does things and it'll be a smoother, less buggy, and faster experience.

## Why Yii?

The Yii framework was created by Qiang Xue and first released in 2008. “Yii” is pronounced like “Yee”, and is an acronym for “Yes, it is!”. From Yii's official documentation:

Is it fast?...Is it secure?...Is it professional?...Is it right for my next project?...Yes, it is!

“Yii” is also close to the Chinese character “Yi”, which represents easy, simple, and flexible.

Mr. Xue was also the founder of the [Prado framework](#), which took its inspiration from the popular [ASP.NET](#) framework for Windows development. In creating Yii, Mr. Xue took the best parts of Prado, [Ruby on Rails](#), [CakePHP](#), and [Symfony](#) to create a modern, feature-rich, and very useable PHP framework.

At the time of this writing, the current, stable release of the Yii framework is 1.1.14. It is expected that version 2 of the Yii framework will be complete in the first half of 2014.

## What Yii Has to Offer

Being a framework, Yii offers all the strengths and weaknesses that frameworks in general have to offer (as already detailed). But what does Yii offer, in particular?

Like most frameworks, Yii uses pure Object-Oriented Programming (OOP). Unlike some other frameworks, Yii has always required version 5 of PHP. This is significant, as PHP 5 has a vastly improved and advanced object structure compared with the older PHP 4 (let alone the archaic and rather lame object model that existed way back in PHP 3). For me, frameworks that were not written specifically for PHP 5 and greater aren't worth considering.

Yii uses the de facto standard Model-View-Controller (MVC) architecture pattern. If you're not familiar with it, Chapter 1, “[Fundamental Concepts](#),” explains this approach in detail.

Almost all Web applications these days rely upon an underlying database. Consequently, how a framework manages database interactions is vital. Yii can work with databases in several different ways, but the standard convention is through Object Relational Mapping (ORM) via Active Record (AR). If you don't know what *ORM* and *AR* are, that's fine: you'll learn well enough in time. The short description is that an ORM handles the conversion of data from one source to another. In the case of a Yii-based application, the data will be mapped from a PHP object variable to a database record and vice versa.

*{TIP}* The excellent Ruby on Rails framework also uses Active Record for its database mapping.

For low-level database interactions, Yii uses PHP 5's [PHP Data Objects](#) (PDO). PDO provides a *data-access abstraction layer*, allowing you to use the same code to interact with the database, regardless of the underlying database application involved.

One of Yii's greatest features is that if you prefer a different approach, you can swap alternatives in and out. For example, you can change:

- The underlying database-specific library
- The template system used to create the output
- How caching is performed
- And much more

The alternatives you swap in can be code of your own creation, or that found in third-party libraries, including code from other frameworks!

Despite all this flexibility, Yii is still very stable, and through caching and other tools, perform quite well. Yii applications will scale well, too, as has been tested on some high-demand sites, such as [Stay.com](#) and [VICE](#).

All that being said, many of Yii's benefits and approaches apply to other PHP frameworks as well. Why *you* should use Yii is far more subjective than a list of features and capabilities. At the end of the day, you should use Yii if the framework makes sense to you and you can get it to do what you need to do.

*{NOTE}* For a full sense of Yii's feature set, see this [book's table of contents](#) online or the [features page](#) at the official Yii site.

As for myself, I initially came to Yii because it requires PHP 5—I find backwards-compatible frameworks to be inherently flawed—and uses the [jQuery](#) JavaScript framework natively. (By comparison, the widely-used [Zend Framework](#) was rather slow to adopt jQuery, in my opinion.) I also love that Yii will auto-generate *a ton* of code and directories for you, a feature that I had come to be spoiled by when using Rails. Yii is also well-documented, and has a great community. Mostly, though, for me, Yii just feels right. And unless you really investigate a framework's underpinnings to see how well designed it is, how the framework feels to you is a large part of the criteria in making a framework selection.

In this book and [my blog](#), I'm happy to discuss what Yii has to offer: why you should use it. The question I can't really answer is what advantage Yii has over this or that framework. If you want a comparison of Yii vs. X framework, search online, but remember that the best criteria for which framework you should use is always going to be your own personal experience.

*{TIP}* If you're trying to decide between framework X and framework Y, then it's worth your time to spend an afternoon, or a day, with each to see for yourself which you like better.

The only other PHP framework I've used extensively is Zend. The Zend Framework has a lot going for it and is worth anyone's consideration. To me, its biggest asset

is that you can use it piecemeal and independently (I've often used components of the Zend Framework in Yii-based and non-framework-based sites), but I just don't like the Zend Framework as the basis of an entire site. It requires a lot of work, the documentation is overwhelming while still not being that great, and it just doesn't "feel right" to me.

I really like the Yii framework and hope you will too. But this book is not a sales pitch for using Yii over any other framework, but rather a guide for those needing help.

## Who Is Using Yii?

The Yii framework has a wide international adoption, with extensive usage in (the):

- United States
- Russia
- Ukraine
- China
- Brazil
- India
- Europe

Many open-source apps have been written in Yii, including:

- [Chive](#), an alternative to [phpMyAdmin](#)
- [Zurmo](#), a Customer Relationship Management (CRM) system
- [X2EngineCRM](#), another CRM
- [LimeSurvey2](#), a surveying application

## What You'll Need

Learning any new technology comes with expectations, and this book on Yii is no different. I've divided the requirements into two areas: *technical* and *personal knowledge*. Please make sure you clear the bar on both before getting too far into the book.

### Technical Requirements

Being a PHP framework, Yii obviously requires a Web server with PHP installed on it. Version 1 of the Yii framework requires PHP 5.1 or greater. Version 2 is expected to require PHP 5.3 or later. At the time of this writing, the latest version of PHP is 5.4.11. This book will assume you're using [Apache](#) as your Web server

application. If you're not, see the Yii documentation or search online for alternative solutions when Apache-specific options are presented.

{NOTE} In my opinion, it's imperative that Web developers know what versions they are using (of PHP, MySQL, Apache, etc.). If you don't already, check your versions now!

You'll also want a database application, although Yii will work with all the common ones. This book will primarily use [MySQL](#), but, again, Yii will let you easily use other database applications with only the most minor changes to your code.

All of the above will come with any decent hosting package. But I expect all developers to install a Web server and database application on their own desktop computer: it's the standard development approach and is a far easier way to create Web sites. Oh, and it's all free! If you have not yet installed an \*AMP stack—Apache, MySQL, and PHP—on your computer, I would recommend you do so now. The most popular solutions are:

- [XAMPP](#) on Windows
- [EasyPHP](#) on Windows
- [BitNami](#) on Windows, Linux, or Mac OS X
- [Zend Server](#) on Windows, Linux, or Mac OS X
- [AMPPS](#) on Windows, Linux, or Mac OS X

All of these are free.

To write your code, you'll also need a good text editor or IDE. In theory, any application will do, but you may want to consider one that directly supports Yii, or can be made to support Yii. That list includes (all information is correct at the time of this writing; all prices in USD):

- [Eclipse](#), through the [PDT extension](#), on Windows, Linux, or Mac OS X; free
- [Netbeans](#) on Windows, Linux, or Mac OS X; free
- [PhpStorm](#) on Windows, Linux, or Mac OS X; \$30-\$200
- [CodeLobster](#) on Windows; \$120
- [SublimeText 2](#) on Windows, Linux, or Mac OS X; \$60

“Support” really means recognition for keywords and classes particular to Yii, the ability to perform code completion, and potentially even include Yii-specific wizards.

In case you're curious, I almost exclusively use a Mac, and currently use the excellent [TextMate](#) text editor (only for Mac, \$51). But I've heard nothing but accolades about SublimeText (version 3 is coming out in 2013) and PhpStorm, and plan on trying them both out extensively in the future.

## Your Knowledge and Experience

There are not only technical requirements for this book, but also personal requirements. In order to follow along, it is expected that you:

- Have solid Web development experience
- Are competent with HTML, PHP, MySQL, and SQL
- Aren't entirely uncomfortable with JavaScript *and* jQuery
- Understand that confusion and frustration are a natural consequence of learning anything new (although I'll do my best in this book to minimize the occurrence of both)

The requirements come down to this: using a framework, you'll be doing exactly the kinds of things you have already been doing, just via a different methodology. Learning to use a framework is therefore the act of translating the conventional approach into a new approach.

The book *does not* assume mastery of Object-Oriented Programming, but things will go much more smoothly if you have prior OOP experience. Chapter 1 hits the high notes of OOP in PHP, just in case.

## About This Book

Most of this introduction is about frameworks in general and the Yii framework in particular, but I want to take a moment to introduce this book as a whole, too.

## The Goals of This Book

I had two goals in writing this book. The first is to explain the entirety of the Yii framework in such a way as to convey a sense of the big picture. In other words, I want you to be able to understand *why you do things in certain ways*. By learning what Yii is doing behind the scenes, you will be better able to grasp the context for whatever bits of code you'll end up using on your site. This holistic approach is what I think is missing among the current documentation options.

The second goal is to demonstrate common tasks using real-world examples. This book is, by no means, a cookbook, or a duplication of the [Yii wiki](#), but I would be remiss not to explain how you implement solutions to standard Web site needs. In doing so, though, I'll explain the solutions within the context of the bigger picture, so that you walk away not just learning *how* to do X but also *why* you do it in that manner.

All that being said, there are some things relative to the Yii framework (and Web development in general) that the book will not cover. For example, Yii 1 defines

many of its own data types, used in more advanced applications. Some of these are replicated in PHP's [Standard PHP Library](#), which will be used in Yii 2 instead. This book omits coverage of them, along with anything else I've deemed equally esoteric.

Still, my expectation is that after reading this book, and understanding how the Yii framework is used, you'll be better equipped to research and learn these omissions, should you ever have those needs.

## Formatting Conventions

I've adopted a couple of formatting conventions in writing this book. They should be obvious, but just in case, I'll lay them out explicitly here.

Code font will be presented `like this`, whether it's inline (as in that example) or presented on its own:

```
// This is a line of code.  
// This is another line.
```

Whenever code is presented lacking sufficient context, I will provide the name of the file in which that code would be found, including the directory structure:

```
# protected/views/layouts/home.php  
// This is the code.
```

Sometimes I will also indicate the name of the function in that file where the code would be placed:

```
# protected/models/Example.php::doThis()  
// This is the code within the doThis() function.
```

This convention simply saves me from having to include the `function doThis() {` line every time.

{NOTE} Chapter 3, "A Manual for Your Yii Site," will explain the Yii directory structure in detail.

Within text, URLs, directories, and file names will be in **bold**. References to specific classes, methods, and variables will be in code font: `SomeClass`, `someMethod()`, and `$someVar`. References to array indexes, component names, and informal but meaningful terms will be quoted: the "items" index, the "site" controller, the "urlManager" component, etc.

## How I Wrote This Book

For those of you that care about such things, this book was written using the [Scrivener](#) application running on Mac OS X. Scrivener is far and away the best writing application I've ever come across. If you're thinking about doing any serious amount of writing, download it today!

Images were taken using [Snapz Pro X](#).

The entire book was written using [MultiMarkdown](#), an extension of [Markdown](#). I exported MultiMarkdown from Scrivener.

Next, I converted the MultiMarkdown source to a PDF using [Pandoc](#), which supports its own slight variation on Markdown. The formatting of the PDF is dictated by [LaTeX](#), which is an amazing tool, but not for the faint of heart.

To create the ePub version of the book, I also used Pandoc and the same MultiMarkdown source.

To create the mobi (i.e., Kindle) version of the book, I imported the ePub into [Calibre](#), an excellent open source application. Calibre can convert and export a book into multiple formats, including mobi.

For excerpts of the book to be published online, I again used Pandoc to create HTML from the MultiMarkdown.

This is a lot of steps, yes, but MultiMarkdown gave me the most flexibility to write in one format but output in multiple. Pandoc supports the widest range of input sources and output formats, by far. And research suggested that Calibre is the best tool for creating reliable mobi files.

## About Larry Ullman

I am a writer, developer, consultant, trainer, and public speaker. This is my 24th book, with the vast majority of them related to Web development. My [PHP for the Web: Visual QuickStart Guide](#) and [PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide](#) books are two of the bestselling guides to the PHP programming language. Both are in their fourth editions, at the time of this writing. I've also written [Modern JavaScript: Develop and Design](#), which is thankfully getting excellent reviews.

I first started using the Yii framework in early 2009, a few months after the framework was publicly released. Later that year, I posted a “[Learning the Yii Framework](#)” series on my blog, which has become quite popular. Qiang Xue, the creator of Yii, liked it so much that he linked to my series from the [Yii's official documentation](#). Ever since, the series has had a good amount of publicity and traffic. I have wanted to write this book for some time, but did not have the opportunity to begin until 2012.

While a large percentage of my work is technical writing, I'm an active developer. Most of the Web sites I do are for educational and non-profit organizations, but I also consult on commercial and other projects. I would estimate that I use a framework on maybe 60% of the sites I work on. I don't use a framework all the time because a framework isn't always appropriate. Some of the framework-based sites I create use [WordPress](#) instead of Yii, depending upon the client and the needs.

My Web site is [LarryUllman.com](#). This book's specific set of pages is at [yii.LarryUllman.com](#). You can also find me on Twitter [@LarryUllman](#).

## Getting Help

If you need assistance with your Yii-based site, or with any of this book's material, there are many places to turn:

- The [Yii documentation](#)
- The [official Yii forums](#)
- [My support forums](#)
- The #yii IRC channel on the Freenode network

{NOTE} If you don't have an IRC client (or haven't used IRC before), the Yii Web site graciously provides a [Web-based interface](#).

When you need help, you should always start by looking at the Yii documentation. Over the course of the book, you'll learn how to use the docs to solve your own problems, most specifically the [class reference](#).

If you're still having problems and a quick Google search won't cut it, the Yii forums are probably the best place to turn. They have an active and smart community. Do begin by *searching* the forums first, as it's likely your question has already been raised and answered (unless it's very particular to this book).

Understand that wherever you turn to for assistance, you'll get far better results if you provide all the necessary information, are patient, and demonstrate appreciation for the help.

You *can* contact me directly with questions, but I would strongly prefer that you use my support forums or the Yii forums instead. By using a forum, other people can assist, meaning you'll get help faster. Furthermore, the assistance will be public, which will likely help others down the line.

{NOTE} I check my own support forums three days per week. I check the Yii support forums irregularly, depending upon when I think of it. But in both forums, there are other, very generous, people to assist you. Of the two, the Yii forums have more members and are more active.

## INTRODUCTION

If you ask me for help via Twitter, Facebook, or Google+, I'll request that you use my or the Yii forums. If you email me, I will reply, but it's highly likely that it will take two weeks for me to reply. And the reply may say you haven't provided enough information. And after providing an answer, or not, I'll recommend you use forums instead of contacting me directly. So you *can* contact me directly, but it's far, far better—for both of us—if you use one of the other resources. Don't get me wrong: I want to help, but I strongly prefer to help in the public forums, where my time spent helping might also benefit others.

# Chapter 1

## FUNDAMENTAL CONCEPTS

Frameworks are created with a certain point of view and design approach. Therefore, properly using a framework requires an understanding and comfort with the underlying perspective(s). Towards that end, this chapter covers the most fundamental concepts that you'll need to know in order to properly use the Yii framework.

With Yii, the two most important concepts are Object-Oriented Programming (OOP) and the Model-View-Controller (MVC) pattern. The chapter begins with a quick introduction to OOP, and then explains the MVC design approach. Finally, the chapter covers a couple of key concepts regarding your computer and the Web server application.

I imagine that nothing in this chapter will be that new for some readers. If so, feel free to skip ahead to Chapter 2, “[Starting a New Application](#).“ If you’re confused by something later on, you can always return here. On the other hand, if you aren’t 100% confident about the mentioned topics, then keep reading.

### Object-Oriented Programming

Yii is an object-oriented framework; in order to use Yii, you must understand OOP. In this first part of the chapter, I’ll walk through the basic OOP terminology, philosophy, and syntax for those completely unfamiliar with them.

#### OOP Terminology

PHP is a somewhat unusual programming language in that it can be used both procedurally and with an object-oriented approach. (Java and Ruby, for example, are always object-oriented language and C is always procedural.) The primary difference between procedural and object-oriented programming is one of focus.

All programming is a matter of taking actions with things:

- A form’s data is submitted to the server.
- A page is requested by the user.
- A record is retrieved by the database.

Put in grammatical terms, you have *nouns*—form, data, server, page, user, record, database—and *verbs*: submitted, requested, and retrieved.

In procedural programming, the emphasis is on the *actions*: the steps that must be taken. To write procedural code, you lay out a sequence of actions to be applied to data, normally by invoking functions. In object-oriented programming, the focus is on the *things* (i.e., the nouns). Thus, to write object-oriented code, you start by analyzing and defining what types of things the application will work with.

The core concept in OOP is the *class*. A class is a blueprint for a thing, defining both the information that needs to be known about the thing as well as the common actions to be taken with it. For example, representing a page of HTML content as a class, you need to know the page’s title, its content, when it was created, when it was last updated, and who created it. The actions one might take with a page include stripping it of all HTML tags (e.g., for use in non-Web destinations), returning the initial X characters of its content (e.g., to provide a preview), and so forth.

With those requirements in mind, a class is created as a blueprint. The thing’s data—title, content, etc.—are represented as variables in the class. The actions to be taken with the thing, such as stripping out the HTML, are represented as functions. These variables and functions within a class definition are referred to as *attributes* (or *properties*) and *methods*, accordingly. Collectively, a class’s attributes and methods are called the class’s *members*.

Once you’ve defined a class, you create *instances* of the class, those instances being *object* variables. Going with a content example, one object may represent the Home page and another would represent the About page. Each variable would have its own properties (e.g., title or content) with its own unique values, but still have the same methods. In other words, while the value of the “content” variable in one object would be different from the value of the “content” variable in another, both objects would have a `getPreview()` method that returns the first X characters of that object’s content.

{NOTE} In OOP, you will occasionally use classes without formally creating an instance of that class. In Yii, this is quite common.

The class is at the heart of Object-Oriented Programming and good class definitions make for projects that are reliable and easy to maintain. As you’ll see when implementing OOP (if you have not already), more and more logic and code is pushed, appropriately, into the classes, leaving the usage of those classes to be rather straightforward and minimalistic.

I consider OOP in PHP to be a more advanced concept than traditional procedural programming for this reason: OOP isn’t just a matter of syntax, it’s also about

philosophy. Whereas procedural programming almost writes itself in terms of a logical flow, proper object-oriented programming requires a good amount of theory and design. Bad procedural programming tends not to work well, but can be easily remedied; bad object-oriented programming is a complicated, buggy mess that can be a real chore to fix. On the other hand, good OOP code is easy to extend and reuse.

{NOTE} Programming in Yii is different from non-framework OOP in that most of the philosophical and design issues are already implemented for you by the framework itself. You're left with just using someone else's design, which is a huge benefit to Object-Oriented Programming.

## OOP Philosophy

The first key concept when it comes to OOP theory is *modularity*. Modularity is a matter of breaking functionality into individual, specific pieces. This theory is similar to how you modularize a procedural site into user-defined functions and includable files.

Not only should classes and methods be modular, but they should also demonstrate *encapsulation*. Encapsulation means that how something *works* is well shielded from how it's *used*. Going with a `Page` class example (an OOP class defined to represent an HTML page), you wouldn't need to know *how* a method strips out the HTML from the page's content, just that the method does that. Proper encapsulation also means that you can later change a method's *implementation*—how it works—without impacting code that invokes that method. (For what it's worth, good procedural functions should adhere to encapsulation as well.)

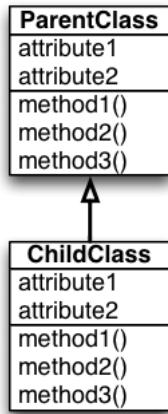
Encapsulation goes hand-in-hand with *access control*, also called *visibility*. Access control dictates where a class's attributes (i.e., variables) can be referenced and where its methods (functions) can be called. Proper usage of access control can improve an application's security and reduce the risk of bugs.

There are three levels of visibility:

- Public
- Protected
- Private

To understand these levels, one has to know about *inheritance* as well. In OOP, one class can be defined as an extension of another, which sets up a parent-child inheritance relationship, also called a *base* class and a *subclass*. The child class in such situations may or may not also start with the same attributes and methods, depending upon their visibility (**Figure 1.1**).

An attribute or method defined as *public* can be accessed anywhere within the class, within derived (i.e., child) classes, or through object instances of those classes. An



**Figure 1.1:** The child class can inherit members from the parent class.

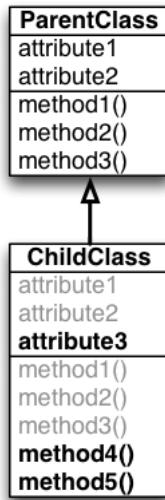
attribute or method defined as *protected* can only be accessed within the class or within derived classes, but not through object instances. An attribute or method defined as *private* can only be accessed within the class itself, not within derived classes (i.e., child or subclasses) or through object instances.

Because object-oriented programming allows for inheritance, another endorsed design approach is *abstraction*. Ideally base classes (those used as parents of other classes) should be as generic as possible, with more specific functionality defined in derived classes (children). The derived class inherits all the public and protected members from the base class, and can then add its own new ones. For example, an application might define a generic `Person` class that has `eat()` and `sleep()` methods. `Adult` might inherit from `Person` and add a `work()` method, among others, whereas `Child` could also inherit from `Person` but add a `play()` method (**Figure 1.2**).

Inheritance can be extended to such a degree that you have multiple generations of inheritance (i.e., parent, child, grandchild, etc.). PHP does not allow for a single child class to inherit from multiple parent classes, however: class `Dog` *cannot* simultaneously inherit from both `Mammal` and `Pet`.

*{TIP}* The Yii framework uses multiple levels of inheritance all the time, allowing you to call a method defined in class C that's defined in class A, because class C inherits from B, which inherits from class A.

Getting into slightly more advanced OOP, child classes can also *override* a parent class's method. To override a method is to redefine what that method does in a child class. This concept is called *polymorphism*: the same method can do different things depending upon the context in which it is called.



**Figure 1.2:** The child class can add new members to the ones it inherited.

## OOP Syntax

With sufficient terminology and theory behind us, let's look at OOP syntax in PHP (specifically PHP 5; earlier versions of the language sometimes did things differently).

Classes in PHP are defined using the `class` keyword:

```
class SomeClass {  
}
```

Within the class, variables and functions are defined using common procedural syntax, save for the addition of visibility indicators:

```
class SomeClass {  
    public $var1;  
    public function doThis() {  
        // Do whatever.  
    }  
}
```

Public is the default visibility and it does not need to be specified as it is in that code, but it is best to be explicit. The class attributes (the variables) can be assigned default values using the assignment operator, as you would almost any other variable.

{TIP} You'll see this syntax to identify a method and the class to which it belongs: `SomeClass::doThis()`. That's shorthand for saying "The `doThis()` method of the `SomeClass` class".

Once you've defined a class, you create an instance of that class—an object—using the `new` keyword:

```
$obj = new SomeClass();
```

Once the object has been created, you can reference public attributes and methods using *object notation*. In PHP, `->` is the object operator (in many other languages it is the period):

```
echo $obj->var1;  
$obj->doThis();
```

Note that, as in that code, object attributes are referenced through the object *without* using the dollar sign in front of the attribute's name (i.e., it's not `echo $obj->$var1`).

Within the class, attributes and methods are accessible via the special `$this` object. `$this` always refers to the current instance of that class:

```
Class SomeClass {  
    public $var1;  
    public function doThis() {  
        $this->var1 = 23;  
        $this->doThat();  
    }  
    private function doThat() {  
        echo $this->var1;  
    }  
}
```

That code also demonstrates how a class can access protected and private members, as protected and private members cannot be accessed directly through an object instance outside of the class.

Some classes have special methods, called *constructors* and *destructors*, that are automatically invoked when an object of that class type is created and destroyed, respectively. These methods must always use the names `*__construct*` and `*__destruct*`. A constructor can, and often does, take arguments, but cannot return any values. A destructor cannot take arguments at all. These special methods might be used, for example, to open a database connection when an object of that class type is created and close it when it is destroyed.

Moving on, *inheritance* is indicated using the `extends` keyword:

```
class ChildClass extends ParentClass {  
}
```

You will see this syntax a lot when working with Yii, as the framework defines all of the base classes that you will extend for individual purposes.

The last thing to know is that, conventionally, class names in PHP use the “upper-camelcase” format: *ClassName*, *ChildName*, and so forth. Methods and attributes normally use “lower-camelcase”: *doThis*, *doThat*, *someVar*, *fullName*, etc. Private attributes normally have an underscore as the first character. These conventions are not required, although they are the ones I will use in this book. By far, the most important consideration is that you are consistent in applying whatever conventions you prefer.

## The MVC Approach

Another core concept when it comes to using the Yii framework is the *MVC* software design approach. MVC, which stands for “model, view, controller”, is an architecture pattern: a way of structuring a program. Although its origins are in the Smalltalk language, MVC has been widely adopted by many languages and particularly by frameworks.

The basic MVC concept is relatively simple to understand, but I find that the actual implementation of the pattern can be tricky. In other words, it can take some time to master *where you put your code*. You must comprehend what MVC is in order to effectively use Yii. To convey both MVC and how it impacts the code you write, let’s look at this approach in detail, explaining how it’s done in Yii, how it compares to a non-MVC approach, and some signs that you may be doing MVC wrong.

### The Basics

Simply put, the MVC approach separates (or, to be more technical, *decouples*) an application’s three core pieces: the data, the actions a user takes, and the visual display or interface. By adopting MVC, you will have an application that’s more modular, easier to maintain, and readily expandable.

{TIP} You can use MVC without a framework, but most frameworks today do apply the MVC approach.

MVC represents an application as three distinct parts:

- Model, which is a combination of the data used by the application and the business rules that apply to that data

- View, the interface through which a user interacts with the application
- Controller, the agent that responds to user actions, makes use of models, and generally does stuff

{NOTE} To be clear, an application will almost always have multiple models, views, and controllers, as will be explained shortly.

You can think of MVC programming like a pyramid, with the model at the bottom, the controller in the middle, and the view at the top. The PHP code should be distributed appropriately, with most of it in the model, some in the controller, and very little in the view. (Conversely, the HTML should be distributed like so: practically all of it in view files.)

I think the model component is the easiest to comprehend as it reflects the data being used by the application. Models are often tied to database tables, where one instance of a model represents one row of data from one table. Note that if you have two related tables, that scenario would be represented by two separate models, not one. You want to keep your models as atomic as possible.

If you were creating a content management system (CMS), logical models might be:

- Page, which represents a page of content
- User, which represents a registered person
- Comment, which represents a user's comment on a page

With a CMS application, those three items are the natural types of data required to implement all the required functionality.

A less obvious, but still valid, use of models is for representing non-permanent sets of data. For example, if your site has a contact form, that data won't be needed after it's emailed out. Still that data must be represented by a model up until that point (in order to perform validation and so forth).

Keep in mind that models aren't just containers for data, but also dictate the rules that the data must abide by. A model might enforce its "email" value to be a syntactically valid email address or allow its "address2" value to be null. Models also contain functions for common things you'll do with that data. For example, a model might define how to strip HTML from its own data or how to return part of its data in a particular format.

Views are also straightforward when it comes to Web development: views contain the HTML and reflect what the user will see—and interact with—in the browser. Yii, like most frameworks, uses multiple view files to generate a complete HTML page (to be explained shortly). With the CMS example, you might have these view files (among many others):

- Primary layout for the site

- Display of a single page of content
- Form for adding or updating a page of content
- Listing of all the pages of content
- Login form for users
- Form for adding a comment
- Display of a comment

Views can't just contain HTML, however: they must also have some PHP (or whatever language) that adds the unique content for a given page. Such PHP code should only perform very simple tasks, like printing the value of a variable. For example, a view file would be a template for displaying a page of content; within that, PHP code would print out the page's title at the right place and the page's content at its right place within the template. The most logic a view should have is a conditional to confirm that a variable has a value before attempting to print it. Some view files will have a loop to print out all the values in an array. The view generates what the user sees, that's it.

Decoupling the data from the presentation of the data is useful for two obvious reasons. First, it allows you to easily change the presentation—the HTML, in a Web page—without wading through a ton of PHP code. As you've no doubt created many pages containing both PHP and HTML, you know how tedious it can be working with two languages simultaneously. Thanks to MVC, you can create an entirely new look for your whole site without touching a line of PHP.

*{TIP}* A result of the MVC approach is a site with many more files that each contain less HTML and PHP. With the traditional Web development approach, you'd have fewer, but longer, files.

A second, and more important, benefit of separating the data from the presentation is that doing so lets you use the same data in many different outputs. In today's Web sites, data is not only displayed in a Web browser, it's also sent in an email, included as part of a Web service (e.g., an RSS feed), accessed via a console (i.e., command line) script, and so forth.

Finally, there's the controller. A controller primarily acts as the glue between the model and the view, although the role is not always that clear. The controller represents *actions*. Normally the controller dictates the responses to user events: the submission of a form, the request of a page, etc. The controller has more logic and code to it than a view, but it's a common mistake to put code in a controller that should really go in a model. A guiding principle of MVC design is “fat model, thin (or skinny) controller”. This means you should keep pushing your code down to the foundation of the application (aka, the pyramid's base, the model). This makes sense when you recognize that code in the model is more reusable than code in a controller.

*{TIP}* Commit this to memory: fat model, thin controller.

To put this all within a context, a user goes to a URL like **example.com/page/1** (the formatting of URLs is a different subject). The loading of that URL is simply a user request for the site to show the page with an ID of 1. The request is handled by a controller. That controller would:

1. Validate the provided ID number.
2. Load the data associated with that ID as a model instance.
3. Pass that data onto the view.

The view would insert the provided data into the right places in the HTML template, providing the user with the complete interface.

## Structuring MVC in Yii

With an understanding of the MVC pieces, let's look at how Yii implements MVC in terms of directories and files. I'll continue using a hypothetical CMS example as it's simple enough to understand while still presenting some complexity.

Each MVC piece—the model, the view, and the controller—requires a separate file, or in the case of views, multiple files. Normally, a single model is entirely represented by a single file, and the same is true for a controller. One view file would represent the overall template and individual files would be used for page-specific subsets: showing a page of content, the form for adding a page, etc.

With a CMS site, there would be one set of MVC pieces for pages, another set for users, and another set for the comments. Yii groups files together by component type—model, view, or controller, not by application component (i.e., page, user, or comment). In other words, the **models** folder contains the page, user, and comment model files; the **controller** folder contains a page controller file, a user controller file, and a comment controller file. The same goes for a **views** folder, except that there's probably multiple view files for each component type.

*{NOTE}* Having exact parallel sets of MVC files for each component in an application is a default initial setting, but a complete live site won't normally have that same rigid symmetry.

For the Yii framework, model files are named *ModelName.php*: **Page.php**, **User.php**, and **Comment.php**. As a convention, Yii uses the singular form of a word, with an initial capital letter. In each of these files there would be defined one class, which is the model definition. The class's name matches that of the file, minus the extension: **Page**, **User**, and **Comment**.

Within the model class, attributes (i.e., variables) and methods (functions) are used to define that class and how it behaves. The class's attributes would reflect the data represented by that model. For example, a model for representing a contact form

might have attributes for the person’s name and email address, the subject, and the content. A model class’s methods serve roles such as returning some of the model’s data in other formats. A framework will also use the model class’s attributes and methods for other, internal roles, such as indicating this model’s relationships to other models, dictating validation rules for the model’s data, changing the model data as needed (e.g., assigning the current timestamp to a column when that model is updated), and much more.

For most models, you’ll also have a corresponding controller (not always, though: you can have controllers not associated with models and models that don’t have controllers). These files go in the **controllers** folder, and have the word “controller” in their name: **PageController.php**, **UserController.php**, and **CommentController.php**.

Each controller is also defined as a class. Within the class, different methods identify possible *actions*. The most obvious actions represent *CRUD* functionality: Create, Read, Update, and Delete. Yii takes this a step further by breaking “read” into one action for listing all of a certain model and another for showing just one. So in Yii, the “page” controller would have methods for: creating a new page of content, updating a page of content, deleting a page of content, listing all the pages of content, and showing just one page of content.

The final component to cover are the views, which is the presentation layer. Again, view files go into a **views** directory. Yii will then subdivide this directory by subject: a folder for page views, another for user views, and another for comment view files. In Yii, these folder names are singular and lowercase. Within each subdirectory are then different view files for different things one does: show (one item), list (multiple items), create (a new item), update (an existing item). In Yii, these files are named simply **create.php**, **index.php**, **view.php**, and **update.php**, plus **\_form.php** (the same form used for both creating and updating an item).

There’s one more view file involved: the layout. This file (or these files, as one site might have several different layouts) establishes the overall template: beginning the HTML, including the CSS file, creating headers, navigation, and footers, ending the HTML. The contents of the individual view files are placed within the greater context of these layout files. This way, changing one thing, like the navigation, for the entire site requires editing only one file. Yii names this primary layout file **main.php**, and places it within the **layouts** subdirectory of **views**. Those individual pieces are then brought into the primary layout file to generate the complete output.

## MVC vs. Non-MVC

To explain MVC (specifically in Yii) in another way, let’s contrast it with a non-MVC approach. If you’re a PHP programmer creating a script that displays a single page of content (in a CMS application), you’d likely have a single PHP file that:

1. Generates the initial HTML, including the HEAD and the start of the BODY

2. Connects to the database
3. Validates that a page ID was passed in the URL
4. Queries the database
5. (Hopefully) confirms that there are query results
6. Retrieves the query results
7. Prints the query results within some HTML
8. Frees the query results and closes the database connection (maybe, maybe not)
9. Completes the HTML page

And that's what's required by a rather basic page! Even if you use included files for the database connection and the HTML template, there's still a lot going on. Not that there's anything wrong with this, mind you—I still program this way as warranted—but it's the antithesis of what MVC programming is about.

Revisiting the list of steps in MVC, that sequence is instead:

1. A controller handles the request (viz., to show a specific page of content)
2. The controller validates the page ID passed in the URL
3. The framework (outside of MVC proper) establishes a database connection
4. The controller uses the model to query the database, fetching the specific page data
5. The controller passes the loaded model data to the proper view file
6. The view file confirms that there is data to be shown
7. The view file prints the model data within some HTML
8. The framework displays the view output within the context of the layout to create the complete HTML page
9. The framework closes the database connection

As you can see, MVC is just another approach to doing what you're already doing. The same steps are being taken, and the same output results, but where the steps take place and in what order will differ.

## Signs of Trouble

I've said that beginners to MVC can easily make the mistake of putting code in the wrong place (e.g., in a controller instead of a model). To help you avoid that, let's identify some signs of trouble up front. You're probably doing something wrong if:

- Your views contain more than just `echo` or `print` and the occasional control structure.
- Your views create new variables.
- Your views execute database queries.

- Your views or your models refer to the PHP superglobals: `$_GET`, `$_POST`, `$_SERVER`, `$_COOKIE`, etc.
- Your models create HTML.
- Your controllers define methods that manipulate a model's data.

As you can tell from that list, the most common beginner's mistake is to put too much programming (i.e., logic) into the views. The goal in a view is to combine the data and the presentation (i.e., the HTML) to assemble a complete interface. Views shouldn't be "thinking" much. In Yii, more elaborate code destined for a view can be addressed using helper classes or widgets (see Chapter 12, "[Working with Widgets](#)").

As already mentioned, another common mistake is to put things in the controller that should go in a model (remember: *fat models, thin controllers*). You can think of this relationship like how OOP works in general: you define a class in one script and then another script creates an instance of that class and uses it, with some logic thrown in. That's what a controller largely does: creates objects (often of models), tosses in a bit of logic, and then passes off the rendering of the output (usually, the HTML) to the view files. This workflow will be explained in more detail in Chapter 3, "[A Manual for Your Yii Site](#)".

## Using a Web Server

Before getting into creating Yii-based applications, there are two more concepts with which you must be absolutely comfortable. The first is your Web server, discussed here, and the second is using the command-line interface, to be discussed next. Understanding how to use both is the only way you can develop using Yii (and, one could well argue, do Web development even without Yii).

*{NOTE}* Technically, it is possible to develop a Yii application without using the command-line, but I would recommend you do use the command-line tool, and you ought to be comfortable in a command-line environment anyway.

### Your Development Server

You can develop Yii-based sites anywhere, but I would strongly recommend that you begin your projects on a development server and only move them to a production server once the project is fairly complete. One reason I say this is that you'll need to use the command-line interface to begin your Yii site, and a production server, especially with cheaper, shared hosting, may not offer that option.

Another reason to use a development server is security: in the process of creating your site, you'll enable a tool called [Gii](#), which should not be enabled on a production

server. Also, errors will undoubtedly come up during development, errors that should never be shown on a live site.

Third, performance: useful debugging tools, such as [Xdebug](#) should not be enabled on live sites, but are truly valuable during the development process.

Fourth, no matter the tools and the setup, it's a hassle making changes to code residing on a remote server. Unless you're using version control (which requires even more learning), having to transfer edited files back and forth is tedious. If you make your computer your development server, your browser will also be able to load pages faster than if it had to go over the Internet.

So my advice is, before going any further, turn your computer into a development server, if you have not already. You can install all-in-one packages such as [XAMPP for Windows](#) or [MAMP for Mac OS X](#), or install the components separately. Whatever you decide, do this now. Once you have a complete site that you're happy with, you can upload it to the production server.

*{TIP}* Some tricks explained in this book will require that you are able to change how the Web server runs (i.e., edit Apache `.htaccess` files). Having your own development server makes this more likely.

## The Web Root Directory

Whether you're working on a production server or a development server, you need to be familiar with the *Web root directory*. This is the folder on the computer (aka the server) where a URL points to. For example, if you're using XAMPP on Windows, with a default installation, the Web root directory is `C:\xampp\htdocs`. This means that the URL `http://localhost:8080/somepage.php` equates to `C:\xampp\htdocs\somepage.php`. If you're using MAMP on Mac OS X, the Web root directory is `/Applications/MAMP/htdocs` (although this is easily changed in the MAMP preferences).

I will occasionally make reference to the Web root directory. Know what this value is for your environment in order to be able to follow those instructions.

## Command Line Tools

The last bit of general technical know-how to have is how to use the command-line tools on your server (even if your server is the same as your computer). The command-line interface is something every Web developer should be comfortable with, but in an age where graphical interface is the norm, many shy away from the command-line. I personally use the command-line daily, to:

- Connect to remote servers

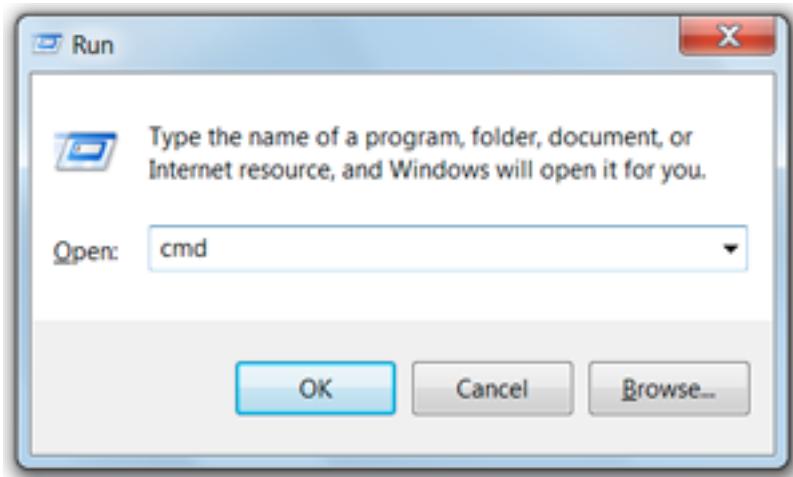
- Interact with a database
- Access hidden aspects of my computer
- And more

But even if you don't expect to do any of those things yourself, in order to create a new Web site using Yii, you'll need to use the command-line once: to create the initial shell of the site. Towards that end, there are three things you must be able to do:

1. Access your computer via the command line
2. Invoke PHP
3. Accurately reference files and directories

### Accessing the Command Line

On Windows, how you access your computer via the command-line interface will depend upon the version of the operating system you have. On Windows XP and earlier, this was accomplished by clicking Start > Run, and then entering `cmd` within the prompt (**Figure 1.3**).

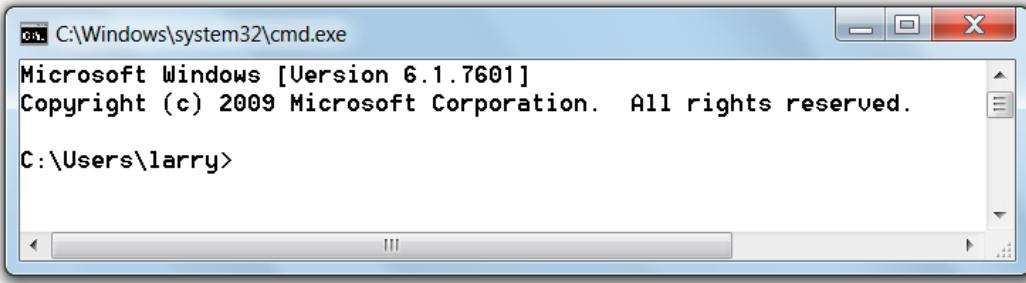


**Figure 1.3:** The `cmd` prompt.

Then click OK.

As of Windows 7, there is no immediate Run option in the Start menu, but you can find it under Start > All Programs > Accessories > Command Prompt, or you can press Command+R from the Desktop.

However you get to the command-line interface, the result will be something like **Figure 1.4**. The default is for white text on a black background; I normally inverse these colors, particularly for book images.



**Figure 1.4:** The command line interface on Windows.

{TIP} The command-line interface on Windows is also sometimes referred to as a “console” window or a “DOS prompt”.

On Mac OS X, the command-line interface is provided by the Terminal application, within the Applications/Utilities folder. On Unix and Linux, I’m going to assume you already know how to find your command-line interface. You’re using \*nix after all.

## Invoking PHP

Once you’ve got a command-line interface, what can you do? Thousands of things, of course, but most importantly for the sake of this book: invoke PHP. A thing the beginning PHP programmer does not know is that PHP itself comes in many formats. The most common use of PHP is as a *Web server module*: an add-on that expands what that Web server can do. There is also a PHP *executable*: a version that runs independently of any Web server or other application. This executable can be used to run little snippets of PHP code, execute entire PHP scripts, or even, as of PHP 5.4, act as its own little Web server. It’s this executable version of PHP that you’ll use to run the script that creates your first Yii-based Web application.

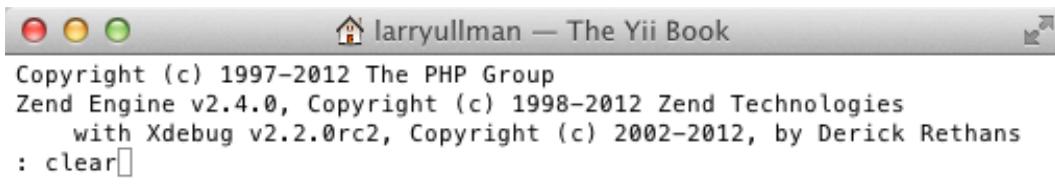
On versions of \*nix, including Mac OS X, referencing the PHP executable is rarely a problem. On Windows, it might be. To test your setup, type the following in your command-line interface and press Enter/Return:

```
php -v
```

If you see something like in **Figure 1.5**, you’re in good shape.

If you see a message along the lines of ‘*php*’ is not recognized as an internal or external command, operable program, or batch file., there are two logical causes:

1. You have not yet installed PHP.
2. You have installed PHP, but the executable is not in your *system path*.



**Figure 1.5:** The result if you can invoke the PHP executable.

If you have not yet installed PHP, such as even installing XAMPP, do so now. If you *have* installed PHP in some way, then the problem is likely your path. The *system path*, or just *path*, is a list of directories on your computer where the system will look for executable applications. In other words, when you enter `php`, the system knows to look for the corresponding `php` executable in those directories. If you have PHP installed but your computer does not recognize that command, you just have to inform your computer as to where PHP can be found. This is to say: you should add the PHP executable directory to your path. To do that, follow these steps (these are correct as of Windows 7; the particulars may be different for you):

1. Identify the location of the **php.exe** file on your computer. You can search for it or browse within the Web server directory. For example, using XAMPP on Windows, the PHP executable is in **C:\xampp\php**.
2. Click Start > Control Panel.
3. Within the Control Panel, click System and Security.
4. On the System and Security page, click Advanced System Settings.
5. On the resulting System Properties window, click the Environment Variables button on the Advanced tab.
6. Within the list of Environment Variables, select Path, and click Edit.
7. Within the corresponding window, edit the variable's value by adding a semi-colon plus the full path identified in Step 1.
8. Click OK.
9. Open a new console window to recognize the path change (i.e., any existing console windows will still complain about PHP not being found).

Now the command `php -v` should work in your console. Test it to confirm, before you go on.

{NOTE} If you have trouble with these steps, turn to the support forums for assistance.

## Referencing Files and Directories

Finally, you must know how to reference files and directories from within the command-line interface. As with references in HTML or PHP code, you can use an

*absolute* path or a *relative* one.

Within the operating system, an absolute path will begin with **C:\** on Windows and **/** on Mac OS X and \*nix. An absolute path will work no matter what directory you are currently in (assuming the path is correct).

A relative path is relative to the current location. A relative path can begin with a period or a file or folder name, but not **C:\** or **/**. There are special shortcuts with relative paths:

- Two periods together move up a directory
- A period followed by a slash (**./**) starts in the current directory

## Chapter 2

# STARTING A NEW APPLICATION

Whether you skipped Chapter 1, “[Fundamental Concepts](#),” because you know the basics, or did read it and now feel well-versed, it’s time to begin creating a new Web application using the Yii framework. In just a couple of pages you’ll be able to see some of the power of the Yii framework, and one of the reasons I like it so much: Yii will do a lot of the work for you!

In this chapter, you’ll take the following steps:

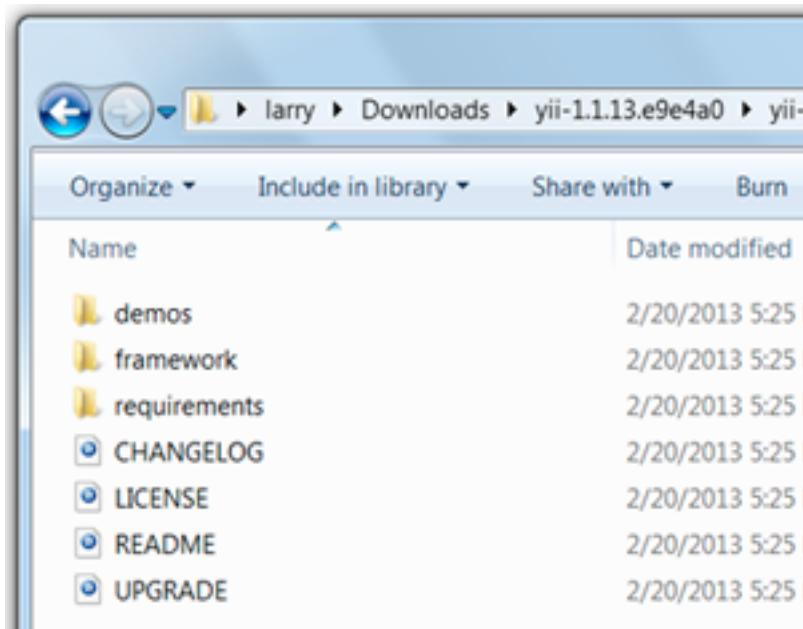
1. Download the framework
2. Confirm that your server meets the minimum requirements
3. Install the framework
4. Build the shell of the site
5. Test what you’ve created thus far

These are generic, but static steps, to be taken with each new Web site you create. In Chapter 4, “[Initial Customizations and Code Generations](#),” you’ll have Yii begin creating code more specific to an individual application.

### Downloading Yii

To start, download the latest stable version of the Yii framework. At the time of this writing, that’s 1.1.13. The file you download will be named something like *yii-version.release.ext* (e.g., [yii-1.1.13.e9e4a0.tar.gz](#) or [yii-1.1.13.e9e4a0.zip](#)). Expand the downloaded file to create a folder of stuff ([Figure 2.1](#)).

You should read the README and LICENSE docs, of course, but the folders are the most important. The **demos** folder contains four Web applications written using Yii: a blog, the game Hangman, a basic “Hello, World!”, and a phone book. The



**Figure 2.1:** The contents of the downloaded Yii framework folder.

demos are great for seeing working code as you're trying to write your own. The **framework** folder is what's required by any Web site using Yii. The **requirements** folder is something simple and brilliant, as you'll see in a moment.

## Testing the Requirements

The first thing you need in order to use the Yii framework is access to a Web server with PHP installed, of course. But if you're reading this, I'm going to assume you have access to a PHP-enabled server (if not, [see my recommendations](#)). Note that the Yii framework does require PHP 5.1 or above (in version 1 of Yii; version 2 requires PHP 5.4). Fortunately, the framework will test your setup for you, as you're about to see.

To confirm that your server meets the minimum requirements for using Yii, follow these steps:

1. Copy the **requirements** folder from the Yii download to your Web root directory.
2. Load *yourURL/requirements* in your Web browser (e.g., <http://localhost/requirements/>).
3. Look at the output to confirm that your setup meets the minimum requirements (**Figure 2.2**).
4. If your server does *not* meet the minimum requirements, reconfigure it, install the necessary components, etc., and retest until your setup does meet the requirements.

## Yii Requirement Checker

### Description

This script checks if your server configuration meets the requirements for running Yii Web applications. It checks if the server is running the right version of PHP, if appropriate PHP extensions have been loaded, and if `php.ini` file settings are correct.

### Conclusion

Your server configuration satisfies the minimum requirements by Yii. Please pay attention to the warnings listed below if your application will use the corresponding features.

### Details

Name	Result	Required By	Memo
PHP version	Passed	Yii Framework	PHP 5.1.0 or higher is required.
<code>\$_SERVER</code> variable	Passed	Yii Framework	
Reflection extension	Passed	Yii Framework	
PCRE extension	Passed	Yii Framework	
SPL extension	Passed	Yii Framework	
DOM extension	Passed	<code>CHTMLPurifier</code> , <code>CWzslGenerator</code>	
PDO extension	Passed	All <a href="#">DB-related classes</a>	
PDO SQLite extension	Passed	All <a href="#">DB-related classes</a>	This is required if you are using SQLite database.
PDO MySQL extension	Passed	All <a href="#">DB-related classes</a>	This is required if you are using MySQL database.
PDO PostgreSQL extension	Passed	All <a href="#">DB-related classes</a>	This is required if you are using PostgreSQL database.
PDO Oracle extension	Warning	All <a href="#">DB-related classes</a>	This is required if you are using Oracle database.
PDO MSSQL extension ( <code>pdo_mssql</code> )	Warning	All <a href="#">DB-related classes</a>	This is required if you are using MSSQL database from MS Windows.
PDO MSSQL extension ( <code>pdo_iblib</code> )	Passed	All <a href="#">DB-related classes</a>	This is required if you are using MSSQL database from GNU/Linux or other UNIX.
PDO MSSQL extension ( <code>pdo_sqlsrv</code> )	Warning	All <a href="#">DB-related classes</a>	This is required if you are using MSSQL database with the driver provided by Microsoft.
Memcache extension	Passed	<code>CMemCache</code>	
APC extension	Warning	<code>CApcCache</code>	
Mcrypt extension	Passed	<code>CSecurityManager</code>	
SOAP extension	Passed	<code>CWebService</code> , <code>CWebServiceAction</code>	This is required by encrypt and decrypt methods.
GD extension with FreeType support or ImageMagick extension with PNG support	Passed	<code>CCaptchaAction</code>	
Ctype extension	Passed	<code>CDateFormatter</code> , <code>CDatetimeParser</code> , <code>CTextHighlighter</code> , <code>CHtmlPurifier</code>	

passed failed warning

**Figure 2.2:** This setup meets Yii's minimum requirements.

Assuming your setup passed all the requirements, you're good to go on. Note that you don't necessarily need every extension: you only need those marked as required by the Yii framework, plus PDO and the PDO extension for the database you'll be using. (If you're not familiar with it, PDO is a database abstraction layer, making your Web sites database-agnostic.) The other things being checked may or may not be required, depending upon the needs of the actual site you're creating.

Yii's testing of the requirements is a simple thing, but one I very much appreciate. It also speaks to what Yii is all about: being simple and easy to use. Do you want to know if your setup is good enough to use Yii? Well, Yii will tell you!

Assuming your setup meets the requirements, you can now install the framework for use.

## Installing the Framework

Installing the Yii framework for use in a project is just a matter of copying the **framework** folder from the Yii download to an appropriate location. For security reasons, this should *not* be within your Web root directory.

{WARNING} Keep the Yii **framework** folder outside of your Web root directory, if at all possible.

If you're going to be using Yii for multiple sites on the same server, place the **framework** folder in a logical directory relative to every site. That way, when you update the framework, you'll only need to replace the files in one place.

As an added touch, you could place the **framework** folder in a directory whose name reflects the version of the framework in use, such as **C:\xampp\yii-1-1-12\** on Windows. Wherever you move (or copy) the **frameworks** folder to, make a note of that location, as you'll need to know it when you go to create your Yii-based application.

## Building the Site Shell

Once you've installed the framework, you can use it to build the shell of the site. Doing so requires executing a PHP script from the command-line interface. This is done via the Yii framework's **yiic** file. This is an executable that is run using the computer's command-line PHP and that really just executes the **yiic.php** script.

{TIP} If you'll be putting the site on a server that you do not have command-line access to, then you should install a complete Web server (Apache, PHP, MySQL, etc.) on your computer, run through these steps, then upload the finished project once you've completed it.

Depending upon your system, you may be able to execute the **yiic** file using just **yiic** or using **./yiic** (i.e., run the **yiic** command found in the current directory). Or you can more explicitly call the PHP script using **php yiic** or **php yiic.php**. My point here is that if at first you don't succeed using the instructions to follow, try appropriate variations until you get it right for your system.

{NOTE} In somewhat rare situations, the version of PHP used to execute the command line script will not be the same one that passed the Yii requirements. If that's the case for you, you'll need to explicitly indicate the PHP executable to be used (i.e., the one installed with your Web server).

Once you know you've figured out the proper syntax for invoking `yiic`, you follow that by "webapp", which is the command for "create a new Web application". Follow this with the path to the Web application itself. This can be either a relative or an absolute path (again, see Chapter 1), but must be within the Web root directory.

As an example, assuming that the **frameworks** folder is one step below the Web root directory, which I'll call *htdocs*, the command would be just

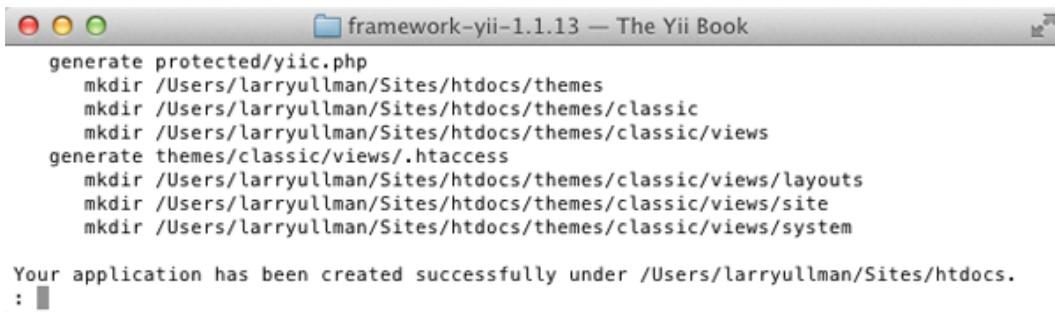
```
./yiic webapp ../htdocs
```

or

```
php yiic webapp ../htdocs
```

Or whatever variation on that you need to use.

You'll be prompted to confirm that you want create a Web application in the given directory. Enter Y (or Yes) and press Return. After lots of lines of information, you should see a message saying that the application has successfully been created (**Figure 2.3**).



The screenshot shows a terminal window titled "framework-yii-1.1.13 — The Yii Book". The window contains the following text:

```
generate protected/yiic.php
  mkdir /Users/larryullman/Sites/htdocs/themes
  mkdir /Users/larryullman/Sites/htdocs/themes/classic
  mkdir /Users/larryullman/Sites/htdocs/themes/classic/views
generate themes/classic/views/.htaccess
  mkdir /Users/larryullman/Sites/htdocs/themes/classic/views/layouts
  mkdir /Users/larryullman/Sites/htdocs/themes/classic/views/site
  mkdir /Users/larryullman/Sites/htdocs/themes/classic/views/system

Your application has been created successfully under /Users/larryullman/Sites/htdocs.
: [REDACTED]
```

**Figure 2.3:** The shell of the Yii application has been built!

Here, then, is the complete sequence:

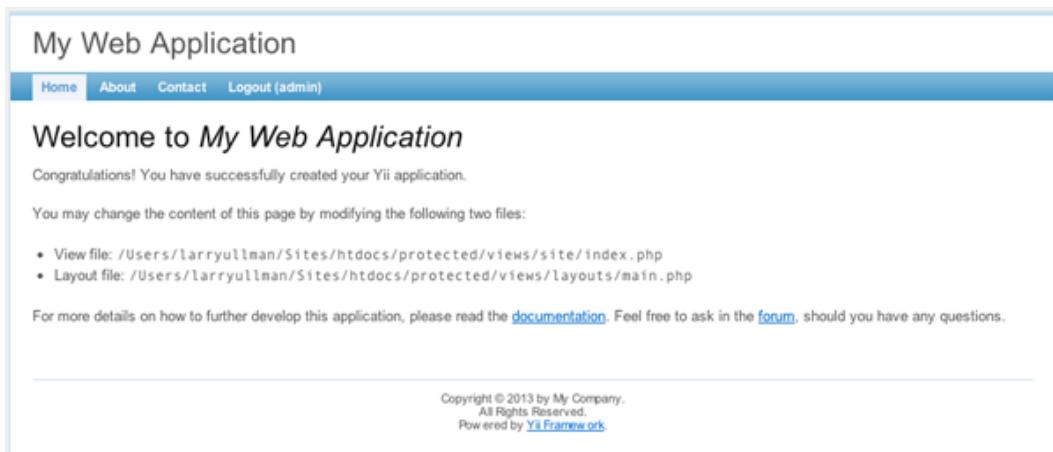
1. Access your computer using the command-line interface. See [Chapter 1](#) if you don't know how to do this.
2. Move into the **framework** directory using the command `cd /path/to/framework`. The `cd` command stands for "change directory". Change the `/path/to/framework` to be accurate for your environment.
3. Create the application using `yiic webapp /path/to/directory` or whatever variation is required. Again, change the `/path/to/directory` to be appropriate for your environment. You may also need to invoke PHP overtly.
4. Enter "Y" at the prompt.

{TIP} If you'll be using Git with your site, the `yiic` tool can create the necessary Git files, too (e.g., `.gitignore` and `.gitkeep`). Just add "git" after the path to the destination directory.

{NOTE} If the PHP executable does not have permission to create the necessary site files in the destination directory, you'll need to change the permissions on that directory. To do so, enter this command from the command-line interface: `chmod -R 755 /path/to/directory`. You may have to preface this with `sudo`, depending upon your environment. It is uncommon that you'll need to do this, however.

## Testing the Site Shell

Unless you saw an error message when you created the site shell, you can now test the generated result to see what you have. To do so, load the site in your browser by going through a URL, of course (**Figure 2.4**).



**Figure 2.4:** The shell of the generated site.

As for functionality, the generated application already includes:

- A home page with further instructions (see Figure 2.4)
- A contact form, complete with CAPTCHA
- A login form
- The ability to greet a logged-in user by name
- Logout functionality

It's a very nice start to an application, especially considering you haven't written a line of code yet! Do note that the contact form will only work once you've edited the configuration to provide your email address. For the login, you can use either demo/demo or admin/admin (username/password). Lastly, the exact look of the application may differ from one version of the Yii framework to another.

So that's the start of a Yii-based Web application. For every site you create using Yii, you'll likely go through these steps. In the next chapter, I'll explain how the site you've just created works.

## Chapter 3

# A MANUAL FOR YOUR YII SITE

Now that you have generated the basic shell of a Yii-based site, it's a good time to go through exactly what it is you have in terms of actual files and directories. This chapter, then, is a manual for your Yii-based Web application. In it, you'll learn what the various files and folders are for, the conventions used by the framework, and how the Yii site works behind the scenes. Reading this chapter and understanding the concepts taught herein should go a long way towards helping you successfully and easily use the Yii framework.

### The Site's Folders

The `yiic` command-line tool generates the shell of the site, including several folders and dozens of files. Knowing how to use the Yii framework therefore begins with familiarizing yourself with the site structure.

In the folder where the Web application was created, you'll find the following:

- **assets**, used by the Yii framework to make necessary resources available
- **css**
- **images**
- **index.php**, a “bootstrap” file through which the entire Web site will be run
- **index-test.php**, a development version of the bootstrap file
- **protected**, where all the site-specific PHP code goes
- **themes**, for theming your site, as you would with a WordPress blog

Of these folders, you'll use **css** and **images** like you would on a standard HTML or PHP-based site. Conversely, you'll never directly do anything with the **assets** folder: Yii uses it to write cached versions of Web resources there. For example, modules

and components will come with necessary resources: CSS, JavaScript, and images. Rather than forcing you to copy these resources to a public directory—and to avoid potential naming conflicts, Yii will automatically copy these resources to the **assets** directory as they are needed. Yii will also provide a copy of the jQuery JavaScript framework there. Note that you should never edit files found within **assets**. Instead, on the rare occasion you have that need, you should edit the master file that gets copied to **assets** (this will mean more later in the book). You can delete entire folders within **assets** to have Yii regenerate the necessary files, but do not delete individual files from within subfolders.

*{WARNING}* The assets folder must be writable by the Web server or else odd errors will occur. This shouldn't be a problem unless you transfer a Yii site from one server to another and the permissions aren't correct after the move.

The **themes** folder can be ignored unless you implement themes. I don't personally use themes that often, and don't formally cover the subject in the book. For more on this subject, see the [Yii documentation](#).

The **protected** folder is the most important folder: you'll edit code found in that folder to change the look and behavior of the site. Unlike the other files and folders, the **protected** folder does not actually have to be in the Web root directory. In fact, for security purposes it is recommended to move it elsewhere (as [explained in the next chapter](#)).

*{TIP}* The **protected** folder is known as the *application base directory* in the Yii documentation.

Within the **protected** folder, you'll find these subfolders:

- **commands**, for `yiic` commands
- **components**, for defining necessary site components
- **config**, which stores your application's configuration files
- **controllers**, where your application's controller classes go
- **data**, for storing the actual database file (when using [SQLite](#)) or database-related files, such as SQL commands
- **extensions**, for third-party extensions (i.e., non-Yii-core libraries)
- **messages**, which stores messages translated in various languages
- **migrations**, for automating database changes
- **models**, where your application's model classes go
- **runtime**, where Yii will create temporary files, generate logs, and so forth
- **tests**, where you'd put unit tests
- **views**, for storing all the view files used by the application

You'll also find three scripts related to the `yiic` tool.

{WARNING} The **protected/runtime** folder must be writable by the Web server.

The **views** folder has some predefined subfolders, too. One is **layouts**, which will store the template for the site's overall look (i.e., the file that begins and ends the HTML, and contains no page-specific content). Within the **views** folder, there will also be one folder for each *controller* you create. In a CMS application, you would have controllers for pages, users, and comments. Each of these controllers gets its own folder within **views** to store the view files specific to that controller.

Again, all of these are within the **protected** folder, also known as the *application directory*. The vast majority of everything you'll do with Yii throughout the rest of this book and as a Web developer will require making edits to the contents of the **protected** folder.

## Referencing Files and Directories

Because the Yii framework adds extra complexity in terms of files and folders, the framework uses several aliases to provide easy references to common locations.

Alias	References
system	<b>framework</b> folder
zii	Zii library location
application	<b>protected</b> folder
webroot	directory where you can find the index file
ext	<b>protected/extensions</b>

If you're using modules in your site—to be covered in Chapter 19, “Extending Yii”, there will be aliases for each module as well.

As for an example of how these aliases are used, if you were to take a peek at the **protected/config/main.php** file, to be discussed in great detail in the next chapter, you'd see this code:

```
'import'=>array(
    'application.models.*',
    'application.components.*',
),
```

That code imports all of the class definitions found in the **protected/models** and **protected/components** folders, because “application” is an alias for the **protected** folder.

## Yii Conventions

The Yii framework embraces the “convention over configuration” approach (also promoted by [Ruby on Rails](#)). What this means is that although you *can* make your own decisions as to how you do certain things, it’s preferable to adopt the Yii conventions. Fortunately, none of the conventions are that unusual, in my experience.

If you really don’t like doing something a certain way, Yii does allow you to change the default convention, but doing so requires a bit more work (i.e., code) and increases the potential for bugs. For example, if you want to organize your **protected** directory in another manner, such as move the view files to another directory, you can, you just need to take a couple more steps.

Let’s first look at the conventions Yii expects within the PHP code and then turn to the underlying database conventions.

### PHP Conventions

First, Yii recommends using upper-camelcase for class names—*SomeClass*—and lower-camelcase for variables and functions: *someFunction*, *someVar*, etc. Camelcase, in case it’s not obvious, uses capital letters instead of underscores to break up words; lower- and upper-camelcase differ in whether the first letter is capitalized or not. Private variables in classes (i.e., attributes) are prefixed with an underscore: \*\$someVar\*. All of these conventions are fairly common among OOP developers.

Additionally, any controller class name must also end with the word “Controller” (note the capitalization): *MyController*.

Files that define classes should have the same name, including capitalization, as the classes they define, plus the **.php** extension: the *MyController* class gets defined within the **MyController.php** file. Again, this is normal in OOP.

You’ll also find that Yii prefixes almost all of *its* classes with a letter to avoid collision issues (i.e., the name of one class conflicting with another). Most Yii classes are prefaced with a capital “C”—**CMenu**, **CModel**, except for *interfaces*, which are appropriately prefaced with a capital “I”: **IAction** or **IWebUser**.

*{TIP}* An *interface* is a special type of class that dictates what methods a class that implements that interface must have defined. Put another way, an interface acts like a contract: in order for objects of this class type to be usable in a certain manner, the class must have these methods.

### Database Conventions

The Yii convention is for the database to use all lowercase letters for both table names and column names, with words separated by underscores: *comment*,

*first\_name*, etc. It is recommended that you use singular names for your database tables—*user*, not *users*, although Yii will not complain if you use plural names. Whatever you decide, consistency is the most important factor (i.e., consistently singular or consistently plural).

You can also prefix your table names to differentiate them from other tables that might be in your database but not used by the Yii application. For example, your Yii site tables might all begin with *yii\_* and your blog tables might begin with *wp\_*.

## How Yii Handles a Page Request

The next thing to learn about your new site is how the Yii framework handles something as basic as a page request. Learning this workflow will go a long way towards understanding the greater Yii context.

In a non-framework site, when a user goes to **http://www.example.com/page.php** in her Web browser, the server will execute the code found in **page.php**. Any output generated by that script, including HTML outside of the PHP tags, will be sent to the browser. In short, there's a one-to-one relationship: the user requests that page and it is executed. The process is not that simple when using Yii (or any framework).

First, whether it's obvious or not, all requests in a Yii-based site will actually go through **index.php**. This is called the “bootstrap” file. With Yii, and some server configuration, all of these requests will be funneled through the bootstrap file:

- **http://www.example.com/**
- **http://www.example.com/index.php**
- **http://www.example.com/index.php?r=site**
- **http://www.example.com/index.php?r=site/login**
- **http://www.example.com/site/login/**
- **http://www.example.com/page/35/**

Note that other site resources, such as CSS, images, JavaScript, and other media, will *not* be accessed via the bootstrap file, but the site's core functionality—the PHP code—will.

Let's look at what the bootstrap file does.

### The Bootstrap File

The contents of the **index.php** file, automatically generated by the **yiic** command, will look something like this:

```
1 <?php
2 // change the following paths if necessary
3 $yii=dirname(__FILE__).'/../yii-dir/framework/yii.php';
4 $config=dirname(__FILE__).'/protected/config/main.php';
5
6 // remove the following lines when in production mode
7 defined('YII_DEBUG') or define('YII_DEBUG',true);
8 // specify how many levels of call stack should be shown in
9 // each log message
10 defined('YII_TRACE_LEVEL') or define('YII_TRACE_LEVEL',3);
11
12 require_once($yii);
13 Yii::createWebApplication($config)->run();
```

Line 3 identifies the location of the Yii framework directory, and specifically the **yii.php** script within it. The next line identifies the configuration file to use for this application. By default, that configuration file is found within the **config** directory of the **protected** folder. The next two lines (7 and 10) establish the debugging behavior.

*{TIP}* The **index-test.php** bootstrap file mostly differs in that it includes an alternate configuration file and is meant to be used in conjunction with unit testing.

At the end of the script, the **yii.php** page is included (line 12). This script defines the **Yii** class. The final line invokes the **createWebApplication()** method of the **Yii** class. This method is provided with the location of the configuration file. The method will return a “Web application” object (technically, in Yii, an object of type **CWebApplication**). The Web application object has a **run()** method, which starts the application. The last line of code is just a single line version of these two steps:

```
$app = Yii::createWebApplication($config);
$app->run();
```

*{NOTE}* The **Yii::createWebApplication()** syntax is an example of calling a class method directly through the class, without a class instance (i.e., an object). There are OOP design reasons for taking this approach, made possible by defining the method as “static”.

That’s all that’s happening in the bootstrap file: a Web application object is created and started, using the configuration settings defined in another file. Everything that will happen from this point on happens within the context of this application object. What happens next depends upon the *route*, but let’s look at the application object in more detail first.

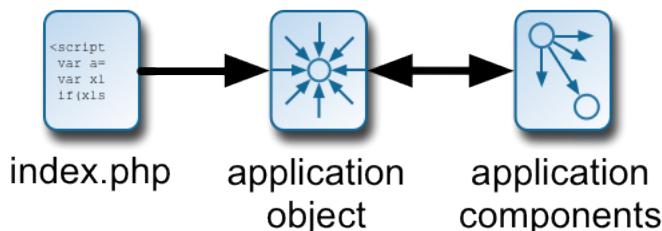
## The Application Object

So what does it mean to say that the Web site runs through the application object? First, the application object manages the components used by the site. For example, the “db” component is used to connect to the database and the “log” component handles any logging required by the site. I’ll get back to components in a couple of pages, just understand here that components are made available to the site through the application object.

The second important task of the application object is to handle the user request. By “user request”, I mean the viewing of a particular page, the submission of a form, and so forth. The handling of the user request is known as *routing*: reading the user’s request and getting the user to the desired end result.

Before explaining routing, let’s get a bit more technical about the application object itself. Within your PHP code, you can access the application object by invoking the static `app()` method of the `Yii` class. This is to say: `Yii::app()`. Whether you need to access the name of the application in a view file (e.g., to set the page title), store a value in a session, or get the identity of the current user, that will be done through `Yii::app()`. This is what I mean when I say that the Web application object is the “context” through which the site runs.

Visually, the bootstrap file’s operations can be portrayed as in (**Figure 3.1**).



**Figure 3.1:** The index page creates an application object which loads the application components.

## Routing

As already mentioned, in a *non-framework site*, the user request will be quite literal (e.g., `http://www.example.com/page.php`). Yii also uses the URL to identify requests, but all requests instead go through `index.php`. To convey the specific request, the route is appended to the URL as a variable. In the default Yii behavior, the request syntax is `index.php?r=ControllerID/ActionID`. All that’s happening there is that a GET variable is passed to `index.php`. The variable is indexed at `r`, short for “route”, and has a value of `ControllerID/ActionID`.

Controllers, as explained in the section on the MVC approach, are the agents in an application: they handle requests and implement the work to be done. In the default site shell created by the `yiic` command, there will be one controller: `site`. In keeping

with Yii conventions, the “site” controller is defined in a class called *SiteController* in a file named **SiteController.php**, stored in the **protected/controllers** directory. The ID of this controller is the name of the class, minus the word “Controller”, all in lowercase. Hence: “site”.

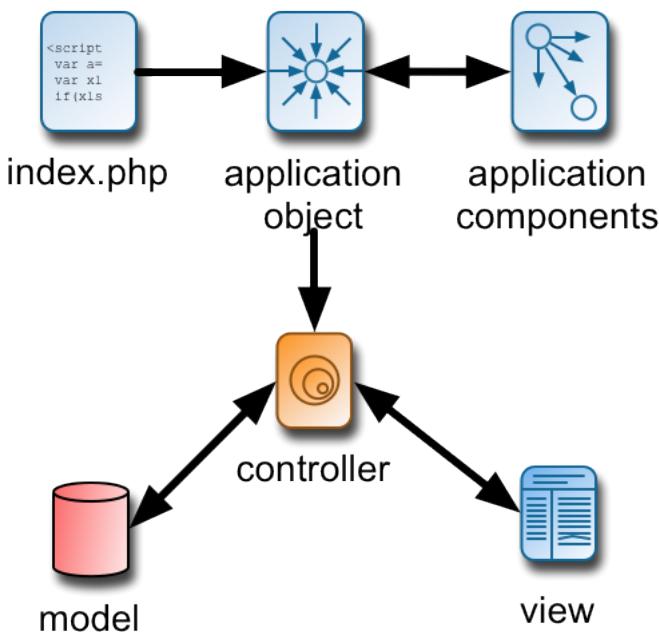
Every controller can have multiple *actions*: specific things done with or by that controller. Four of the actions defined by default in the site controller are: error, login, logout, and contact. As you’ll learn about in much more detail in Chapter 7, “[Working with Controllers](#),” actions are created by defining a method within the controller named “action” plus the action name: `actionError()`, `actionLogin()`, `actionLogout()`, and `actionContact()`. The action ID is the name of the function, minus the initial “action”, all in lowercase. Hence: “error”, “login”, “logout”, and “contact”.

Putting this all together, when the user goes to this URL:

<http://www.example.com/index.php?r=site/login>

That is a request for the “login” action of the “site” controller. Behind the scenes, the application object will read in the request, parse out the controller and action, and then invoke the corresponding method accordingly. In this case, that URL has the end result of calling the `actionLogin()` method of the **SiteController** class.

That’s all there is to routing: calling the correct method of the correct controller class. The controller method itself takes it from there: creating model instances, handling form submissions, rendering views, etc. (**Figure 3.2**).



**Figure 3.2:** Subsequent steps involve the correct controller being called, accessing models, and rendering views.

There are a couple more things to know about routes. First, if an action is not specified, then the default action of the controller will be executed. This is normally the “index” action, represented by the `actionIndex()` method. A request with a controller but no action would be of the format `http://www.example.com/index.php?r=site`.

Second, if neither an action nor a controller is indicated, Yii will execute the default action of the default controller. This is the “index” action of the “site” controller, generated by `yiic`.

Third, many requests will require additional information to be passed along. For example, a CMS site will have a “page” controller responsible for creating, reading, updating, and deleting pages of content. Each of these tasks constitutes an “action”. Three of those—all but “create”—also require a page identifier to know which page of content is being read, updated, or deleted. In such cases, the request URL will become of the format `http://www.example.com/index.php?r=page/delete&id=25`.

Fourth and finally, although the default request syntax is—

`http://www.example.com/index.php?r=ControllerID/ActionID`

This format is commonly altered for Search Engine Optimization (SEO) purposes. With just a bit of customization, the format can be changed to:

`http://www.example.com/index.php/ControllerID/ActionID/`

Taken a step further, you can drop the `index.php` reference and configure Yii to accept `http://www.example.com/ControllerID/ActionID/`. Using the examples already explained, resulting URLs might be:

- `http://www.example.com/site/`
- `http://www.example.com/site/login/`
- `http://www.example.com/page/create/`
- `http://www.example.com/page/delete/id/25/`

You’ll see how this `URL manipulation` is done in Chapter 4, “Initial Customizations and Code Generations.”

## Chapter 4

# INITIAL CUSTOMIZATIONS AND CODE GENERATIONS

After you've created the shell of your Web application, and once you're fairly comfortable with what Yii has generated for you, it's time to start tweaking what was generated to customize your site. First, you'll want to change how your application runs. The first half of the chapter will explain how you do that and introduce the most common settings you'll want to adjust.

Then, it's time to have Yii generate more code for you. But instead of creating just a generic site template, you'll have Yii build boilerplate code based upon the particulars of the database schema you'll be using for the application.

### Enabling Debug Mode

When developing a site, the first thing you'll want to do is make sure that the debugging mode is enabled. This is done (for the entire site) in the bootstrap file, thanks to this line:

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

Written out less succinctly, that line equates to:

```
if (!defined('YII_DEBUG')) {  
    define('YII_DEBUG', true);  
}
```

In short: set debugging to true if it's not already set. This is the default for any newly generated site, but you may want to check that this line is present, just in case you're working with a site someone has already edited, or in case Yii later changes

this default. With debugging enabled, Yii will report problems to you should they occur (and they will).

In **index.php**, the next line of code dictates how many levels of *call stack* are shown in a message log:

```
defined('YII_TRACE_LEVEL') or define('YII_TRACE_LEVEL',3);
```

The call stack is a history of what files, functions, etc., are included, invoked, and so forth. With a framework, the simple loading of the home page could easily involve a dozen actions. In order to restrict the logged (i.e., recorded) data to the freshest, most useful information, the call stack is limited by that line to just the most recent three actions. If you find that's too much or not enough information when you are debugging, just change that value.

While you're checking your debugging settings, I'd recommend that you also confirm that PHP's *display\_errors* setting is enabled. If it's not, then parse errors will result in a blank screen.

{TIP} You can check your PHP's *display\_errors* setting by calling the `phpinfo()` function.

Note that both of these recommendations are for sites that you are developing. Due to the extra debugging information and logging, sites running with these settings will be slower. A production site on a live server should have Yii's debugging mode disabled (by removing that line of code in **index.php**) and PHP's *display\_errors* setting turned off. You'll read more on what else you should do before going live in Chapter 24, “[Shipping Your Project](#).”

## Moving the Protected Folder

Next, for security purposes, you ought to move your **protected** folder outside of the Web root directory. This isn't mandatory, as there is an **.htaccess** file within **protected** to prevent direct access, but if you can move the **protected** folder, you should.

{NOTE} Some (cheaper) hosting environments will not allow you to put things outside of the Web root directory. In such cases, just leave the **protected** folder where it is and don't edit the **index.php** file.

Assuming that the folder **C:\xampp\htdocs\** is my Web root directory (where the site is located), then I would move **protected** to **C:\xampp**. After doing that, the bootstrap file has to be updated so it can find the **protected** folder. Change this line:

```
$config=dirname(__FILE__).'/protected/config/main.php';
```

to

```
$config=dirname(__FILE__).'../protected/config/main.php';
```

The difference is the addition of the two periods before “/protected”, saying to go up one directory to find the **protected** folder.

You can now save the index file and reload the site in your Web browser to confirm that everything still works.

## Basic Configurations

Aside from ensuring that debugging is enabled and possibly changing the location of your **protected** folder, the rest of your site configuration will go within a configuration file. Let’s first look at where the configuration files are and how they work, and then walk through the most important changes to make.

### The Configuration Files

If you look in the **protected/config** directory, you’ll find that three configuration files have been generated for you:

- **console.php**, for configuring console applications
- **main.php**, for the production site
- **test.php**, for testing mode

If you look at **index.php**, you’ll see that it includes **main.php** as its configuration file. The **index-test.php** bootstrap is exactly the same as **index.php**, except that **index-test.php** includes the test configuration file. However, the test configuration file just includes the main configuration file, then also enables the **CDbFixtureManager** component, used for unit testing. The **index-test.php** file also omits the call stack limitation.

*{TIP}* Chapter 21, “Testing Your Applications,” will explain how to perform unit testing in Yii.

If you open the main configuration file in your text editor or IDE, you’ll see that all it does is return an array of name=>value pairs. The first question you may have is: How do I know what names to use and what values (or value types)? Over the rest of this chapter, I’ll explain the most important names and values, but the short

answer is: Any writable property of the `CWebApplication` class can be configured here. Okay, how'd I know that?

As explained in the previous chapter, the bootstrap file creates a Web application object through which the entire site runs. That object will actually be an instance of type `CWebApplication`. The configuration file, therefore, configures this object. And by “configures”, I mean that the configuration file sets the values for the object’s public, writable properties. In other words, the configuration file is a way to tell the Yii framework: when you go to create an object of this type, use these values. That’s all that’s happening in the configuration file, but it’s vital.

For example, `CWebApplication` has a `name` property, which takes a string as the name of the application. By default, `yic` creates this for you:

```
return array(
    'name'=>'My Web Application',
    // Lots of other stuff.
);
```

All you have to do is change the value of the `name` element in that array and you’ll have successfully changed the name of your Web application. (The application name, by the way, is used in page titles and other places.)

As the configuration file is extremely important, you really have to master how it works. To do that, I would first recommend that you be *very* careful when making edits. Because the whole file returns an array, and because many of the values will also be arrays, you’ll end up with nested arrays within nested arrays. A failure to properly match parentheses and use commas to separate items will result in a parse error.

{TIP} You may want to always start by making a duplicate of your existing, working configuration file, before performing new edits.

My second tip is to learn how to read the [Yii class documentation](#), starting with the page for `CWebApplication`. For example, I said that the configuration file can be used for any writable property of that class; using the docs, you can find out what properties exist, what types of values they expect, and whether or not they are writable. **Figure 4.1** shows the manual’s description of `name`:

You can see that the property expects a string value and that it defaults to “My Application” (although the configuration file overwrites that value with “My Web Application”). Now you know that `name` must be assigned a string.

Conversely, look at the documentation for `request` (**Figure 4.2**):

This is a *read-only* property, meaning you cannot assign it a new value in the configuration file.

**name** property

```
public string $name;
```

the application name. Defaults to 'My Application'.

**Figure 4.1:** The Yii docs for the `name` property of the `CWebApplication` class.

**request** property *read-only*

```
public CHttpRequest getRequest()
```

Returns the request component.

**Figure 4.2:** The Yii docs for the `request` property of the `CWebApplication` class.

With this introduction to the configuration file in mind, let's go through the most common and important configuration settings for new projects. Throughout the course of the book, you'll also be introduced to a few other configuration settings, as appropriate.

## Configuring Components

Rather than walk through the configuration file sequentially, I'm going to go in order of most important to least. Arguably the most important section is *components*. Components are application utilities that you and/or Yii have created. To start, you'll configure how your application uses Yii's predefined components.

### Predefined Components

The Yii framework defines [16 core application components](#) for you, representing common needs. Just some of those are:

- authManager, for role-based access control (RBAC)
- cache, for caching of site materials
- clientScript, through which client-side tools—JavaScript and CSS—can be managed
- db, which provides that database connection

- request, for working with user requests
- session, for working with sessions
- user, which represents the current user

Those names are the corresponding configurable `CWebApplication` properties. For each component, Yii defines a class that does the actual work.

In this chapter, I'll explain the basic configuration of the components that are most immediately needed. Throughout the rest of the book, I'll introduce other predefined components as warranted.

## Enabling and Customizing Components

Components are made available to the Yii application, and customized, via the configuration file's "components" section:

```
return array(
    'name'=>'My Web Application',
    'components' => array(
        ), // End of components array.
        // Lots of other stuff.
);
```

Within the "components" section of the configuration file, each component is declared and configured using the syntax:

```
'componentName' => array( /* configuration values */)
```

The names of the predefined components are: "authManager", "cache", and the others already mentioned, plus a few more listed in the manual (and discussed, when appropriate, in this book). The name is also the component's ID.

As for the values, they will vary from one component to the next. To know what configuration is possible for a component, you'll need to look at the underlying class that provides that component's functionality. For example, the `db` component provides a database connection. The associated class is `CDbConnection`. In other words, when a database connection is required, an object of `CDbConnection` type will be created. The "db" element of the "components" section of the configuration file can set the values of that object's properties.

Looking at the [Yii API reference](#) (aka, the class documentation), you can see that the `CDbConnection` class has a public, writable `username` property. Therefore, that property's value can be assigned in your configuration file:

```
'components' => array(
    'db' => array(
        'username' => 'someuser'
    ) // End of db array.
), // End of components array.
```

With that line in the configuration file, when the site needs a database connection, Yii will create an instance of `CDbConnection` type, using “`someuser`” as the value of the object’s `username` property.

{NOTE} Just as the whole configuration file can only assign values to the writable properties of the `CWebApplication` object, individual configurations can only assign values to the writable properties of the associated class (e.g., the writeable properties of `CDbConnection`).

Over the next few pages, I’ll explain the most important ways to configure the most important components. But first, there are two more things to know about these application components.

First, Yii wisely only creates instances of application components when the component is used. For example, if you configure your Web site to have a database component, Yii will still only create that component on pages of your site that use the database. Thanks to Yii’s automatic management of components, your site will perform better without you having to perform tedious tweaks and edits on a page-by-page basis (i.e., to turn components on and off).

Still, Yii can be told to *always* create an instance of a component. This is done through the “preload” element of the main configuration array:

```
'preload'=>array('log'),
```

By default, the logging component is always loaded. To always load other components, you would just add those component IDs to that array. Although, for performance reasons, you should only do so sparingly.

The second thing to know about application components is how to access them in your code (e.g., in controllers). Components are available in your code via `Yii::app()->ComponentID`, where the *ComponentID* value comes from the configuration file. For example, in theory, you could change the database username on the fly (although you never would):

```
Yii::app()->db->username = 'this username';
```

With this understanding of how components in general are configured, let’s look at a handful of the most important components when starting a new Yii application.

## Connecting to the Database

Unless you are not using a database, you'll need to establish the database connection before doing anything else. (And if you're not using a database, there's probably a good argument that you shouldn't be using a framework, either.) Establishing the database connection is accomplished through the “db” component, as already mentioned. In the default configuration file created by Yii, a connection to an SQLite database is established:

```
'db'=>array(
    'connectionString' =>
        'sqlite:' . dirname(__FILE__) . '/../data/testdrive.db'
),
```

If you are using SQLite for your project, just change that line so that it correctly points to the location of your SQLite database. If you're not using SQLite, comment out or remove those three lines of code and take a look at the lines following it in the configuration file:

```
'db'=>array(
    'connectionString' => 'mysql:host=localhost;dbname=testdrive',
    'emulatePrepare' => true,
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
),
```

By default, those lines will be disabled as they are surrounded by the `/*` and `*/` comment tags. Remove those tags to enable this configuration. Then you'll need to change the values to match your setup.

The connection string is a *DSN* (Database Source Name), which has a precise format. It starts with a keyword indicating the database application being used, like “mysql”, “pgsql” (PostgreSQL), “mssql” (Microsoft’s SQL Server), or “oci” (Oracle). This keyword is followed by a colon, then, depending upon the database application being used, and the server environment, any number of parameters, each separated by a semicolon:

- mysql:host=localhost;dbname=test
- mysql:port=8889;dbname=somedb
- mysql:unix\_socket=/path/to/mysql.sock;dbname=whatever

Indicating the database to be used is most important. Depending upon your environment, you may also have to set the port number or socket location. For me,

when using MAMP on Mac OS X, I had to set the port number as it was not the expected default (of 3306). On Mac OS X *Server*, I had to specify the socket, as the expected default was not being used there. Also do keep in mind that you'll need to have the proper PHP extensions installed for the corresponding database, like PDO and PDO MySQL.

You should obviously change the username and password values to the proper values for your database. You may or may not want to change the character set.

{NOTE} If you don't know what your database connection values are—the username and password, then this book might be too advanced for you in general. This is fairly basic MySQL knowledge, which is assumed by this book.

And that's it! Hopefully your Yii site will now be able to interact with your database. You'll know for sure shortly.

## Managing URLs

Next, I want to look at the *urlManager* component. This component dictates, among other things, what format the site's URLs will be in.

**Creating SEO-friendly URLs** As explained in Chapter 3, “[A Manual for Your Yii Site](#),” the default URL syntax is:

`http://www.example.com/index.php?r=ControllerID/ActionID`

For SEO purposes, and because it looks nicer for users, you'll probably want URLs to be in this format instead:

`http://www.example.com/index.php/ControllerID/ActionID/`

To do that, just enable the “*urlManager*” component and customize its behavior. Yii nicely provides the right syntax for you in the main configuration file, you just need to remove the comment tags from around the following:

```
'urlManager'=>array(
    'urlFormat'=>'path',
    'rules'=>array(
        '<controller:\w+>/<id:\d+>'=>'<controller>/view',
        '<controller:\w+>/<action:\w+>/<id:\d+>'
            =>'<controller>/<action>',
        '<controller:\w+>/<action:\w+>'
            =>'<controller>/<action>',
    ),
),
```

Note that you don't have to do anything to Apache's configuration (i.e., to the Web server itself) for this to work. By using this component, any links created within the site will use the proper syntax as well (you'll learn much more about how this works in Chapter 7, “[Working with Controllers](#)”).

If you want, you can test this change already. After removing the comment tags around that code, save the configuration file, and then reload the home page in your Web browser. Click on “Contact” and you should see that the URL is now `http://www.example.com/index.php/site/contact` instead of `http://www.example.com/index.php?r=site/contact`.

**Hiding the Index File** If you want to take your URL customization further, it's possible to configure “urlManager”, along with an Apache `.htaccess` file, so that `index.php` no longer needs to be part of the URL:

`http://www.example.com/ControllerID/ActionID/.`

To do this, you have to add a `mod_rewrite` rule to an `.htaccess` file stored in your Web root directory (i.e., in the same directory as `index.php`). The contents of that file should be:

```
<ifModule mod_rewrite.c>
# Turn on the engine:
RewriteEngine on

# Don't perform redirects for files and directories that exist:
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# For everything else, redirect to index.php:
RewriteRule ^(.*)$ index.php/$1
</ifModule>
```

If you're not familiar with `mod_rewrite`, you can look up oodles of tutorials online. Note that this modification will only work if your Web server allows for configuration overrides using `.htaccess`. If not, then you may be a bit over your head anyway. You can either look online for how to allow for overrides via `.htaccess`, skip this step for now, or ask for help in my [support forums](#).

Once you've implemented the `mod_rewrite` rules, to test if `mod_rewrite` is working, go to any other file in the Web directory (e.g., an image or your CSS script) to see if that loads. Then go to a URL for something that *doesn't* exist (e.g., `www.example.com/varmit`) and see if the contents of the index page are shown instead (most likely with an error message).

Finally, you must tell the URL manager not to show the bootstrap file by setting the `showScriptName` property to false:

```
'urlManager'=>array(
    'showScriptName'=>false,
    'urlFormat'=>'path',
    'rules'=>array(
        '<controller:\w+>/<id:\d+>'=>'<controller>/view',
        '<controller:\w+>/<action:\w+>/<id:\d+>'=>'<controller>/<action>',
        '<controller:\w+>/<action:\w+>'=>'<controller>/<action>',
    ),
),
```

## Logging

A third component I'd recommend you configure before you begin working is *log*. The log component writes pertinent information to a text file. If you're not in the habit of implementing logging on your projects, you're missing out on something wonderfully useful. Logs serve two excellent purposes:

1. They allow you to investigate a problem after the fact (i.e., without attempting to recreate the problem yourself, which can be anywhere from not easy to impossible).
2. They allow you to see errors and problems without them being visible to public users.

The logging component is enabled by default, and, as already mentioned, is set to always be loaded. If you want to quickly test it, you just need to create an error. For example, by changing the site URL from **index.php/site/contact** to **index.php/site/contacts**, you'll create a page not found (404) exception (**Figure 4.3**).

As you can see in the image, Yii reports that there is no action that matches “contacts”. Yii also logs this occurrence. To view that, open the **protected/runtime/application.log** file in any text editor (**Figure 4.4**).

That's one example of logging, but as already said, this is enabled by default. I recommend that while you're debugging a project, you also enable **CWebLogRoute**. This tool will output tons of useful details to each rendered page. The code for enabling it is already in the main configuration file, you just need to remove the comment tags from around it. Here's the relevant logging code in its entirety:

```
'log'=>array(
    'class'=>'CLogRouter',
    'routes'=>array(
        array(
            'class'=>'CFileLogRoute',
```

# My Web Application

Home   About   Contact   Login

[Home](#) » Error

## Error 404

The system is unable to find the requested action "contacts".

**Figure 4.3:** The default page not found response.

```
X application.log
1 2013/02/22 14:43:31 [error] [exception.CHttpException.404] exception 'CHttpException' with message
. 'The system is unable to find the requested action "contacts".' in
. /Users/larryullman/Sites/framework-yii-1.1.13/web/CController.php:483
2 Stack trace:
3 #0 /Users/larryullman/Sites/framework-yii-1.1.13/web/CController.php(270):
. CController->missingAction('contacts')
4 #1 /Users/larryullman/Sites/framework-yii-1.1.13/web/CWebApplication.php(282):
. CController->run('contacts')
5 #2 /Users/larryullman/Sites/framework-yii-1.1.13/web/CWebApplication.php(141):
. CWebApplication->runController('site/contacts')
6 #3 /Users/larryullman/Sites/framework-yii-1.1.13/base/CApplication.php(169):
. CWebApplication->processRequest()
7 #4 /Users/larryullman/Sites/htdocs/index.php(13): CApplication->run()
8 #5 {main}
9 REQUEST_URI=/index.php/site/contacts
10 ---
```

**Figure 4.4:** The logging information for the page not found exception.

```
        'levels'=>'error, warning',
),
array(
    'class'=>'CWebLogRoute',
),
),
),
),
```

With the same error in place (the request for a page/action that does not exist), **Figure 4.5** shows some of the output generated by `CWebLogRoute`.

Application Log			
Timestamp	Level	Category	Message
14:43:31.181961	trace	system.CModule	Loading "log" application component in /Users/larryullman/Sites/htdocs/index.php (13)
14:43:31.183257	trace	system.CModule	Loading "request" application component in /Users/larryullman/Sites/htdocs/index.php (13)
14:43:31.184525	trace	system.CModule	Loading "urlManager" application component in /Users/larryullman/Sites/htdocs/index.php (13)
14:43:31.188972	trace	system.CModule	Loading "coreMessages" application component in /Users/larryullman/Sites/htdocs/index.php (13)
			exception 'HttpException' with message 'The system is unable to find the requested action "contacts".' in /Users/larryullman/Sites/framework-yii-1.1.13/web/CController.php:48 Stack trace: #0 /Users/larryullman/Sites/framework-yii-1.1.13/web/CController.php(278): CController->missingAction('contacts') #1 /Users/larryullman/Sites/framework-yii-1.1.13/web/CWebApplication.php(282): CController->run('contacts') #2 /Users/larryullman/Sites/framework-yii-1.1.13/web/CWebApplication.php(141): CwebApplication->runController('site/contacts') #3 /Users/larryullman/Sites/framework-yii-1.1.13/base/CApplication.php(169): CwebApplication->processRequest() #4 /Users/larryullman/Sites/htdocs/index.php(13): CApplication->run() #5 {main} REQUEST_URI=/index.php/site/contacts ...
14:43:31.191114	error	exception.CHttpException.404	

**Figure 4.5:** The same error as in Figure 4.3, with extra Web logging.

## Modules

After you've configured the necessary components, there are a few other configuration settings you should look at. One is under the “modules” section. Modules are essentially mini-applications within a site. You might create an administration module or a forum module. Chapter 19, “Extending Yii,” will cover creating modules in detail, but to begin developing your site, you'll want to enable one of Yii's modules: Gii.

Gii is a Web-based tool that you'll use to generate boilerplate model, view, and controller code for the application (based upon your database tables). Gii is a

wonderful tool, and a great example of why I love Yii: the framework does a lot of the development for you.

To enable Gii, just remove the comment tags–/\* and \*/–that surround this code:

```
'gii'=>array(
    'class'=>'system.gii.GiiModule',
    'password'=>'Enter Your Password Here',
    // If removed, Gii defaults to localhost only.
    // Edit carefully to taste.
    'ipFilters'=>array('127.0.0.1','::1'),
),
```

Next, enter a secure password in that code, one that only you will know. The *ipFilters* option let's you declare through what IP addresses Gii can be accessed. Understand, however, that Gii should not be used on a live site.

*{TIP}* If you're using a very secure development server, like your own local machine, you can set the password to "false" (the Boolean, without any quotes), allowing you to use Gii without authentication.

With Gii enabled and configured, you'll be able to use it later in this chapter.

*{NOTE}* Gii was added to Yii in version 1.1.2, and it replaces functionality previously made available using the command-line Yii tools.

## Parameters

At the end of the configuration file, there's a "params" element. This is where user-defined parameters can be established. One will already be there for you:

```
'params'=>array(
    // this is used in contact page
    'adminEmail'=>'webmaster@example.com',
),
```

Change this to your email address, so that you receive error messages, contact form submissions, or whatever. Understand that for those emails to be sent, your Web server must be configured properly. On a live server, that shouldn't be a problem, but on your own test system, you may need to install a mail server or configure PHP to use an SMTP server.

You can add other name=>value pairs here, if you'd like:

```
'params'=>array(
    // this is used in contact page
    'adminEmail'=>'webmaster@example.com',
    'something' => 23,
),
```

By setting this parameter, in your site's code (e.g., in a controller), you'll be able to globally access the parameter value via `Yii::app()->params['something']`. You'll see examples of this later in the book.

## Developing Your Site

Over the course of the book, I'll work with different practical examples so that you can use real-world code to learn new things. In Part 4 of the book, I'll create a couple of examples in full (or mostly full), in order to show how all of the ideas come together in context. But the primary example to be used throughout the book is a Content Management System (CMS). CMS is a fairly generic term that applies to so many of today's Web sites. I would describe CMS as a moderately complex application to implement, and so it makes for a good example in this book.

In the next several pages, you'll learn not just how to design a CMS site, but also how to approach designing any new project.

### Identifying the Needed Functionality

Simply put, projects are a combination of *data*, *functionality*, and *presentation*. Games have a lot more of the latter two and many Web-based projects focus on the data, but those are the three elements, in varying percentages. When you go to start a new project, it's in one of these three areas that you must begin. And you should always start with the functionality, as that dictates everything else. The functionality is what a Web site or application must be able to do, along with the corollary of what a user must be able to do with the Web site or application. The functionality needs to be defined in advance. Use a paper and pen (or note-taking application), and write down everything the project requires:

- Presentation of content
- User registration, login, logout
- Search
- Rotating banner ads
- Et cetera

Try your best to be exhaustive, and to perform this task without thinking of files and folders, let alone specific code. Be as specific as you can about what the project has to be able to do, down to such details as:

- Show how many users are online
- Cache dynamic pages for improved performance
- Not use cookies or only use cookies
- Have sortable tables of data

The more complete and precise the list of requirements is, the better the design will be from the get-go, and you'll need to make fewer big changes as the project progresses.

{NOTE} The development process and the site's functionality will be dictated by your business goals, too: how much money you're able to spend, how much money you'd like to make (and through what means), etc. But for a developer, and for the purposes of this book, the site's functionality is most important.

With a CMS, the most obvious functionality is to present content for people to view. This implies related functionality:

- Someone should be able to create new content
- Someone should be able to edit existing content

(Maybe content should also be deletable, but I'd rather make content no longer live than remove it entirely.)

This is a fine start, but the CMS would be better if people could also comment on content. So there's another bit of functionality to be implemented.

And I should define what I mean by "content". For most of the Web, content is in the form of HTML, even if that HTML includes image and video references. But it would be nice if the content could also have files that are downloadable. This feature adds a few more requirements:

- The ability to upload files
- The ability to associate files with pages of content (i.e., the files will be linked somewhere on the site)
- The ability to download files
- The ability to change a previously uploaded file

And to better distinguish between a page of content and file content, let's start calling the page of content a "page".

But I'm not done yet: let's assume that all of the content is publicly viewable, but there ought to be limits as to who can create and edit content. More functionality:

- Support for different user types

- Only certain user types can author content
- Only certain user types can edit content (e.g., the original author, plus administrators)
- Only certain user types can assign types to users

As you can see, one initial goal—present content—has quickly expanded into over a dozen requirements. As I said, this is a moderately complex example, but will work well for the purposes of this book.

## Next Steps

Once you've come up with the functionality (with the client, too, if one exists), it's time to start coding and creating files and folders. You can start that process from one of two directions: the data or the presentation. In other words, you can begin with the user interface and work your way down to the code and database or you can begin with the database and work your way up to the user interface. I'm a developer and a data-first person, but let's look at the presentation approach, too.

If you're a designer, or are working with clients that think primarily in visual terms, it makes sense to begin any new project with how it will look. You may want to start with a wireframe representation, or actual HTML, but create a series of pages or images that provides a usable basis for how the site will appear from a user interface perspective. You don't need to create every page, and in a dynamically driven site you actually shouldn't, but address the key and common parts. The end goal is the HTML, CSS, and media, in a final or nearly final state. Once you've done that, and the client has accepted it, you can work your way backwards through the functionality and data.

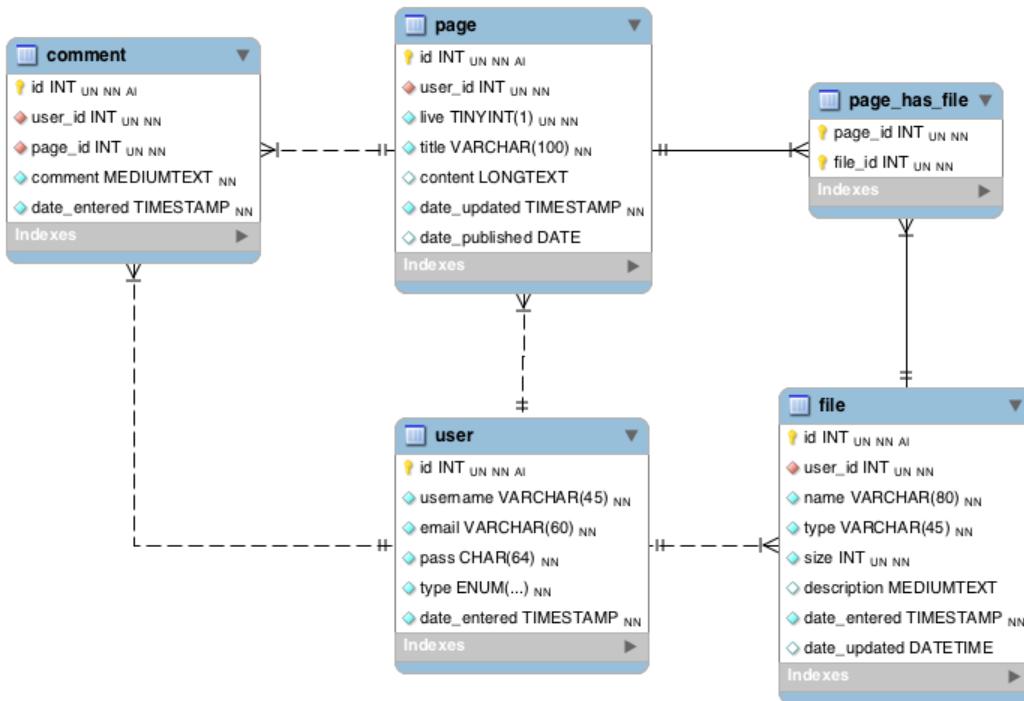
If you're a developer, like me, incapable of thinking in graphical terms, it makes sense to begin any new project from the perspective of the data: what will be stored and how the stored information will be used. For this task, you'll want to use a paper and pen, or a modeling tool such as the [MySQL Workbench](#), but the goal is to create a database schema. Always err on the side of storing too much information, and always err on the side of complete normalization (when using a relational database). Once you've done that, I normally populate the database with some sample data. This allows me to then create the functionality that ties the data into a sample presentation. By doing so, I, and the client, can confirm that the site looks and works as it should. From there, you can implement more functionality, and then have the entire presentation and interface finalized.

With the CMS site, I already have a sense of data used by the site: pages, users, comments, and files. As a quick check, I can look back over the needed functionality and confirm that everything that the site must be able to do will involve just those four types of things.

## Defining the Database

Now that the functionality has been identified, and I know what data the site will use, it's time to create the database itself. Using a relational database application such as MySQL, one would go through the process of *normalizing* a database. It's beyond the scope of this book to explain that process here (and it's the kind of thing I would assume you already know), but if you're not familiar with database normalization, search online for tutorials or check out my “[PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide](#)” book.

**Figure 4.6** shows the database schema, as designed in the [MySQL Workbench](#).



**Figure 4.6:** The CMS database schema.

I'll now walk through the tables, and the corresponding SQL commands, individually. You should notice that I'm keeping with the Yii [database conventions](#): singular table names, all lowercase table and column names, and *id* for the primary keys. Also, every table will be of the InnoDB type—MySQL's current default storage engine, and use the UTF8 character set.

{NOTE} You can download the complete SQL commands, along with some sample data, from the account page on the book's Web site.

```
CREATE TABLE IF NOT EXISTS yii_cms.user (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    username VARCHAR(45) NOT NULL,
    email VARCHAR(60) NOT NULL,
    pass CHAR(64) NOT NULL,
    type ENUM('public','author','admin') NOT NULL,
    date_entered TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    UNIQUE INDEX username_UNIQUE (username ASC),
    UNIQUE INDEX email_UNIQUE (email ASC)
)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
```

The `user` table stores information about registered users. The table will store the user's username, which must be unique, her email address, which must also be unique, and her password. Users can be one of three types, with *public* being the default (MySQL treats the first item in an ENUM column as the default).

New user records can be created using this SQL command:

```
INSERT INTO user (username, email, pass) VALUES ('<username>',
'<email>', SHA2('<password><username><email>', 256))
```

As you can see, the stored password is salted by appending both the user's name and the user's email address to the supplied password, with the whole string run through the `SHA2()` method, using 256-bit encryption (which returns a string 64 characters long). If your version of MySQL does not support `SHA2()`, you can use another encryption or hashing function.

The `page` table stores a page of HTML content:

```
CREATE TABLE IF NOT EXISTS yii_cms.page (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    user_id INT UNSIGNED NOT NULL,
    live TINYINT(1) UNSIGNED NOT NULL DEFAULT 0,
    title VARCHAR(100) NOT NULL,
    content LONGTEXT NULL,
    date_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    date_published DATE NULL,
    PRIMARY KEY (id),
    INDEX fk_page_user_idx (user_id ASC),
    INDEX date_published (date_published ASC),
    CONSTRAINT fk_page_user
        FOREIGN KEY (user_id )
        REFERENCES yii_cms.user (id )
```

```
    ON DELETE CASCADE
    ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

{NOTE} When revising this text for release version 0.5, I made a slight alteration to the `page` table. Specifically, I swapped the order of the two date columns and changed the `date_entered` TIMESTAMP NOT NULL column to `date_published` DATE NULL.

There's nothing too revolutionary here, save for the use of the foreign key constraint, as there's a relationship between `page` and `user`. MySQL supports foreign key constraints when using the InnoDB type. This particular constraint says that when the `user.id` record that relates to this table's `user_id` column is deleted, the corresponding records in this table will also be deleted (i.e., the changes will cascade from `user` into `page`). You may not want to cascade this action; you could have the `user_id` be set to NULL instead:

```
CREATE TABLE IF NOT EXISTS yii_cms.page (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    user_id INT UNSIGNED NULL,
    /* other columns and indexes */
    CONSTRAINT fk_page_user
        FOREIGN KEY (user_id )
        REFERENCES yii_cms.user (id )
        ON DELETE SET NULL
        ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

Note that setting the `user_id` value to NULL is only possible if the column allows for NULL values as in the above modified SQL.

New page records can be created using this SQL command:

```
INSERT INTO page (user_id, title, content) VALUES
(23, 'This is the page title.', 'This is the page content.')
```

When the page is ready to be made public, you'd change its `live` value to 1 and set its `date_published` column to the publication date.

Next, there's the `comment` table, with relationships to both `page` and `user`:

```
CREATE TABLE IF NOT EXISTS yii_cms.comment (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    user_id INT UNSIGNED NOT NULL,
    page_id INT UNSIGNED NOT NULL,
    comment MEDIUMTEXT NOT NULL,
    date_entered TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    INDEX fk_comment_user_idx (user_id ASC),
    INDEX fk_comment_page_idx (page_id ASC),
    INDEX date_entered (date_entered ASC),
    CONSTRAINT fk_comment_user
        FOREIGN KEY (user_id )
        REFERENCES yii_cms.user (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION,
    CONSTRAINT fk_comment_page
        FOREIGN KEY (page_id )
        REFERENCES yii_cms.page (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

Again, there are foreign key constraints here, but nothing new.

New comment records can be created using this SQL command:

```
INSERT INTO comment (user_id, page_id, comment) VALUES
(23, 149, 'This is the comment.')
```

Next, there's the `file` table, for storing information about uploaded files. It relates to `user`, in that each file is owned by a specific user:

```
CREATE TABLE IF NOT EXISTS yii_cms.file (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    user_id INT UNSIGNED NOT NULL,
    name VARCHAR(80) NOT NULL,
    type VARCHAR(45) NOT NULL,
    size INT UNSIGNED NOT NULL,
    description MEDIUMTEXT NULL,
    date_entered TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    date_updated DATETIME NULL,
    PRIMARY KEY (id),
```

```
INDEX fk_file_user1_idx (user_id ASC),
INDEX name (name ASC),
INDEX date_entered (date_entered ASC),
CONSTRAINT fk_file_user
    FOREIGN KEY (user_id )
        REFERENCES yii_cms.user (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

The file's name, type (as in MIME type), and size would come from the uploaded file itself. The description is optional.

New file records can be created using this SQL command:

```
INSERT INTO file (user_id, name, type, size, description)
VALUES (23, 'somefile.pdf', 'application/pdf', 239085,
'This is the description')
```

Finally, the `page_has_file` table is a middleman for the many-to-many relationship between `page` and `file`:

```
CREATE TABLE IF NOT EXISTS yii_cms.page_has_file (
    page_id INT UNSIGNED NOT NULL,
    file_id INT UNSIGNED NOT NULL,
    PRIMARY KEY (page_id, file_id),
    INDEX fk_page_has_file_file_idx (file_id ASC),
    INDEX fk_page_has_file_page_idx (page_id ASC),
    CONSTRAINT fk_page_has_file_page
        FOREIGN KEY (page_id )
        REFERENCES yii_cms.page (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION,
    CONSTRAINT fk_page_has_file_file
        FOREIGN KEY (file_id )
        REFERENCES yii_cms.file (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
```

New `page_has_file` records can be created using this SQL command:

```
INSERT INTO page_has_file (page_id, file_id) VALUES (23, 82);
```

And there you have the entire sample database. No doubt there are things you might do differently and, if so, feel free to edit my design as you'd prefer it to be. Just remember to factor in your edits when working with the code later in the book.

This database also makes a couple of assumptions. First, only logged-in users can make comments. If you'd want to allow *anyone* to post comments, then you wouldn't tie the comments to the `user` table, instead storing the information about the person making the comment in `comment`.

Second, this database doesn't support the option of categorizing or tagging content. That's easy enough to implement, however. You can either add that functionality yourself, or perhaps I'll add that to the fuller implementation of this project in Chapter 22, “[Creating a CMS](#).”

## Foreign Key Constraints in MyISAM Tables

In the previous section, in which I outline the database schema, I made repeated references to the foreign key constraints in place. Foreign key constraints are beneficial in databases as they help to insure data integrity. It's anywhere from messy to outright bad if a record in one table remains after a related record in another table is removed. However, there is another benefit to foreign key constraints in Yii-based applications beyond just data integrity.

In Yii, a model will be based upon a database table. In situations where one database table is related to another, such as `user` to `comment`, it's helpful to recognize the relationship in the corresponding model files, too (i.e., in the PHP code). For example, if the `Comment` model is identified as being related to `User` through its `user_id` property, then instances of `Comment` type can use knowledge of that relationship to, for example, easily retrieve the `username` associated with the `user_id` of the current comment. For this reason, Yii will automatically read the foreign key constraints and use them to identify relationships in models, as you're about to see in the section on using Gii.

The problem is that MySQL only enforces foreign key constraints in InnoDB tables (when *every* table involved uses the InnoDB storage engine). This may be a problem as MyISAM was the default storage engine for years, and you may still be using it. If so, you can't use foreign key constraints. Still you *can* indicate to Yii that a relationship exists between two tables by adding a comment to the related column. Here is the `page` table, without the foreign key constraint but with the comment:

```
CREATE TABLE IF NOT EXISTS yii_cms.page (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    user_id INT UNSIGNED COMMENT
        "CONSTRAINT FOREIGN KEY (user_id) REFERENCES User(id)",
```

```
/* other columns and indexes */
)
ENGINE = MyISAM
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci;
```

The comment that's part of the `user_id` column indicates that this column relates to the `id` column of the `user` table (or, technically, that `user_id` references the `id` property of the `User` model to be created by Yii).

To be clear, this comment has no effect on MySQL at all, but when you generate the models for these tables, Yii will automatically add the code to reflect the proper relationships.

*{TIP}* If you don't add this comment, it's not a big deal as you can write the code to indicate the relation yourself, but it's nice that Yii will do it for you.

## Creating the Database

Once you've defined your database in SQL terms, you'll need to create it in MySQL (or whatever database application you're using). You should do that now. You can use whatever tools you'd like, and the SQL commands I used are available to download from this [book's Web site](#).

Before continuing, you should also double-check your main configuration file to confirm that it is set to connect to the proper database used by the application. As already mentioned, for the purposes of this book, I'm calling this the `yii cms` database.

## Generating Code with Gii

Once you've created your database, and configured your Yii site to connect to it, it's time to fire up Gii. Again, the purpose of Gii is to create the fundamental model, view, and controller files required by the site. Most of what you'll learn in Part 2 of this book will be how to edit these files to tweak them to your particular needs.

### Gii Requirements

Before going any further, you should go through the following checklist (if you have not already):

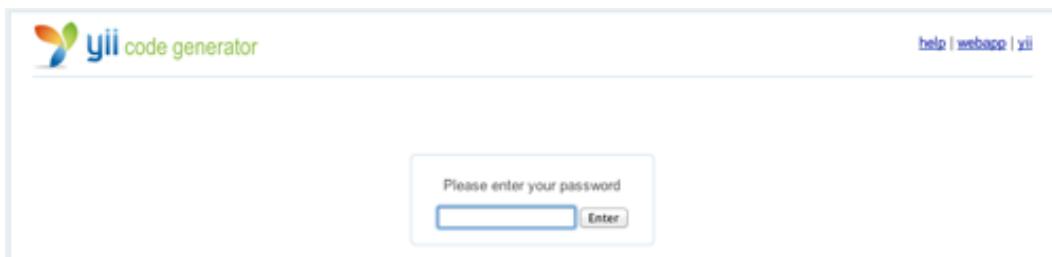
1. Confirm that your Yii installation meets the [minimum requirements](#).

2. Have your database design as complete as possible. Because Gii does so much work for you, it's best not to have to make database changes later on. If done properly, after creating your database tables following these next steps, you won't use Gii again for the project.
3. Enable Gii, using the [instructions provided earlier in this chapter](#) (and remember the password you identified).
4. Be using a development server (ideally).

Preferably, you've enabled Gii on a development server, you'll use it, and then disable it, and then later put the site online.

Assuming you understand all of the above and have taken the requisite steps, you should now load Gii in your browser. Assuming your site is to be found at [www.example.com/index.php](http://www.example.com/index.php), the Gii tool is at [www.example.com/index.php/gii/](http://www.example.com/index.php/gii/). This URL also assumes you're using the URL management component in Yii. If not, head to [www.example.com/index.php?r=gii](http://www.example.com/index.php?r=gii) instead.

Using that address, you should be taken to the login screen (**Figure 4.7**).



**Figure 4.7:** The Gii login page.

Enter your Gii password (established in the configuration file), and click Enter. Assuming you entered the correct password, you'll see a splash page and a list of options (**Figure 4.8**).

As you can see, Gii can be used to generate:

- Controllers
- CRUD functionality
- Forms
- Models
- Modules

Over the next couple of pages, you'll use two of these options: models and then CRUD.



## Welcome to Yii Code Generator!

You may use the following generators to quickly build up your Yii application:

- [Controller Generator](#)
- [Crud Generator](#)
- [Form Generator](#)
- [Model Generator](#)
- [Module Generator](#)

**Figure 4.8:** The Gii home page.

### Generating Models

The first thing you'll want to do is generate the models. Click the “Model Generator” link. On the following page (**Figure 4.9**):

1. Enter \* as the table name.
2. Click Preview.
3. In the preview (**Figure 4.10**), deselect the **PageHasFile.php** model, which is not needed by this application.
4. Click Generate.

The \* is a shortcut to have Gii automatically model every database table. If you'd rather generate a model at a time, you can enter a table name in the first field and work through Steps 2 and 3, and then repeat for every other table. You can also deselect models in the preview if you don't want them generated.

After clicking Generate, you should then see a message indicating that the code was created. You can check for the new files within the **protected/models** directory to confirm this. If you see an error about an inability to write the file, you'll need to modify the permissions on the **protected/models** directory to allow the Web server to write there (**Figure 4.11**).

For the CMS example, these steps generate four files within the **protected/models** directory:

- **Comment.php**
- **File.php**
- **Page.php**

## Model Generator

This generator generates a model class for the specified database table.

*Fields with \* are required. Click on the highlighted fields to edit them.*

**Database Connection \***

db

**Table Prefix**

[empty]

**Table Name \***

\*

**Base Class \***

CActiveRecord

**Model Path \***

application.models

**Build Relations**

**Code Template \***

default (/Users/larryullman/Sites/framework-yii-1.1.13/gii/generators/model/templates/default)

**Preview**

**Figure 4.9:** The form for auto-generating a model file.

<b>Code File</b>	<b>Generate</b> <input type="checkbox"/>
<a href="#">models/Comment.php</a>	new <input checked="" type="checkbox"/>
<a href="#">models/File.php</a>	new <input checked="" type="checkbox"/>
<a href="#">models/Page.php</a>	new <input checked="" type="checkbox"/>
<a href="#">models/PageHasFile.php</a>	new <input type="checkbox"/>
<a href="#">models/User.php</a>	new <input checked="" type="checkbox"/>

**Figure 4.10:** The preview of the files to be created.

Preview

There was some error when generating the code. Please check the following messages.

```
Generating code using template "/Users/larryullman/Sites/framework-yii-1.
generating models/Comment.php
    Unable to write the file '/Users/larryullman/Sites/htdocs/proj
generating models/File.php
    Unable to write the file '/Users/larryullman/Sites/htdocs/proj
generating models/Page.php
    Unable to write the file '/Users/larryullman/Sites/htdocs/proj
skipped models/PageHasFile.php
generating models/User.php
    Unable to write the file '/Users/larryullman/Sites/htdocs/proj
done!
```

**Figure 4.11:** You'll see errors if Yii cannot create files in the **models** directory.

- **User.php**

In Chapter 5, “[Working with Models](#),” you’ll start using and editing the generated code.

## Generating CRUD

With the models created, the next step is to have Gii generate complete CRUD functionality. “CRUD” stands for Create, Retrieve (or Read), Update, and Delete. In other words, everything you’d do with database content. This ability of Yii to write this code for you is a wonderful feature, in my opinion, saving you lots of time and energy.

To start, click the “Crud Generator” link. On the following page (**Figure 4.12**):

1. Enter “Page” as the Model Class.
2. Retain “page” as the Controller ID (Gii will automatically populate this field for you).
3. Click Preview.
4. Click Generate (**Figure 4.13**).

If all went well, that one step will create the controller file for the **Page** model, plus a view directory for its view files, and eight specific view files:

- **\_form.php**

## Crud Generator

This generator generates a controller and views that implement CRUD operations for the specified data model.

Fields with \* are required. Click on the highlighted fields to edit.

<b>Model Class *</b>	Page	Controller ID is case-sensitive. CRUD controllers are often named after the model class name that they are dealing with. Below are some examples:
<b>Controller ID *</b>	page	<ul style="list-style-type: none"> <li>post generates PostController.php</li> <li>postTag generates PostTagController.php</li> <li>admin/user generates admin/UserController.php. If the application has an admin module enabled, it will generate UserController (and other CRUD code) within the module instead.</li> </ul>
<b>Base Controller Class *</b>	Controller	
<b>Code Template *</b>	default (/Users/larryullman/Sites/framework-yii-1.1.13/gii/gene	
<input type="button" value="Preview"/>		

**Figure 4.12:** Generating CRUD functionality for pages.

Code File	Generate <input checked="" type="checkbox"/>
<a href="#">controllers/PageController.php</a>	new <input checked="" type="checkbox"/>
<a href="#">views/page/_form.php</a>	new <input checked="" type="checkbox"/>
<a href="#">views/page/_search.php</a>	new <input checked="" type="checkbox"/>
<a href="#">views/page/_view.php</a>	new <input checked="" type="checkbox"/>
<a href="#">views/page/admin.php</a>	new <input checked="" type="checkbox"/>
<a href="#">views/page/create.php</a>	new <input checked="" type="checkbox"/>
<a href="#">views/page/index.php</a>	new <input checked="" type="checkbox"/>
<a href="#">views/page/update.php</a>	new <input checked="" type="checkbox"/>
<a href="#">views/page/view.php</a>	new <input checked="" type="checkbox"/>

**Figure 4.13:** The preview of the files to be generated for Page CRUD functionality.

- `_search.php`
- `_view.php`
- `admin.php`
- `create.php`
- `index.php`
- `update.php`
- `view.php`

The controller will be explained in detail in Chapter 7, “[Working with Controllers](#).“ The view files will be covered in Chapter 6, “[Working with Views](#).“ For your knowledge now, I’ll explain that the form file is used to both create and update records. The search script is a custom search form. The `_view.php` file is a template for showing an individual record. The admin script creates a tabular listing of the model’s records, with links to CRUD functionality. The index script is really intended for a public listing of the records. The view script is used to show the specifics of an individual record. And the create and update files are wrappers to the form page, with appropriate headings and such.

You’ll also see that the resulting page will offer up a link to go test the generated files.

*{TIP}* If you know you won’t need certain functionality, such as the ability to create a model type, deselect the corresponding checkboxes in the preview table before generating the code.

Again, if you see a permission error, as in Figure 4.11, you’ll need to correct the permissions on the `protected/views` and `protected/controllers` folders, too.

Once those steps works for the `Page` model, repeat the process for `Comment`, `User`, and `File`. You don’t need to create CRUD functionality for the `PageHasFile` class.

*{NOTE}* You will have situations where you’d have a model for a table but not want CRUD functionality, so don’t assume you always take both steps.

And that’s it! You can click “logout”, then click “webapp” to return to the home page. You should then disable Gii by editing the main configuration file.

You can confirm that what you did worked by checking out the new directories and files or by going to a URL. Depending upon whether or not you added “urlManager” to the application’s configuration, the URL would be something like `www.example.com/index.php/user/` or `www.example.com/index.php?r=user`. You will see that there are no records to list yet, and also that you can’t add any new records without logging in (the default is admin/admin). But, to be clear, you won’t want to add any new records until you make some edits anyway, starting in Chapter 5.

{TIP} If you use Yii a lot, and have a few of your own ways of doing things, you may want to look into how you can [customize the Gii generated output](#).

## Chapter 5

# WORKING WITH MODELS

Part 1 of the book, “Getting Started,” introduces the underlying philosophies of the Yii framework and provides an overview of how a Yii-based site is organized and how it functions. You also saw how to create the initial shell of an application, and how to have Yii create a ton of code for you. In Part 2 of the book, you’ll expand that knowledge so that you will understand how to customize the generated code. The combination of generated code and your alterations is how Yii-based sites are created: have `yiic` and Gii create the boilerplate materials, and then edit those files to make the code specific for the application.

The process of learning about the core Yii concepts begins with the three pieces of MVC design: models, views, and controllers. In this chapter, you’ll read about models in greater detail. You’ll learn what the common model methods do, and how to perform standard edits. Many of the examples will assume you’ve already created the CMS database and code explained in Part 1. If you have not already, you might want to do so now.

Also, the focus in this chapter is obviously on models, but there are two types of models you’ll work with: those based upon database tables and those not (the alternative types are normally based on a form). So as to keep the chapter to a reasonable length, and to not overwhelm you with technical details, most of the chapter covers subjects relevant to both model types. A bit of the material will only apply to database-specific models, with much more such material to follow in Chapter 8, “[Working with Databases](#).”

### The Model Classes

By default, model classes in a Yii-based application go within the `protected/models` directory. Each file defines just one model as a class, and each file uses the name of the class it defines, followed by the `.php` extension.

{TIP} You can break a model into one base class and multiple derived classes. This is sometimes necessary for large applications where not all the model's methods are needed everywhere in the site (e.g., when using modules).

In Yii, every model class must inherit directly from the `CModel` class, or, more commonly, from a subclass. Yii defines two subclasses for you: `CActiveRecord` and `CFormModel`. `CActiveRecord` is the basis of models tied to database tables. `CFormModel` is the basis of models *not* tied to database tables, instead tied to HTML forms.

{TIP} Another way of differentiating between the two model types is that `CActiveRecord` models *permanently* store data. Conversely, `CFormModel` models *temporarily* represent data, such as from the time a contact form is submitted to when the contact email is sent, at which point the data is no longer needed.

For example, if you have Yii create your model code and site shell for the CMS example and steps outlined in Chapter 4, “[Initial Customizations and Code Generations](#),” you'll end up with six model classes:

- `ContactForm.php` and `LoginForm.php`, both of which extend `CFormModel`
- `Comment.php`, `File.php`, `Page.php`, and `User.php`, all of which extend `CActiveRecord`

Even though these six classes represent two different types of models—those associated only with an HTML form and those associated with a database table, all models serve the same purposes. First, models store data. Second, models define the business rules for that data. All models are used in essentially the same way, too, as you'll see when you learn more about how controllers use models.

Let's first look at the model classes from an overview perspective and then go into their code in more detail.

The two classes that extend `CFormModel` have this general structure:

```
class ClassName extends CFormModel {  
  
    // Attributes...  
    public $someAttribute;  
  
    // Methods...  
    public function rules() {}  
    public function attributeLabels() {}  
}
```

The classes that extend `CActiveRecord` have this general structure:

```
class ClassName extends CActiveRecord {

    // Methods...
    public function model($className=__CLASS__) {}
    public function tableName() {}
    public function rules() {}
    public function relations() {}
    public function attributeLabels() {}
    public function search() {}

}
```

As you can see, two of the methods—`rules()` and `attributeLabels()`—are common to both model types. Also, as both model types are inherited (indirectly) from `CModel`, other methods are common to both model types but aren't included in these specific model definitions. You'll see some of those later in the chapter.

Further, the `CFormModel` models will always have declared attributes. These are used to temporarily represent the model data. Conversely, `CActiveRecord` models don't initially need explicit attributes, as the data is stored in the database and then loaded into attributes made available on the fly through Active Record. The `CActiveRecord` models also define four other methods that `CFormModel` models do not have. Two of those—`model()` and `tableName()`—are obvious and don't need further illumination. The other two methods—`relations()` and `search()`—will be explained in this and subsequent chapters.

Over the rest of this chapter you'll learn what these methods do, and how you might want to edit them. I'll also explain how you can add your own attributes and methods when needed, just as you can in any class.

*{TIP}* Most of the model's functionality isn't defined in your model class, but rather in a parent class. For example, `CActiveRecord` defines a `save()` method for saving data to the database. Since that functionality is defined already for you (in a parent class), the goal of your specific model should be to tweak and expand that core, inherited functionality when needed.

## Establishing Rules

Perhaps the most important method in your models is `rules()`. This method returns an array of rules by which the model data must abide. Much of your application's security and reliability depends upon this method. In fact, this method alone represents a key benefit of using a framework: built-in data validation. Whether a

new record is being created or an existing one is updated, you won't have to write, replicate, and test the data validation routines: Yii will handle them for you, based upon the rules.

{NOTE} Your database tables will have built-in rules, too, such as requiring a value (i.e., NOT NULL) or restricting a number to being non-negative (i.e., UNSIGNED). However, while those rules protect the integrity of your data, and assist in performance, violating these rules won't necessarily result in error messages end users can see, unlike the Yii model rules.

The `rules()` method, like many methods in Yii, returns an array of data:

```
public function rules() {
    return array(/* actual rules */);
}
```

The `rules()` method needs to return an array whose elements are also arrays. Those arrays use the syntax `array('attributes', 'validator', [other parameters])`.

The attributes are the class attributes (for `CFormModel`) or table column names (for `CActiveRecord`) to which the rule should apply. To apply the same rule to multiple attributes, just separate them by commas as part of one string value.

The validator value is a single string, referring to a built-in Yii validator or one of your own creation (or a third-party's creation). For an easy example, there's the "required" validator:

```
# protected/models/File.php
public function rules() {
    return array (
        array('name', 'type', 'size', 'required')
    ); // End of return statement.
} // End of method.
```

That one rule says that values for the `name`, `type`, and `size` attributes (in this case, table columns) are required.

Some validators take parameters that further dictate the terms. For example, the "length" validator can take a "max" parameter:

```
array('name', 'length', 'max'=>80),
```

That code is from the `File.php` model. You may notice that Yii will have already generated rules like this based upon the underlying table definition: the `name` column in the `file` table is a `VARCHAR(80)`.

(For simplicity sake, and to reduce the amount of code, I'm going to forgo the function definition and `return array()` statement from here on out, for the most part.)

If you want to pass multiple parameters to a validator, you do so as separate arguments:

```
array('age', 'numerical', 'integerOnly'=>true, 'min'=>13, 'max'=>100),
```

(Also note that you don't have to use the `array()` method explicitly in these parameter statements.)

With an understanding of how rules are syntactically defined, let's look at more of the validators, and then get into more custom rules.

## Available Validators

Yii has defined more than a [dozen common validators](#) for you. These are:

- boolean
- captcha
- compare
- date
- default
- email
- exist
- file
- filter
- in
- length
- match
- numerical
- required
- safe
- type
- unique
- unsafe
- url

Each of these names is associated with a defined Yii class that actually performs the validation . If you look at the [Yii class docs](#) for any of them (linked through the [CValidator](#) page), you can find the parameters associated with that validator. The parameters are listed as the class's properties. For example, the “required” class has a `requiredValue` property (**Figure 5.1**).

**requiredValue** property

```
public mixed $requiredValue;
```

the desired value that the attribute must have. If this is null, the validator will validate that the specified attribute does not have null or empty value. If this is set as a value that is not null, the validator will validate that the attribute has a value that is the same as this property value. Defaults to null.

**Figure 5.1:** The details for the “`requiredValue`” property of the `CRequiredValidator` class.

Using that information, you now know that you can set a specific required value when using this rule:

```
array('acceptTerms', 'required', 'requiredValue'=>1),
```

(In case it’s not obvious, that particular bit of code is how you would verify that someone has checked an “acceptance of terms” checkbox, which results in the associated variable having a value of 1.)

Looking at the other validators, the “boolean” validator confirms that the value is Boolean-like. I say “Boolean-like”, because it’s not looking for PHP’s true/false values, but rather 1 or 0. This makes sense if you think about it, as MySQL, for example, stores Booleans as 1 or 0, and HTML doesn’t have true/false Booleans either. The Yii generated code uses the “boolean” validator for the “remember me” option in the `LoginForm` class:

```
array('rememberMe', 'boolean'),
```

The “captcha” validator is used with the `CCaptchaAction` class to implement captcha form validation. I’ll discuss this more in Chapter 12, “[Working with Widgets](#).”

The “compare” validator compares a value against another value and confirms equality. The second value can either be another attribute or an external value. For example, a registration form often has two passwords. The second password, which I might call “`passCompare`” would be represented as an attribute in the model, but not stored in the database table:

```
class User extends CActiveRecord {  
    // Add passCompare as an attribute:  
    public $passCompare;
```

The `rules()` method would then return this array, among others:

```
array('pass', 'compare', 'compareAttribute'=>'passCompare')
```

The “compare” validator’s `strict` property takes a Boolean indicating if a strict comparison is required: both the value and the type must match. This is false, by default. The `operator` property takes the comparison operator you’d like to use: `==`, `!=`, `>`, `<`, `<=`, and `>=`.

The “date” validator confirms that the provided value is a date, time, or datetime. Its `format` property dictates the exact format the value must match, with the default being “MM/dd/yyyy”. As in that string, the format is dictated using special characters, outlined in the Yii docs for the [CDateTimeParser](#) class. The characters are largely what you’d expect; you mostly have to adjust for whether values include leading zeros or not.

The “default” validator is not a true restriction, but rather establishes a default value for an attribute *should one not be provided*. I’ll return to it in a few pages.

The “email” and “url” validators compares the value against proper regular expressions for those syntaxes. You can customize these in a few ways. For example, you can use the `validSchemes` property to list the acceptable URL schemes (“http” and “https” are the defaults).

```
array('email', 'email'),
array('website', 'url'),
```

The “exist” validator is for very specific uses. It confirms that the provided value exists in a table. You’ll normally see this with foreign key-primary key relationships wherein the value provided for a foreign key in Table A must exist as a primary key value in Table B. I’ll show a real-world example of this in a few pages.

The “file” validator is for validating an uploaded file. This is a bit more complex of a process, and so I’ll cover its usage in Chapter 9, “[Working with Forms](#)”.

The “filter” validator isn’t a true validator, but actually a processor through which the data can be run. I’ll explain it in more detail later in this chapter.

The “length” validator is used on strings and confirms that the number of characters is more than, fewer than, or equal to a specific number:

```
array('pass', 'length', 'max'=>20),
```

{TIP} All numbers used for sizes, ranges, and lengths are inclusive.

The minimum and maximum can also be combined to create a range:

```
array('pass', 'length', 'min'=>6, 'max'=>20),
```

To require a string of a specific length, use the `is` property:

```
array('stateAbbr', 'length', 'is'=>2),
```

The “in” validator confirms that a value is within a range or list of values. You can provide the range or list as an array assigned to the `range` attribute:

```
array('stooge', 'in', 'range'=>array('Curly', 'Moe', 'Larry')),  
array('rating', 'in', 'range'=>range(1,10)),
```

The “match” validator tests a value against a regular expression. You assign the specific regular expression to the `pattern` attribute:

```
array('pass', 'match', 'pattern'=>'/^[\w0-9_-]{6,20}$/'),
```

The “numerical” validator confirms that the value is a number: integer or rational. You can further customize this by setting its `integerOnly` property to true, or by assigning `min` and `max` values:

```
php array('age', 'numerical', 'integerOnly'=>true, 'min'=>13,  
'max'=>110),
```

The “required” validator will catch both null values and empty values. You’ve already seen an example of it:

```
array('user_id, name, type, size, date_entered', 'required'),
```

Keep in mind that “required” only insures that the attribute has a value; the other rules more specifically restrict what the value must be. This also means, for example, that applying the “email” validator to an attribute without also applying “required”, means that value *can* be null (or empty), but if it has a value, it must match the email address pattern.

{WARNING} Be sure to also apply “required”, on top of any other rule when the attribute must have a value.

The “safe” and “unsafe” validators are used to flag attributes as being safe to use without any other rules applying, or unsafe to use. For example, the `description` column in the `file` table can have a null value, and its allowed value—any text—doesn’t lend itself to any other validation. But without *any* validation, Yii will consider `description` to be unsafe. On the other hand, an email address is already considered to be “safe” because it must abide by the email rule.

“Unsafe” isn’t just a label, however. When a form is submitted, Yii quickly maps the form data onto corresponding model attributes. This process is called “massive assignment”. But Yii will only perform massive assignment for attributes that are considered to be safe. This means that, without any other validation rules, the `description` value from the form, for example, will not be assigned to the corresponding model attribute, and therefore won’t end up in the database. The fix is to apply the “safe” validator to `description` to force Yii to treat it as safe to massively assign:

```
array('description', 'safe'),
```

All that being said, in the particular case of an optional description, you’d likely want to filter it through PHP’s `strip_tags()` function, as a security measure. In fact, I would generally recommend that you try to apply at least one validator to every attribute and in the rare cases you cannot, that you at least apply a filter. And once you’ve applied the filter, then you no longer need to declare the attribute as safe.

*{NOTE}* Rarely do attributes need to be labelled as “unsafe”.

The “type” validator used to be a catchall, in case another validator didn’t fit the bill, but thanks to the addition of the “date” validator (in Yii 1.1.7), there’s little need for “type” now. The “numeric” validator can catch numbers and “length” can validate a string’s size. If, for whatever reason, you need to validate that a value is just, say, an array (without further validating the array’s values), then “type” would be useful.

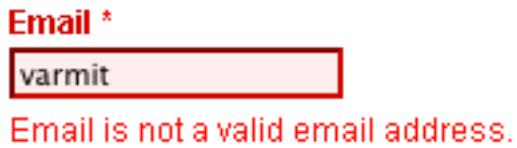
The “unique” validator requires that the value be unique for all corresponding records in the associated database table. You would use this to insure unique email addresses, for example:

```
array('email', 'unique'),
```

And that’s an introduction to all of Yii’s built-in validators. At the end of this section of the chapter, I’ll put this information together within the context of the CMS site to show some practical rules for its models. But first, there’s more to learn about rules...

## Changing Error Messages

As you’ve already seen, many validators take additional parameters, which map to public properties of the underlying validator class. There are also parameters common to every validator, as they all extend the `CValidator` class. One such



**Figure 5.2:** The default error message for an invalid email address.

parameter is “message”. This attribute stores the error message returned when the attribute does not pass a particular validation (**Figure 5.2**).

As with almost everything in Yii, if you don’t like the default response, you can easily change it. To change the default error message for an attribute, assign a new value to the `message` property:

```
array('email', 'email',
    'message'=>'You must provide an email address
    to which you have access.'),
array('pass', 'match', 'pattern'=>'/^[\w\.-]{6,20}$/,
    'message'=>'The password must be between 6 and 20 characters
    long and can only contain letters, numbers, the underscore,
    and the hyphen.'),
```

Within your new `message` value, you can use the special placeholder `{attribute}` to have Yii automatically insert the offending attribute.

```
array('pass', 'match', 'pattern'=>'/^[\w\.-]{6,20}$/,
    'message'=>'The {attribute} must be between 6 and 20 characters
    long and can only contain letters, numbers, the underscore,
    and the hyphen.'),
```

If you also set a `requiredValue` attribute for the item in question, your error message can indicate what the required value is via `{value}`.

A couple of validators have more specific error messages you can customize. The “length” validator has `tooLong` and `tooShort` properties for those specific error messages. Similarly, the “numeric” validator has `tooBig` and `tooSmall`:

```
array('age', 'numerical', 'integerOnly'=>true,
    'min'=>13, 'max'=>110,
    'tooSmall'=>'You must be at least 13 years old to use
    this site.'),
```

## Setting Default Values

As previously mentioned, the “default” validator is not a true validator but is instead used to set default values for an attribute should one not be provided. Default rules

are normally implemented when an attribute should be provided with a value *but not by the user*.

As an appropriate example of this, you could use “default” to set a default user type. The CMS database defines the `user.type` column as `ENUM('public', 'author', 'admin')`. Of course, when a new user registers, the user would not indicate her user type; that’s something only the administrator would set. Now, technically, if an `ENUM` column is set as `NOT NULL`, MySQL will automatically use the first possible value as the default, so you could get away with *not* providing a type value. However, it’s best to be as explicit as you can when programming and not rely upon assumptions about external behavior. (And, of course, you may not be using MySQL.)

A better solution is to assign the `type` property a default value:

```
array('type', 'default', 'value'=>'public')
```

If no value is provided, then “public” will be used. But when a value *is* provided, such as when an administrator updates an account and changes the user’s type in the process, that provided value will be used instead.

{TIP} Another good example of default values is to set date and time values, as demonstrated in the validation scenarios section later in this chapter.

You can also use the `default` validator to set empty values to `NULL`. For example, the `file.description` column can be `NULL`. If no value is provided for that element in the form, then its value will be an empty string when saved in the database. An empty string is not technically the same as `NULL`, and won’t be properly represented in queries that use `IS NULL` conditionals. The solution is to set a default value of `NULL`:

```
array('description', 'default', 'value'=>NULL)
```

## Creating Your Own Validator

Thus far, I’ve only covered the built-in validators, but Yii allows you to create your own, too. The more advanced way to do so is to create a new class that extends `CValidator`. A more simple approach (and more appropriate approach much of the time) is to define a new method in the same model. That model method can then perform the validation. The `LoginForm` class created by the `yiic` script does just that, defining an `authenticate()` method:

```
public function authenticate($attribute,$params) {
    if(!$this->hasErrors()) {
        $this->_identity=new UserIdentity($this->username,
            $this->password);
        if(!$this->_identity->authenticate())
            $this->addError('password',
                'Incorrect username or password.');
    }
}
```

What's going on in that code is a bit complicated for the beginner, but will be explained in Chapter 11, “[User Authentication and Authorization](#).“ You may not want to get bogged down in the particulars of that code, and instead focus on the ability to define your own method as a validator. The method must take two arguments: the attribute being validated and the validation parameters (which should be an array). Once defined, the method is used as the validator name. Here’s that rule from `LoginForm`:

```
array('password', 'authenticate'),
```

As in the `authenticate()` example, all your validation method has to do in order to indicate a problem is add an error to the model instance object (aka `$this`). Yii will use the presences of errors, or lack thereof, as an indicator of whether or not all the validation tests have been passed by the provided data. The `addError()` method takes two arguments: the attribute to which the error applies and an error message.

As another example, the `File` class has a `type` attribute, which corresponds to `file.type` in the database. This column/attribute is meant to store the MIME type of a file: `application/pdf`, `audio/mp4`, `video/ogg`, or `application/msword`. When the file is uploaded, the PHP code can retrieve this value, and these values will be used when PHP sends the file back to the browser (i.e., when the user downloads the file).

A site would normally want to restrict the kinds of files that can be uploaded to certain file types. Older versions of the framework had no built-in validator to do that, so you would have defined your own method for that purpose:

```
# protected/models/File.php
public function validateFileType($attr, $params) {

    // Allow PDFs and Word docs:
    $allowed = array('application/pdf', 'application/msword');

    // Make sure this is an allowed type:
```

```
if (!in_array($this->type, $allowed)) {
    $this->addError('type',
        'You can only upload PDF files or Word docs.');
}
} // End of validateFileType() method.
```

Once defined, the validating method can be applied:

```
public function rules() {
    return array(
        // Other rules.
        array('type', 'validateFileType'),
    );
}
```

{TIP} More current versions of Yii have a validator that can check a file's type, as will be explained in Chapter 9.

## Applying Filters

As already explained, the “filter” validator is not a true validator, but rather a way to run a value through a function. This processing will occur *prior* to any other validation. When you have attributes whose values don’t align with any other validator, I would strongly recommend that you consider filtering that data for extra security. Common examples would be addresses or comments, both of which don’t fit any regular expression but should be sanitized for safe usage. Or, going with the CMS example, the `File` class’s `description` attribute should be stripped of any HTML or PHP code:

```
array('description', 'filter', 'filter' => 'strip_tags')
```

As another example, you can run values through the `trim()` function, if you’d like:

```
array('username, email, pass', 'filter', 'filter' => 'trim')
```

You can even write your own filtering function, if you need something more custom. The function needs to take one argument—the value being filtered—and return a value:

```
public function filterValue($v) {
    // Do whatever to $v.
    return $v;
}
```

## Validation Scenarios

Another thing to learn when it comes to rules are *validation scenarios*. Validation scenarios are a way to restrict when a rule should or shouldn't apply. A scenario is indicated using the syntax '`on`' => '`scenarioName`'. If you want a rule to apply to multiple scenarios, just separate each with a comma.

By default, rules apply under all scenarios. In order to change when a rule applies, you need to know what the possible scenarios are.

{TIP} Instead of using “on” to specify a scenario, you can use “except” to have a rule apply to every scenario but the one(s) indicated.

Simply put, a scenario is a label that describes how a model is currently being used. (Technically, `scenario` is a writable property of the `CModel` class.) The `CActiveRecord` class defines two scenarios for you: `insert` and `update`. This means that when you create a new record in a database table, the model is in the “`insert`” scenario. When you update a record in the database table, the model is in the “`update`” scenario. Scenarios are useful when code in a model needs to take extra, or just different, steps under different conditions.

Take, for example, the `File` class (and `file` database table), which has `date_entered` and `date_updated` attributes (and columns). When a new file record is *created*, the `date_entered` attribute should be set to the current date and time. But this should only happen when a new file record is being created; in all other situations, the `date_entered` should be left alone. To properly address the range of possibilities, a rule can be established to set this attribute's value, but should do so only upon inserts. Similarly, the `File` class's `date_updated` attribute should be set to the current date and time whenever the file is updated, but not when it's first created. Thus, you have two different steps that should occur in two different situations. Scenarios to the rescue!

The specific code to solve this particular problem is:

```
array('date_entered', 'default',
    'value'=>new CDbExpression('NOW()'),
    'on'=>'insert'),
array('date_updated', 'default',
    'value'=>new CDbExpression('NOW()'),
    'on'=>'update'),
```

The `new CDbExpression('NOW()')` bit of code will be explained in Chapter 8. For now, just understand that it says to use the MySQL `NOW()` function for this column's value in the database query.

If you want to take this scenario rule a step further, you can set the “`default`” validator's `setOnEmpty` property to false:

```
array('date_entered', 'default',
      'value'=>new CDbExpression('NOW()'),
      'setOnEmpty'=>false, 'on'=>'insert'),
array('date_updated', 'default',
      'value'=>new CDbExpression('NOW()'),
      'setOnEmpty'=>false, 'on'=>'update')
```

That code says that a value should be set for the given attribute whether it already has a value or not (given the specific scenario, of course). In other words, even if your code sets the `date_entered` value to tomorrow, by disabling `setOnEmpty`, the current date and time will always be used. If, however, you wanted to allow for a user-provided value, only overriding that if one is not provided, you would instead use the first example bit of code (that does not change `setOnEmpty`).

Those are two scenarios built into Active Record, but Yii let's you define your own. For example, a `User` object would be created when a person registers or when he logs in. During the registration process, all of the information is required, and you would often compare the password with a confirmed version of the password. But during the login process, only the email address and password are necessary. To address these different uses, you would create two scenarios: `register` and `login`.

Creating a scenario is easy, although it's done not in the model itself but when an instance of that model is created. To flesh out this specific example, I need to turn to controllers a bit (and two chapters early)...

The registration of a new user would likely be done through the `actionCreate()` method of the `UserController` class, as registration is literally the creation of a new user. That method begins with this line of code:

```
$model=new User;
```

To convert that into a scenario, provide the constructor (the class method that's automatically called when a new object of that class is created) with the name of the scenario:

```
$model=new User('register');
```

Now there is a `register` scenario! Any rules set to apply during the `register` scenario will only be invoked within this one circumstance. You might also create user-related scenarios for changing passwords (where a second password would again be necessary and compared) or for changing other user settings.

*{TIP}* Scenarios can also be set on existing instances using the code  
`$model->scenario = 'value';`.

The Yii generated code already creates a scenario in this manner: `search`. This scenario is used by the `CGridView` widget used in the “admin” view. That page is intended as an administrator’s page for viewing records. The associated rule, again created by Gii, will look like this:

```
array('id, username, email, pass, type, date_entered', 'safe',
    'on'=>'search'),
```

The purpose of that rule is to make certain values safe to use for searching. You would want to remove any attribute listed there that *should not* be a search criteria. In Chapter 12, I’ll cover the search scenario in more detail, and I’ll also discuss the related `search()` method defined by Gii in `CActiveRecord` models.

## Putting It All Together

With all of this information in mind, how do you then go about defining rules for a model? Here’s what you should do:

- **Identify required attributes.** This should be obvious and easy, but think in terms of information *required from the user*. You only establish rules for fields (i.e., models attributes) whose data may be provided by users. You wouldn’t, for example, declare a rule for a primary key field, whose value will be automatically created by the database.
- **Validate the values in the most restrictive way possible.** The required rule ensures that the attribute has a value, but most attributes can be further restricted. Add rules in this order:
  1. *Validate anything you can to a specific value.* It’s not often the case that an attribute must have a specific value, or one of a possible set of values, but if so, check for that. For example, the `type` attribute of `User` can only be “public”, “author”, or “admin”.
  2. *Validate anything else remaining to a strict pattern, if you can.* For example, an email address or a URL must match a (pre-defined) pattern. You might also create patterns for matching usernames, passwords, and so forth.
  3. *Validate anything else remaining to a strict type, if you can.* For example, validate to numbers or numeric types.
  4. *Validate numbers to an appropriate range, if you can.* The easiest and most common check is for a positive value. Ages, quantities, prices, and so forth, must all be greater than 0. Ages, however, would also have a logical maximum, such as 100 or 120.

- **Apply filters as appropriate.** At the very least, you'll likely want to apply filters to any remaining attribute not covered by a validation rule.
- **Be as conservative as you can with safe lists.** If you've thought carefully about the applicable validation rules, there should hopefully be only a rare few attributes that also need to be forcibly marked as safe. Even better, only mark attributes as safe in specific scenarios.
- **Be as conservative as you can with the search list.** Chapter 12 will explain the usage of the search scenario in more detail.
- **Customize descriptive error messages, if needed.** This is more of a user interface issue, but something to also consider.

With all of this in mind, I'll present the rules I would initially set for the CMS site's four models. If you're the kind of person that likes to test yourself, take a crack at customizing the appropriate rules first, before looking at mine. You can check your answers by downloading my code from the [book's download page](#) (on the "Downloads" tab of your account page).

{NOTE} Later in the chapter, you'll learn how to handle the foreign key columns such as `comment.user_id`, `comment.page_id`, etc.

```
# protected/models/Comment.php::rules()
// Required attributes (by the user):
array('comment', 'required'),

// Must be in related tables:
array('user_id', 'page_id', 'exist'),

// Strip tags from the comments:
array('comment', 'filter', 'filter'=>'strip_tags'),

// Set the date_entered to NOW():
array('date_entered', 'default',
    'value'=>new CDbExpression('NOW()'), 'on'=>'insert'),

// The following rule is used by search().
// Please remove those attributes that should not be searched.
array('id', 'user_id', 'page_id', 'comment', 'date_entered',
    'safe', 'on'=>'search'),
```

{NOTE} To save space, I'm only showing the arrays that are returned as part of the primary array in the `rules()` methods.

And here is Page:

```
# protected/models/Page.php::rules()
// Only the title is required from the user:
array('title', 'required'),

// User must exist in the related table:
array('user_id', 'exist'),

// Live needs to be Boolean; default 0:
array('live', 'boolean'),
array('live', 'default', 'value'=>0),

// Title has a max length and strip tags:
array('title', 'length', 'max'=>100),
array('title', 'filter', 'filter'=>'strip_tags'),

// Filter the content to allow for NULL values:
array('content', 'default', 'value'=>NULL),

// Set the date_updated to NOW() every time:
array('date_updated', 'default',
    'value'=>new CDbExpression('NOW()')),

// date_published must be in a format that MySQL likes:
array('date_published', 'date', 'format'=>'yyyy-MM-dd'),

// The following rule is used by search().
// Please remove those attributes that should not be searched.
array('id', 'user_id', 'live', 'title', 'content', 'date_entered',
    'date_published', 'safe', 'on'=>'search'),
```

And here is User:

```
# protected/models/User.php::rules()
// Required fields when registering:
array('username', 'email', 'pass', 'required', 'on'=>'insert'),

// Username must be unique and less than 45 characters:
array('email', 'username', 'unique'),
array('username', 'length', 'max'=>45),

// Email address must also be unique (see above), an email address,
// and less than 60 characters:
array('email', 'email'),
```

```
array('email', 'length', 'max'=>60),  
  
// Password must match a regular expression:  
array('pass', 'match', 'pattern'=>'/^([a-z0-9_-]{6,20})$/i'),  
  
// Password must match the comparison:  
array('pass', 'compare', 'compareAttribute'=>'passCompare',  
      'on'=>'register'),  
  
// Set the type to "public" by default:  
array('type', 'default', 'value'=>'public'),  
  
// Type must also be one of three values:  
array('type', 'in', 'range'=>array('public', 'author', 'admin')),  
  
// Set the date_entered to NOW():  
array('date_entered', 'default',  
      'value'=>new CDbExpression('NOW()'),  
      'on'=>'insert'),  
  
// The following rule is used by search().  
// Please remove those attributes that should not be searched.  
array('id', 'username', 'email', 'pass', 'type', 'date_entered',  
      'safe', 'on'=>'search'),
```

You also have to add one attribute to User:

```
# protected/models/User.php  
class User extends CActiveRecord {  
    public $passCompare; // Needed for registration!  
    // Et cetera
```

{TIP} The “user” validation rules will also change depending upon how you plan on handling logging in (see Chapter 11) and updating user accounts.

And here are the rules from the File model:

```
# protected/models/File.php::rules()  
// name, type, size are required (sort of come from the user)  
array('name', 'type', 'size', 'required'),  
  
// description is optional; must be filtered
```

```
// and set to NULL when empty:  
array('description', 'filter', 'filter'=>'strip_tags'),  
array('description', 'default', 'value'=>NULL),  
  
// Maximum length on the name:  
array('name', 'length', 'max'=>80),  
  
// Type must be of an appropriate kind:  
array('type', 'validateFileType'),  
  
// Set the date_entered to NOW():  
array('date_entered', 'default',  
    'value'=>new CDbExpression('NOW()'), 'on'=>'insert'),  
  
// Set the date_updated to NOW():  
array('date_updated', 'default',  
    'value'=>new CDbExpression('NOW()'), 'on'=>'update'),  
  
// The following rule is used by search().  
// Please remove those attributes that should not be searched.  
array('id', 'user_id', 'name', 'type', 'size', 'description', 'date_entered',  
    'date_updated', 'safe', 'on'=>'search'),
```

Those rules also refer to the “validateFileType” filter, explained earlier in the chapter. Three of the file attributes—its name, type, and size—are not actually provided by the user directly, but come from the file the user uploaded. I’ll explain how to get those into the attributes in Chapter 9.

## Changing Labels

Moving out of the rules, on a much more trivial note, let’s look at the `attributeLabels()` method. This method returns an associative array of fields and the labels the site should use for those fields. The labels will appear in forms, error messages, and so forth. For example, in a form that asks the user for an email address, should that form field say “Email”, “E-mail”, “E-mail Address”, or whatever? Rather than editing the corresponding HTML (in the view file), the MVC approach says to put this knowledge into the model itself. By doing so, editing one file will have the desired effect wherever the field’s label is used.

The Yii framework uses the `attributeLabels()` method for this purpose, and it does a great job of generating reasonable labels for you. For example, given a column name of “date\_updated”, Yii will generate the label “Date Updated”. Foreign key columns, such as “user\_id” become references to the associated class: “User”. Still, you may want to customize these labels more. To do so, just edit the values returned

by `attributeLabels()`. Note that you only want to edit the *values*, not the array indexes.

For example, in the `File.php` model, used to represent an uploaded file, I would change the `attributeLabels()` definition to:

```
public function attributeLabels() {
    return array(
        'id' => 'ID',
        'user_id' => 'Uploaded By',
        'name' => 'File Name',
        'type' => 'File Type',
        'size' => 'File Size',
        'description' => 'Description',
        'date_entered' => 'Date Entered',
        'date_updated' => 'Date Updated',
    );
}
```

After making those edits, you'll see that all of the view files reflect the new changes (**Figure 5.3**).

{NOTE} The file upload (or create) form would actually be much different in the live site, as the file's name, type, and size would come from the uploaded file itself.

(Unless you've made edits to the `LoginForm` model, you can access the page shown in Figure 5.3 by first logging in as admin, and then going to <http://www.example.com/index.php/file/create/>.)

When editing these values, remember that they aren't just relevant on forms such as that in Figure 5.3. For example, you won't have the user provide the `date_entered` value, instead that will be automatically created by the database. That might lead you to think there's no need to have a "Date Entered" label, but that label will be useful on a page that shows the information about an already uploaded file.

Also, when you have a model that's not based upon a database, you'll need to add the attribute names and values to the `attributeLabels()` method yourself. This is also true when you add attributes to a database-based model:

```
# protected/models/User.php::attributeLabels()
return array(
    'id' => 'ID',
    'username' => 'Username',
    'email' => 'Email',
    'pass' => 'Password',
```

## Create File

Fields with \* are required.

Uploaded By

File Name \*

File Type \*

File Size \*

Description

Date Entered

Date Updated

**Figure 5.3:** The form for adding a new file, with its new labels.

```
'type' => 'Type',
'date_entered' => 'Date Entered',
'comparePass' => 'Password Confirmation'
);
```

## Watching for Model Events

Thus far, the chapter has been examining the model methods created by Gii. But there are methods *not* generated for you but still common to Yii models to be discussed. I'm specifically thinking of:

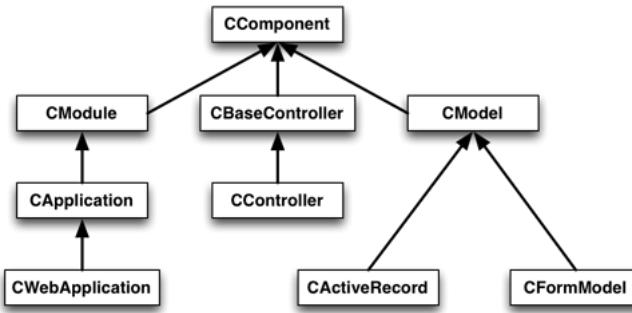
- `afterConstruct()`
- `afterDelete()`
- `afterFind()`
- `afterSave()`
- `afterValidate()`
- `beforeDelete()`
- `beforeFind()`
- `beforeSave()`
- `beforeValidate()`

These methods are used to handle model-related events. Before looking at the usage of these methods, let's first look at event handling in Yii in general.

### The CComponent Class

Something I thought about discussing in Chapter 3, “[A Manual for Your Yii Site](#),” but later changed my mind about, is the concept of *components*. Discussion of components can get a bit complex (which is why I removed it from Chapter 3), but components are an important subject, and it’s time they were introduced to you.

Unlike the *application* components configured in Chapter 4 (such as the database component, the “urlManager” component, and so forth), I’m talking about generic components here. Components are the key building block in the Yii framework. It all starts with Yii’s `CComponent` class. Most of the classes used in Yii are descendants of the base `CComponent` class. For example, the application object will be of type `CWebApplication`. That class is derived from `CComponent` (although there are other classes in between). Controllers are of type `CController`, which inherits from `CBaseController`, which inherits from `CComponent`. `CActiveRecord` inherits from `CModel`, which inherits from `CComponent`, and the same inheritance path apply to `CFormModel` (**Figure 5.4**)



**Figure 5.4:** Part of Yii's class inheritance structure, with **CComponent** at the top.

Knowing that the component is the basic building block is important to your use of Yii. Because of the nature of inheritance in OOP, functionality defined in **CComponent** will be present in every derived class, which is to say most of the classes in the framework.

The **CComponent** class provides three main tools:

- The ability to get and set attributes
- Event handling
- Behaviors

Of these three, I want to discuss events now. This coverage will be specific to models, but understand that any class that inherits from **CComponent** supports events (which is to say most classes).

## Event Handling in Yii

Event programming isn't necessarily familiar territory to PHP developers, as PHP does not have true events the way, say, JavaScript does. In PHP, the only real event is handling the request of a PHP script (through a direct link or a form submission). The result of that event occurrence is that the PHP code in that script is executed. Conversely, in JavaScript, which continues to run so long as the browser window is open, you can have your code watch for, and respond to, all sorts of events (e.g., a form's submission, the movement of the cursor, and so forth). Thanks to the **CComponent** class, Yii adds additional event functionality to PHP-based Web site.

{NOTE} To be perfectly clear, events in Yii still only occur during the execution of a script. Once a complete browser page has been rendered, no other events can occur until another PHP script is requested. Therefore, it may help to think of events in Yii as being similar to the concept of database triggers more so than to events in JavaScript.

Event handling in any language starts by declaring “when this event happens with this thing, call this function”. In Yii, you can create your own events, and I’ll perhaps discuss that at another point in the book (in a chapter to be named later). But models have their own predefined events you can watch for: before a model is saved, after a model is saved, before a model is validated, after a model is validated, and so forth (the available events will depend upon the model type). Yes: each of the methods previously mentioned correspond to an event that Yii will watch for with your models.

In many situations, you’ll want to make use of events when something that happens with an instance of model A should also cause a reaction in model B. You’ll see examples of this in time. But watching for events can be a good way to take some extra steps within a single model, too.

For example, you might want to do something special before a model instance is saved. To do so, just create a `beforeSave()` method within the model:

```
# protected/models/SomeModel.php
protected function beforeSave() {
    // Do whatever.
    return parent::beforeSave();
}
```

As you can see in that minimal example, it’s a best practice to call the parent class’s same event handler (here, `beforeSave()`) just before the end of the method. Doing so allows the parent class’s event handler to also take any actions it needs to, just in case. (If you don’t do this, then any default behavior in the parent class method won’t be executed.)

As a real-world example of using an event with a model, the `page.user_id` value needs to be set to the current user’s ID when a new page record is created. One way to do that is to create a `beforeValidate()` event handler that sets the attribute’s value:

```
# protected/models/Page.php
protected function beforeValidate() {
    if(empty($this->user_id)) { // Set to current user:
        $this->user_id = Yii::app()->user->id;
    }
    return parent::beforeValidate();
}
```

This does assume that the current user’s ID is available through `user->id`, but other than that, it will work fine. And because this method checks for an empty `user_id` attribute first, the event handler will not have an impact on the attribute’s value when a page is being updated.

{TIP} In Chapter 11, you'll learn about `Yii::app()->user->id`.

The same concept can be applied to the `user_id` and `page_id` attributes in `Comment` and the `user_id` attribute in `File`. See the downloadable code for examples of all of these.

As another example, earlier in the chapter you saw how to set the two date/time column values via scenarios:

```
# protected/models/AnyModel.php::rules()
array('date_entered', 'default',
    'value'=>new CDbExpression('NOW()'),
    'on'=>'insert'),
array('date_updated', 'default',
    'value'=>new CDbExpression('NOW()'),
    'on'=>'update'),
```

An alternative solution would be to use the `beforeSave()` method and set the values within it. To test whether this is an insertion of a new record or an update of an existing one, the code can check the `isNewRecord` property of the model:

```
# protected/models/AnyModel.php
public function beforeSave() {
    if ($this->isNewRecord) {
        $this->created = new CDbExpression('NOW()');
    } else {
        $this->modified = new CDbExpression('NOW()');
    }
    return parent::beforeSave();
}
```

Which approach you take for setting values—validation scenarios or events—is largely a matter of preference, as both can do the trick. The argument for using event handling is that you are moving more of the logic out of the rules and into new methods, which can make for cleaner code.

{TIP} To get the automatically incremented primary key value for the new record just created, refer to `$this->primaryKey`. You might need to do this in an `afterSave()` event handler.

As a final note on this concept, if the event that's about to take place *shouldn't* occur—for example, the model should not be saved for some reason, just return false in the event handler method.

## Relating Models

To wrap up this discussion of models, another key model method is `relations()`. This method is used by `CActiveRecord` models to indicate one model's relationship to other models. If your database is designed properly, this method will already be properly filled out, again thanks to Gii.

{TIP} Revisit Figure 4.6 in Chapter 4 if you don't recall the relationships between the various tables in the CMS example.

Here's what the `relations()` method in the `Comment` model looks like, with the Gii-generated code:

```
# protected/models/Comment.php
public function relations() {
    return array(
        'page' => array(self::BELONGS_TO, 'Page', 'page_id'),
        'user' => array(self::BELONGS_TO, 'User', 'user_id'),
    );
}
```

This method returns an array. Each array index is the relation's name. The relation's name, is a made up value, that should be obviously meaningful.

Each value is another array, starting with the relationship type, followed by the related model, followed by the attribute in *this* model that relates to that model (i.e., the foreign key to that model's primary key). The above code indicates that `Comment` belongs to `Page` via the `page_id` attribute. In other words, each comment belongs to a page, and the association is made through `page_id`. The same relationship exists with `User`.

Here's how this will come into play: When loading a record for a `Page`, you can also load any of its related models. In this case, the `Page` instance can load the comments associated with that page. This will allow the page to also display those comments (without you doing any other work). Furthermore, since `Comment` is related to `User`, the user's name can also be loaded and shown. You'll see examples of this in subsequent chapters. But for now, let's look into the model relations in great detail.

{TIP} The relationships are also needed by the "exists" validator which confirms that a foreign key value in this table exists as a primary key in another.

## Relationship Types

The possible relationships are:

- HAS\_ONE
- BELONGS\_TO
- HAS\_MANY
- MANY\_MANY

{TIP} The relationships are indicated by constants defined within the CActiveRecord class. That's why each is prefaced with `self::` when used in the `relations()` method.

You've already seen an example of `BELONGS_TO`. This constant represents the "one" side of a one-to-many relationship (e.g., page "belongs to" user). The other model in that relationship will have a `HAS_MANY` relationship to this one:

```
# protected/models/User.php
public function relations() {
    return array(
        'comments' => array(self::HAS_MANY, 'Comment', 'user_id'),
        'files' => array(self::HAS_MANY, 'File', 'user_id'),
        'pages' => array(self::HAS_MANY, 'Page', 'user_id'),
    );
}
```

As you can see, a `User` can have many comments, files, and pages in the system.

When there is a one-to-one relationship between two models, the relations are `BELONGS_TO` and `HAS_ONE` (instead of `HAS_MANY`). One-to-one relationships in databases aren't that common as one-to-one relations can alternatively be combined into a single table. But, as a hypothetical example, if you had an e-commerce site that used subscriptions to access content, you could opt to store the subscription information separate from the user information. But each user could only have a single subscription and each subscription could only be associated with a single user. Again, this isn't common, but Yii supports that arrangement when it exists.

Note that these relation definitions can be automatically created by Gii based upon one of two things found in your database:

- Foreign key constraints
- Comments used to indicate relationships for tables that don't support foreign key constraints

If Gii doesn't generate this code for you, or if you just need to alter the relations later, you can add the right relation definitions that match the situation.

## Handling Many-to-Many Relationships

Finally, there's the `MANY_MANY` relationship. In a normalized, relational database, a many-to-many relationship between two tables is handled by creating an *intermediary* table. Both of the original two tables will then have a one-to-many relationship with the intermediary. This is the case in the CMS example, with the `page_has_file` table.

When using Gii to create the boilerplate code in Chapter 4, the `page_has_file` table was purposefully *not* modeled, as the PHP code will never need to create an instance of that table's records. (The table only has two columns: `page_id` and `file_id`.) You might think that you would have to model that table and then indicate its relationship to the other two models in order for the models to use the table, but thankfully, Yii supports a different syntax for the common situation of a many-to-many relationship between two models:

```
# protected/models/AnyModel.php::relations()
return array(
    'relationName' => array(self::MANY_MANY, 'Model',
        'intermediary_table(fk1, fk2)')
)
```

Again, you give the relationship a name as the index. The value is an array, with the first value being the type: here, `MANY_MANY`. Next, you name the other model to which this model relates. Next, instead of identifying the foreign key in this model that relates to the other model, you name the intermediary table and the corresponding foreign keys.

Putting this together, here are the corresponding relations for `File`:

```
# protected/models/File.php
public function relations() {
    return array(
        'user' => array(self::BELONGS_TO, 'User', 'user_id'),
        'pages' => array(self::MANY_MANY, 'Page',
            'page_has_file(file_id, page_id)'),
    );
}
```

And here's `Page`:

```
# protected/models/Page.php
public function relations() {
    return array(
        'comments' => array(self::HAS_MANY, 'Comment', 'page_id'),
        'user' => array(self::BELONGS_TO, 'User', 'user_id'),
```

```
    'files' => array(self::MANY_MANY, 'File',
        'page_has_file(page_id, file_id)'),
);
}
```

As the Gii-generated comments also indicate, even though proper relationship definitions were probably created for you, you'll want to inspect them yourself, just to be sure.

*{TIP}* More advanced relationship issues will be covered in Chapter 18, “[Advanced Database Issues](#).”

## Chapter 6

# WORKING WITH VIEWS

Part 2 of the book focuses on the core concepts within the Yii framework. The very core of the core of Yii is the MVC–model, view, controller–design approach. The previous chapter explained *models* in some detail. Models represent the data used by an application. In this chapter, you’ll look at *views* in equal detail. Users interact with applications through the views. For Web sites, this means that views are a combination of HTML and PHP that help to create the desired output that the user will see in her browser. To me, models are complicated in design but easy to use. Conversely, views are simple in design but can be challenging for beginners to get comfortable with because of how they are implemented.

In this chapter, you’ll learn everything you need to know in order to both comprehend and work with views. You’ll also encounter several recipes for performing specific tasks. As in the previous chapter, many of the examples will assume that you’ve created the CMS example explained in Chapter 4, “[Initial Customizations and Code Generations](#).”

Finally, understand that views are *rendered*–loaded and their output sent to the browser–through controllers. Controllers will be covered in the next chapter, but there’s a bit of a “chicken and the egg” issue in discussing the two subjects. As best as I can, I’ll keep explanations in this chapter to the view files themselves, but some discussion of the associated controllers will inevitably sneak in.

### The View Structure

When you use the command-line and Gii tools to create a new Web application, you’ll generate a series of files and folders. By default, all of the view files will go in the **protected/views** directory. This directory is subdivided into a **layouts** directory plus one directory for each *controller* you’ve created. Those directory names match the controller IDs: **comment**, **file**, **page**, **site**, and **user** in the CMS example application.

Within the **layouts** directory, you'll find these three files:

- **column1.php**
- **column2.php**
- **main.php**

For each of the controllers you created via Gii (i.e., all of the directories except for **site**), you'll find these files:

- **\_form.php**
- **\_search.php**
- **\_view.php**
- **admin.php**
- **create.php**
- **index.php**
- **update.php**
- **view.php**

As you can see, view files are designed to be broken down quite atomically, such that, for example, the form used to both create and edit a record is its own file, and that file can be included by both **create.php** and **update.php** (those two files start by changing the headings above the form). As with most things in OOP, implementing atomic, decoupled functionality goes a long way towards improving reusability. But the individual view files are only part of the equation for creating a complete Web page. The individual view files get rendered within a layout file. Although most of your edits will take place within the individual view files, in order to comprehend views in general, you must understand how Yii assembles a page.

## Where Views are Referenced

In Chapter 3, “[A Manual for Your Yii Site](#),” I discuss *routing* in Yii: how the URL requested by the Web browser becomes the generated page. For example, when the user goes to this URL:

**`http://www.example.com/index.php?r=site/index`**

That is a request for the “index” action of the “site” controller. (Because “site” is the default controller and “index” is the default action, the URL **`http://www.example.com/`** would have the same effect.) Behind the scenes, the application object will read in the request, parse out the controller and action, and then invoke the corresponding method accordingly. In this case, that URL has the end result of calling the **actionIndex()** method of the **SiteController** class:

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

The only thing that method does is invoke the `render()` method of the `$this` object, passing it a value of “index”. The `render()` method, defined in the `CController` class, is called any time the site needs to render a view file within the site’s layout. The first argument to the method is the view file to be rendered, without its `.php` extension. By default, the view file will be pulled from the current controller’s view directory: `protected/views/ControllerID/viewName.php`. In this case, “index” means that `protected/views/site/index.php` will be rendered.

{TIP} As `$this` always refers to the current object, within a controller `$this` refers to the current instance of that controller.

That’s where the view file is referenced and how Yii decides it is time to create a Web page. Now let’s look at the rendering process itself.

## Layouts and Views

In order to understand what Yii does behind the scenes to create a complete HTML page, it may help to begin by looking at how templates are used in a *non-framework* PHP site.

### The Premise of Templates

When you begin creating dynamic Web sites using PHP, you’ll quickly recognize that many parts of an HTML page will be repeated throughout the site. At the very least, this includes the opening and closing HTML and BODY tags. But within the BODY, there are normally repeating elements: the header, the navigation, the footer, etc. To create a template system, you would pull all of those common elements out of individual pages and put them into one (or more) separate files. Then each specific page can include these files before and after the page-specific content (**Figure 6.1**):

```
<?php
include('header.html');
// Add page-specific content.
include('footer.html');
```

This is the approach I would use on non-framework-based sites. It’s easy to generate and maintain. If you need to change the header or the footer for the entire site, you only need to edit the one corresponding file.



Figure 6.1: A templated page.

## Templates in Yii

When using Yii for your site, you'll still use a template system, but it's not so simple and direct as that just outlined. In a non-framework site, the executed PHP scripts tend to be accessed directly (i.e., the user goes to `view_page.php` or `add_page.php`). In Yii, everything runs through the bootstrap file, `index.php`. As explained in Chapter 3, the bootstrap file creates an application object and runs it. It's up to that application object to assemble all the necessary pieces together. Of those pieces, the *layout* files constitute all the common elements, everything that's not page-specific.

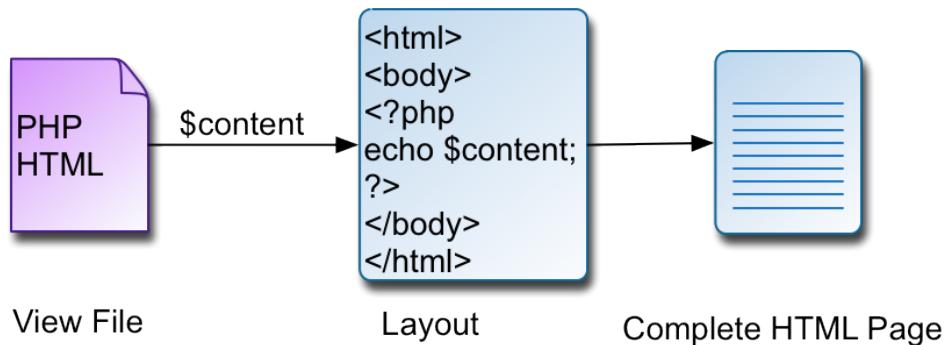
In your Yii-generated site, you'll find the `protected/views/layouts/main.php` file. This is the primary page layout. If you open it, you'll see that it begins with the DOCTYPE and opening HTML tag, then has the HTML HEAD and all its jazz, then this page starts the BODY, and finally the page contains the footer material and the closing tags. This one file acts as both the header and the footer.

{TIP} The main layout file obviously has much more to it, and I'll explain the key pieces throughout this chapter.

In the middle of the body of the page, you'll see this line:

```
<?php echo $content; ?>
```

This is the most important line of code in the entire layout file. Its job is to pull the page-specific content into the template. For example, when the `SiteController` class's `actionIndex()` method is invoked, it renders the "index" view file, which is to say `protected/views/site/index.php`. In that situation, the contents of that view file are assigned to the `$content` variable and then printed at that spot in the layout file. This is what it means to "render" a view file. If the view file has any PHP code, that code will be executed and its results also assigned (inline) to the `$content` variable (**Figure 6.2**).



**Figure 6.2:** How Yii renders a complete page by pulling an individual view file into the layout.

{TIP} “Rendering” just means compiling all the pieces together, including static text (HTML and such) and the output from executed PHP code.

Now you won't find an assignment to the `$content` variable anywhere in your code and do note that it's just `$content`, not `$this->content` or `$model->content`. Yii just uses this variable to identify the rendered view content that will be inserted into the layout. All the other HTML and PHP in the layout is the template for the entire site; the value of `$content` is what makes page X different from page Y.

That's the principle involved, but the process is made more complicated by the various ways that content can be wrapped in Yii. This is where the `column1.php` and `column2.php` files from the `layouts` directory come into play. I'll explain what's happening there later in the chapter, but for now, let's move on to the basics of editing view files.

{TIP} Also pertinent to views is Chapter 12, “[Working with Widgets](#).” Widgets are used to generate more custom HTML (and JavaScript) without embedding too much logic directly in a view file.

## Editing View Files

Now that you (hopefully) have a sense of how individual view files and the primary layout file work together to create a full Web page, let's look at the individual view files in more detail. Over the next several pages, I'll explain

- What variables exist in view files, and how they get there
- How to set the page's title in the browser
- The syntax commonly used in view files
- How to create absolute links to resources
- How to create links to other site pages
- How to prevent Cross-Site Scripting (XSS) attacks

One thing I'm *not* going to explain in any detail in this chapter is the use of forms within view files. Forms are a specific enough topic that they get their own coverage in Chapter 9, “[Working with Forms](#).”

### Variables in the View

Views use a combination of PHP and HTML to create a complete page, just as in many non-framework PHP pages. But it's not often that a view will contain *only* HTML; most of the dynamic functionality comes from the values of variables. A common point of confusion, however, is how variables get to the view file in the first place.

In a traditional, non-framework PHP script, PHP and HTML are intermixed, making it easy to reference variables:

```
<?php  
$var = 23;  
?  
<!-- HTML -->  
<?php echo $var; ?>  
<!-- HTML -->
```

Even if you use included files, you won't have problems accessing variables (outside of functions, that is), as the included code has the same scope as the page that included it.

But in a Yii-based site, the structure and sequence is not so straightforward, as already explained. Further, you will rarely create variables in the view files themselves. No, most of the variables used in view files will come from the controller that rendered the view. This is not that direct either, however. For example, say you change the `actionIndex()` method of the `SiteController` class to:

```
public function actionIndex() {
    $num = 23;
    $this->render('index');
}
```

You *might* think that the `protected/views/site/index.php` script could then make use of `$num`, but that is not the case. Variables must be passed to the view deliberately. To do so, pass an array as the second argument to the `render()` method:

```
$this->render('index', array('num' => $num));
```

Now, `index.php` can make use of `$num`, which will have a value of 23. Note that you can use any valid variable name for the array index, and the resulting variable will exist in the view file. This is equally acceptable, albeit confusing:

```
$this->render('index', array('that' => $num));
```

Now, `index.php` has a `$that` variable with a value of 23.

An important exception to the rule that variables must be formally passed to the view file is `$this`. `$this` is a special variable in OOP that always refers to the current object. It never needs to be formally declared. As a view file gets rendered by a controller, `$this` in a view file always refers to the current controller. In `protected/views/site/index.php`, `$this` refers to the current instance of the `SiteController` class.

The view files generated by Gii have comments at the top of them that indicate the variables that were passed to the view file. For `protected/views/site/index.php`, that's:

```
<?php
/* @var $this SiteController */
```

For `protected/views/user/create.php`, you'll see:

```
<?php
/* @var $this UserController */
/* @var $model User */
```

This is a simple but brilliant touch that makes it easier to know what variables you can work with within a view.

{TIP} Follow the Yii framework's lead and add, or modify, the comments at the top of the view file when you change what variables are passed to it.

As in that user example, the relevant model instance will normally be passed to the view file, too. The code Gii generates passes the model instance as the `$model` variable. Here's the relevant parts of the `actionCreate()` method from `UserController`:

```
public function actionCreate(){
    $model=new User;
    $this->render('create',array(
        'model'=>$model,
    ));
}
```

This means that within the view, you can access any of the model's public properties via `$model->propertyName`. With the `User` class defined in the CMS example, you could therefore greet the user by username in a view using:

```
<?php echo $model->username; ?>
```

{WARNING} That code will work, but later in the chapter you'll learn a slightly more secure approach.

You can also call any of a model's public methods via `$model->methodName()`. For example, the `CModel` class defines the `getAttributeLabel()` method which returns the label (defined in the model's `attributeLabels()` method) for the provided attribute:

```
<?php echo $model->getAttributeLabel('user_id'); ?>
```

## Setting Page Titles

As previously mentioned, within a view file, the `$this` variable refers to an instance of the controller that rendered the file. As `$this` will be an object, you can access any `public controller property` via `$this->controllerProperty`. There aren't that many of them, and certainly few you would *need* to access within a view, but `pageTitle` is useful. Its value will be placed between the `TITLE` tags in the HTML, and therefore used as the browser window title.

```
<?php $this->pageTitle = 'About This Site'; ?>
```

Or, in the CMS example, you might want the browser window title to match the title of the content for a single page. That would have been passed to the page as `$model`:

```
<?php $this->pageTitle = $model->title; ?>
```

By default, the `pageTitle` value will be the application's name (defined in the config file) plus something about the current page. For example, for the `protected/views/user/create.php` page, the title will end up being *My Web Application - Create User* unless changed.

If you want to use the application's name in the title, it's available via `Yii::app()->name`. This comes from the configuration file, explained in Chapter 4:

```
<?php $this->pageTitle = Yii::app()->name . ' :: ' . $model->title; ?>
```

{TIP} Because the page title is set by assigning a value to the controller instance, it can also be set within the controller action, if you'd rather.

## Alternative PHP Syntax

The view files are just PHP scripts, and so you can write PHP code in them as if they were any other type of PHP script. While you *could* do that, view files are also part of the MVC paradigm, which has its own implications. Specifically, the emphasis in view files should be on the output, the HTML. Towards that end, the PHP code written in views is embedded more so than in non-MVC sites. For example, a `foreach` loop in non-MVC code might be written as so:

```
<?php
foreach ($list as $value) {
    echo "<li>$value</li>";
}
?>
```

In Yii, that same code would be normally written as:

```
<?php foreach ($list as $value): ?>
<li><?php echo $value; ?></li>
<?php endforeach; ?>
```

In this particular example, with so little HTML, using three separate PHP blocks may seem ridiculous, but the important thing to focus on is the alternative `foreach` syntax. Instead of using curly brackets, a colon begins the body of the loop and the `endforeach;` closes it.

The same approach can be taken with conditionals:

```
<?php if(true): ?>
<div><h2>True!</h2>
<p>Hey! This is true.</p>
</div>
<?php else: ?>
<div><h2>False!</h2>
<p>Hey! This was not true.</p>
</div>
<?php endif; ?>
```

Naturally, the `else` clause is optional.

Again, these are just syntactical differences, common in MVC, but not required. Yii uses its own template system by default, but allows you to [use alternative systems](#), if you'd rather. For example, you can use [Smarty](#).

## Linking to Resources

If you look at the `protected/views/layouts/main.php` file, you'll see that the CSS files for the site are stored where you'd expect them to be, within a `css` subdirectory of the Web root. However, the layout file does not use a relative path to the CSS scripts:

```
<!-- NOT THIS! -->
<link rel="stylesheet" type="text/css"
      href="css/screen.css" media="screen, projection" />
```

While you may be in the habit of using relative URLs for CSS, JavaScript, and other resources on your sites, relative URLs are not a good idea when using Yii. The reason has nothing to do with Yii and everything to do with how Web servers and browsers work. Say you change how URLs are formatted in Yii (see Chapter 4), so that the user might end up at `http://www.example.com/index.php/site/login`. Or better yet: `http://www.example.com/site/login`. In both of those cases, the request to load the CSS file using `href="css/screen.css"` means that the browser will request the file `http://www.example.com/site/login/css/screen.css`. That file, of course, does not exist.

The solution is to use an *absolute* path to all CSS, JavaScript, images, and so forth. This *could* be as simple as:

```
<!-- NOT THIS! -->
<link rel="stylesheet" type="text/css"
      href="/css/screen.css" media="screen, projection" />
```

The initial slash before `css/screen.css` says to start in the Web root directory.

That will work, but it leaves you open to another problem. Like me, you may develop a site on one server and then deploy it to the live server. On the development server, the URL may be something like `http://localhost/sitename/`. In that case, the proper absolute path would be `/sitename/css/screen.css`. If you used that value on your local server, you would have to change it when the site is deployed to the production server.

Rather than having to double check all your references when you move the site, and to generally make your site much more flexible, just have the Yii application insert the proper absolute path for you. That value can be found in `Yii::app()>request->baseUrl`:

```
<link rel="stylesheet" type="text/css"
      href="<?php echo Yii::app()->request->baseUrl; ?>/css/
main.css" />
```

Note that the `Yii::app()->request->baseUrl` value does not end with a slash, so you must add that.

{TIP} Always use absolute paths for external resources.

Understand that you should *not* use this approach to create links to other pages within your site. Only use `Yii::app()->request->baseUrl` to reference resources that should not be loaded through the bootstrap (`index.php`) file. For links to other pages, there are better solutions.

## Linking to Pages

As you should well know by now, every page within a Yii site goes through the bootstrap file. The URL for site pages will be in one of the following formats, depending upon how the “urlManager” component is configured:

- `http://www.example.com/index.php?r=ControllerID/ActionID`
- `http://www.example.com/index.php/ControllerID/ActionID/`
- `http://www.example.com/ControllerID/ActionID/`

Because the URL format is dictated by the “urlManager”, and because you may need to change this format later, you don’t want to hardcode links to other pages within your views. Instead, have Yii create the entire correct URL for you.

The right tool for this job is the `link()` method of the `CHtml` class (**Figure 6.3**). This class defines oodles of helpful methods for you, although most are related to creating HTML forms.

**link() method**

<pre>public static string link(string \$text, mixed \$url='#', array \$htmlOptions=array ())</pre>		
<b>\$text</b>	string	link body. It will NOT be HTML-encoded. Therefore you can pass in HTML code such as an image tag.
<b>\$url</b>	mixed	a URL or an action route that can be used to create a URL. See <a href="#">normalizeUrl</a> for more details about how to specify this parameter.
<b>\$htmlOptions</b>	array	additional HTML attributes. Besides normal HTML attributes, a few special attributes are also recognized (see <a href="#">clientChange</a> and <a href="#">tag</a> for more details.)
<b>{return}</b>	string	the generated hyperlink

**Figure 6.3:** The class documentation for the `CHtml::link()` method.

As you can see in the figure, the first argument is the text or HTML that should be linked. This can be straight text, such as “Home Page”, or HTML. This means that you can use `link()` to turn an image into a link.

The second argument is the URL to use. You could provide a hardcoded value here, but that again defeats the purpose of having flexible links. Instead, you should provide the proper *route*. This must be provided as an array, even if it’s an array of one argument. For example, the route for the home page, which by default is the “index” action of the “site” controller would be “site/index”:

```
<?php echo CHtml::link('Home', array('site/index')); ?>
```

The route to the page for creating a user (i.e., registration), would be “user/create”:

```
<?php echo CHtml::link('Register', array('user/create')); ?>
```

*{NOTE}* Routes are in the format *ControllerID/ActionID*.

Understand that this only works if the route is provided as an array. If you provide a string as the second argument to `CHtml::link()`, it will be treated as a literal string URL value:

```
<?php
// This result is ALWAYS site/index:
echo CHtml::link('Home', 'site/index'); ?>
```

That URL *may* work, depending upon how the “urlManager” is configured, but will break if you ever change the routing configuration.

If you need to pass additional parameters to the routing, just add those to the array. This next bit of code creates a link to the page that has an ID value of 23:

```
<?php echo CHtml::link('Something', array('page/view', 'id' => 23)); ?>
```

The resulting output will be one of the following, depending upon the configuration:

```
<a href="/index.php?r=pageview&id=23">Something</a>
<a href="/index.php/page/view/id/23">Something</a>
<a href="/page/view/id/23">Something</a>
```

As already stated, the value being linked (i.e., that which the user would click upon) can be HTML, too:

```
<?php
echo CHtml::link('', array('page/view', 'id' => 23)); ?>
```

The third parameter to `link()` is for additional HTML options. You could use this, for example, to set the link's class:

```
<?php echo CHtml::link('Something', array('page/view', 'id' => 23),
    array('class' => 'btn btn-info')); ?>
```

Results in:

```
<a href="/index.php/page/view/id/23"
    class="btn btn-info">Something</a>
```

{TIP} The `CHtml::link()` method takes additional HTML options not tied to HTML attributes. These allow you to associate JavaScript events with a link, and will be discussed in Chapter 14, “[JavaScript and jQuery](#).”

Sometimes you'll need to create a URL for a page without creating the entire HTML link code. For example, you may want to use the link's URL for the link text, or just include a URL in some other text or the body of an email. In those cases, you wouldn't want to use `CHtml::link()`, nor would you want to use `Yii::app()->request->baseUrl` (which does handle routing). The trick in such cases is to use the `CController::createAbsoluteUrl()` method. It's available via `$this`, of course:

```
<?php
$url = $this->createAbsoluteUrl('page/view', array('id' => 23));
echo CHtml::link($url, array('page/view', 'id' => 23)); ?>
```

As you can see in that code, `createAbsoluteUrl()` takes the route as a *string*, not an array, as its first argument. Additional parameters can be provided as an array to the second argument. (I've spread the entire code out over two lines for extra clarity, but that's not required.)

*{TIP}* To link to an anchor point on a page, pass '`#`' => '`anchorId`' as a parameter.

## Preventing XSS Attacks

A few pages ago, I demonstrated a line of code but mentioned that the code could be implemented more securely:

```
<?php echo $model->comment; ?>
```

That may seem harmless, but if a malicious user entered HTML in a comment, that HTML would be added to the page (assuming that tags weren't stripped out prior to storing the value). If the HTML included the `SCRIPT` tags, the associated JavaScript would be executed when this page was loaded. That is the premise behind Cross-Site Scripting (XSS) attacks: JavaScript (or other code) is injected into Site A so that valuable information about Site A's users will be passed to Site B.

*{NOTE}* Reprinting user-provided HTML tags on a page is not only a security concern, but it can also mess up the appearance and functionality of the page.

Fortunately, XSS attacks are ridiculously easy to prevent. In straight PHP, you would send data through the `htmlspecialchars()` function, which converts special characters into their corresponding HTML entities. In Yii, you can use `CHtml::encode()` to perform the same role (it's just a wrapper on `htmlspecialchars()`). You'll see this method used liberally (and appropriately) in the view files generated by Gii:

```
<b><?php echo CHtml::encode($data->getAttributeLabel('id'));  
?>:</b>  
<?php echo CHtml::link(CHtml::encode($data->id),  
array('view', 'id'=>$data->id)); ?>
```

As a rule of thumb, any value that comes from an external source that will be added to the page's HTML (including the HEAD), should be run through `CHtml::encode()`. “External source” includes: files, sessions, cookies, passed in URLs, provided by forms, databases, Web services, and probably two or three other things I didn't think of.

{WARNING} You should use `encode()` when dynamically printing the page's title, too.

`CHtml::encode()`, or `htmlspecialchars()`, is fine, but it's not an ideal solution in all situations. For example, in the CMS site, each page has a `content` attribute that stores the page's content. This content will be HTML, so you can't apply either of these methods to it. Obviously, pages of content should only be created by trusted administrators, but you can still make the content safe to display without being vulnerable to XSS attacks. That solution is to use the `ChtmlPurifier` class, which is a wrapper to the [HTML Purifier](#) library:

```
<?php
$purifier = new ChtmlPurifier();
$data = $purifier->purify($page->content);
echo $data;
?>
```

HTML Purifier is able to strip out malicious code while retaining useful, safe code. This is a big improvement over the blanket approach of `htmlspecialchars()`. Moreover, HTML Purifier will also ensure that the HTML is standards compliant, which is a great, added bonus (particularly when non-Web developers end up submitting HTML content).

The biggest downside to using `ChtmlPurifier` is performance: it's slow and tedious for it to correctly do everything it does. For that reason, I would recommend using `ChtmlPurifier` to process data before it's saved to the database. In other words, you'd add its invocation as a `beforeSave()` method of the model instead of putting it into your view. Alternatively, you could use fragment caching to cache just the HTML Purifier output.

{TIP} `ChtmlPurifier` can be customized as to what tags and values are allowed (i.e., considered to be safe).

## Working with Layouts

As explained earlier in the chapter, complete HTML pages are created in Yii by compiling together a view file within the layout file. The past several pages have focused on the individual view files: the variables that are accessible in them, the alternative PHP syntax used within view files, and how to properly and securely perform common tasks. Now let's look at layouts in detail.

As a reminder, the layout is the general template used by a page. It's a wrapper around an individual view file. And, to be precise, the result of an individual view file will be inserted into the layout as the `$content` variable.

{TIP} You can also change the entire look of a site using themes. I've never personally felt the need to use them, but if you're curious, the Yii guide [explains them well](#).

## Creating Layouts

Although the default template that the generated Yii site has is fine, you'll likely want to create your own, more custom site. By this point, you should have the knowledge to do that, but here's the sequence I would take:

1. Create a new file in the **protected/views/layouts** directory.

I would recommend creating a new file, named whatever you think is logical, and leaving the **main.php** file untouched, for future reference.

2. Drop in your HTML code:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>untitled</title>
    <!--[if lt IE 9]>
        <script src="js/html5.js"></script>
    <![endif]-->
</head>
<body>
</body>
</html>
```

That is a basic HTML5 template, and it includes a locally hosted copy of the [html5shiv](#).

3. As the very first line of the page, add:

```
<?php /* @var $this Controller */ ?>
```

This is the kind of documentation that Gii puts into the default layout for you. You really ought to put it in your layout file as a reminder of what variables are available to this page.

4. Replace the page's title with:

```
<?php echo CHtml::encode($this->pageTitle); ?>
```

This inserts the value of the controller's `pageTitle` attribute between the HTML TITLE tags. For extra security, it's sent through the `encode()` method first.

5. Include your CSS files using:

```
<link rel="stylesheet" href="<?php echo  
Yii::app()->request->baseUrl; ?>/css/styles.css">
```

This method for providing an absolute reference to a CSS script was explained earlier in the chapter. Obviously you'll need to change the specific filename to match your site.

6. Repeat Step 5 for any JavaScript:

```
<script src="<?php echo  
Yii::app()->request->baseUrl; ?>/js/scripts.js"></script>
```

7. In the proper location between the BODY tags, print the content:

```
<?php echo $content; ?>
```

{WARNING} Never apply `encode()` when printing the `$content` variable! It will contain HTML that must be treated as such.

8. Make any other necessary changes to implement your template.

9. Save the file.

Once you've created the layout file, you can tell your site to use it.

## Changing Layouts

To change layouts in Yii, assign a new value to the `layout` property of the controller. That's really simple to do, but there are several places you can take that step.

To broadly change the layout used for every page of your site, edit the `protected/-components/Controller.php` file:

```
class Controller extends CController {  
    public $layout='//layouts/your-layout';
```

The **Controller** class is created when you make a new site, and all new controllers created by Gii will extend it. Thus, changing the property here impacts every controller. Note the syntax used: *//layouts/your-layout*. The “//” indicates to start in the default **views** directory. Then, “layouts” means go into the **layouts** directory, and “your-layout” says to use the file named **your-layout.php**. Change this last value to match the filename of your layout file.

If different controllers are going to use different layouts, you can still set a default layout in **protected/components/Controller.php**, but override that value in individual controllers:

```
# protected/controllers/UserController.php  
class UserController extends Controller {  
    public $layout='//layouts/your-other-layout';
```

Now this one controller will use **your-other-layout.php**.

You can also change the layout for specific controller actions (and therefore different view files):

```
# protected/controllers/SiteController.php  
public function actionIndex() {  
    $this->layout = '//layouts/home';  
    $this->render('index');
```

And that's all there is to it. When **protected/views/site/index.php** is rendered, it will use the **protected/views/layouts/home.php** template.

There is one more way in which you can change the default layout of the entire site: by assigning a value to “layout” in the configuration file (i.e., assign a new value to the **layout** property of **CWebApplication**). The following table shows all the options, in order from having the biggest impact to the smallest.

Location	Applies to
config/main.php	Every controller and view
components/Controller.php	Every controller that inherits from <b>Controller</b>
controllers/SomeController.php	Every view in <b>SomeController</b>
SomeController::actionSomething()	The view rendered by <b>actionSomething()</b>

Also note that layout changes made by code lower in the table override values established higher in the table. For example, a layout change in **SomeController.php**

overrides the value set in **Controller.php** or the configuration file.

## Rendering Views Without the Layout

Sometimes you want to render view files *without* using a layout. Two logical reasons to do so are when:

- One view file is being rendered as part of another
- A view file's output won't be HTML

For an example of the first situation, take a peek at any of the “create” view files. You'll see something like this:

```
<h1>Create User</h1>
<?php echo $this->renderPartial('_form', array('model'=>$model)); ?>
```

Both the “create” and the “update” processes make use of the same form, the latter is just pre-populated with the existing data. Since multiple files will use this same form, the logical approach is to create the form as a separate file and then include it wherever it's needed. That's what's happening in **protected/views/user/create.php** above. However, if both the **create.php** and **\_form.php** view files were rendered within the template, the template would be doubled up and the result would be a huge mess.

Yii is prepared for such situations and provides the **renderPartial()** for them. Just like the **render()** method, the first argument should be the name of the view file (without an extension) to be rendered, and the second argument can be used to pass along variables.

*{TIP}* Most of the time you use **renderPartial()** within one view file, you'll want to pass along the variables it received to the other view file.

The second common need to render a view file *without* the layout (i.e., to use **renderPartial()**) is because the view file is not outputting HTML. That would be the case if you were creating a Web service that were to output plain text, XML, or JSON data. Remember that views are not just for Web pages, but for any interface with the site. That interface could be a Web service accessed by client-side JavaScript.

## Rendering Views From Other Controllers

As mentioned already, the file being rendered comes from the directory associated with the current controller. For example, when updating a post record, the URL

is something like `http://www.example.com/index.php/post/update/id/23`. This calls the `actionUpdate()` method of the `PostController` class. That method renders the “update” view, which is to say `protected/views/post/update.php`. But there are cases where you’ll need to render view files from other subdirectories.

For example, say you’ve created an `EmailController` class that defines the formatting for all the emails sent by the site. When a purchase is completed, which would be an action of the `ShoppingCartController` class, the method may want to send the confirmation email (arguably, the `EmailController` could do that, too, but I’m demonstrating a concept here). To generate the body of the email, the `ShoppingCartController` class method can render the `EmailController` class’s view file using:

```
// Use a simpler layout:  
$this->layout = '/layouts/email';  
// Render the email/purchase.php view:  
$this->render('//email/purchase', array('order' => $order));
```

That code is close to what’s required, but will actually render the output in the browser. Instead, the code should *return* the rendered result so that it can be used in an email. That’s possible if you provide the `render()` method with a third argument of true:

```
// Use a simpler layout:  
$this->layout = '/layouts/email';  
// Render the email/purchase.php view:  
$body = $this->render('//email/purchase',  
    array('order' => $order), true);  
// Use $body in an email!
```

## Alternative Content Presentation

There are a couple more ways that you might present content to the user that merit discussion in this chapter. In fact, one of these topics will go a long way towards helping you to understand what’s going on in the layout system implemented by the `yiic` command.

### Using Content Decorators

*Content decorators* are a less heralded but interesting feature of Yii. Content decorators allow you to hijack the view rendering process and add some additional stuff around the view file being rendered. It works by invoking the controller’s `beginContent()` and `endContent()` methods. The `beginContent()` method takes

as an argument the view file into which *this* content should be inserted (i.e., treat this output as the value of \$content in that view):

```
<?php $this->beginContent('//directory/file.php'); ?>
// Content.
<?php $this->endContent(); ?>
```

This feature is best used for handling more complex embedded or nested layouts, although it can be used in other ways, too (there's an interesting example in [Alexander Makarov's book](#)).

For example, say you wanted some pages in your site to use the full page width for the content (**Figure 6.4**).



**Figure 6.4:** The page-specific content goes across the entire width of the browser.

But some pages should have a sidebar (**Figure 6.5**).



**Figure 6.5:** The page-specific content shares the browser width with a sidebar.

You *could* create two different layout files and use each accordingly. However, most of the common elements would then be unnecessarily repeated in two separate files, making maintenance a bit harder. The solution is to *decorate* the page specific

content with the sidebar on those pages that ought to have it. Rather than showing you new code that illustrates this point, I'll recommend that you turn to the files generated by `yiic`, which does this very same thing.

More recent versions of Yii will automatically create three layout files when you create a new Web app:

- `views/layouts/column1.php`
- `views/layouts/column2.php`
- `views/layouts/main.php`

Although all three are layout files, you don't have three different template files. The `main.php` file still creates the DOCTYPE and HTML and HEAD and so forth. The `column1.php` and `column2.php` files are decorators that create variations on how the page-specific content gets rendered. Here is the entirety of `column1.php`:

```
<?php $this->beginContent('//layouts/main'); ?>
<div id="content"><?php echo $content; ?></div>
<?php $this->endContent(); ?>
```

Again, you have the magic `echo $content` line there, but all `column1.php` does is wrap the page-specific content in a DIV.

The `column2.php` file starts off the same, but adds another DIV (which includes some widgets) before `$this->endContent()`:

```
<?php $this->beginContent('//layouts/main'); ?>
<div class="span-19">
<div id="content"><?php echo $content; ?></div>
<!-- content --></div>
<div class="span-5 last">
<div id="sidebar">
<?php $this->beginWidget('zii.widgets.CPortlet',
    array('title'=>'Operations'));
$this->widget('zii.widgets.CMenu', array(
    'items'=>$this->menu,
    'htmlOptions'=>array('class'=>'operations'),
));
$this->endWidget();
?></div>
<!-- sidebar --></div>
<?php $this->endContent(); ?>
```

To be absolutely clear on what's happening, the `protected/components/Controller.php` class sets `column1.php` as the default layout file. When, say, the

`actionIndex()` method of the `SiteController` class is invoked, it renders the `protected/views/site/index.php` file. The rendered result from `index.php` will be passed to the layout file, `column1.php`. This means the `$content` variable in `column1.php` represents the rendered result from `index.php`.

Then, because `column1.php` uses `beginContent()` the rendered result from `column1.php` will be passed to the `main.php` layout file (because it's provided as an argument to `beginContent()`). This means the `$content` variable in `main.php` represents the rendered result from `column1.php`.

Personally I think this default layout approach is a bit complicated for the Yii newbie (yiibie?), but it is invaluable in situations where the content around the page-specific content needs to be adjusted dynamically.

## Using Clips

Somewhat similar to decorators are *clips*. Whereas a decorator provides a way to wrap one piece of content within other content, a clip provides a mechanism for dynamically defining some content that can be used as needed later. In short, it's an alternative to using `renderPartial()` with a hardcoded view file. Clips can be more dynamic than view files, and provide a way to take potentially complex logic out of views.

To create a clip, wrap the content within `beginClip()` and `endClip()` calls. Provide a unique clip identifier to the former method invocation:

```
# protected/controllers/SomeController.php
public function actionSomething() {
    $this->beginClip('stockQuote');
    echo 'AAPL: $533.25';
    $this->endClip();
    $this->render('something');
}
```

Anything outputted between the `beginClip()` and `endClip()` method calls will be stored in the controller under that clip identifier. Again, to be clear: that `echo` statement won't actually send the data to the browser. (This is similar to output buffering in PHP.)

Presumably, the clip content would be more dynamic than that, such as performing a Web service call to fetch the actual stock price. Note that the clip does not need to be passed to the view like other variables, because it's defined as part of the controller.

To use a clip in a view file, first check that it exists, and then print it:

```
<?php
# protected/views/some/something.php
if (isset($this->clips['stockQuote'])) {
    echo $this->clips['stockQuote'];
}
```

## Chapter 7

# WORKING WITH CONTROLLERS

The third main piece of the MVC design approach is the *controller*. The controller acts as the agent, the intermediary that handles user and other actions. The previous chapter mentioned controllers as they pertain to views, and in this chapter, you'll learn all the other fundamental aspects of this MVC component.

### Controller Basics

To best understand the role that controllers play, it may help to think about what a site may be required to do. Say you have an e-commerce site: you may have created a model named *Product*, which can represent each product being sold. There are several actions that can be taken with products:

- Creating one
- Updating one
- Deleting one
- Showing one
- Showing multiple

The first three of these are actions that would only be taken by an administrator. The last two are how customers would interface with products on the site (i.e., maybe first seeing all the products in a category, then viewing a particular one). Furthermore, the presentation for a single product or multiple products will likely be different for the customer than it would be for the administrator. So you have many different uses of the same model within the site.

{NOTE} Understand that I'm just focusing on the *product* aspect of an e-commerce site here. The act of adding an item to a shopping cart would fall under the purview of a `ShoppingCart` class.

In the MVC approach, the way to address the complexity of doing many different things with one model type is to create a controller that will handle all the interactions with an associated model. Hence, the `Product` model has a pal in the `ProductController` controller. The controller is implemented as a class.

For the controller to know which step it's taking (e.g., updating a product vs. showing multiple products), one method is defined for each possibility. One method of the controller fetches a single product whereas another method fetches *all the products* and another is called when a product is created. In Yii, as already mentioned, these methods all begin with the word *action*.

The code created by Gii's scaffolding tool defines these methods for you:

- `actionCreate()`, for creating new model records
- `actionIndex()`, for listing every model record
- `actionView()`, for listing a single model record
- `actionUpdate()`, for updating a single model record
- `actionDelete()`, for deleting a single model record
- `actionAdmin()`, for showing every model record in a format designed for administrators

The generated code defines a few other methods:

- `filters()`
- `accessRules()`
- `loadModel()`
- `performAjaxValidation()`

As you saw in Chapter 4, “[Initial Customizations and Code Generations](#),” the generated code can already handle all of the CRUD functionality and even implements basic security that prevents anyone but an administrator from creating, updating, and deleting records. This is a wonderful start to any Web site, and one of the reasons I like Yii. But there’s much more to know and do with controllers.

In this chapter, I’m going to explain three of these methods and, more importantly, the valuable relationship that the controller has with the model and the views. You’ll also learn a few new tricks, such as how to create static pages and how to define more elaborate routing possibilities.

*{TIP}* Chapter 14, “[JavaScript and jQuery](#),” will explain the `performAjaxValidation()` method.

Chapter 11, “[User Authentication and Authorization](#),” is also germane to controllers, but more advanced. After reading this chapter, you may want to also consider reading [Alex Markarov’s book](#), which has oodles of controller tips and tricks (among other goodies).

## Gii Generated Controllers

All controllers in a Yii based site must extend the `CController` class (which, in turn, extends `CBaseController`, which extends `CComponent`). The controllers automatically generated by the `yiic` command line script and the Gii module add another layer to this hierarchy.

First, the `yiic` command creates the `protected/components/Controller.php` class that extends `CController`:

```
<?php
class Controller extends CController {
    public $layout='//layouts/column1';
    public $menu=array();
    public $breadcrumbs=array();
}
```

As you can see in the comments, this is a customized base controller class. It does three things:

1. Sets a default layout for every controller
2. Creates an attribute to be used for an HTML menu
3. Creates an attribute to be used for breadcrumbs

You don't *have* to extend this `Controller` component class when you create your own controllers, but this class does provide an easy way to customize how *all* of your controllers function (i.e., without hacking the `CController` class in the framework files). Towards that end, you should probably consider editing `protected/components/Controller.php` to suit your needs. For example, you should change the `$layout` value to your actual layout file (as explained in the previous chapter).

## Setting the Default Action

The "index" action is the default for every controller in Yii. This means that when an action is not specified, the `actionIndex()` method of the controller will be called. To change that behavior, just assign a different action value to the `$defaultAction` property within the controller class:

```
class SomeController extends Controller {
    public $defaultAction = 'view';
```

Note that you're using the action ID here, not the name of the method: it's just `view`, not `actionView` or `actionView()`.

With that line, the URL `http://www.example.com/index.php/some` calls the `actionView()` method whereas `http://www.example.com/index.php/some/create` calls `actionCreate()`.

## Setting the Default Controller

Although you can set the default action value within a controller, you cannot set the default *controller* in that way (which makes sense when you think about it). Just as the “index” action is the default in Yii, the “site” controller is the default controller. In other words, if no controller is specified in the URL (i.e., by the user request), the “site” controller will be invoked (and, therefore, the “index” action of that controller).

If you want to change the default controller, add this code to the main configuration array:

```
# protected/config/main.php
return array(
    /* Other stuff. */
    'defaultController' => 'YourControllerId',
    /* More other stuff. */
);
```

Note that you use the controller ID here (i.e., you drop off the *controller* part and use an initial lowercase letter). For example, to make “SomeController” the default, you’d use just “some” as the value.

Also note that you add this line so that it’s a top-level array element being returned (i.e., it does not go within any other section; it’s easiest to add it between the “basePath” and “name” elements, for clarity):

```
# protected/config/main.php
return array(
    'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',
    'defaultController' => 'some',
    'name'=>'My Blog',
    /* More other stuff. */
);
```

## Revisiting Views

As explained in the previous chapter, controllers create the site’s output by invoking the `render()` method:

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

The first argument to the method is the view file to be rendered, without its `.php` extension. By default, the view file will be pulled from the current controller's view directory: `protected/views/ControllerID/viewName.php`.

The second argument to `render()` is an array of data that can be sent to the view file. In many methods, a model instance is being passed along:

```
public function actionCreate() {
    $model = new Page;
    $this->render('create', array('model'=>$model));
}
```

As also explained in the previous chapter, that code will create a variable named `$model` in the “create” view file.

The `render()` method takes an optional third argument, which is a Boolean indicating if the rendered result should be returned to the controller instead of sent to the Web browser. This would be useful if you wanted to render the page and then send the output in an email or write it to a text file on the server (to act as a cached version):

```
$body = $this->render('email', 'data' => $data, true);
```

As also mentioned in the previous chapter, sometimes you'll want to render a view file *without* incorporating the layout. To do that, invoke `renderPartial()`. The `renderPartial()` method is also used for Ajax calls, where the layout isn't appropriate.

And finally, you'll sometimes need to render a view file from another directory (i.e., not this controller's directory). To change directories, preface the view file's name with `//`, which means to start in `protected/views`. The following code renders `protected/views/users/profile.php`:

```
$this->render('//users/profile', 'data' => $data);
```

## Making Use of Models

The most common thing that a controller does is create an instance of a model and pass that instance off to a view. Knowing how to do that is therefore vital to programming in Yii. I'm going to present the most standard, simple ways of doing

so in this chapter, and Chapter 8, “[Working with Databases](#),” will delve into the more complicated ways to get model instances. There are three basic ways that a controller will create a model instance:

- Creating a new, empty model instance
- Loading an existing model instance (i.e., a previously stored record)
- Retrieving *every* model instance (i.e., all previously stored records)

## Creating New Model Instances

The first way controllers create model instances is quite simple, and just uses standard object-oriented code:

```
public function actionCreate() {  
    $model = new Page;  
    // Etc.
```

{TIP} The parentheses often used when creating a new class instance are optional, meaning that `new Page` and `new Page()` are equivalent.

## Loading a Single Model

The second scenario—loading a single model instance from the database—is required by multiple actions:

- `actionView()`
- `actionUpdate()`
- `actionDelete()`

Since three (or possibly more) methods will perform this task, the code generated by Gii does the smart thing and creates a new controller method for that purpose:

```
public function loadModel($id) {  
    $model = Page::model()->findByPk($id);  
    if ($model==null) {  
        throw new CHttpException(404, 'The requested page  
            does not exist.');
```

}

```
    return $model;  
}
```

{NOTE} In order to enhance readability, some of the code that I present will be formatted slightly differently than the original created by Yii.

As you can see, this method takes an ID value as its lone argument. This should be the primary key value of the model to be loaded. The first line of the method is obviously the most important. It attempts to fetch the record with the provided primary key value:

```
$model = Page::model()->findByPk($id);
```

Let's take a moment to understand what's going on there, as it's both common in the framework and important. The goal is to fetch the record from the database table and turn it into a model object (an instance of the class). Fetching the record is done via the `findByPk()` method. This is defined in the `CActiveRecord` class, which all database models should extend. In theory, you *could* create an instance of the class and then use that instance's method:

```
$model = new Page;  
$model = $model->findByPk($id);
```

That *would* work, but it's a bit verbose, redundant, and illogical. The alternative is to use a *static class instance*. A static class instance is a more advanced OOP concept. Understanding static *class* instances is easier if you first understand static *methods*.

A *standard* class method is invoked through an object instance:

```
$obj = new ClassName();  
$obj->someMethod();
```

Some class methods are designed to be useful *without* a class instance. These are called *static*, and are defined using that keyword:

```
class SomeClass {  
    public static function name() {  
    }  
}
```

Because that method is *public* and *static*, it can be called without creating a class instance:

```
echo SomeClass::name();
```

This is what's happening in the first part of that code: `Page::model()`. That specific part of the code calls the static `model()` method of the `Page` class. (The `model()` method is specifically defined within the `CActiveRecord` class, which all database models should extend.)

As another example of a static class method that you would have seen in Yii, the `app()` method of the `YiiBase` class is also static. This is why the `index.php` bootstrap file uses the code `Yii::app()`.

The problem is that only *static* methods can be invoked this way; non-static methods still require a class instance. This creates a dilemma, as there are times where it'd be useful to call non-static methods without having to create a class instance first. The solution Yii came up with is to create a *static class instance*. A static class instance has these characteristics:

- It is immutable (i.e., its properties cannot be changed)
- It is not created with the `new` keyword
- It can be used to invoke non-static methods

One such non-static method is `findByPk()`.

Putting this all together, that one line of code is the same as:

```
// Get a static class instance:  
$static = Page::model();  
// Invoke the (non-static) method:  
$model = $static->findByPk($id);
```

For brevity sake, the two lines are combined into one by *chaining* the result of one method call to another. Specifically, `Page::model()` returns a static class instance, on which the `findByPk()` method is called.

{TIP} The `::` is known as the scope resolution operator.

Moving on in the controller's `loadModel()` method, after attempting to load the model instance, the method next checks that the value isn't NULL. If it is NULL, which means that no model could be created given the provided primary key value, an exception will be thrown:

```
if ($model==null) {  
    throw new CHttpException(404, 'The requested page does  
        not exist.');//  
}
```

I'll talk about exceptions in more detail at the end of the chapter.

Finally, the fetched model is returned by the method: `return $model;`. Note that due to the way exceptions work, this line will never be executed if an exception is thrown by the preceding line.

Here's an example of how this method is used:

```
public function actionView($id){  
    $this->render('view',array(  
        'model'=>$this->loadModel($id),  
    ));  
}
```

Anytime you need to load a model instance in your controller, simply call the `loadModel()` method, providing it with the primary key value of the record to retrieve.

## Loading Every Model

The final way a controller may create model instances is to load *every* model instance (i.e., every database record). That's easily done via the `findAll()` method of the `CActiveRecord` class. This method returns an array of model objects. Because this method is not static, it cannot be called directly through the class name (as in `ClassName::findAll()`). And, of course, you're not going to use an instance of the class either (it wouldn't make sense to create an actual model based upon a single database record in order to fetch every database record). The solution once again is to use the static class instance returned by the `model()` method:

```
$models = Page::model()->findAll();
```

Surprisingly, you won't find this code in that generated for you by `yiic` and `Gii`. The reason is that the two methods that will fetch multiple models—`actionIndex()` and `actionAdmin()`—use alternative solutions for fetching all the records.

The `actionIndex()` method ends up using a `CActiveDataProvider` object which will fetch all the model instances. That object is used by the `CListView` widget in the view file. Chapter 12, “[Working with Widgets](#),” talks about `CListView` in detail.

The `actionAdmin()` method uses the “search” scenario to fetch the applicable models. That scenario also returns a `CActiveDataProvider` object. This is also covered in detail in Chapter 12.

## Joining Models

Finally, there's the issue of how a controller can fetch joined models. I'm using the phrase “joined models” to refer to two models that have a relationship, as defined by the `relations()` method of each model class.

When you have two related tables, such as `comment` and `user`, you can use a `JOIN` in an SQL query to fetch information from both tables, such as the comment itself (from the `comment` table) and the user's name (from `user`). But when using the

`CActiveRecord` methods for fetching records (e.g., `findByPk()`), how you fetch the necessary information from the related table is not obvious. But Yii is brilliant, and has prepared two solutions.

The first solution is called “lazy loading”. Lazy loading triggers Yii to perform a relational query on demand:

```
// Perform one query of the comment table:  
$comment = Comment::model()->findByPk(1);  
// Run another query to get the associated username:  
$user = $comment->user->username;
```

This code works because the `Comment` class has a declared relationship with `User`. Usage of the `user->username` triggers another query to fetch the related record. That code is simple to use (and can even be used in a view file), but is not very efficient. Two queries are required when just one would work using straight SQL. With a page that shows 39 comments, that means there would be 40 total queries!

A better alternative is “eager loading”. When you know you’ll need related models ahead of time, you can tell Yii to fetch those, too, as part of the primary query:

```
// Perform one query of the comment table:  
$comment = Comment::model()->with('user')->findByPk(1);  
// No query necessary to do this:  
$user = $comment->user->username;
```

The value to be provided to the `with()` method is the name of the relation as defined within the `Comment` class. This works whether you’re fetching one record or multiple. As another example, the `Page` class in the CMS example is related to `User`, in that each page is owned by a user:

```
# protected/models/Page.php::relations()  
return array(  
    'comments' => array(self::HAS_MANY, 'Comment', 'page_id'),  
    'user' => array(self::BELONGS_TO, 'User', 'user_id'),  
    'files' => array(self::MANY_MANY, 'File',  
        'page_has_file(page_id, file_id)'),  
);
```

This means you can fetch every page with every page author in one step:

```
$pages = Page::model()->with('user')->findAll();
```

There are the very basics of joining models in a controller. It can get far more complicated, as will be explained in Chapter 8.

## Handling Forms

Two of the controller methods are used to both display and handle a form: create and update. The structure of both is virtually the same, except that the one begins with a new model from scratch and the other fetches its model from the database (and they obviously render different view files). Here's the `actionCreate()` method:

```
public function actionCreate(){
    $model=new Page;

    if(isset($_POST['Page'])) {
        $model->attributes=$_POST['Page'];
        if ($model->save()) {
            $this->redirect(array('view',
                'id'=>$model->id));
        } // save()
    } // isset()

    $this->render('create',array('model'=>$model));
}
```

First, the method creates a new model instance. In `actionUpdate()`, that would be `$model=$this->loadModel($id);` instead, as that method is working with an existing record, not a new one.

Next, the method checks if the form has been submitted. If so, then `$_POST['ClassName']` will be set. Chapter 9, “Working with Forms,” goes into forms in detail, but know now that `$_POST['ClassName']` will be an array:

- `$_POST['ClassName']['someAttribute']`
- `$_POST['ClassName']['anotherAttribute']`
- Et cetera

If the form has been submitted, then the model's attributes will be assigned the values from the form, *but only for attributes that are safe thanks to the model's rules*. This was explained in Chapter 5, “Working with Models.”

Next, if the model can be saved, then the user is redirected to the view page (i.e., the page for viewing a specific record). Know that the model can only be saved if the data passed all the business rules defined in the model.

If the model cannot be saved, or if the form has not yet been submitted, then the corresponding view file—“create” or “update”—is rendered, passing along the model instance.

## Basic Access Control

Access control is an integral aspect of any Web site, dictating what a user can and cannot do based upon factors of your choosing. There are a couple of ways of implementing access control in Yii, starting with basic access control. This approach, which I'll cover here, is similar to the Access Control Lists (ACL) used by operating systems. This approach is simple to implement in Yii. In fact, Gii will have set up a basic access control structure for you, to be explained now.

{NOTE} Chapter 11 will discuss the more advanced alternative, Role-Based Access Control (RBAC).

### The Fundamentals

The code generated by Gii defines the `accessRules()` method, which provides for basic access control. For the access rules to apply, the method must first be set as a filter:

```
public function filters() {
    return array('accessControl');
}
```

I'll cover filters later in the chapter, but that code, created by Gii in CRUD-related controllers, enables the access control filter. The `accessRules()` method then defines who can do what.

For the “what” options, you have your actions: admin, create, delete, index, update, and view. In almost all situations, only administrators should be able to execute the admin and delete actions, but maybe every user can do index. It's helpful to remember that although this code is in the controller, it really dictates who can do what with the associated models (i.e., who can create, read, update, etc. the model records).

{WARNING} When you add new actions to a controller, be sure to update your access rules!

Your “who” depends upon the situation, but to start there are three general types of users, each represented by a symbol:

- ?, anonymous (i.e., not logged-in) users
- @, logged-in users
- \*, any user, logged-in or not

Depending upon the login system in place, you may also have levels of users, like admins, or specific user names to target.

The `accessRules()` method combines all this information and returns an array of values. The values are also arrays, indicating permissions (“allow” or “deny” are the only options), actions, and users. Each value array is of this syntax:

```
array(
    '<permission>',
    'actions' => array('action1', 'action2'),
    'users' => array('user1', 'user2')
);
```

This syntax comes from the `CAccessRule` class which dictates what's possible for a rule: what attributes exist to which you can assign values. Only the permission—“allow” or “deny”—is required.

*{TIP}* Both the actions and the users are case-insensitive, but I think it's best to treat them all as if they were case-sensitive regardless.

Here are some unedited rules created by Gii:

```
# protected/controllers/PageController.php::accessRules()
array('allow', // allow all users to perform 'index' and
      // 'view' actions
      'actions'=>array('index','view'),
      'users'=>array('*'),
),
array('allow', // allow authenticated user to perform
      // 'create' and 'update' actions
      'actions'=>array('create','update'),
      'users'=>array('@'),
),
array('allow', // allow admin user to perform 'admin' and
      // 'delete' actions
      'actions'=>array('admin','delete'),
      'users'=>array('admin'),
),
array('deny', // deny all users
      'users'=>array('*'),
),
```

Those rules represent the default settings, where anyone can perform index and view actions, meaning that anyone can list all records or view individual records of the associated model. The next section allows any logged-in user to perform create

and update actions. Next, only the “admin” user can perform admin and delete actions. To be clear, the use of “admin” here doesn’t refer to a user *type*, but a specific user name (based upon the default accepted login values of user/user and admin/admin). Finally, a global deny for all users is added to cover any situation that wasn’t explicitly defined. This is just a good security practice.

Understand that these rules are checked in order from top down. Once a rule applies, that’s it: the rules do not continue to be evaluated. This means that if you put a “deny all” rule first, no one could do anything. On the other hand, if no rules apply, then the action is allowed. For this reason, you should always do a “deny all” last, just in case you missed something (from a security perspective, it’s far better to accidentally deny an action than to accidentally allow one).

You’ll want to customize the rules to each controller and situation. For example, in a CMS site, anyone (or more specifically, anonymous users) needs to perform create actions with a `User` model and controller, as that constitutes registration. But perhaps only logged-in users can create comments, and only certain users or user types can create posts.

## Identifying Users

With basic access control, there are four ways to identify or restrict to what users a rule applies:

- By authentication status (anyone, only anonymous, or only logged-in)
- By username
- By IP address
- By role

The first two possibilities have already been mentioned. The use of authentication status is the most broad (i.e., whether or not the user is logged-in). Don’t use authentication status if it’s possible to be more specific! For example, if you only want to allow the admin user (by name) to delete records, don’t allow all logged-in users delete capability.

Setting rules by specific user name is very particular and effective, but only appropriate for sites with a small number of relatively static users. If a site only has one administrator, and that administrator will never change, then using the name of that administrator is appropriate.

Similarly, setting rules by IP address is logical if an administrator will only ever be accessing the site from a specific IP address. To do that, set a “ips” value:

```
array(
    'allow',
    'actions' => array('admin', 'delete'),
```

```
'users' => array('@'),
'ips' => array('127.0.0.1')
);
```

That code says that the user must be logged-in and coming from the 127.0.0.1 IP address. This, of course, is the IP address for localhost, which means that the administrator is allowed to perform those actions when working on the same computer as the site. To allow remote access, you would need to add the IP address of the administrator's computer (or network).

{NOTE} IP addresses are a great way to allow for access via localhost, but using IP addresses to identify users over networks is problematic for two reasons. First, a person's IP address can change frequently, depending upon her Internet access provider. Second, multiple people accessing the same site from the same network (e.g., a company, organization, or school) may all have the same IP address.

Finally, you can identify users by roles. This is an implementation of Role-Based Access Control (RBAC), and will be explained in Chapter 11.

## Other Restrictions

There are two more ways you can restrict who can do what. The first is to use the “verbs” index. It takes an array of request types to which the rule applies. For example, if you have a page that both shows some information and displays a form, you might allow everyone to perform a GET request (i.e., see the information) but only logged in users can submit the form (perform a POST request):

```
array(
    // Anyone can "GET" the page
    'allow',
    'actions' => array('someAction'),
    'users' => array('*'),
    'verbs' => array('GET')
),
array(
    // Must be logged in to POST
    'allow',
    'actions' => array('someAction'),
    'users' => array('@'),
    'verbs' => array('POST')
);
```

(Of course, you'd also want to code your view so that it doesn't show the form to those that can't submit it, but this rule is acting as a security measure.)

Or, as another example, you might have a cron or other automated process that regularly performs HEAD requests of an action to confirm that the site is up and running. That kind of request would be restricted to anonymous users.

Finally, there's the "expression" option. It can be used to define a PHP expression whose executed value dictates whether or not the rule will apply. You can use this to establish conditionals with a Boolean result. This example is from the Yii documentation:

```
array(
    'allow',
    'actions' => array('someAction'),
    'expression'=>'!$user->isGuest && $user->level==2',
),
```

A CMS example with different user types—admin, editor, writer, and other—might let everyone but “other” (i.e., readers) create content:

```
array(
    'allow',
    'actions' => array('create'),
    'expression'=>'isset($user->type) &&
        ($user->type !== "other")',
),
```

Expressions can be very useful, but they also start blurring the lines with roles.

## When Access is Denied

You should also be aware of what happens when Yii denies access to an action. If the user is not logged in and the rule requires that she be logged in, the user will be redirected to the login page by default. After successfully logging in, the user will be redirected back to the page she had been trying to request.

If the user *is* logged in but does not have permission to perform the task, an HTTP exception is thrown using the error code 403. That value matches the “forbidden” HTTP status code value. Towards the end of the chapter you'll see how exceptions are handled.

To change an error message associated with an exception, assign a value to the “message” index:

```
array('allow',
    'actions' => array('someAction'),
    'users'=>array('admin'),
    'message'=>'You are not allowed to do this thing
        you are trying to do.'
),
)
```

## Understanding Routes

A topic critical to controllers, although not dictated within the actual controller code are *routes*. Routes are how URLs map to the controller and action to be invoked. Chapter 3, “[A Manual for Your Yii Site](#),” introduced the basic concept and Chapter 4 explained how to configure the “urlManager” component to change how routes are formatted. Let’s now look at the topic in greater detail.

### Path vs. Get

URLs in Yii are going to be in one of two formats:

`http://www.example.com/index.php?r=ControllerID/ActionID`

or

`http://www.example.com/index.php/ControllerID/ActionID/`

The first format is the default, and is called the “get” format, as values are passed as if they were standard GET variables (because, well, they are). The second format is the “path” format, in which the values appear as if they are part of the path (i.e., as if they map to directories on the file system). As explained in Chapter 4, this format is enabled by uncommenting the appropriate part of the configuration file:

```
# protected/config/main.php
// Other stuff.
'components'=>array(
    'urlManager'=>array(
        'urlFormat'=>'path',
        // Etc.
    ),
    // More other stuff.
)
```

This is fairly basic information and is hopefully well ingrained to you by now. The important part of this concept is how you define the rules for dictating the paths.

Note that, for simplicity sake, I’m going to assume you’re using the path format from here on out. Further, the rules I’ll discuss only apply to the “path” part of the URL: that after the schema (e.g., http or https) and the domain (e.g., www.example.com/).

So in the rest of this explanation, I'll generally begin my demonstration URLs without the assumed `http://www.example.com/`.

{TIP} You can actually make your rules apply to the whole URL, including the domain and/or subdomain. See the class docs for the [CurlManager](#) class for an example.

## Route Rules

These are the default rules set in the configuration file:

```
'rules'=>array(
    '<controller:\w+>/<id:\d+>' => '<controller>/view',
    '<controller:\w+>/<action:\w+>/<id:\d+>' =>
        '<controller>/<action>',
    '<controller:\w+>/<action:\w+>' =>
        '<controller>/<action>',
),
)
```

What exactly is going on there and, more importantly, how do you understand these rules enough for you to be able to implement your own?

The “urlManager” rules apply both when reading in a URL and when creating a URL (e.g., as a link). The rules serve two purposes: identifying the *route*—the controller and action—and reading in or passing along any parameters.

Each rule uses a name=>value pair to associate a value with a route. For example, here's a simple rule: `'home' => 'site/index'`. With that rule in place, the URL `http://www.example.com/home` will call the “index” action of the “site” controller. And similarly, if you go to create a URL to the “site/index” route, Yii will set that URL as `http://www.example.com/home`.

Obviously hardcoding literal strings to routes has limited appeal. To make your rules more flexible, apply regular expressions and *named placeholders* as the values. That syntax is: `<PlaceholderName:RegEx>`. Naturally, you do need to understand regular expressions to follow this approach.

As an example, let's return to the last rule defined by default in the configuration file:

```
'<controller:\w+>/<action:\w+>' => '<controller>/<action>',
```

The very first part of that is `<controller:\w+>`. That means the rule is looking for what's called a regular expression “word”, represented by `\w+`. In plainer English, that particular regular expression looks for a string of one or more alphanumeric

characters and the underscore. Once a “word” is matched, the rule labels that combination of characters as *controller*.

Next, the rule looks for a literal slash.

Next, the rule looks for another “word”: \w+. Once found, the rule labels that combination as *action*.

The labels—the named placeholders—can then be used to establish the route. (Think of this like *back referencing* in regular expressions, if you’re familiar with that concept.)

This rule therefore equates a URL of **anyword1/anyword2** with the “anyword1/anyword2” route. Hopefully, that literal association makes sense, but remember that this is a *flexible* literal association, unlike the hardcoded one shown earlier.

{TIP} You may have an easier time understanding routes and regular expressions if you’re familiar with using Apache’s `mod_rewrite` tool as routes in Yii are used to the same effect.

## Handling Parameters

Once you understand the concepts I’ve covered thus far, there’s one new twist to introduce: parameters. Many actions will require them, such as the “view” and “update” actions that need to accept the ID value of the record being viewed or updated. Those particular values will always be integers, you can use the \d+ regular expression pattern to match them. That’s what’s going on in the first rule:

```
'<controller:\w+>/<id:\d+>' => '<controller>/view',
```

That rule looks for a “word”, followed by a slash, followed by one or more digits. The matching “word” gets mapped to the controller’s “view” action. Hence, the URL **page/42** is associated with the route “page/view”. But what about the 42?

The digits part is a named *parameter*, labelled “id”. That value does not get used as part of the actual route: it’s neither a controller nor an action. Where does it go? Well, it will be passed as a parameter to the action method:

```
# protected/controllers/PageController.php
public function actionView($id) {
    // Etc.
```

When Yii executes the “view” action, it invokes that method, passing along the parameter, as you would pass a parameter to any function call. In this particular case, the *route* will be “page/view” but the `actionView()` method of the “page”

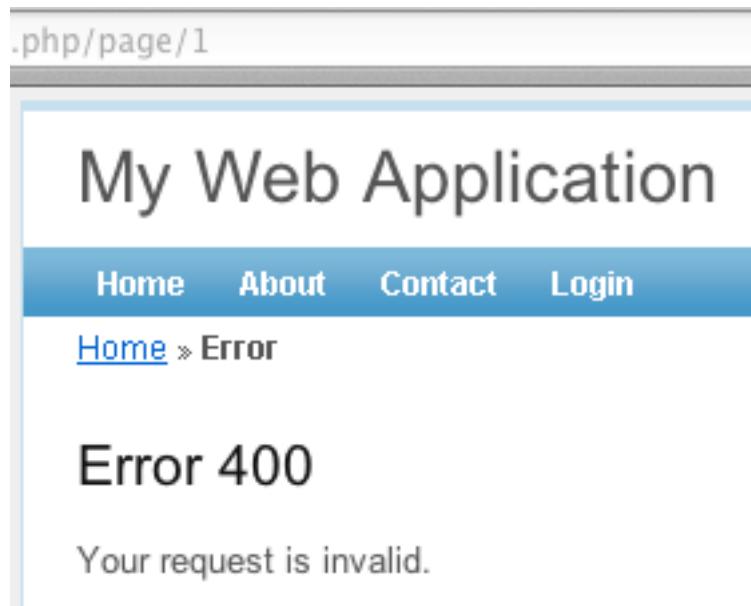
controller will also be able to use `$id`, which will have a value of 42. There's one little hitch...

Normally, it does not matter what names you give your parameters in a function:

```
function test($x) {}  
$z = 23;  
test($z); // No problem.  
$x = 42;  
test($x); // Still fine.
```

However, when you've identified a parameter in a rule that's not part of the route, it will only be passed to an action if that action's parameter is named the same. The earlier example code works, but this action definition with that same rule will throw an exception (**Figure 7.1**):

```
# protected/controllers/PageController.php  
public function actionView($x) {  
    // Etc.
```



**Figure 7.1:** This exception is actually caused by a misnamed method parameter.

{NOTE} Placeholders, as I'm calling them, and parameters are created in a rule in exactly the same way. But for clarity sake, I'm using two different terms to distinguish between matches that will be used in routes (i.e., placeholders) and those passed to action methods (i.e., parameters).

Looking at this in more detail, let's say you want to support more than one parameter. For example, you have a user verification process, wherein the user clicks a link in an email that takes them back to the site to verify the account. The link should pass two values along in the URL to uniquely identify the user: a number and a hash (a string of characters). The rule to catch that could be:

```
'verify/<x:\d+>/<y:\w+>' => 'user/verify',
```

That rule says that the literal word “verify”, followed by a slash, followed by a digit, followed by another slash, followed by a regular expression “word” should be routed to “user/verify”. With the named parameters, the `actionVerify()` method should be written to accept two parameters, `$x` and `$y`:

```
# protected/controllers/UserController.php
public function actionVerify($x, $y) {
    // Etc.
```

Strange as it may seem, you can put those parameters in either order, and it will still work. You can even write the method with just one or no parameters and the the action can still be invoked without error (although the action would not be passed both parameters in that case). Really, the only thing you can't do is define the method to take parameters with different names than those in the rule.

As one more example of this, to really hammer the point home, let's say you want a way to view a user's profile by name: `user/myUserName`, `user/yourUserName`, etc. That rule could be:

```
# protected/config/main.php
'user/<username:\w+>' => 'user/view',
```

That rule associates the “user/view” route with that value, and creates a named parameter of “username”. (Note that the regular expression pattern for matching the actual username will depend upon what characters you allow in a username.)

Now for the action, which would presumably retrieve the user's profile from the database using the username:

```
# protected/controllers/UserController.php
public function actionView($username) {
    // Etc.
```

Returning to the default rules created by `yiic`, the “edit” and “update” actions are handled by the last one:

```
'<controller:\w+>/<action:\w+>/<id:\d+>' =>
'<controller>/<action>',
```

That rule would catch `page/edit/42` or `page/delete/42` (as well as `page/view/42`). And, again, each action should be defined so that it takes a parameter specifically named `$id`.

Understand that the rules are tested in order from top down, the first rule that constitutes a match will be equated to its route. Also, from both a comprehension standpoint, and for better performance, you should try to have as few rules as possible.

## Case Sensitivity

Before getting into a couple more examples, I should clarify that routing rules are case-sensitive by default. This means that although the regular expression `\w+` will match `Post`, `post`, or `pOsT`, only `post` will match a controller in your application.

As explained previously, the controller ID is the name of the controller class minus the “Controller” part, with the first letter in lowercase. Thus, `SiteController` becomes “site”, but `SomeTypeController` would become “someType”. The same is true for actions, except you start by dropping the initial “action” part.

*{TIP}* Routes can be made to be case-insensitive. Still, I think it best to have them be and treat them as case-sensitive.

## Creating URLs

As I said at the beginning of this discussion, the routing rules come into play when parsing URLs into routes and also when creating URLs based upon routes. You should always have Yii create URLs to any page that gets run through the bootstrap file! This is done using either the `createUrl()` or the `createAbsoluteUrl()` method of the `CController` class.

Within a controller, or within its view files, you can access those methods via `$this`. Note that both methods just create and return a URL, not an HTML link!

The first argument to both is a string indicating the route. The current controller and action are assumed, so `$this->createUrl('')` returns the URL for the current page.

If you just provide an action ID, you’ll get the URL for that action of the current controller:

```
# protected/controllers/PageController.php
public function actionDummy() {
    $url = $this->createUrl('index'); // page/index
```

If you provide a controller, the URL will be to that route:

```
# protected/controllers/PageController.php
public function actionDummy() {
    $url = $this->createUrl('user/index'); // user/index
```

If the URL expects named parameters to be passed, add those in an array as the second argument:

```
# protected/controllers/PageController.php
public function actionDummy() {
    $url = $this->createUrl('view', array('id' => 42));
    // page/view id=42
```

To really hammer home the point, for that URL to work, the `actionView()` method of the controller should be written to accept an `$id` parameter, which presumably also matches the corresponding rule.

You can also create URLs through `Yii::app()`:

```
$url = Yii::app()->createUrl('page/view',
    array('id' => 42));
```

The main difference between the two versions of the `createUrl()` method is that the `CController` version can be used *without* referencing a controller ID, whereas the `Yii::app()` version (i.e., that defined in `CApplication`) requires a controller ID be provided.

## Tapping Into Filters

Another method defined in controllers by Gii is `filters()`. Filters let you identify code to be executed before and/or after an action is executed. An example is the “accessControl” filter, which is run prior to an action, and confirms that the user has authority to perform the action in question. You can also use filters to:

- Restrict access to an action to HTTPS only or a certain request type (Ajax, GET, POST)
- Start and stop timers to benchmark performance

- Implement compression
- Perform any other type of setup that should apply to one or more actions

Filters are created by defining methods in your controller or by creating a separate class that extends `CFilter`. The controller method uses the naming scheme “filter” plus whatever filter name. `CController` has a `filterAccessControl()` method defined for you, whose usage has already been explained.

There are two other filters defined for you in `CController`: `filterAjaxOnly()` and `filterPostOnly()`. These filters are used to prevent an action from executing unless it was requested via Ajax and POST accordingly. The Gii generated code uses the latter to force deletions via POST:

```
public function filters() {
    return array(
        // perform access control for CRUD operations:
        'accessControl',
        // we only allow deletion via POST request:
        'postOnly + delete',
    );
}
```

Looking at the syntax there, the `filters()` method returns an array. Each array value should be a *string*. The string starts with the filter’s name, which is the name of the associated method to call minus its “filter” preface.

Then, within the string, you can optionally dictate to what actions the filter should or should not be applied. A plus sign means the filter applies to the following action or actions, separating multiple via commas:

```
return array(
    // perform access control for CRUD operations:
    'accessControl',
    // we only allow deletion & updates via POST request:
    'postOnly + delete,update',
);
```

A minus sign would mean that the filter would apply to every action *except* the one(s) named. Note that the filters are run in the order they are listed, so in that code, the “accessControl” filter is executed, then “postOnly”.

As an example of writing your own filter, the following function can confirm that an HTTPS connection was used to access a resource. As previously mentioned, all filtering methods must begin with “filter”. The method needs to take one parameter, which will be a *filter chain*. This variable will be used to allow the action to take place by invoking its `run()` method (i.e., continue the filtering and such). To test

if HTTPS was used, you can invoke the `getIsSecureConnection()` method of the `CHttpRequest` object, available via `Yii::app()->getRequest()`. Here's the entire code:

```
public function filterHttpsOnly($fc) {
    if (Yii::app()->getRequest()->getIsSecureConnection()) {
        $fc->run();
    } else {
        throw new CHttpException(400,
            'This page needs to be accessed securely.');
    }
}
```

{NOTE} In this particular example, I would also have Apache be configured to force HTTPS for the URLs in question, with this Yii filter acting as a safety net.

To use that filter in a controller, you would code:

```
public function filters() {
    return array(
        'accessControl',
        // account must be accessed over HTTPS:
        'httpsOnly + account',
    );
}
```

{TIP} Explaining how to create filters as a separate class is a bit too advanced and esoteric at this point, but see the [corresponding section](#) of the Yii Guide, if you're curious.

## Showing Static Pages

Next up, let's look at a different way of rendering view files: using *static pages*. The difference between a static page and a standard page in the site is that the static page does not change based upon any input. In other words, a dynamic page might display information based upon a provided model instance, but a static page just displays some hard-coded HTML (in theory).

If you only have a single static page to display, the easy solution is to treat it like any other view file, with a corresponding controller action:

```
<!-- # protected/controllers/views/some/about.php -->
<h1>About Us</h1>
<p>spam, spam, spam...</p>
```

And:

```
# protected/controllers/SomeController.php
public function actionAbout() {
    $this->render('about');
}
```

The combination of those two files means that the URL—

`http://www.example.com/index.php/some/about`

—will load that about page. As I said, this is a simple approach, and familiar, but less maintainable the more static pages you have.

An alternative and more professional solution is to register a “page” action associated with the `CViewAction` class. This is done via the controller’s `actions()` method:

```
# protected/controllers/SiteController.php
public function actions() {
    return array(
        'page' => array('class' => 'CViewAction')
    );
}
```

*{TIP}* You can have any controller display static pages, but it makes sense to do so using the “site” controller.

The `CViewAction` class defines an action for displaying a view based upon a parameter. By default, the determining parameter is `$_GET['view']`. This means that the URL `http://www.example.com/index.php?r=site/page&view=about` or, if you’ve modified your URLs, `http://www.example.com/index.php/site/page/view/about`, is a request to render the static `about.php` page.

*{NOTE}* You must also adjust your access rules to allow whatever users (likely everyone) to access the “page” action. Or, you can do what the “site” controller does: not implement access control at all.

By default, `CViewAction` will pull the static file from a `pages` subdirectory of the controller’s view folder. Thus, to complete this process, create your static files within the `protected/views/site/pages` directory.

If, for whatever reason, you want to change the name of the subdirectory from which the static files will be pulled, assign a new value to the `basePath` attribute:

```
# protected/controllers/SiteController.php
public function actions() {
    return array(
        'page' => array('class' => 'CViewAction',
                         'basePath' => 'static')
    );
}
```

You can also create a nested directory structure. For example, say you wanted to have a series of static files about the company, stored within the `protected/views/site/pages/company` directory. To refer to those files, just prepend the value of `$_GET['view']` with “company.”: `/site/page/view/-company.board` would display the `protected/views/site/pages/company/board.php` page.

By default, if no `$_GET['view']` value is provided, `CViewAction` will attempt to display an `index.php` static file. To change that, assign a new value to the `defaultView` property:

```
# protected/controllers/SiteController.php
public function actions() {
    return array(
        'page' => array('class' => 'CViewAction',
                         'defaultView' => 'about')
    );
}
```

To change the layout used to encase the view, assign the alternative layout name to the `layout` attribute in that array.

## Exceptions

One of the great things about Object-Oriented Programming is that you’ll never see another error, although there are exceptions (pun!). Exceptions, in case you’re not familiar with them, are errors turned into object variables, that’s all. This is fairly standard OOP stuff, but you’ll need to be comfortable with exceptions in order to properly use the framework.

In many situations, the framework itself will automatically create an exception for you. You may have seen this already, as in Figure 7.1.

Sometimes you’ll want to raise an exception yourself. To do so, you *throw* it:

```
if /* some condition */ {
    throw new CException('Something went wrong');
}
```

You've actually seen code similar to this in the chapter already. Notice that the thing being thrown is actually an object of type `CException`. Written more verbosely, you could do this:

```
$e = new CException('Something went wrong.');
throw $e;
```

{TIP} When an exception is thrown, whether by you or automatically by the framework, no subsequent code will be executed.

Yii defines three classes for exceptions:

- `CException`
- `CDbException`
- `CHttpException`

The first is a generic exception class, a wrapper of PHP's built-in `Exception` class. When creating an exception of this type, provide the constructor (the method called when an instance of this class is created) with a string message. Optionally, you can provide an error number as the second argument. It's up to you to give this number meaning.

The `CDbException` class is for exceptions related to database operations.

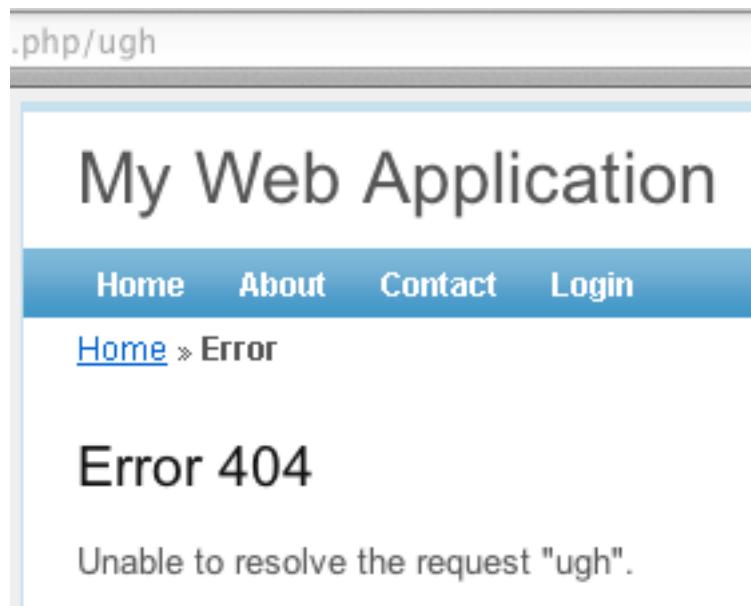
The `CHttpException` class is for exceptions related to HTTP requests. This exception type also has a status code value. You could use it to indicate, for example, a page not found (**Figure 7.2**):

```
throw new CHttpException(404, 'Page not found.');
```

Previous examples in this chapter also use `CHttpException` for forbidden access messages.

{TIP} In time you'll probably want to create your own exception class by extending `CException`.

In standard PHP code, exceptions are used with `try...catch` blocks, where any exceptions that occur within the `try` block are to be caught by a matching `catch`. You don't have to formally create `try...catch` blocks in your Yii code, though. Yii



**Figure 7.2:** Because the “ugh” controller does not exist, a 404 exception is thrown.

will automatically catch the exceptions and handle them differently based upon the type.

When a `CHttpException` occurs, Yii will look for a corresponding view file to use. The name of the file will match the HTTP status code associated with the error: `errorXXX.php`, where XXX is the code. The framework will look for that view file in these directories in this order:

- `WebRoot/themes/ThemeName/views/system`
- `WebRoot/protected/views/system`
- `yii/framework/views`

So if you are using a theme, Yii will check that directory first. If not, Yii will check within a `system` subdirectory of your application’s `views` folder. If you have not yet created an appropriate view file there, Yii will use the default view that comes with the framework files.

For all other exception types, by default, Yii uses the “site” controller’s “error” action to handle exceptions. This is true regardless of the controller in which the exception actually occurred. You can change the behavior by configuring the “errorHandler” component in your configuration file:

```
# protected/config/main.php
// Other stuff.
'components' => array(
```

```
'errorHandler' => array ('errorAction' =>
    'ControllerId/ActionId')
)
```

The default error handling action method looks like:

```
public function actionError() {
    if($error=Yii::app()->errorHandler->error) {
        if(Yii::app()->request->isAjaxRequest) {
            echo $error['message'];
        } else {
            $this->render('error', $error);
        }
    }
}
```

The error will be an array with these elements:

Index	Stores the...
code	HTTP status code
file	name of the PHP script where the error occurred
line	line number in the PHP script on which the error occurred
message	error message
source	source code context in which the error occurred
trace	call stack leading up to the error
type	error type

This means that within your custom view file, you can refer to these values to report whatever to the end user. How you access those values is a bit tricky, however, as Yii passes the error object to the view in a somewhat unconventional way. Traditionally, you would do this:

```
$this->render('error', array('error' => $error));
```

In that case, `$error` in the view is an array, just as it is in the controller. However, the code Yii uses to pass the error along is this:

```
$this->render('error', $error);
```

Because `$error` is passed as an array, its individual elements will be immediately available in the view file as `$code`, `$file`, `$line`, etc. This is equivalent to coding:

```
$this->render('error', array(
    'code' => $error['code'],
    'file' => $error['file'],
    // etc.
));
```

Keep in mind that end users should never see detailed error messages. They are for the developer's benefit only.

As a convenience, Yii will automatically log exceptions to **protected/runtime/application.log** (or your other default log file). This allows you to go back in and view all the exceptions that occurred.

## Chapter 8

# WORKING WITH DATABASES

A database is at the core of most dynamic Web applications. In previous chapters, you've been introduced to the basics of how to interact with a database using Yii. At a minimum, this entails:

- Configuring Yii to connect to your database
- Creating models based upon existing tables
- Using Active Record to create, read, update, and delete records

This combination is a great start and can provide much of the needed functionality for most sites. But in more complicated sites, you'll need to be able to interact with the database in more low-level or custom ways.

In this chapter, you'll learn all of the remaining core concepts when it comes to interacting with databases using Yii. In Part 3, "Advanced Topics," a handful of other subjects related to databases will be covered, although those will be far more complex and less commonly needed than those discussed here.

As with most things in Yii, there are many ways of accomplishing the same task. Attempting to explain every possibility tends to confuse the reader, in my experience. So, for the sake of clarity, this chapter will discuss only the approaches and methods that I think are most practical and common.

### Debugging Database Operations

Whenever you begin working with a database, you introduce more possible causes of errors. Thus, you must learn additional debugging strategies. When using PHP to run queries on the database, the problems you might encounter include:

- An inability to connect to the database
- A database error thrown because of a query
- The query not returning the results or having the effect that you expect
- None of the above, and yet, the output is still incorrect

On a non-framework site, you just need to watch for database errors to catch the first two types of problems. There's a simple and standard approach for debugging the last two types:

1. Use PHP to print out the query being run.
2. Run the same query using another interface to confirm the results.
3. Debug the query until you get the results you want.

When using a framework, these same debugging techniques are a little less obvious, in part because you may not be directly touching the underlying SQL commands. Thankfully, Yii will still be quite helpful, if you know what switches to flip.

First of all, while developing your site, enable `CWebLogRoute`:

```
# protected/config/main.php "components" section
'log'=>array(
    'class'=>'CLogRouter',
    'routes'=>array(
        array(
            'class'=>'CFileLogRoute',
            'levels'=>'error, warning',
        ),
        array(
            'class'=>'CWebLogRoute',
        ),
    ),
),
```

This will show, in your browser, everything being done by the framework including what queries are being run (**Figure 8.1**).

But there's one more thing you should do to make debugging SQL problems easier...

Many queries will use *parameters*, separating the core of the query from the specific (and often user-provided) values used by it. To see the entire query, with the parameter values in place, you must also set the `CDbConnection` class's `enableParamLogging` attribute to true:

```
Page.findByPk()
in /Users/larryullman/Sites/htdocs/protected/controllers/PageController.php
(155)
in /Users/larryullman/Sites/htdocs/protected/controllers/PageController.php
(55)
in /Users/larryullman/Sites/htdocs/index.php (13)

Querying SQL: SELECT * FROM `page` `t` WHERE `t`.`id`='1' LIMIT 1
in /Users/larryullman/Sites/htdocs/protected/controllers/PageController.php
(155)
in /Users/larryullman/Sites/htdocs/protected/controllers/PageController.php
(55)
in /Users/larryullman/Sites/htdocs/index.php (13)
```

**Figure 8.1:** Here, the Web log router is showing one of the queries required to fetch a page record.

```
# protected/config/main.php "components" section
'db'=>array(
    'connectionString' =>
        'mysql:host=localhost;dbname=test',
    'emulatePrepare' => true,
    'enableParamLogging' => true,
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
),
```

Now you'll be able to see the entire query in your output, including the query's parameter values.

{WARNING} Public display of errors and detailed logging are terrible for a site's security and performance. Both should only be used during the development of a site.

## Database Options

There are two broad issues when it comes to having a Yii based site interact with a database: the database application in use and how the interactions are performed.

MySQL is by far the most commonly used database application, not just for Yii, but for PHP, too. And, it's the only database application I'm using in this book (I

*think*, perhaps I'll find the time and need to discuss one or two others in Part 3). You tell Yii what database application, and what specific database, is to be used via the "db" component, configurable in the primary configuration file as you just saw in the previous bit of code.

Yii can work with other database applications, too, including:

- [PostgreSQL](#)
- [SQLite](#)
- Microsoft's [SQL Server](#)
- [Oracle](#)

To change database applications, modify the "connectionString" value (in the configuration file) to match the application in use. To find the proper connection string DSN (Database Source Name) value, see the [PHP manual page](#) for the PHP Data Object (PDO) class. Yii uses PDO for its connections and low-level interactions.

Once you've configured Yii for your database application and specific database, the "db" component can pretty much be forgotten about. But you can easily switch databases or database applications later on, if needed (e.g., you might change the database name when you go from the development site to the production version).

Regardless of what database application you're connected to, there are actually three different ways in Yii you can interact with it:

- Active Record
- Query Builder
- Database Access Object (DAO)

The Active Record approach is what you have already seen. For example, this line of code was explained in the previous chapter:

```
$model = Page::model()->findByPk($id);
```

That line performs a SELECT query, retrieving one record using the primary key value.

The two alternatives to Active Record are the Query Builder and Data Access Objects (DAO). To best understand the three options, when you would use them, and how, let's look at each in great detail. As in the previous chapters, I'll assume you've created the CMS example first mentioned in Chapter 2, "[Starting a New Application](#)." After covering all three options in great detail, I'll provide some tips as to when you should use which approach.

## Using Active Record

If you're reading this book sequentially, and I really hope you are, you've already learned about and used Active Record. In Yii, Active Record is implemented in the `CActiveRecord` class. Every model based upon a database table will extend `CActiveRecord` by default. Still, you may not really understand what Active Record is or how to use it to its fullest potential.

Active Record is simply a common architectural pattern for relational databases (first identified by Martin Fowler in 2003). Active Record provides CRUD—Create, Read, Update, and Delete—functionality for database records. Active Record is used for Object Relational Mapping (ORM): converting a database record into a usable programming object and vice versa. An instance of the `CActiveRecord` class therefore can represent a single record from the associated database table.

{TIP} Active Record cannot be used with every database application, but does work with MySQL, SQLite, PostgreSQL, SQL Server, and Oracle.

### Creating New Records

A new Active Record object can be created the way you would create any object in PHP:

```
$model = new Page();
```

(That code assumes that the `Page` class extends `CActiveRecord`.)

Assuming this is the CMS example, a new page record can be created by assigning values to the properties; the class properties each corresponding directly to a database column:

```
$model->user_id = 1;
$model->title = 'This is the title';
// And so forth.
```

Understand that in the code created by Gii, the controllers automatically populate the object's properties using the form values, but you can hardcode value assignments as in the above.

To create the new record in the database, invoke the `save()` method:

```
$model->save();
```

The `save()` method is also how you would update an existing record, after having changed the necessary property values. Remember that whether you're creating a new record, or updating an existing one, the INSERT/UPDATE will only occur if the model's data passes all of the validation routines established in the model class. This was explained in Chapter 5, “[Working with Models](#).”

To differentiate between inserting a new record and updating an existing one, you can invoke the `getIsNewRecord()` method, which returns a Boolean. The catch is that you can only reliably use it before the record is saved. Once the record is saved, `getIsNewRecord()` will return false, because the record is no longer new:

```
$new = $model->getIsNewRecord();
if ($model->save()) {
    if ($new) {
        $message = 'The thing has been created';
    } else {
        $message = 'The thing has been updated.';
    }
}
```

## Retrieving A Record

One example of retrieving existing records using Active Record has already been shown and explained in this book:

```
$model = Page::model()->findPk($id);
```

The `findPk()` method needs to be provided with a primary key value and returns a single row. If no matching primary key exists, the method returns NULL.

Active Record supports several different methods that allow you to retrieve records using different criteria. The most basic of these is `find()` ([Figure 8.2](#)).

<b>find() method</b>		
<pre>public CActiveRecord find(mixed \$condition='', array \$params=array( ))</pre>		
<b>\$condition</b>	mixed	query condition or criteria. If a string, it is treated as query condition (the WHERE clause); If an array, it is treated as the initial values for constructing a <code>CDbCriteria</code> object; Otherwise, it should be an instance of <code>CDbCriteria</code> .
<b>\$params</b>	array	parameters to be bound to an SQL statement. This is only used when the first parameter is a string (query condition). In other cases, please use <code>CDbCriteria::params</code> to set parameters.
<b>{return}</b>	CActiveRecord	the record found. Null if no record is found.

**Figure 8.2:** The class specification for the `find()` method.

There's also `findAll()`, `findAllByAttributes()`, `findAllByPk()`, `findAllBySql()`, `findByAttributes()`, `findByPk()`, and `findBySql()`. All of these methods can be grouped into two categories: methods that return every record, and methods that only return one (at most). Beyond that distinction, the methods differ in how they go about selecting the record(s) to be returned.

To use any of the `find*`() methods, you must comprehend the arguments they take. Many of these methods take one argument, which is used to set the selection's conditions, and another argument for passing condition parameters. The conditions argument can be a string, an array, or a `CDbCriteria` object. Let's look at some examples.

If the conditions value is a simple string, it will be used as the clause following `WHERE`. This line is equivalent to the `findByPk()` method:

```
// Works, but dangerous:  
$model = Page::model()->find("id=$id");
```

With both that line of code and the `findByPk()`, the result is the query `SELECT * FROM page WHERE id=X` (as in Figure 8.1). But there is one important difference between that use of `find()` and the use of `findByPk()`: this last bit of code leaves you open to *SQL injection attacks*. A more secure solution is to use a parameter:

```
// Works safely:  
$model = Page::model()->find('id=:id', array(':id'=>$id));
```

The difference may be subtle between the two, but the latter is more secure as the query won't break even if an inappropriate ID value is provided. Notice that this example uses a *named* parameter—`:id`, which must match the index value of the parameters argument. Also, the parameters argument is only ever used in an example like this, where the condition is a string. The condition can also be set as an array or as a `CDbCriteria` object. The array just maps to the `CDbCriteria` object, so let's look at `CDbCriteria` next.

## Using `CDbCriteria`

The `CDbCriteria` class lets you customize queries through an object. To start, create a `CDbCriteria` instance:

```
$criteria = new CDbCriteria();
```

Then you can customize it by assigning values to various properties, the most important of which are listed in the following table.

Property	Sets
condition	The WHERE clause
limit	The LIMIT value
offset	The offset value in a LIMIT
order	The ORDER BY clause
params	The variables to be bound to the parameters
select	The columns to be selected

There are also properties for grouping and aggregating results, to be discussed later in the book.

{TIP} The first thing you should do to become more comfortable with Active Record is master usage of CDbCriteria.

As an easy example to begin, the same `findByPk()` query can be accomplished in this manner:

```
$criteria = new CDbCriteria();
$criteria->condition = 'id=:id';
$criteria->params = array(':id'=>$id);
$model = Page::model()->find($criteria);
```

To perform the same query using `find()` without formally creating a `CDbCriteria` object, just pass `find()` an array equivalent to what you would do with `CDbCriteria`:

```
$model = Page::model()->find(array('condition' => 'id=:id',
    'params' => array(':id'=>$id)));
```

{NOTE} When you're finding a record using the primary key, it makes the most sense to use the `findByPk()` method. These other examples are for simple, comparative demonstrations.

As another example, this code might be used as part of the login process:

```
$criteria = new CDbCriteria();
$criteria->select = 'id, username';
$criteria->condition = 'email=:email AND pass=:pass';
$criteria->params = array(':email'=>$email, ':pass'=>$pass);
$model = User::model()->find($criteria);
```

## Retrieving Multiple Records

The `find()` method only ever returns a single row (at most). If multiple rows should be returned by a query, use `findAll()` instead. Its signature is the same (**Figure 8.3**).

<b>findAll() method</b>		
<pre>public array findAll(mixed \$condition='', array \$params=array( ))</pre>		
<b>\$condition</b>	mixed	query condition or criteria.
<b>\$params</b>	array	parameters to be bound to an SQL statement.
<b>{return}</b>	array	list of active records satisfying the specified condition. An empty array is returned if none is found.

**Figure 8.3:** The class specification for the `findAll()` method.

The `findAll()` method will return an array of objects, if one or more records match. If no records match, `findAll()` returns an empty array. This differs from `find()`, which returns `NULL` if no match was found.

## Deleting Records

So far, you've seen how to perform INSERT, UPDATE, and SELECT queries, using different Active Record methods. Sometimes you'll also need to run DELETE queries. This is easily done.

If you have a model instance, you can remove the associated record by invoking the `delete()` method on the object:

```
$model = Page::model()->findPk($id);  
$model->delete();
```

*{TIP}* The model object and its values (stored in its attributes) remain until the variable is unset by PHP (e.g., when a function or script terminates).

If you don't yet have a model instance, you can remove the record by calling `deleteByPk()` or `deleteAll()`:

```
$model = Page::model()->deleteByPk($id);
```

Or:

```
$criteria = new CDbCriteria();
$criteria->condition = 'email=:email';
$criteria->params = array(':email'=>$email);
$model = User::model()->deleteAll($criteria);
```

As you can see in that second example, the `deleteAll()` method takes the same arguments as `find()` or `findAll()`.

{TIP} The `updateAll()` method can be used like `deleteAll()` to update multiple records at once.

## Counting Records

Sometimes, you don't actually need to return rows of data, but just determine how many rows apply to the given criteria. If you just want to see how many rows *would be* found by a query, use Active Record's `count()` method. It takes the criteria as the first argument and parameters as the second, just like `find()`:

```
// Find the number of "live" pages:
$criteria = new CDbCriteria();
$criteria->condition = 'live=1';
$count = Page::model()->count($criteria);
```

This is equivalent to running a `SELECT COUNT(*) FROM page WHERE live=1` query and fetching the result into a number.

If you don't care how many rows would be returned, but just want to confirm that at least one would be, use the `exists()` method:

```
$criteria = new CDbCriteria();
$criteria->condition = 'email=:email';
$criteria->params = array(':email'=>$email);
if (User::model()->exists($criteria)) {
    $message = 'That email address has already been
               registered.';
} else {
    $message = 'That email address is available.';
}
```

## Working With Primary Keys

Normally, the primary key in a table is a single unsigned, not NULL integer, set to automatically increment. When a query does not provide a primary key value—which it almost always shouldn't, the database will use the next logical value. In

traditional, non-framework PHP, you're often in situations where you'll immediately need to know the automatically generated primary key value for the record just created. With MySQL, that's accomplished by invoking the `mysqli_insert_id()` or similar function.

In Yii, it's so simple and obvious to find this value that many people don't know how. After saving the record, just reference the primary key property:

```
$model->save();
// Use $model->id
```

It's that simple.

## Scopes

Web sites will commonly use SELECT queries repeatedly configured in the same manner. For example, a site might want to show the most recently posted comments or the most active users. You already know how to write such queries using Active Record: configure a `CDbCriteria` object and provide it to the `findAll()` method.

Yii allows you to “bookmark” queries using *named scopes*. A named scope is a previously defined configuration associated with a name. Referencing that name invokes that same configuration.

{NOTE} Named scopes only apply to SELECT queries.

To define a named scope, create a `scopes()` method in your model definition (or edit an existing one, if applicable):

```
# protected/models/SomeModel.php
class SomeModel {
    // Other stuff.
    public function scopes() {
        // Definition.
    }
}
```

The method needs to return an array. The array's indexes should be the name of the scope, and its values should be an array of criteria:

```
# protected/models/Comment.php
public function scopes() {
    return array(
        'recent' => array(

```

```
        'order' => 'date_entered DESC',
        'limit' => 5
    );
}
}
```

To use a named scope, reference the scope as if it were a class method, just before the `find()` or `findAll()`:

```
$comments = Comment::model()->recent()->findAll();
```

The end result will be the criteria identified in the named scope passed to the `findAll()` method as if it were the first argument to the `findAll()` call.

{TIP} You can parameterize a named scope: define it so that you can change a value used in the scope. See the [Yii Guide](#) for details.

Sometimes you'll have criteria that should be applied to every query. In that case, create (or edit) the `defaultScope()` method of the model class:

```
# protected/models/Page.php
public function defaultScope() {
    return array (
        'condition' => 'live=1',
        'limit' => 5,
        'order' => 'date_published DESC'
    );
}
```

Notice that the default scope just returns an array of criteria, not a multidimensional array as in `scopes()`.

With that particular default scope, for the most recently published, live pages, you'd probably want to create a named scope for other queries as well. For example, you would want other scopes for showing non-live pages (for the administrator).

## Performing Relational Queries

The use of Active Record to this point has largely been for retrieving records from a single table, but in modern relational databases, it's rarely that simple. Active Record is quite capable of performing JOINs—selecting data across multiple tables, it's just a question of how (as always). I briefly introduced this concept in Chapter 7, “[Working with Controllers](#),” but now it's time to cover the subject in sufficient detail.

{NOTE} You can download my code from your account page at the book's Web site.

The first thing you'll need to do is make sure you've properly identified all of your model (and therefore, table) relationships in your model definitions. Chapter 5 went through this in detail. Remember that the names assigned to the defined relationships will be used when performing queries. Once you've done that, you can use *lazy loading* or *eager loading* in Active Record to reference values from related tables. This was explained in Chapter 7. Here's the difference in terms of code:

```
// Lazy:  
$pages = Page::model()->findAll();  
$user = $page->user->username;  
// Eager:  
$pages = Page::model()->with('user')->findAll();  
$user = $page->user->username;
```

Of the two approaches, eager loading is clearly better for a couple of reasons. First, if you know you'll want access to related data, you should overtly request it (i.e., it's bad programming form to rely upon lazy loading). Second, eager loading is far more efficient. When Yii performs lazy loading, it runs two separate SELECT queries: in the above, one on the `page` table and another on `user`. With eager loading, Yii actually performs a JOIN across the two, resulting in a single SELECT query.

{NOTE} Lazy and eager loading work on any of the `find*` methods.

Getting a bit more complicated, you can use `with()` to JOIN *multiple* tables. Just pass the other relation names to `with()`, separated by commas:

```
$page = Page::model()->with('comments', 'user')->findByPk($id);  
$user = $page->user->username;
```

But what will that query return? Obviously that query returns a single `page` record, along with all the comments associated with that page (because the "comments" relationship is defined within the `Page` model). But the "user" reference may cause confusion because both the `Comment` model and the `Page` model have a relationship named "user". To understand the result, let's investigate the executed query thanks to `CWebLogRoute` (**Figure 8.4**).

If you look at that query, you'll noticed a couple of things. For one thing, Yii uses aliases to the nth degree. That's fine, but you do need to know that Yii uses the alias "t" for the primary table by default. In this case, that's `page`.

Moving on, the first JOIN is:

```
Querying SQL: SELECT `t`.`id` AS `t0_c0`, `t`.`user_id` AS `t0_c1`,
`t`.`live` AS `t0_c2`, `t`.`title` AS `t0_c3`, `t`.`content` AS `t0_c4`,
`t`.`date_updated` AS `t0_c5`, `t`.`date_published` AS `t0_c6`,
`comments`.`id` AS `t1_c0`, `comments`.`user_id` AS `t1_c1`,
`comments`.`page_id` AS `t1_c2`, `comments`.`comment` AS `t1_c3`,
`comments`.`date_entered` AS `t1_c4`, `user`.`id` AS `t2_c0`,
`user`.`username` AS `t2_c1`, `user`.`email` AS `t2_c2`, `user`.`pass` AS
`t2_c3`, `user`.`type` AS `t2_c4`, `user`.`date_entered` AS `t2_c5` FROM
`page` `t` LEFT OUTER JOIN `comment` `comments` ON
(`comments`.`page_id`=`t`.`id`) LEFT OUTER JOIN `user` `user` ON
(`t`.`user_id`=`user`.`id`) WHERE (`t`.`id`='1')
in /Users/larryullman/Sites/htdocs/protected/controllers/PageController.php
(55)
in /Users/larryullman/Sites/htdocs/index.php (13)
```

**Figure 8.4:** A JOIN across three tables.

```
`page` `t` LEFT OUTER JOIN `comment` `comments`  
ON (`comments`.`page_id`=`t`.`id`)
```

This is a LEFT OUTER JOIN, which means that the `page` record will be returned whether or not there are comments for it (which is what you would want). An INNER JOIN would only return the `page` record if it also had comments (which is what you don't want). The JOIN equates the `comments.page_id` column with `page.id`, which is correct.

Now let's look at the next JOIN. It joins the previous results with `LEFT OUTER JOIN user user ON (t.user_id=user.id)`. There is probably at least one problem with this JOIN. First, this query is returning the user that owns this page, as you can tell by this second LEFT OUTER JOIN's clause. That may be what you want, but what if you actually wanted the user that posted each comment? To do that, you must specify which "user" relationship you want:

```
$page = Page::model()->with('comments', 'comments.user')->findPk($id);  
$user = $page->user->username;
```

That's a solution, but the query should actually retrieve *both* users: the one that owns the page and the user associated with each comment. In theory, you could do this:

```
$page = Page::model()->with('user', 'comments',  
    'comments.user')->findPk($id);  
$user = $page->user->username;
```

In my opinion, the clearest way to handle this situation is to make sure that no two models use the same relationship names. For example, if you change your model definitions to:

```
# protected/models/Page.php::relations()
return array(
    'pageComments' => array(self::HAS_MANY,
        'Comment', 'page_id'),
    'pageUser' => array(self::BELONGS_TO,
        'User', 'user_id'),
    'pageFiles' => array(self::MANY_MANY, 'File',
        'page_has_file(page_id, file_id)'),
);
```

And:

```
# protected/models/Comment.php::relations()
return array(
    'commentPage' => array(self::BELONGS_TO,
        'Page', 'page_id'),
    'commentUser' => array(self::BELONGS_TO,
        'User', 'user_id'),
);
```

Then this works, and is much more clear:

```
$page = Page::model()->with('pageUser', 'pageComments',
    'pageComments.commentUser')->findPk($id);
// Note the change from "user" to "pageUser":
$user = $page->pageUser->username;
```

But there's a secondary problem with this particular example: neither a page nor a comment can be created *without* an associated user. An OUTER JOIN from `comment` to `user` or from `page` to `user` is imprecise. Both should be INNER JOINS. To fix that, you need to know how to customize relational queries.

## Customizing Relational Queries

Providing the names of relationships to the `with()` method is an easy and direct way to perform a JOIN and fetch the information that you need. But you'll inevitably need to know how to customize the resulting queries. For example, as just indicated, the default is for OUTER JOINS to be performed, which is not always appropriate. Or, as another example, if you know you will only be needing certain bits of information, there's no reason to select every column from the related table.

*{TIP}* Every SELECT query you run should ideally be limited to selecting only the information you actually need.

To customize the related query, pass an array to `with()`. The array's index should be the relation name; the array's value should be the customization. This should be a series of name=>value pairs. The allowed names are listed in the following table, and are similar to the attributes used with `CDbCriteria`.

Index	Sets
alias	An alias for the related table
condition	The WHERE clause
group	A GROUP BY clause
having	The HAVING clause
join	An additional JOIN clause
joinType	The type of JOIN to perform
limit	The LIMIT value
on	An ON clause
offset	The offset value in a LIMIT clause
order	The ORDER BY clause
params	The variables to be bound to the parameters
scopes	The scope to use
select	The columns to be selected

For example, here's how a `Page` query would also just select the page owner's username:

```
$page = Page::model()->with(array('pageUser' =>
    array('select'=>'username'))
)->findPk($id);
```

{TIP} As a reminder, use the output from `CWebLogRoute` to verify what query is actually being executed.

All that code is doing is customizing how the related `User` model is fetched along with `Page`. Every column from the `page` table is retrieved, along with the matching `user` record, but only the `username` column is retrieved from `user`.

In some situations, you don't need to actually fetch the related models, but you want to use them in some manner. For example, a query may only want to fetch the pages that have comments. In that case, set "select" to false:

```
$pages = Page::model()->with(array('pageComments' =>
    array('select'=>false)))->findAll();
```

As another example, if you wanted to fetch the associated comments in order of ascending comment date, you would do this:

```
$page = Page::model()->with(array('pageComments' =>
    array('order'=>'date_entered')))->findByPk($id);
```

You can use scopes with related models, too. Perhaps you've defined a "recent" scope in `Comment` that fetches the five most recently entered comments. In other words, that scope orders the selection by the `date_entered` DESC, and applies a LIMIT of 5. You could use that in your relational query:

```
$page = Page::model()->with(array('pageComments' =>
    array('scopes'=>'recent'))
))->findByPk($id);
```

(To be clear, because of the JOIN across `page` and `comment`, the scope should return the most recently entered five comments associated with this page.)

If comments had an `approved` column, you could factor that in:

```
$page = Page::model()->with(array(
    'pageComments' => array('scopes'=>'recent',
        'condition' => 'approved=1')
))->findByPk($id);
```

To apply a scope to the primary table (i.e., the target model) while using a relational query, invoke the primary table's scope *before* the `with()`:

```
$page = Page::model()->scopeName()->with(array(
    'pageComments' => array('scopes'=>'recentComments',
        'condition' => 'approved=1')
))->findByPk($id);
```

Moving on, the "joinType" index can be used to specify the type of JOIN to be performed across the two tables. The default, as you've already seen, is a LEFT OUTER JOIN. Let's make that an INNER JOIN where appropriate:

```
$page = Page::model()->with(array(
    'pageUser' => array('joinType' => 'INNER JOIN'),
    'pageComments',
    'pageComments.commentUser' =>
        array('joinType' => 'INNER JOIN')
))->findByPk($id);
```

## Preventing Column Ambiguity

With JOINS, a common problem is a database error complaining about an ambiguous column name. Relational databases often use the same name in related tables; using that name in a SELECT, WHERE, or ORDER clause causes confusion. Preventing such errors is easily done using the dot syntax: `table_name.column_name`.

The trick to doing this in Yii is that Yii automatically creates aliases for table and column names. In order to use the proper dot syntax, you must understand Yii's system, which is simple:

- The primary table's alias is “t”
- The alias for any other table is the relationship name

That is all.

## Statistical Queries

Another common need with relational queries are “statistical queries”: where information *about* related records is needed, not the related data itself. With the CMS example, you might want to find out:

- How many pages a user has created
- How many comments a user has posted
- How many comments a page has
- How many files a user has uploaded

When hand coding SQL queries, you'd use a COUNT() to fetch this information. When using Active Record, you instead create a new relation identifying the particular situation, specifying the relationship as STAT:

```
# protected/models/Page.php::relations()
return array(
    'pageComments' => array(self::HAS_MANY,
        'Comment', 'page_id'),
    'pageUser' => array(self::BELONGS_TO,
        'User', 'user_id'),
    'pageFiles' => array(self::MANY_MANY, 'File',
        'page_has_file(page_id, file_id)'),
    'commentCount' => array(self::STAT,
        'Comment', 'page_id')
);
```

{NOTE} Statistical queries can only be run when there is a HAS\_MANY or MANY\_MANY relationship between the two models.

With the new relationship defined, you can use it like you would any other relationship declaration. This code retrieves a single page and the number of comments for that page:

```
$page = Page::model()->with('commentCount')->findByPk($id);
```

Now `$page->commentCount` will represent the statistical value (i.e., COUNT(\*)).

You can further customize how statistical queries are executed, such as changing the selection from COUNT(\*) to something else, or adding a conditional. See the [Yii Guide](#) for details.

## Using Query Builder

Active Record provides the most common ways to interact with the database in Yii, but not the only way. Yii's Query Builder is the first logical alternative to Active Record that you'll use. Query Builder is a system for using objects to create and execute SQL commands. It's best used for building up dynamic SQL commands on the fly. Although Query Builder is a different beast than Active Record, many of the same ideas, and even the `CDbCriteria` class will apply to Query Builder, too.

{NOTE} Like most things in Yii, there's more than one way to use Query Builder. I'll present the most direct, easy to understand approach here.

### Simple Queries

Whereas all the Active Record interactions go through the model classes or objects, Query Builder works through `Yii::app()->db`. `Yii::app()->db` refers to the "db" component customized in the configuration file. One of the "db" component's attributes is a `CDbConnection` instance, which provides the database connection. Another is `CDbCommand`, used to make an SQL command.

To start using Query Builder, you create a new `CDbCommand` object:

```
$cmd = Yii::app()->db->createCommand();
```

To execute simple queries—those that don't return results, invoke the corresponding method on the `CDbCommand` object:

- `delete()`

- `insert()`
- `update()`

These methods all take the table name as the first argument. The `delete()` method takes the WHERE condition as its second, and bound parameters as its third:

```
$cmd->delete('file', 'id=:id', array(':id'=>$id));
```

The `insert()` method takes an array of column=>value pairs as its second argument:

```
$cmd->insert('some_table',
    array('some_col'=>$val1, 'num_col' => $val2));
```

Understand that Yii will automatically take care of parameter binding for you, so you don't have to worry about SQL injection attacks when using this approach.

The `update()` method takes an array of column=>value pairs as its second argument, the WHERE condition as its third, and bound parameters as its fourth:

```
$cmd->update('some_table',
    array('some_col'=>':col1', 'num_col' => ':col2'),
    ':id=:id',
    array(':col1' => 'blah', ':col2' => 43, ':id'=>$id));
```

{WARNING} If you insert or update records using Query Builder, you don't get the benefits of data validation that Active Record offers.

All three methods return the number of records affected by the action.

## Building SELECT Queries

Where Query Builder really shines is in SELECT queries. These are built up by assigning values to any number of command properties.

Property	Sets or returns
from	The tables to be used
join	A JOIN
limit	A LIMIT clause without an offset
offset	A LIMIT clause with an offset
order	The ORDER BY clause
select	The columns to be selected
where	A WHERE clause

{NOTE} There are also properties for creating more complex queries that use GROUP BY clauses, UNIONs, and so forth. I'll get to those later.

For an easy example, and one that's *not* a good use of Query Builder, let's retrieve the title and content for the most recently updated live page record (**Figure 8.5**):

```
$cmd->select = 'title, content';
$cmd->from = 'page';
$cmd->where = 'live=1';
$cmd->order = 'date_published DESC';
$cmd->limit = '1';
```

```
Querying SQL: SELECT `title`, `content`
FROM `page`
WHERE live=1
ORDER BY `date_published` DESC LIMIT 1
in /Users/larryullman/Sites/htdocs/protected/controllers/SiteController.php
(118)
in /Users/larryullman/Sites/htdocs/index.php (13)
```

**Figure 8.5:** The resulting query, run in the browser.

Once you've defined the query, for SQL commands like SELECT that return results (or should), invoke `query()`.

```
$result = $cmd->query();
```

The `query()` method will return a `CDbDataReader` object, which you can use in a loop:

```
foreach ($result as $row) {
    // Use $row['column_name'] et al.
}
```

If your query is only going to return a single row, you can just use `queryRow()` instead:

```
$cmd = Yii::app()->db->createCommand();
$cmd->select = '*';
$cmd->from = 'user';
$cmd->where = "id=$id";
$row = $cmd->queryRow();
```

When you have a query that only returns a single *value*, you can use `queryScalar()`:

```
$cmd = Yii::app()->db->createCommand();
$cmd->select = 'COUNT(*)';
$cmd->from = 'page';
$num = $cmd->queryScalar();
```

## Using Methods Instead of Attributes

For every attribute you can use to customize a command, there's also a method. This earlier example:

```
$cmd->select = 'title, content';
$cmd->from = 'page';
$cmd->where = 'live=1';
$cmd->order = 'date_published DESC';
$cmd->limit = '1';
```

can also be written as:

```
$cmd->select('title, content');
$cmd->from('page');
$cmd->where('live=1');
$cmd->order('date_published DESC');
$cmd->limit('1');
```

The end result is the same; which you use is a matter of preference. Still, some people like the method approach because you can “chain” multiple method calls together, resulting in a single line of code:

```
$cmd->select('title, content')->from('page')->
where('live=1')->order('date_published DESC')->limit('1');
```

If you prefer more clarity, you can spread out the chaining over multiple lines:

```
$cmd->select('title, content')
->from('page')
->where('live=1')
->order('date_published DESC')
->limit('1');
```

This appears to be thoroughly unorthodox, but it's syntactically legitimate. But understand that this only works if you omit the semicolons for all but the final line.

You can even combine the creation of the command object and its execution on the database into one step:

```
$num = Yii::app()->db->createCommand()
    ->select('COUNT(*)')
    ->from('page')
    ->queryScalar();
```

{TIP} If you want one more way to use Query Builder, you can pass an array of attribute=>value pairs to the `createCommand()` method instead.

At any point in time, you can find the final query to be run (perhaps for debugging purposes) by referencing `$cmd->getText()` (**Figure 8.6**):

```
echo $cmd->getText();
```

```
SELECT `title`, `content` FROM `page` WHERE live=1 ORDER BY `date_published` DESC LIMIT 1
```

**Figure 8.6:** The complete and actual query that was run.

## Setting Multiple WHERE Clauses

If you're dynamically building up a query, you might be dynamically defining the WHERE clause, too. For example, you might have an advanced search page that allows the user to choose what criteria to include in the search. The `where` attribute of the `CDbCommand` object takes a string, so you could dynamically define that string. Or you could let Yii do that for you.

`CDbCommand` defines two methods for building up the WHERE clause: `andWhere()` and `orWhere()`. The former adds an AND clause to the exiting WHERE conditional, and the latter adds an OR:

```
$cmd = Yii::app()->db->createCommand();
$cmd->select = 'id, title';
$cmd->from = 'page';
$cmd->where = 'live=1';
if (isset($_POST['author'])) {
    $cmd->andWhere('user_id=:uid',
        array(':uid', $_POST['author']));
}
// And so on.
```

## Performing JOINS

The last thing to learn about Query Builder is how to perform JOINS. The `from` attribute or `from()` method takes the name of the initial table on which the SELECT query is being run. You can provide it with more than one table name to create a JOIN:

```
$cmd = Yii::app()->db->createCommand();
$cmd->select = 'page.id, title, username';
$cmd->from = 'page, user';
$cmd->where = 'page.user_id=user.id';
// And so on.
```

Or you can use the `from()` method:

```
$cmd = Yii::app()->db->createCommand();
$cmd->select('page.id, title, username');
$cmd->from('page, user');
$cmd->where('page.user_id=user.id');
// And so on.
```

The `from()` method (or attribute), as well as `select()`, `order()`, and others, can take its argument (or value) as a string or an array.

You can also use the `join()`, `leftJoin()`, `rightJoin()`, `crossJoin()`, and `naturalJoin()` methods to perform JOINS. The first three methods all take as their arguments: the name of the table to join, the conditions, and an array of parameters:

```
$cmd = Yii::app()->db->createCommand();
$cmd->select('page.id, title, username');
$cmd->from('page');
$cmd->join('user', 'page.user_id=user.id');
// And so on.
```

The `crossJoin` and `naturalJoin()` methods just take the name of the table being joined to as its lone argument.

## Using Database Access Objects

The third way you can interact with a database in Yii is via Data Access Objects (DAO). This is a wrapper to the PHP Data Objects (PDO). DAO provides the most direct way of interacting with the database in Yii, short of tossing out the framework altogether and invoking the database extension functions directly!

## Simple Queries

As with Query Builder, one starts by creating a `CDbCommand` object. Unlike with Query Builder, you should actually provide the SQL command to the constructor:

```
$q = 'SELECT * FROM table_name';
$cmd = Yii::app()->db->createCommand($q);
```

For simple queries, which do not return results, invoke the `execute()` method to actually run the command:

```
$q = 'DELETE FROM table_name WHERE id=1';
$cmd = Yii::app()->db->createCommand($q);
$cmd->execute();
```

This method returns the number of rows affected by the query:

```
if ($cmd->execute() === 1) {
    $msg = 'The row was deleted.';
} else {
    $msg = 'The row could not be deleted';
}
```

## SELECT Queries

SELECT queries return results, and are therefore not run through `execute()`. There are many ways you can execute a SELECT query and handle the results. One option is to use the `query()` method:

```
$q = 'SELECT * FROM table_name';
$cmd = Yii::app()->db->createCommand($q);
$result = $cmd->query();
```

The `query()` method will return a `CDbDataReader` object, which you can use in a loop:

```
foreach ($result as $row) {
    // Use $row['column_name'] et al.
}
```

If your query is only going to return a single row, you can just use `queryRow()` instead:

```
$q = 'SELECT * FROM table_name WHERE id=1';
$cmd = Yii::app()->db->createCommand($q);
$row = $cmd->queryRow();
```

When you have a query that only returns a single value, you can use `queryScalar()`:

```
$q = 'SELECT COUNT(*) FROM table_name';
$cmd = Yii::app()->db->createCommand($q);
$num = $cmd->queryScalar();
```

## Returning Objects

Except for `queryScalar()`, the mentioned methods all result in arrays (e.g., within the `foreach` loop, `$row` will be an array, `queryRow()` returns an array, etc.). If you'd rather fetch results into an object, set the fetch mode:

```
$q = 'SELECT * FROM page WHERE id=1';
$cmd = Yii::app()->db->createCommand($q);
$cmd->setFetchMode(PDO::FETCH_CLASS, 'Page');
$model = $cmd->queryRow();
// Use $model->title et al.
```

As you can see in that code, this allows you to fetch results as an object type of your choosing.

## Parameter Binding

In order to use DAO securely, you must use bound parameters to prevent SQL injection attacks. Unlike Active Record and Query Builder, DAO will not take the necessary steps on its own. To do so, first make sure that the “`emulatePrepare`” option is set to true in your database configuration:

```
# protected/config/main.php
// Lots of other stuff.
'db'=>array(
    'connectionString' =>
        'mysql:host=localhost;dbname=test',
    'emulatePrepare' => true,
```

{WARNING} It was reported to me that setting “`emulatePrepare`” to true causes bugs in PostgreSQL (thanks, David!).

With that set, you can use bound parameters by using named or unnamed parameters (i.e., question marks) in place of variables (or values), in your query. I would recommend you go the named route. Use unique identifiers as the placeholders in your query, and then bind those to variables using the `bindParam()` method:

```
$q = 'INSERT INTO table_name (col1, col2)
      VALUES (:col1, :col2)';
$cmd = Yii::app()->db->createCommand($q);
$cmd->bindParam(':col1', $some_var, PDO::PARAM_STR);
$cmd->bindParam(':col2', $other_var, PDO::PARAM_INT);
$cmd->execute();
```

{TIP} Depending upon a few factors, there may also be a performance benefit to using bound parameters.

The data type is identified by a PDO constant:

- `PDO::PARAM_BOOL`
- `PDO::PARAM_INT`
- `PDO::PARAM_STR`
- `PDO::PARAM_LOB` (large object)

This is an example of binding *input parameters*; you can also perform outbound parameters, too (see the [Guide](#) for details).

## Choosing an Interface Option

Now that you've seen the three main approaches for interacting with the database—Active Record, Query Builder, and Data Access Objects, how do you decide which to use and when?

Active Record has many benefits. It:

- Creates usable model objects
- Has built-in validation
- Requires no knowledge or direct invocation of any SQL
- Handles the quoting of values automatically
- Prevents SQL Injection attacks automatically via parameters
- Supports behaviors and events

All of these benefits come at a price, however: Active Record is the slowest and least efficient way to interact with the database. This is because Active Record has to perform queries to learn about the structure of the underlying database table.

A second reason not to use Active Record is that, as is the case with many things, using it for very basic tasks is a snap, but using it for more complex situations can be a challenge.

But before giving up on Active Record entirely, remember that Yii does have ways of improving performance (e.g., using caching), and that ease of development is one of the main reasons to use a framework anyway. In short, when you need to work with model objects and are performing basic tasks, try to stick with Active Record, but be certain to implement caching.

*{TIP}* One rule of thumb is to stick with Active Record for creating, updating, and deleting records, and for selecting under 20 at a time.

Query Builder's benefits are that it:

- Handles the quoting of values automatically
- Prevents SQL injection attacks automatically via parameters
- Allows you to perform JOINs easily, without messing with Active Record's relations
- Offers generally better performance than Active Record

The downsides to Query Builder are that it's a bit more complicated to use and does not return usable objects. Query Builder is recommended when you have dynamic queries that you might build on the fly based upon certain criteria.

Finally, there's Direct Access Objects. With DAO, you're really just using PDO, which might be enough of a benefit for you, particularly when you're having trouble getting something to work using Active Record or Query Builder. Other benefits include:

- Probably the best performance of the three options (depending upon many factors)
- Ability to use the SQL you've known and loved for years
- Ability to fetch into specific object types
- Ability to fetch records into arrays, for easy and fast access

On the other hand, DAO does not provide the other benefits of Active Record, is not as easy for creating dynamic queries on the fly as with Query Builder, and does not automatically prevent SQL injection attacks via bound parameters. I would recommend using DAO when you have an especially tough, complex query that you're having a hard time getting working using the other approaches.

## Common Challenges

To conclude this chapter, I thought I would cover a couple of specific, common challenges when it comes to working with databases. These challenges are independent of the database approach in use: Active Record, Query Builder, and DAO.

At this point in the book's progression, I've only come up with two challenges that haven't already been covered: performing transactions and using database functions. Other possible topics have been moved to Part 3 of the book. If, after reading this chapter, something's still not clear, let me know and I'll see about adding coverage of that topic to subsequent updates of the book.

### Performing Transactions

In relational databases, there are often situations in which you ought to make use of transactions. Transactions allow you to only enact a sequence of SQL commands if they all succeed, or undo them all upon failure.

Transactions are started in Yii by calling the `CDbConnection` object's `beginTransaction()` method. That's accessible through the "db" component:

```
$trans = Yii::app()->db->beginTransaction();
```

Then you proceed to execute your queries and call `commit()` to enact them all or `rollback()` to undo them all. It would make sense to execute your code within a `try...catch` block in order to most easily know when the queries should be undone:

```
$trans = Yii::app()->db->beginTransaction();
try {
    // All your SQL commands.
    // If you got to this point, no exceptions occurred!
    $trans->commit();
} catch (Exception $e) {
    // Use $e.
    // Undo the commands:
    $trans->rollback();
}
```

That is the code you would use with Query Builder or DAO. To use transactions with Active Record, begin the transaction through the model's `dbConnection` property:

```
$model = new SomeModel;
$trans = $model->dbConnection->beginTransaction();
```

Everything else is the same.

Know that, depending upon the database application in use, certain commands have the impact of automatically committing the commands to that point. See your database application's documentation for specifics.

{NOTE} MySQL only supports transactions when using specific storage engines, such as InnoDB. The MyISAM storage engine does not support transactions.

## Using CDbExpression

Many, if not most, queries use database function calls for values. For example, the `date_updated` column in the `file` table would be set to the current timestamp upon update. You can do this in your SQL command for the table, depending upon the database application in use and the specific version of that database application, but you would also normally just invoke the `NOW()` function for that purpose (in MySQL):

```
UPDATE file SET date_updated=NOW(), /* etc. */ WHERE id=42
```

In theory, you might think you could just do this in Yii:

```
# protected/models/File.php beforeSave()
$this->date_updated = 'NOW();
```

But that won't work, for a good reason: for security purposes, Yii sanitizes data used for values in its Active Record and Query Builder queries (Yii does so with table and column names, too). Thus, the string '`NOW()`' will be treated as a literal string, not a MySQL function call.

In order to use a MySQL (or other database application) function call, you must use a `CDbExpression` object for the value. That syntax is:

```
# protected/models/File.php beforeSave()
$this->date_updated = new CDbExpression('NOW());
```

If the MySQL function takes an argument, such as the password to be hashed, use parameters:

```
# protected/models/User.php beforeSave()
$this->pass = new CDbExpression('SHA2(:pass)', 
    array(':pass' => $this->pass));
```

As another example, if you wanted to get a random record from the database, you would use a `CDbExpression` for the ORDER BY value:

```
$cmd = Yii::app()->db->createCommand();
$cmd->select = '*';
$cmd->from = 'user';
$cmd->order = new CDbExpression('RAND()');
$cmd->limit = 1;
$row = $cmd->queryRow();
```

To select a formatted date, use the `DATE_FORMAT()` call as part of the selection:

```
$cmd = Yii::app()->db->createCommand();
$cmd->select = array('*',
    new CDbExpression('DATE_FORMAT(date_entered, "%Y-%m-%d")')
);
$cmd->from = 'user';
$cmd->order = new CDbExpression('RAND()');
$cmd->limit = 1;
$row = $cmd->queryRow();
```

## Chapter 9

# WORKING WITH FORMS

HTML forms are one of the key pieces of any Web site. As is the case with many things, creating forms while using a framework such as Yii is significantly different than creating forms using standard HTML alone. In this chapter, you'll learn what you need to know to create HTML forms when using the Yii framework. You'll comprehend the fundamentals of forms in Yii and see a few recipes for common form needs.

## Understanding Forms and MVC

Before getting into the code, let's take a minute to think about the MVC architecture. A form is part of the view component, as a form is an aspect of the user interface. Forms, though, are almost always associated with specific models. A contact form may have its own model, not tied to a database table (in which case the model extends `CFormModel`), and a form for employees or departments will be based upon a model that is tied to a database table (in which case the model extends `CActiveRecord`, most likely). Whether the model extends `CFormModel` or `CActiveRecord`, the important thing to remember is that the form is tied to a model. This is significant because it's the model that dictates what form elements should exist, controls validation of the form data, and even defines the form's labels (e.g., "First Name" for the `firstName` attribute), and so forth.

*{TIP}* With the MVC approach, forms should be associated with models.

There are situations where you might have a form not associated with a model, but that is extremely rare.

When you use Gii to auto-generate CRUD functionality for a model, the framework creates a form for you in a file named `_form.php`. This file in turn gets included by other view files (any view file in Yii that starts with an underscore is intended to be an include). Naturally, the controller dictates which primary view file gets

rendered. Also understand that the same `_form.php` file is intended to be used whether the form is for creating new records or updating existing ones. Yii will take care of populating the form's elements with the model's current value when an update is being performed.

Because forms are normally tied to models, you'll need access to a model instance when you go to create the form. Before getting to the view and its form, let's be clear as to how the view accesses the specific model. A controller may have this code:

```
public function actionCreate() {
    $model=new Page;
    /* Code for validation and redirect upon save. */
    // If not saved, render the create view:
    $this->render('create',array(
        'model'=>$model
    ));
}
```

The `create.php` view file will include `_form.php`, passing along the model instance in the process:

```
<?php echo $this->renderPartial('_form',
    array('model'=>$model)); ?>
```

So now `_form.php` has access to the model instance and can create the form tied to that model. Once the form view file has access to the model, it can create form elements in one of two ways:

- Invoking the `CHtml` class methods directly
- Using the `CActiveForm` widget

I'll explain both approaches, but focus more on the later, which is the current default approach in Yii.

Of course, to be fair, you *could* create an HTML form using raw HTML, without Yii at all. The downside to that approach is it creates no tie-in between the model's validation rules, errors, labels, etc., and the form. By creating the form using Yii, labels will be based upon the model definitions (meaning that changing just the model changes reference to attributes everywhere), invalid form values can automatically be highlighted, and much, much more. Plus, it's not hard to use Yii to create a form, once you understand how.

## Creating Forms without Models

The older Yii approach for creating a form was simply a matter of invoking the appropriate `CHtml` methods. This class is used statically (i.e., not through a model instance), and defines everything you need to make form elements. To start, you would create the opening FORM tag:

```
<?php echo CHtml::beginForm(array('search'), 'get'); ?>
```

The method's first argument is the tag's "action" attribute value. Yii will turn this into an appropriate route. The above, for example, would be submitted to one of the following, depending upon the configuration:

- `http://www.example.com/index.php?r=search`
- `http://www.example.com/index.php/search`
- `http://www.example.com/search`

Thus, the form gets submitted to the search controller, and the default action of that controller.

The second argument to `beginForm()` is the value of the form's "method" attribute.

Next, start adding form elements. There's a method for each type. For example, the `label()` method creates an HTML LABEL (**Figure 9.1**).

### `label()` method

public static string <code>label(string \$label, string \$for, array \$htmlOptions=array( ))</code>		
<code>\$label</code>	string	label text. Note, you should HTML-encode the text if needed.
<code>\$for</code>	string	the ID of the HTML element that this label is associated with. If this is false, the 'for' attribute for the label tag will not be rendered.
<code>\$htmlOptions</code>	array	additional HTML attributes. The following HTML option is recognized: <ul style="list-style-type: none"><li>• required: if this is set and is true, the label will be styled with CSS class 'required' (customizable with <code>CHtml::\$requiredCss</code>), and be decorated with <code>CHtml::beforeRequiredLabel</code> and <code>CHtml::afterRequiredLabel</code>.</li></ul>
<code>{return}</code>	string	the generated label tag

**Figure 9.1:** The Yii class docs for the `CHtml::label()` method.

The `textField()` method creates a text input. Toss in a submit button, and you've got yourself a search box:

```
<?php echo CHtml::label('Search', 'terms'); ?>
<?php echo CHtml::textField('terms'); ?>
<?php echo CHtml::submitButton('Go!'); ?>
```

Finally, close the form:

```
<?php echo CHtml::endForm(); ?>
```

The end result would be the following HTML:

```
<form action="/index.php/page/search" method="get">
<label for="terms">Search</label>
<input type="text" value="" name="terms" id="terms" />
<input type="submit" name="yt0" value="Go!" />
</form>
```

It's not practical, or a good use of book space, to explain each `CHtml` method in detail in this book. Instead, I'll cover how you create forms in practice, and leave it up to you to look up the details and options in the [Yii class reference](#) for the `CHtml` class. Later in the chapter, I'll specifically walk through a few of the more common but difficult needs. But rather than leave you entirely on your own, there are a few things you ought to know up front about the `CHtml` methods...

## Using `CHtml`

When creating “action” attributes for your forms, they need to be mapped to proper routes for your site and controllers. To do so, Yii uses the `CHtml::normalizeUrl()` method, which does the following:

- Uses the current URL when an empty string is provided
- If a non-empty string is provided, that string is used as a URL without change
- If an array is provided, the array’s values are used as in the `CController::createUrl()` method, explained in Chapter 7, “[Working with Controllers](#)”

In case you don’t remember what I wrote in that chapter, basically, the first element in the array is treated as the action (e.g., “view”) or controller and action (e.g., “search/view”). Any other array elements will be used as parameters passed in the URL.

Next, you should know that many of the `CHtml` methods take a final argument that can be used to provide additional HTML attributes. For example, to apply a class to an input, you would do this:

```
<?php echo CHtml::textField('terms', '',
    array('class' => 'input-medium search-query')); ?>
```

(The second argument to the `textField()` method is its value.)

Third, `CHtml` has methods for creating useful elements that don't correspond to a single HTML form element. For example, the `checkBoxList()` method creates a sequence of checkboxes and `radioButtonList()` does the same thing with radio buttons (**Figure 9.2**):

```
<?php echo CHtml::label('Your Interests:', 'interests'); ?>
<?php echo CHtml::checkBoxList('interests', '',
    array('PHP', 'MySQL', 'JavaScript', 'CSS',
        'Yii Framework')); ?>
```

---

Your Interests:

PHP  
 MySQL  
 JavaScript  
 CSS  
 Yii Framework

**Figure 9.2:** The `checkBoxList()` method creates multiple checkboxes from an array.

Fourth, and similarly, `CHtml` defines methods related to other HTML aspects, not just forms. For example, the `cssFile()` method creates a link to a CSS file:

```
<?php echo CHtml::cssFile('css/mycss.css'); ?>
```

Or, the `image()` method creates an HTML IMG tag and `link()` creates an HTML A tag.

## Using “Active” Methods

The final thing you should know about `CHtml` is that each method in `CHtml` used to create form elements is repeated with a version prefaced with “active”: `activeLabel()`, `activeTextField()`, `activeSubmitButton()`, and so forth. The non-active versions are used *without* a model reference. The active versions all take a model instance as their first argument:

```
<div class="row">
    <?php echo CHtml::activeLabel($model, 'username'); ?>
    <?php echo CHtml::activeTextField($model, 'username') ?>
</div>
```

Note that each element's name, such as "username" in the above needs to be exactly the same as a corresponding attribute in the model. Doing so ties the model's definition—its labels, validation routines, and so forth—to the form elements. If you attempt to create an element that doesn't correlate to a model attribute, you'll get an error (**Figure 9.3**).

## CException

Property "User.name" is not defined.

**Figure 9.3:** An exception is thrown by attempting to create a form element without a matching model attribute.

And here's another benefit of tying a model to a form: if the form is being used for an update, the values will automatically be pre-populated/pre-selected/pre-checked based upon the existing model instance! As you should know, that alone requires a lot of code and logic in traditional programming.

## Using CActiveForm

Just using the "active" methods to tie a form to a model is great, but you can improve upon that approach. As of Yii 1.1.1, forms can be created using the `CActiveForm` class as a widget. Although widgets won't be formally covered until Chapter 12, "[Working with Widgets](#)," using `CActiveForm` as a widget is easy enough to grasp and implement that I can explain it here.

You always start using `CActiveForm` by invoking the controller's `beginWidget()` method. Provide to it the name of the widget class:

```
<?php $form = $this->beginWidget(' CActiveForm ') ; ?>
```

(That code goes in the `_form.php` file.) Now `$form` is an object of the `CActiveForm` type and it can be used to generate the form itself.

From there, the `CHtml` "active" methods have been wrapped within methods in the `CActiveForm` class, although you no longer have to use the "active" part. For example, `activeTextField()` is now just `textField()`. The `activeLabel()` becomes `labelEx()`, however:

```
<div>
    <?php echo $form->labelEx($model,'firstName'); ?>
    <?php echo $form->textField($model,'firstName',
        array('size'=>20,'maxlength'=>20)); ?>
    <?php echo $form->error($model,'firstName'); ?>
</div>
```

The model is still passed as the first argument to these methods. The model attribute involved is passed as the second.

By using `CActiveForm`, instead of just `CHtml`, you can now easily implement:

- Server-side validation
- Client-side validation via JavaScript
- Client-side validation via Ajax

The first two will be implemented automatically for you. Just test it for yourself to see. Disable JavaScript and test that, too. Chapter 14, “[JavaScript and jQuery](#),” will discuss Ajax and Ajax form validation.

Returning to the widget itself, you can customize the behavior of the form by passing an array of values when creating it:

```
<?php $form = $this->beginWidget(' CActiveForm', array(
    'id'=>'user-form',
    'enableAjaxValidation'=>false,
    'focus'=>array($model,'firstName'),
)); ?>
```

The various `CActiveForm` properties can be found in the documentation. Commonly you won’t need to customize any properties, but you can change the form’s “method” and “action” attributes, or add additional HTML to the opening FORM tag.

Finally, the form needs a submit button. Unlike the other form elements, this isn’t tied to a model; it is created with just the `CHtml` class:

```
<div>
    <?php echo CHtml::submitButton(
        $model->isNewRecord ? 'Create' : 'Save'); ?>
</div>
```

Then the form is closed by “ending” the widget:

```
<?php $this->endWidget(); ?>
```

Those are the basics for using `CActiveForm`. Once you've taken the above steps, form elements will be pre-populated when updating a record, errors will be clearly indicated upon form submission, and so forth.

## Using Form Builder

Another way of creating forms in Yii is to use the Form Builder. Form Builder in Yii is somewhat similar to using PEAR's [HTML\\_QuickForm](#), if you're familiar with that. Form Builder is useful if you want to create forms dynamically or if you'd just rather not rely upon so much hard coded HTML and PHP in your views. For an example of using Form Builder, I'll recreate a contact form.

### Getting Started

To start, you need to create a new `CForm` object. This would be done in a controller:

```
# protected/controllers/SiteController.php
public function actionContact() {
    $model = new ContactForm;
    $form = new CForm(<configuration>, $model);
}
```

When creating the `CForm` object, you need to pass to the constructor two values:

- Configuration details for the form
- An instance of the model to associate with the form

For the model instance, let's assume as an example that you want a contact form for the site (and don't want to use the view created by `yiic`). Here's part of the `ContactForm` model definition that Yii will create for you:

```
# protected/models/ContactForm.php
class ContactForm extends CFormModel {
    public $name;
    public $email;
    public $subject;
    public $body;
```

{TIP} For the sake of simplicity, I'm ignoring the captcha for this example.

Since the model has these four attributes, the form should have four elements, plus a submit button.

As with other forms tied to models, Form Builder will perform validation of the submitted data, populate the form with existing data when applicable, use the model's label values, and so forth.

The “configuration” argument is how you dictate what elements the form has. There are a couple of ways you can configure the form. The first is to pass the constructor an array:

```
# protected/controllers/SomeController.php
public function actionContact() {
    $model = new ContactForm;
    $form = new CForm(array(/* index=>value */), $model);
}
```

The second way to configure the `CForm` object is to have a file store an array and pass the path to that file to the constructor. There's a good argument to taking this approach, as it results in cleaner code that's easier to edit. As this information will eventually be used to create a view, it makes sense to put the array in a view file:

```
# protected/views/site/contactForm.php
<?php
return array(
    /* index => value */
);
```

Once you've created the file that returns an array, tell `CForm` to use it:

```
# protected/controllers/SiteController.php
public function actionContact() {
    $model = new ContactForm;
    $form = new CForm('application.views.site.contactForm',
        $model);
}
```

## Configuring `CForm`

However you pass an array to the `CForm` constructor, the key is what that array contains. The array's indexes will always include two values: “elements” and “buttons”. These two values correspond to the two categories into which `CForm` sorts all form elements. The submit button and a reset button (if you're using those, which you generally shouldn't) go into the latter category; everything else is an element.

You can also set indexes for any other writable property of the `CForm` class. This includes:

- `action`, dictating to what page the form should be submitted
- `method`, dictating the method (the default is POST)
- `title`, used as a legend value in a fieldset

To create elements and buttons, you assign an array to those indexes. Within that inner array, each form element is indexed by its name. Again, these names/indexes should match the corresponding model attributes. Here's what the code looks like considering what I've explained thus far:

```
return array(
    'title' => 'Contact Us',
    'elements' => array(
        'name' => array(),
        'email' => array(),
        'subject' => array(),
        'body' => array()
    ),
    'buttons' => array(
        'submit' => array()
    )
);
```

That code defines the form's title and then identifies four elements and one button. The four elements align with the four attributes in the model. The order in which the elements and buttons are listed dictates the order in which they are displayed in the form.

{TIP} You can create subforms by specifying another `CForm` object as an element type. The [Guide](#) has examples of this.

Next, the code creates the arrays for the individual elements and buttons. Each element and button is represented as an array, which should always have a "type" index. For element types, the possible values are:

- `text`
- `hidden`
- `password`
- `textarea`
- `file`
- `radio`

- checkbox
- listbox
- dropdownlist
- checkboxlist
- radiolist
- url
- email
- number
- range
- date

Note that there are again Yii variations here on common HTML elements, such as *checkboxlist* and *listbox*. The last five options are all new to HTML5. And, if that's not enough, you can use **CInputWidget** or **CJuiInputWidget** widgets for the types, too (widgets will be covered in Chapter 12).

For buttons, the possible types are:

- htmlButton
- htmlReset
- htmlSubmit
- submit
- button
- image
- reset
- link

The first three options all use HTML BUTTON tags. Generally speaking, you'll use the "submit" type.

The individual element and button arrays will have other indexes depending upon the item in question. For example, the "list" types—checkboxlist, dropdownlist, and radio list—all need "items". For buttons, you'll normally want to set the "label" index, which is the text that appears on the button. All of the other indexes that you can customize are the writable attributes of the **CFormInputElement** and **CFormButtonElement** classes. These classes use **CHtml** methods to actually create the various elements; your existing knowledge of **CHtml** applies to using **CForm**, too.

With all this in mind, here's the complete customization of the contact form (without the **return array()** part):

```
'title' => 'Contact Us',
'elements' => array(
    'name' => array(
        'type' => 'text',
```

```
        'maxlength' => '80'
    ),
    'email' => array(
        'type' => 'email',
        'hint' => 'If you want a reply, you must...',
        'maxlength' => '80'
    ),
    'subject' => array(
        'type' => 'text',
        'maxlength' => '120'
    ),
    'body' => array(
        'type' => 'textarea',
        'attributes' => array('rows'=>20, 'cols'=>80)
    )
),
'buttons' => array(
    'submit' => array(
        'type' => 'submit',
        'label' => 'Submit'
    )
)
```

{TIP} By default, the model's `attributeLabels()` values will be used for the labels for the elements. You can change this using the "label" index.

## Displaying the Form

Once you've created a customized `CForm` object, it's ready to be rendered in a view. First, in your controller, pass the `CForm` and model objects to the view file:

```
# protected/controllers/SiteController.php
public function actionContact() {
    $model = new ContactForm;
    $form = new CForm('application.views.site.contactForm',
        $model);
    $this->render('contact', array('model' => $model,
        'form' => $form));
}
```

Then, in the view file, just do this:

```
<?php echo $form; ?>
```

{TIP} You can customize how the form is displayed by extending the `CForm` class to override the `render()` method. See the [Guide](#) for details.

## Handling Form Submissions

Now that you've successfully displayed the form (hopefully), the final step is to handle the form's submission. This occurs in the controller, of course. You can check for a form's submission by calling the form object's `submitted()` method, providing to it the name of the submit button: `$form->submitted('submit')`. The default value is "submit", so if you name your button that, you can just use `$form->submitted()`. To see if the form data passed all validation (as defined in the model), invoke the `validate()` method.

Putting this altogether, your controller might look like this:

```
# protected/controllers/SiteController.php
public function actionContact() {
    $model = new ContactForm;
    $form = new CForm('application.views.site.contactForm',
        $model);
    if ($form->submitted() && $form->validate()) {
        // Send the email!
        $this->render('emailSent');
    } else {
        $this->render('contact', array('model' => $model,
            'form' => $form));
    }
}
```

Note that Yii will automatically take care of error reporting and making the form sticky should it be displayed again after a submission that does not pass validation.

The last thing you need to know is how to access the submitted data. That will be available in the model, through its attributes:

```
// Get the "to" from the Yii configuration file:
$to = Yii::app()->params['adminEmail'];
// Compose the body:
$body = "Name: {$model->name}
        Email: {$model->email}
        Message: {$model->body}";
mail($to, $model->subject, $body);
```

{WARNING} That code will send the email, but to make it more secure, I'd test and scrub the submitted values for possible spam attempts.

## Common Form Needs

This book is purposefully not intended as a recipe book, such as [Alexander Makarov's book](#), and my hope is that the material in this chapter to this point gives you enough information that you would be able to figure out whatever it is you need to do from this point forward. On the other hand, there are still a few common needs and points of confusion that could stand to be addressed. Over the rest of the chapter, I'll do just that.

If, after reading this chapter or working with forms on your own, you find that there are still some significant, outstanding mysteries when it comes to using forms, let me know. I can also add more of these "how to's" in subsequent releases.

{TIP} I'm not going to explain CAPTCHA in this chapter as it requires a widget. You'll see how to use it in Chapter 12.

### Working with Checkboxes

Checkboxes can sometimes be challenging to work with, due to the way they represent values. When you enter "lycanthropy" in a text box, you know that text box's value will be "lycanthropy". But when you check a checkbox, what value will it have? And, more importantly, how can that value be properly mapped to a model attribute?

The answer to the first question is simple: if a checkbox is checked, the resulting PHP variable will have the same value as the checkbox's "value" attribute:

```
Receive Updates?  
<input type="checkbox" name="updates" value="yes">
```

If that box is checked, then `$_POST['updates']` will have a value of "yes". But what happens if the checkbox is not checked? In that case, `$_POST['updates']` *will not have a value* (i.e., the variable won't be "set"). And this creates one problem: the database, and the corresponding model attribute, may use true/false, Y/N, or 1/0 to represent whether or not that checkbox was checked. You can easily set the affirmative value—true, Y, 1, but how do you set the negative (i.e., non-checked) value?

A second problem arises when updating models. You can pre-check a checkbox by adding the "checked" attribute to the element:

```
Receive Updates? <input type="checkbox" name="updates"
    value="yes" checked>
```

How do you make that happen when the model might use true, Y, or 1 for its affirmative values?

In order to answer all these questions, let's run through some specific examples, starting with simply accessing checkboxes.

{TIP} Most of the information with respect to checkboxes equally applies to radio buttons.

### Implementing “Remember Me?” Functionality

The code generated by the `yiic` script makes use of a checkbox already: the login form presents a “Remember Me?” element. This is an optional checkbox: the user can log in whether she checks the box or not. If the user does check the box, the login cookie will be extended.

The checkbox is created in the view form using this code:

```
<?php echo $form->checkBox($model, 'rememberMe') ; ?>
```

This checkbox is mapped to a model attribute. This means the controller can find the attribute's value using `$model->rememberMe`. But what will that value be if checked? What will it be if not checked?

The first thing you need to know about checkboxes in Yii is that if a checkbox is *not* provided with a value, its default value will be 1. In other words, Yii will set a checkbox's “value” attribute to 1, unless you specify otherwise. But Yii does something very clever: it creates a default value, too. The trick can be seen in the rendered HTML:

```
<input id="ytLoginForm_rememberMe" type="hidden" value="0"
    name="LoginForm[rememberMe]" />
<input name="LoginForm[rememberMe]"
    id="LoginForm_rememberMe" value="1" type="checkbox" />
```

First, there's a hidden input with a value of 0 and the name “`LoginForm[rememberMe]`”. Then comes the checkbox with a value of 1 *and the same name*. Whenever you have two form elements with the same name, the value of the second element will overwrite the value of the first. In this case, if the checkbox is checked, then `$_POST['LoginForm']['rememberMe']` ends up being 1, because the value of the checkbox overwrites the value of the hidden form element. If the checkbox is *not*

checked, then `$_POST['LoginForm']['rememberMe']` ends up being 0, because the checkbox will not be set, leaving the hidden form element's value intact. Clever stuff!

To change the checked and unchecked values, pass a third argument to the `checkBox()` method:

```
<?php echo $form->checkBox($model, 'rememberMe',
    array('value' => 'Y', 'uncheckValue'=>'N')); ?>
```

### Agreeing To Terms

The “remember me?” example is one in which the checking of the box is optional, but what if it’s required? A logical example is the pervasive “I agree to these terms (that I did not even consider reading)” checkbox.

Enforcing this requirement is simple, and is done in the same way you enforce any requirement on a model attribute: using the model rules. This code was presented in Chapter 4, “[Initial Customizations and Code Generations](#)”:

```
array('acceptTerms', 'required', 'requiredValue'=>1,
    'message'=>'You must accept the terms to register.'),
```

Assuming that your model has the `acceptTerms` attribute, and that your form creates that corresponding checkbox, the form now mandates that the user check the box (**Figure 9.4**).



**Figure 9.4:** The resulting error message if the terms checkbox is not checked by the user.

### Checkboxes and Updates

The last remaining question, then, is how to handle updates and checkboxes. In *theory*, you could just add the “checked” attribute to the view code that creates the form. But you would have to do this conditionally, based upon what value the corresponding model attribute has, if any.

Yii has predicted and solved this dilemma, however: the framework will automatically check the box for you if the corresponding model attribute has a “true” value, in PHP terms. This includes the Boolean true, as well as any non-zero number, as well as any non-empty string. If your attribute has any of the following values, Yii will check the box on updates:

- true
- 1
- ‘Yes’
- ‘Y’

That’s great, but the problem is that ‘N’, ‘No’, and ‘false’ would also qualify as “true” values.

If you’re using values for your checkboxes that do not easily correlate to Booleans, such as Y/N or Yes/No, the solution is to convert the values from non-Booleans to Booleans before rendering the form. You’ll want to *perform* that conversion in the controller, before an update occurs, but the particular functionality should be defined within the model itself. Here’s how I would do that...

To start, create a `convertToBooleans()` method. It should define an array of all attributes that need the conversion and then loop through that array. Within the loop, the attribute’s value should be converted to a Boolean:

```
# protected/models/SomeModel.php
public function convertToBooleans() {
    $attributes = array('receiveUpdates',
        'receiveOffers', ...);
    foreach ($attributes as $attr) {
        $this->$attr = ($this->$attr === 'Y') ? true : false;
    }
}
```

The list of strings must exactly match the names of the corresponding model attributes, but that’s all you need to do to make this work.

Now the controller should invoke this method, but only when an update is being performed:

```
# protected/controllers/SomeController.php
public function actionUpdate($id) {
    $model=$this->loadModel($id);
    $model->convertToBooleans();
    // Rest of the method (show & handle the form).
```

Now the checkbox(es) in the form will be automatically checked as appropriate. Note that even with that code, the form should end up assigning Y/N or Yes/No to the model attributes, as that’s what would be stored in the database:

```
<?php echo $form->checkBox($model, 'attributeName',
    array('value' => 'Y', 'uncheckValue'=>'N')); ?>
```

## Working with Passwords

If you have any kind of user-type model, you presumably have a password attribute, required for logging in. This alone leads to two problems to solve:

- Confirming the password during registration
- Securely storing the password

Let's quickly look at both dilemmas.

### Confirming Passwords

By now you've certainly been asked to confirm your password upon registration many thousands of times. The theory is that it ensures that the user knows exactly what her password was because she had to enter it twice. That's a theory, anyway. Before showing you how to do this in Yii, I'd like to put forth the case that it's really unnecessary.

Many sites are forgoing the password confirmation these days, and with good reason: it's an extra hassle that provides no extra security. Sure, in theory you'll know your password better because you entered it twice, but how many times have you done that just to forget it later anyway? A lot, right? So how about just taking the user's password once, and if the user forgets it, have a good "forget password" system in place. That's it. But if you still want to do a password confirmation...

Let's assume you have a `User` model, which extends `CActiveRecord`. In the database, there's a `pass` column for storing the password. The first thing you should do is add an attribute to the model itself:

```
# protected/models/User.php
class User extends CActiveRecord {
    public $passCompare; // Needed for registration!
```

Then, add a rule that says that the two password attributes must match:

```
# protected/models/User.php::rules()
return array(
    // Other rules.
    array('pass', 'compare', 'compareAttribute'=>'passCompare',
          'on'=>'insert'),
);
```

You'll notice that this rule only applies during the "insert" scenario. This means that the rule will only apply when the model is being saved for the first time (in the `actionCreate()` method of the `UserController`).

Next, your view file must also display the second password input:

```
<div class="row">
<?php echo $form->labelEx($model, 'passwordCompare'); ?>
<?php echo $form->passwordField($model, 'passwordCompare',
    array('size'=>60, 'maxlength'=>64)); ?>
</div>
```

And that will do it (**Figure 9.5**)!

The figure shows a registration form with two password input fields. The first field is labeled "Password" and contains six asterisks. The second field is labeled "Pass Compare" and is empty. A red error message "Password must be repeated exactly." is displayed between the fields.

**Figure 9.5:** Password confirmation is now part of the registration form (for better or for worse).

{TIP} If you want the ability for users to change their passwords (which only makes sense), change the password compare rule to also apply to the “update” scenario.

## Securing Passwords

Another common issue regarding passwords is how you secure them. Typically, one would store the *hashed* version of a password upon registration, not the actual password itself. Then, when the user goes to login, the submitted login password is hashed using the same code, and the two hashes are compared. The simplest way of doing all this is to use a MySQL function:

```
INSERT INTO users VALUES (NULL, 'trout', 't@example.com',
    SHA2('bypass', 512))
```

And:

```
SELECT * FROM users WHERE (email='t@example.com' AND
    pass=SHA2('bypass', 512))
```

(Obviously most of those values would be represented as PHP variables in a real site; I’m demonstrating the underlying commands).

You may be wondering how you can take this same approach when using Yii? If you’re just using a MySQL function as in the above, you can set the password using a `CDbExpression`, as discussed in Chapter 8, “[Working with Databases](#)”:

```
# protected/models/User.php
public function beforeSave() {
    if ($this->isNewRecord) {
        $this->pass = new CDbExpression('SHA2(:pass, 512)',
            array(':pass' => $this->pass));
    }
    return parent::beforeSave();
}
```

That code uses a `CDbExpression` to set the value of the password using the `SHA2()` function but only for new records. That's obviously how the registration (i.e., `INSERT`) would work; you'd need to do a similar thing with `SHA2()` upon login.

That approach will work, and may be secure enough for some sites, but `SHA2()` has been cracked and it also requires a relatively current version of MySQL with SSL support enabled. An alternative, and probably more secure, approach would be to hash the password in PHP.

There are a number of ways to hash data in PHP, which approach you use is really a matter of the level of security you need to reach for your site. I often use the `hash_hmac()` function, added to the language as of PHP 5.1.2. Its first argument is the algorithm to use, the second is the password to encrypt, and the third is a key for added security:

```
$pass = hash_hmac('sha256', $pass, 'lvkj23mn5j25KJE5r');
```

This is much more secure than just using `SHA2()`, but requires that the same key be used for all interactions (i.e., both registration and login). Normally I would securely store that key as a constant in a configuration file, thereby separating the key from the code that uses it. If you want to take that route in Yii, you can set the key as a parameter in the configuration file:

```
# protected/config/main.php
// Lots of other stuff.
$params=>array(
    'adminEmail'=>'webmaster@example.com',
    'encryptionKey' => 'lvkj23mn5j25KJE5r',
),
```

Then your code can use this key like so:

```
$pass = hash_hmac('sha256', $pass,
    Yii::app()->params['encryptionKey']);
```

There are arguments for using the `hash()` method, or `crypt()` with the Blowfish algorithm, instead. For added security, you can salt the password: add extra data to it to increase the password's length and uniqueness.

To use a PHP hashing function in your model upon registration, add a `beforeSave()` event handler, as described in Chapter 5, “[Working with Models](#)”:

```
# protected/models/User.php
public function beforeSave() {
    if ($this->isNewRecord) {
        $this->pass = hash_hmac('sha256', $this->pass,
            Yii::app()->params['encryptionKey']);
    }
    return parent::beforeSave();
}
```

To use the security manager upon login attempts is just a matter of calling `hash_hmac()` again and comparing the results. You'll see this in Chapter 11, “[User Authentication and Authorization](#).”

## Handling File Uploads

Next, let's look at how to handle uploaded files in Yii using the MVC approach. In non-framework PHP, handling uploaded files, while not hard, is a different process than handling other types of form data. I'm going to assume that you already know how to do that, and are able to set the correct permissions on folders, and do the other things that need to be in place for any PHP script to handle an uploaded file.

### Defining the Model

The CMS project already has a good model to use for this example: `File`. But that model works a bit differently than many uses of uploaded files, so let's come up with a new hypothetical: say users can upload an avatar image. It can be in JPG or PNG formats (no animated GIFs!), and must be smaller than 100KB in size. To start, set the rules in the model:

```
# protected/models/User.php::rules()
// Other rules
array('avatar', 'file', 'allowEmpty' => true,
    'maxSize' => 102400, 'types' => 'jpg, jpeg, png')
```

This rule uses the “file” validator, which in turn uses the `CFileValidator` class. Its writable attributes include:

- `allowEmpty`, whether the file is required

- `maxFiles`, the number of files that can be uploaded
- `maxSize`, the maximum number of bytes allowed
- `minSize`, the minimum number of bytes required
- `types`, the allowed file extensions (case-insensitive)

Thus, the above code says that the file is optional, but if provided must be under 100KB, and has to use the `.jpg`, `.jpeg`, or `.png` extension.

Plus, you can set error messages using the `tooLarge`, `tooMany`, `tooSmall`, and `wrongType` attributes:

```
# protected/models/User.php::rules()
// Other rules
array('avatar', 'file', 'allowEmpty' => true,
    'maxSize' => 102400, 'types' => 'jpg, jpeg, png',
    'tooLarge' => 'The avatar cannot be larger than 100KB.',
    'wrongType' => 'The avatar must be a JPG or PNG.')
```

And that takes care of all the validation!

## Creating the Form

Next, let's turn to the view file that displays the form. I'm going to demonstrate this for `CActiveForm`, but it should be easy enough for you to translate to using `CHtml` directly, or using Form Builder.

As you should know, in order for PHP to be able to handle an uploaded file, the form must use the “`enctype`” attribute, with a value of `multipart/form-data`. In order to have your Yii form do that, set the “`htmlOptions`” when invoking `beginWidget()`:

```
$form = $this->beginWidget('CActiveForm',
    array(
        'enableAjaxValidation' => false,
        'htmlOptions' =>
            array('enctype' => 'multipart/form-data'),
    )
);
```

{WARNING} Forgetting to set the “`enctype`” attribute is a common cause of problems when attempting to upload files. Also make sure that PHP is configured to allow for big enough file uploads to match your application's needs.

Also remember that such forms must use the POST method, which is the default. And “enableAjaxValidation” is disabled because uploaded files cannot be validated via Ajax (more on Ajax validation in Chapter 14, “[JavaScript and jQuery](#)”)

To create the file input in the form, invoke the `fileField()` method:

```
<?php echo $form->fileField($model, 'avatar'); ?>
```

## Handling the Uploaded File

Finally, there’s the controller, which should handle the uploaded file. The code created by Gii will look like this:

```
# protected/controllers/UserController.php
public function actionCreate() {
    $model=new User;
    if(isset($_POST['User'])){
        $model->attributes=$_POST['User'];
        if($model->save()){
            $this->redirect(array('view',
                'id'=>$model->id));
        }
    }
    $this->render('create',array('model'=>$model));
}
```

That code takes care of everything except for the uploaded file. File uploads are handled using the `CUploadedFile` class, which is the Yii equivalent to PHP’s `$_FILES` array. Invoke its `getInstance()` method to access the uploaded file:

```
$model->attributes=$_POST['User'];
$model->avatar =
CUploadedFile::getInstance($model, 'avatar');
```

With that in place, you can now save the model instance (i.e., store the record in the database):

```
$model->attributes=$_POST['User'];
$model->avatar =
CUploadedFile::getInstance($model, 'avatar');
if ($model->save()) {
```

As with any other model attribute, the model cannot be saved if it does not pass validation, including the validation of the file.

Finally, after saving the model, save the physical file to the server’s file system:

```
if ($model->save()) {  
    $model->avatar->saveAs('path/to/destination');
```

When the file is optional, as in my example, you should check that the attribute isn't null before attempting to save the file:

```
if ($model->save()) {  
    if ($model->avatar !== null) {  
        $model->avatar->saveAs('path/to/dest');  
    }  
}
```

Of course, you need to set the “path/to/destination” to something meaningful.

### Setting the File's Name

The “path/to/destination” value provided to the `saveAs()` method needs to indicate both the destination directory for the file *and* the file's name.

For the destination, for security reasons, it's best to store uploaded files outside of the Web root directory, or at least in a non-public Web directory. I always try to go outside of the Web root directory, but in a Yii site, using a subfolder of **protected** makes equal sense. For this example, let's assume you've created an **avatars** directory in **protected** (and set the permissions accordingly). Having done so, you can use this code to get a reference to that directory:

```
$dest = Yii::getPathOfAlias('application.avatars');
```

{NOTE} Once you store an uploaded file so that it's not directly available in the browser, you then need to use a *proxy script* to display it in the browser. I'll explain how to do that in Yii in Chapter 16, “[Leaving the Browser](#).”

Next, there's the file's name to determine. The `CUploadedFile` object will have the following useful properties:

- `error`, an error code, if an error occurred
- `extensionName`, the file's extension (from its original name)
- `name`, the file's original name on the user's computer
- `size`, the size of the uploaded file in bytes
- `tempName`, the path and name of the file as it was initially stored on the server
- `type`, the file's MIME type

As the model attribute is assigned the value of the CUploadedFile object, you could do this:

```
$model->avatar->saveAs($dest . '/' . $model->avatar->name);
```

For improved security, though, it's also best to rename uploaded files. In this particular situation, where the uploaded file is directly related (in a one-to-one manner) with a user record, I'd be inclined to use the unique user's ID for the file's name, although with its original extension:

```
$model->avatar->saveAs($dest . '/' . $model->id . '.' .
    $model->avatar->extensionName);
```

The only caveat here is that the file's extension isn't entirely reliable. Neither is the MIME type. I'll introduce an alternative approach in just a few pages.

## Handling Updates

Files that are uploaded in association with a model (such as an avatar for a user) pose a logical problem when it comes to updating the model. A user might change other pieces of information, such as her password, without touching the uploaded file.

To handle this situation, you must first address the validation rules in the model. If the file is required, you would want to make sure it's required only upon insertions of new records:

```
# protected/models/User.php::rules()
// Other rules
array('avatar', 'file', 'allowEmpty' => false,
    'maxSize' => 102400, 'types' => 'jpg, jpeg, png',
    'on' => 'insert'),
array('avatar', 'file', 'allowEmpty' => true,
    'maxSize' => 102400, 'types' => 'jpg, jpeg, png',
    'except' => 'insert')
```

Next, I normally reveal something about the file on the update form to let the user know what the current file is. This could be the current image, in the case of an avatar, or the original file name and size for something not visual. How you do this depends upon the file, your models, and so forth, but you should be able to figure that part out.

Then in the controller that handles the form's submission, you can actually use the same code already explained to handle the uploaded file. If the file was required, then it will only pass validation if provided. If the file was not required, then you don't want to blindly do this:

```
$model->avatar =  
    CUploadedFile::getInstance($model, 'avatar');
```

That would overwrite any existing file value with a NULL value (which would be bad). Instead, check for the presence of an uploaded file first:

```
$model->attributes=$_POST['User'];  
$upload = CUploadedFile::getInstance($model, 'avatar');  
if ($upload !== null) $model->avatar = $upload;  
if ($model->save()) {  
    if ($upload !== null) {  
        $model->avatar->saveAs('path/to/destination');  
    }  
}
```

The `CUploadedFile` instance is first assigned to a local variable. Then that local variable will be used as a flag to know whether or not a new file was uploaded.

## Uploading Multiple Files

If you have the need to upload multiple files with one model instance, there are a couple of ways you can do that. If the files are different, such as an avatar and a resumé, those would be two different attributes and you can handle each separately in the same way as you handle the one.

If the user could provide multiple files for the same attribute, things get more complicated but not that much more complicated. First, in the model, use the same validation rules you otherwise would, but also set the `maxFiles` attribute:

```
# protected/models/SomeModel.php::rules()  
// Other rules  
array('images', 'file', 'allowEmpty' => true,  
    'maxSize' => 102400, 'types' => 'jpg, jpeg, png',  
    'maxFiles' => 3)
```

Then, in the form, create the correct number of file inputs, giving each the rather unconventional name “[attributeName]”:

```
<?php echo $form->fileField($model, '[images'); ?>  
<?php echo $form->fileField($model, '[images'); ?>  
<?php echo $form->fileField($model, '[images'); ?>
```

(You'd add labels and formatting to your form, too, of course.)

Finally, in the controller, instead of calling `CUploadedFile::getInstance()`, call `CUploadedFile::getInstances()`, which will return an array of uploaded file instances, usable in a loop:

```
$files = CUploadedFile::getInstances($model, 'images');
foreach ($files as $file) {
    if ($file !== null) {
        // Use $file->saveAs()
    }
}
```

## More Secure Uploading

As you can tell, it's pretty easy to upload files in Yii, and there's even built-in validation. However, that validation is based upon information provided to PHP by the user and the browser: the file's MIME type, its extension, and so forth. In PHP, more secure validation can be accomplished using the [Fileinfo](#) functions. These functions, built into PHP as of version 5.3, use a file's *magic bytes*—actual data stored in the file itself—to determine its type. In Yii, you can use the [CFileHelper](#) class. If Fileinfo is available, [CFileHelper](#) will use that extension. If not, [CFileHelper](#) will use the `mime_content_type()` function or just the file's extension (as the worst case scenario).

{WARNING} The actual behavior of [CFileHelper](#) will depend upon the OS and configuration in use. On some systems, [CFileHelper](#) will throw an exception if the [Fileinfo](#) extension is not available.

You can use this class directly, if you want, but Yii will automatically use it if you set the “mimeTypes” property in the rule:

```
# protected/models/User.php::rules()
// Other rules
array('avatar', 'file', 'allowEmpty' => true,
      'maxSize' => 102400,
      'mimeType' => 'image/jpeg, image/pjpeg, image/png')
```

As with the extensions, the MIME types are case-insensitive, but you have to be careful as to what values you use. A file with an extension of `.jpeg` may have a MIME type of either `image/jpeg` or `image/pjpeg`. Search online for a complete list of MIME types by file type.

## Working with Lists

There are several HTML elements that use lists of data:

- Check box list

- Drop down list
- List box
- Radio button list

These elements take arrays for their data, but you can also use *list data*. List data is created by the `CHtml::listData()` method, and is a good way to populate one of these elements using existing models.

For example, say you want to create a drop down list that allows an administrator to change the owner of a page (reflected in the `Page` model's `user_id` attribute). You could populate that drop down list using this code:

```
<?php echo $form->dropDownList($model, 'user_id',
CHtml::listData(User::model()->findAll(), 'id', 'username')
); ?>
```

The first argument to `listData()` is an array of models. This can be returned by the `findAll()` method. The second argument is the index or attribute in the data to use for the list's values. The third argument is the index or attribute in the data to use for the list's displayed text.

This will work for you just fine, and even pre-select the right option when performing an update. Still, there are two ways you could improve upon this:

- Only display the users that have the authority to be owner's of a page
- Only fetch the `User` class's `id` and `username` attributes, as those are the only two being used

Both issues can be addressed using *scopes*, as explained in Chapter 8:

```
# protected/models/User.php
public function scopes() {
    return array(
        'authorsForLists' => array(
            'select' => 'id, username',
            'order' => 'username ASC',
            'condition' => 'type!="public"'
        )
    );
}
```

Then the `listData()` call can be changed to:

```
<?php echo $form->dropDownList($model, 'user_id',
CHtml::listData(
    User::model()->authorsForLists()->findAll(),
    'id', 'username')
); ?>
```

And by the way: the update form will still automatically select the correct author name from the drop down list! How nice is that?

## Forms for Multiple Models

There is one last subject that ought to be covered in this chapter: working with multiple models in the same form. In a somewhat trivial way, you just saw an example of this: in which the attribute in one model is related to another. Next, I'll expand on that example in a couple of ways. Finally, I'll demonstrate how to create two model instances using one form.

### Handling Many-to-Many Relationships

The previous example demonstrated using one model to populate a drop down list for another model. In that situation, there was a one-to-many relationship between the two models. A more complicated situation exists when there's a many-to-many relationship between two models, such as `Page` and `File` in the CMS site. Because of the many-to-many relationship between these two, neither `Page` nor `File` would have a foreign key to the other. Instead, a junction table is used: `page_has_file`. The table itself has only two columns: `page_id` and `file_id`. Each file associated with a page is represented by a record in this table.

On the form for adding (or updating) a page, you'd need to be able to select multiple files to associate with the page:

```
<div>
<?php echo $form->labelEx($model, 'files'); ?>
<?php echo $form->dropDownList($model, 'files', CHtml::listData(
    File::model()->findAll(), 'id', 'name'),
    array('multiple'=>'multiple', 'size'=>5)
); ?>
<?php echo $form->error($model, 'files'); ?>
</div>
```

That `dropDownList()` method will create a drop-down of size 5 (five items will be shown), populated using the list of files, and the user will be able to select multiple options. If you were to run this code, though, you'd get an error as `Page` doesn't have a `files` attribute (**Figure 9.6**).

## CException

Property "Page.files" is not defined.

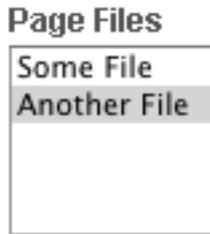
**Figure 9.6:** Yii complains when you attempt to create a form element for a model that does not have that attribute.

But there's an easy solution here...

In the `Page` model definition, the model's relationship is identified within the `relations()` method (these relations are updated after going through Chapter 8):

```
# protected/models/Page.php::relations()
'pageComments' => array(self::HAS_MANY,
    'Comment', 'page_id'),
'pageUser' => array(self::BELONGS_TO, 'User', 'user_id'),
'pageFiles' => array(self::MANY_MANY, 'File',
    'page_has_file(page_id, file_id)'),
'commentCount' => array(self::STAT, 'Comment', 'page_id')
```

Not only does this result in `Page` having a "pageFiles" relationship, it also results in `Page` having a "pageFiles" attribute, as if it were a column in the database. All you need to do to fix this is change the code in the form to use "pageFiles" instead of "files". Then the form will work and the menu will even select files that have been associated with that page (**Figure 9.7**).



**Figure 9.7:** The file associated with this page is selected in the drop down box.

Furthermore, you can also add "pagesFiles" to the `attributeLabels()` array of the `Page` model to give this relationship a new label that would be used in the form. You can even add a rule to the `Page` model to make sure this part of the form validates.

{TIP} When you create a relation, that relation becomes an attribute of the model.

Now that the form is working—in this case allowing you to select multiple files to be associated with a page, you’ll need to update the controller to handle the file selections. Within the `actionCreate()` method (of `PageController`), after the page has been saved, loop through each file and add that record to the `page_has_file` database table. This is a great time to use a prepared statement and a transaction.

```
# protected/controllers/PageController.php
public function actionCreate() {
    $model=new Page;
    if(isset($_POST['Page'])) {
        $model->attributes=$_POST['Page'];
        if($model->save()) {
            foreach ($_POST['Page']['pagesFiles'] as $file_id) {
                // Save to the `page_has_files` table.
            }
        }
    }
    // Et cetera
```

Personally, I would be inclined to use Direct Access Objects and prepared statements within the `foreach` loop to perform that task:

```
# protected/controllers/PageController.php::actionCreate()
if ($model->save()) {
    $q = "INSERT INTO page_has_file (page_id, file_id)
          VALUES (:model->id, :file_id)";
    $cmd = Yii::app()->db->createCommand($q);
    foreach ($_POST['Page']['pageFiles'] as $file_id) {
        $cmd->bindParam(':file_id', $file_id,
                         PDO::PARAM_INT);
        $cmd->execute();
    }
    // Et cetera
```

Now, when a new page is created, one new record is created in `page_has_file` for each selected file.

There’s still one more consideration: the update process. The final step is to update the `page_has_file` table when the form is submitted (and the page is updated). This could be tricky, because the user could add or remove files, or make no changes to the files at all. The easiest way to handle all possibilities is to clear out the existing values (for this page) in the `page_has_file` table, and then add them in anew. To do that, in `actionUpdate()` of the `PageController`, you would have:

```
# protected/controllers/PageController.php::actionUpdate()
if($model->save()) {
    $q = "DELETE FROM page_has_file
        WHERE page_id={$model->id}";
    $cmd = Yii::app()->db->createCommand($q);
    $cmd->execute();
```

Then you use the same foreach loop as in `actionCreate()` to repopulate the table.

### Creating Different Models at Once

The previous example demonstrated how to create and handle a form in which multiple instances of a model is associated with a single instance of another. Sometimes, you may only have a one-to-one relationship but you'll actually want to create new instances of both models at one time. For example, in my *PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide* book I used an example in which an administrator could add works of art as products to be sold in an e-commerce site. In the form for adding a print, the administrator could select an existing artist or add a new artist while adding the print. How you do this is easier than you might think.

In your controller (for whichever model is primary), create instances of both objects and pass them to the view:

```
# protected/controllers/ArtController.php
public function actionCreate() {
    $art = new Art;
    $artist = new Artist;
    $this->render('create', array(
        'artist'=>$artist,
        'art'=>$art,
    ));
}
```

From there, it's rather simple:

- Create elements for both models in the view
- Back in the `actionCreate()` method, mass assign the *primary* model's attributes
- Then, mass assign the *secondary* model's attributes
- If the secondary model is not NULL, save it in the database and assign the new ID value as the foreign key in the *primary* model
- Save the primary model

And that should do it. For added reliability, you could use transactions here, as explained in Chapter 8.

# Chapter 10

## MAINTAINING STATE

One of the first things every Web developer must learn is that the HyperText Transfer Protocol (HTTP) is *stateless*. Every page request and server response is a separate action, with no shared memory. Even the loading of an HTML form and the submission of that form are two separate and technically unconnected steps. (Conversely, a desktop application is one constantly running process.)

The ability to maintain state is required in order to have any of the functionality expected by today's Web sites. Without maintaining state, you can't have users logging in, shopping carts, and much more. Through cookies and sessions, server-side technologies such as PHP provide easy and reliable mechanisms for maintaining state in your applications. When using Yii, you could, of course, still use the standard PHP approaches. But since you're already using the framework, you ought to use the framework.

In this chapter, I'll explain everything you need to know in order to maintain state using Yii. I do assume that you already know the arguments for and against cookies vs. sessions. I won't waste time explaining when and why you would use one over the other, rather just cover the "how".

### Cookies

First up, let's look at cookies: how to create, read, delete, and customize them. You'll also see how to make your site more secure using cookies.

#### Creating and Reading Cookies

To create a cookie in PHP without using a framework, you just call the `setcookie()` function. To create a cookie while using the Yii framework, you don't use `setcookie()`, but rather create a new element in the `Yii::app()->request->cookies`

array. Cookies are in `Yii::app()->request->cookies`, because cookies are part of the HTTP request a browser makes of a Web server.

What you'll want to do to create a cookie is create a new object of type `CHttpCookie`: Yii's class for cookies. Here, then, is the syntax for setting a cookie in Yii:

```
Yii::app()->request->cookies['name'] =  
    new CHttpCookie('name', 'value');
```

You must use the same name in both places, replacing it with the actual cookie name. Remember that the cookie's name, and value, are visible to users in their browsers, so one ought to be prudent about what name you use and be extra mindful of what values are being stored.

*{TIP}* Because the cookie's name must be used twice in the code, you may want to consider assigning the cookie's name to a variable that is used in both instances instead.

Once you've created a cookie, you can access it, using:

```
Yii::app()->request->cookies['name']->value
```

You have to use the extra `->value` part, because the “cookie” being created is actually an object of type `CHttpCookie` (and Yii, internally, takes care of actually sending the cookie to the browser and reading the received cookie from the browser).

*{NOTE}* Remember that cookies are only readable on subsequent pages; cookies are never immediately available to the page that set them.

To test if a cookie exists, just use `isset()` on `Yii::app()->request->cookies['name']`, as you would any other variable:

```
if (isset(Yii::app()->request->cookies['name'])) {  
    // Use Yii::app()->request->cookies['name']->value.  
}
```

## Deleting Cookies

To delete an existing cookie, just unset the element as you would any array element:

```
unset(Yii::app()->request->cookies['name']);
```

To delete all existing cookies (for that site), use the `clear()` method:

```
Yii::app()->request->cookies->clear();
```

## Customizing Cookies

By default, cookies will be set to expire when the browser window is closed. To change that behavior, you need to modify the properties of the cookie. You *can't* do so when you create the `CHttpCookie` object (i.e., the only arguments to the constructor are the cookie's name and value), so you must separately create a new object of type `CHttpCookie`, to be assigned to `Yii::app()->request->cookies` later:

```
$cookie = new CHttpCookie('name', 'value');
```

Then adjust the `expire` attribute:

```
$cookie->expire = time() + (60*60*24); // 24 hours
```

Then add the cookie to the application:

```
Yii::app()->request->cookies['name'] = $cookie;
```

You can manipulate other cookie properties using the above syntax, changing out the specific attribute: `domain`, `httpOnly`, `path`, and `secure`. Each of these correspond to the arguments to the `setcookie()` function. (You can also manipulate the value of the cookie through `$cookie->value` and the cookie's name through `$cookie->name`).

For example, if you want to limit a cookie to a specific domain, or subdomain, use `domain`. To limit a cookie to a specific folder, use `path`. To only transmit a cookie over SSL, set `secure` to true.

If you'd rather not set one attribute at a time, you could instead pass an array to the `configure()` method:

```
$cookie = new CHttpCookie('name', 'value');
$cookie->configure(array(
    'expire' => time() + (60*60*24),
    'domain' => 'subdomain.example.com',
    'path' => 'dir'
));
```

## Securing Cookies

Another way to configure your cookies is to add an extra layer of security by setting Yii’s “enableCookieValidation” to true. This is done by assigning that value within the “request” component section of the main configuration file:

```
# protected/config/main.php
// Other stuff.
'components'=>array(
    'request'=>array(
        'enableCookieValidation'=>true,
    ),
// Other stuff.
```

Cookie validation prevents cookies from being manipulated in the browser. To accomplish that, Yii stores a hashed representation of the cookie’s value when the cookie is sent, and then compares the received cookie’s value against that stored hash to ensure they are the same. Obviously there’s extra overhead required to do this, but in some instances, the extra effort is justified by the extra security.

## Preventing CSRF Attacks

One thing to watch out for when using forms is the potential for Cross-Site Request Forgery (CSRF) attacks. A CSRF works like this:

- Site A does something meaningful by passing a value in a URL
- Site A requires that the user has a cookie from Site A in order to execute that action.
- Malicious site B has some code on it that unknowingly has users make that same request of site A. This could even be an image tag’s `src` attribute.
- If the user still has the cookie from site A, the request will be successful.

CSRF is a blind attack, in that the hacker cannot see the results of the request, but CSRF is amazingly easy to implement. As an example, let’s say that an administrator at your site logs in and does whatever but doesn’t log out. The administrator therefore still has a cookie in his browser indicating access to the site (i.e., the user could open the browser and perform admin tasks without logging in again). Now let’s say that the `src` attribute on malicious site B points to a page on your site that deletes a blog posting. If the administrator with the live cookie loads that page on site B, it will have the same effect as if that administrator went to your site and requested the blog deletion directly. This is not good.

To prevent a CSRF attack on your site, first make sure that all significant form submissions use POST instead of GET. You should be using POST for any form

that changes server content anyway, but a CSRF POST attack is a bit harder to pull off than a GET attack.

Second, set “enableCsrfValidation” to true in your configuration file, under the “request” component:

```
# protected/config/main.php
// Other stuff.
'components'=>array(
    'request'=>array(
        'enableCsrfValidation'=>true,
    ),
// Other stuff.
```

By doing this, Yii will send a cookie with a unique identifier to the user. All forms will then automatically store that same identifier in a hidden input. The form submission will only be handled then if the two identifiers match. With the case of a CSRF attack, the two identifiers will not match because the form’s identifier will not be passed as part of the request. Note that this only works if you’re using `CHtml` to create your forms, including through `CActiveForm` (if you manually create the form tags, Yii won’t insert the necessary code for preventing CSRF attacks).

The most important thing to remember about cookies, which I’ve already stated, is that cookies are visible to the user in the browser. And unless you’re using SSL for the cookies, they are also visible while being transmitted back and forth between the server and the client (which happens on every page request). So be careful of what gets stored in a cookie! If the data is particularly sensitive, use sessions instead of cookies.

## Sessions

With coverage of cookies completed, let’s quickly look at sessions in Yii.

### Using Sessions

The first thing to know about using sessions in Yii is that you don’t have to do anything to enable them, which is to say you don’t have to invoke `session_start()`, as you would in a standard PHP script. This is the behavior with Yii’s “autoStart” session property set to true, which is the default. Even without calling `session_start()` yourself, you could, of course, make use of the `$_SESSION` superglobal array, as you would in a standard PHP script, but it’s best when using frameworks to make total use of the framework. The Yii equivalent to `$_SESSION` is `Yii::app()->session`:

```
Yii::app()->session['name'] = 'value';
echo Yii::app()->session['name']; // Prints "value"
```

And that's all there is to it. To remove a session variable, apply `unset()`, as you would to any other variable:

```
unset(Yii::app()->session['name']);
```

{NOTE} Sessions are stored in `Yii::app()->session`, but cookies are in `Yii::app()->request->cookies`.

There are, of course, session methods you can use instead, if you'd rather:

```
Yii::app()->session->add('name', 'value');
echo Yii::app()->session->itemAt('name'); // Prints "value"
Yii::app()->session->remove('name');
```

Frequently, for debugging purposes, and sometimes to store it in the database, I like to know the user's current session ID. That value can be found in `Yii::app()->session->sessionID`.

If you'll be working with sessions a lot in a script, you may tire of typing `Yii::app()->session['whatever']`. Instead, you can create a shorthand variable pointing to the session:

```
$session = Yii::app()->session;
// Or:
$session = Yii::app()->getSession();
// Use $session['var'].
```

If you do this, just be careful not to assign a value to `$session`, because that value won't be stored in the actual session. Not unless you perform a mass reassignment later on:

```
Yii::app()->session = $session;
```

Those are the basics, and there's nothing really unexpected here once you know where to find the session data. The more complex consideration is how to configure sessions for your Yii application.

## Configuring Sessions

You can change how your Yii site works with sessions using the primary configuration file. Within that, you would add a “session” element to the “components” array, wherein you customize how the sessions behave. The key attributes are:

- `autoStart`, which defaults to true (i.e., always start sessions)
- `cookieMode`, with acceptable values of *none*, *allow*, and *only*, equating to: don’t use cookies, use cookies if possible, and only use cookies; defaults to *allow*
- `cookieParams`, for adjusting the session cookie’s arguments, such as its lifetime, path, domain, and HTTPS-only
- `gCProbability`, for setting the probability of garbage collection being performance, with a default of 1, as in a 1% chance
- `savePath`, for setting the directory on the server used as the session directory, with a default of `/tmp`
- `sessionId`, for setting the session’s, um, name, which defaults to *PHPSESSID*
- `timeout`, for setting after how many seconds a session is considered idle, which defaults to 1440
- `useTransparentSessionId`, if set to true, the session ID will be appended to all URLs and stored in all forms

For all of these, the default values are the same as those that PHP sessions commonly run using, except for `autoStart`.

For security purposes, I normally prefer to take the session out of the default `/tmp` directory, and put them in a directory that only this site will use. While I’m at it, I’ll set `cookieMode` to *only*, and change the session name:

```
# protected/config/main.php
// Other stuff.
'components'=>array(
    'session' => array (
        'cookieMode' => 'only',
        'savePath' => '/path/to/my/dir',
        'sessionId' => 'Session'
    ),
    // Other stuff.
```

The save path, in case you’re not familiar with it, is where the session data is stored on the server. By default, this is a temporary directory, globally readable and writable. Every site running on the server, if there are many (and shared hosting plans can have dozens on a single server), share this same directory. This means

that any site on the server can read any other site's stored session data. For this reason, changing the save path to a directory within your own site can be a security improvement.

## Storing Sessions in a Database

Another customization you can make as to how sessions are used is to store the session data in a database. To do that, change the session class used from the default `CHttpSession` to `CDbHttpSession`:

```
# protected/config/main.php
// Other stuff.
'components'=>array(
    'session' => array (
        'class' => 'system.web.CDbHttpSession',
        'connectionID' => 'db',
        'sessionTableName' => 'session',
    ),
// Other stuff.
```

You can also perform any of the other session configuration changes in that code block, too. The `CDbHttpSession` class extends `CHttpSession`, so it inherits the properties you've already seen.

If you choose this route, Yii can automatically create the table if it does not exist if you set “autoCreateSessionTable” to true. But to be thorough, you should create it yourself first:

```
CREATE TABLE session (
id CHAR(32) PRIMARY KEY,
expire INT,
data BLOB,
KEY (expire)
)
```

Other than the initial configuration differences, everything else about using the database is the same.

*{TIP}* You can also store session data in a cache, but this does require that you've established a caching mechanism first.

## Destroying Sessions

When the user logs out, you may want to formally eradicate the session. To do so, call `Yii::app()->session->destroy()` to get rid of the actual data stored on the server *and* the session:

```
# protected/controllers/SiteController.php::actionLogout()
if (Yii::app()->session->isStarted) {
    Yii::app()->session->clear();
    Yii::app()->session->destroy();
}
```

## Disabling Sessions

If your site will not be using sessions at all, you would want to disable them by adding this code to the “components” section of the configuration file:

```
# protected/config/main.php
// Other stuff.
'session' => array (
    'autoStart' => false,
),
// Other stuff.
```

## Chapter 11

# USER AUTHENTICATION AND AUTHORIZATION

*Authentication* is the process of identifying a user. On Web sites, this is most often accomplished by providing a username/password combination (or email/password), or via a third-party, such as the user's Twitter or Facebook account. Un-authenticated users qualify as anonymous users, or guests.

Related to authentication is *authorization*. Authorization is the process of determining whether the current user is allowed to perform a specific task. Users don't necessarily need to be authenticated to be authorized—for example, an un-authenticated guest can view your home page, but even in those situations, the authorization *is* using authentication (specifically the lack of authentication) to dictate what the user can do.

The previous chapter explained how to maintain state in Yii: storing and retrieving data that continues to be associated with a user as she travels from page to page. And in Chapter 7, “[Working with Controllers](#),” you were introduced to basic access control in Yii. Using it, a site can restrict which users can invoke what controller methods. The material in that chapter involves the basic user authentication generated by the `yiic` command when you first create a site. That generated code allows you to login using either `admin/admin` or `demo/demo`. Now it's time to learn how to fully implement user authentication and authorization in Yii (aka, “auth and auth”).

### Fundamentals of Authentication

As with anything in Yii, authentication is a matter of using the proper classes defined in the framework. And although the authentication classes are easy enough to use, the authentication process can be a bit confusing, due to the number of pieces involved and the role each plays. To hopefully minimize confusion, let's start by looking at the fundamentals of authentication, and the logic flow it entails.

## Key Components

The authentication process is designed to be quite flexible in Yii. Authentication can be performed against:

- Static values (e.g., demo/demo and admin/admin)
- Database tables
- Third-parties (e.g., Facebook or Twitter)
- Lightweight Directory Access Protocol (LDAP)

As already mentioned, the code created by `yiic` uses static values. It does so by creating the `UserIdentity` class, which inherits from `CUserIdentity` (which implements the `IUserIdentity` interface). The main purpose of these classes is to implement an `authenticate()` method that authenticates the user based upon whatever criteria or source you desire.

*{NOTE}* First thing to remember: the `authenticate()` method of the `CUserIdentity` class (or subclass) is used to perform the actual authentication.

A representation of the current user (i.e., the person accessing the current page) is always available through the “user” application component. Consequently, `Yii::app()->user` is a reference to the current user. This is true whether the user is authenticated (logged-in) or not.

By default, the “user” component is an object of type `CWebUser`. The `isGuest` attribute stores a Boolean indicating authentication status:

```
if (Yii::app()->user->isGuest) {
    // Do whatever.
} else {
    // Do this.
}
```

*{NOTE}* Second thing to remember: the representation of the user is stored in the “user” component as an object of type `CWebUser` (or an extended type).

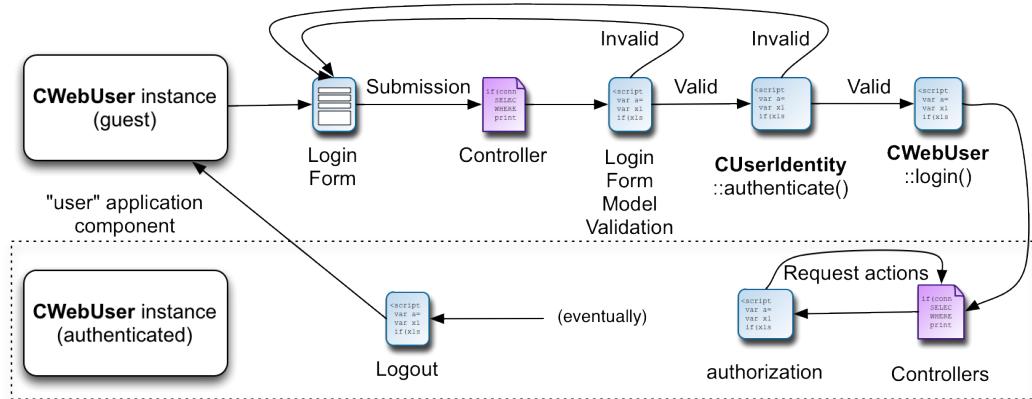
If the data passes authentication, then the `CWebUser` class’s `login()` method is invoked. It saves the authenticated user’s identity in the “user” component. From there on, different controllers can use the saved user identity to determine *authorization*. Controllers will do so in one of two ways:

- Basic access control (list-like)

- Role-Based Access Control (RBAC)

Finally, the user will log out. Logging out involves calling the `logout()` method of `CWebUser`, thereby removing the user's saved identity.

Of course, thrown into this mix, you also have the login form, which is tied to a model, which entails its own validation. **Figure 11.1** shows the logic flow of this entire process.



**Figure 11.1:** The authentication and authorization process.

With that overview in mind, let's now walk through the authentication process using the code generated by `yiic`.

## Default Authentication Process

The default application created by the Yii framework has built-in authentication using hard-coded values. When you generate a new site using Yii's command-line tool, three files for managing authentication are created:

- `protected/components/UserIdentity.php`
- `protected/models/LoginForm.php`
- `protected/views/site/login.php`

And there's also some code added to `protected/controllers/SiteController.php` that comes into play. The controller file gets the action going, of course. The view file is the login form itself. The `LoginForm` model defines the rules and behaviors for the login data. And the `UserIdentity` class defines a component that performs the actual authentication.

The URL to login will be `www.example.com/index.php/site/login` (or a variation on that URL), as Yii puts login/logout functionality in the "site" controller by default. When the user clicks on a link to go to the login page, she'll go through

the “site” controller and call the `actionLogin()` method. That method is defined as (with some comments and Ajax functionality removed):

```
1 # protected/controllers/SiteController.php::actionLogin()
2 $model=new LoginForm;
3 if(isset($_POST['LoginForm'])) {
4     $model->attributes=$_POST['LoginForm'];
5     if($model->validate() && $model->login()) {
6         $this->redirect(Yii::app()->user->returnUrl);
7     }
8 }
9 $this->render('login',array('model'=>$model));
```

First, a new object of type `LoginForm` is created (line 2). That class is defined in the `LoginForm.php` model file. The model extends `CFormModel`, and has three public attributes: `username`, `password`, and `rememberMe`. There’s also a private `_identity` attribute, used upon logging in.

If the form has been submitted (line 3), the form data is assigned to the model’s attributes (line 4). Then a conditional does two things: attempts to validate the data and login the user (line 5). If the data passes validation and the user can be logged in, the user will be redirected to whatever URL got her here in the first place (line 6, more on redirection later in the chapter). If the form has not been submitted, or if the form data does not pass the validation routine, then the login form is displayed, and the `LoginForm` object is passed along to it (line 9). There’s nothing unusual about that form, so I’ll leave it up to you to examine that view file if needed.

The call to the `validate()` method in the above code means that the form data has to pass the validation rules established in the `LoginForm` class. This is basic model validation as defined by the `rules()` method of that model:

```
# protected/models/LoginForm.php::rules()
return array(
    // username and password are required
    array('username', 'password', 'required'),
    // rememberMe needs to be a boolean
    array('rememberMe', 'boolean'),
    // password needs to be authenticated
    array('password', 'authenticate'),
);
```

One little trick here is the `authenticate()` validation requirement. This is an example of a user-defined filter, explained in Chapter 4, “[Initial Customizations and Code Generations](#)”. Here’s that method’s definition:

```
1 # protected/models/LoginForm.php::authenticate()
2 if(!$this->hasErrors()) {
3     $this->_identity=new UserIdentity($this->username,
4         $this->password);
5     if(!$this->_identity->authenticate()) {
6         $this->addError('password',
7             'Incorrect username or password.');
8     } // Did not authenticate.
9 } // Errors exist already!
```

That method actually performs the authentication: compares the submitted values against the required values. To start, it only performs this task if the model does not already have any errors (line 2). No use in attempting validation if a username or password was omitted, right? Next, the method creates a new `UserIdentity` object, passing it the username and password values (line 3). This object is assigned to the internal, private `_identity` attribute. Then the `UserIdentity` class's `authenticate()` method is invoked (line 5). This is the primary authentication method in the site, which performs the actual authentication. If that method returns false, an error is added to the current model. As with all validators, if no error is added, then the model will be considered to have valid data.

{NOTE} In case it's not clear, the purpose of the `LoginForm` model's `authenticate()` method is to invoke the `UserIdentity` class's `authenticate()` method. It's that other method that actually performs the authentication against stored values.

Now, let's look at the `UserIdentity` class, defined in `protected/components/UserIdentity.php`. This class extends `CUserIdentity` which, in turn, implements the `IUserIdentity` interface, as required by Yii for any authentication classes. The `CUserIdentity` class takes two arguments to its constructor: a username and a password. These are then assigned to its attributes.

Note that you'll never use `CUserIdentity` directly. You should extend it to create your own authentication class. The `CUserIdentity` class has an `authenticate()` method which has to be overridden by classes that extend `CUserIdentity`. The `authenticate()` method needs to do whatever steps are necessary to authenticate the user. It must return a Boolean value, indicating success of the authentication. Here's how `UserIdentity` defines the `authenticate()` method:

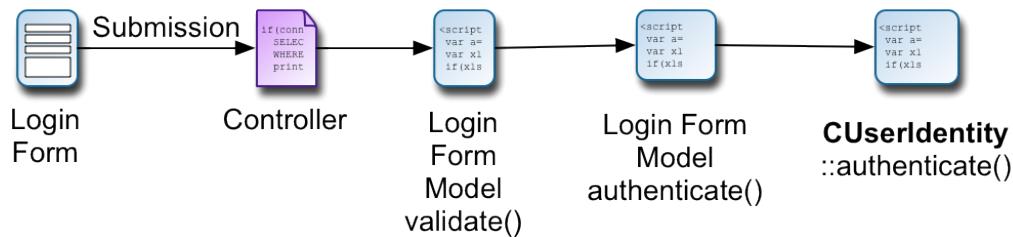
```
1 # protected/components/UserIdentity.php::authenticate()
2 $users=array(
3     // username => password
4     'demo'=>'demo',
5     'admin'=>'admin',
```

```

6 );
7 if(!isset($users[$this->username])) {
8     $this->errorCode=self::ERROR_USERNAME_INVALID;
9 } else if($users[$this->username]!==$this->password) {
10    $this->errorCode=self::ERROR_PASSWORD_INVALID;
11 } else {
12     $this->errorCode=self::ERROR_NONE;
13 }
14 return !$this->errorCode;

```

Before explaining this code, **Figure 11.2** demonstrates the logical flow that got the application this point.



**Figure 11.2:** How the Yii app gets to the `CUserIdentity::authenticate()` method.

As already mentioned, the default code authenticates users against hard-coded values. If you do nothing at all after creating a site, users will be allowed to log into the site with the username/password combinations of `demo/demo` or `admin/admin`. You can see those values in the above code.

After defining those values, the code checks if the current instance's `username` property does not exist in the internal `$users` array. Given the hard-coded values, an alternative would be to simply confirm that `$this->username` equals either “`demo`” or “`admin`”.

If the username isn't a proper value, then the `UserIdentity` class's `errorCode` property is assigned the value of the constant `UserIdentity::ERROR_USERNAME_INVALID`. This constant is defined in `CBaseUserIdentity`, and is inherited down to `UserIdentity`. This table lists the available constants and their values.

Constant	Value
<code>ERROR_NONE</code>	0
<code>ERROR_USERNAME_INVALID</code>	1
<code>ERROR_PASSWORD_INVALID</code>	2
<code>ERROR_UNKNOWN_IDENTITY</code>	100

If that first conditional is false, which means the username *does* exist in the `$users` array, then the next conditional is evaluated: `$users[$this->username] !== $this->password`. In other words, does the submitted password *not* match the assigned password for this user. If the passwords do *not* match, then the `UserIdentity` class's `errorCode` property is assigned the value of the constant `UserIdentity::ERROR_PASSWORD_INVALID`.

Finally, if neither of those conditions is true, then the username must exist in the array and the password for that username must be correct. In that case, the `UserIdentity` class's `errorCode` property is assigned the value of the constant `UserIdentity::ERROR_NONE`.

{NOTE} The logic of the `authenticate()` method is a little less clear because both conditionals test for negative conditions: the username not existing and the password not being correct.

The last thing the method does is return a Boolean indicating whether or not there was an error:

```
return !$this->errorCode;
```

If no error occurred, that statement returns true: the user was authenticated. If there was an error, that statement returns false: the user was not authenticated.

As a reminder, the returned value is used by the `authenticate()` method of the `LoginForm` model:

```
if (!$this->_identity->authenticate())
    $this->addError('password',
        'Incorrect username or password.');
```

Thus, if the login form values don't authenticate in `UserIdentity`, a new error is added to the login form model instance. (And that error can be used on the login form view page.)

All of this code completes the `validate()` process of the `LoginForm` model. Next, return to the “site” controller, which subsequently invokes the model's `login()` method:

```
# protected/controllers/SiteController.php::actionLogin()
if ($model->validate() && $model->login())
```

The `login()` method of the model is responsible for registering the `UserIdentity` object to the “user” component. This allows the entire site to track and recognize an authenticated user. The code generated for you is:

```
1 # protected/models/LoginForm.php::login()
2 if($this->_identity==null) {
3     $this->_identity=new UserIdentity($this->username,
4         $this->password);
5     $this->_identity->authenticate();
6 }
7 if($this->_identity->errorCode==UserIdentity::ERROR_NONE) {
8     $duration=$this->rememberMe ? 3600*24*30 : 0; // 30 days
9     Yii::app()->user->login($this->_identity,$duration);
10    return true;
11 } else {
12     return false;
13 }
```

This code first checks that an authenticated `UserIdentity` object exists (line 2). This is just a precaution, as you wouldn't want to login a non-authenticated user. If no such object exists, a new `UserIdentity` object is created and validated (lines 3-5).

The next conditional further checks that there's no error code (line 7). If no error code is present, then the `UserIdentity` object will be registered with the application by providing it to the `CWebUser::login()` method (line 9). That method needs to be provided with a `IUserIdentity` object. (I'll get to the `$duration` shortly.)

Finally, the function returns true or false. This returns the logic flow back to the controller:

```
# protected/controllers/SiteController.php::actionLogin()
if($model->validate() && $model->login())
    $this->redirect(Yii::app()->user->returnUrl);
```

The user is redirected if she was able to be logged in. If not, the form will be displayed again.

Whew! This probably seems like a lot of code and logic. And, well, it kind of is. But by separating all the pieces of the authentication process, it's an extremely flexible process. Here's what you have to work with:

- A model (`LoginForm`)
- A view (`views/site/login.php`)
- A controller (`SiteController`)
- Validation (through the model)
- Error reporting (through the view)
- Authentication (through `UserIdentity`)
- Registration of the authenticated user with the application (through `Yii::app()->user->login`)

Because these are separate components, you can make changes to one without having to even look at the others. You want to authenticate based upon an email address instead of a username? No problem. You want to authenticate against a database? No problem. You want to change what errors are displayed? No problem.

Hopefully you were able to follow this logic, because you'll need to understand the role each part plays in order to customize the authentication process for your own sites. You'll learn how to do that after first learning a couple more things about the "user" component in Yii.

## Allowing for Extended Login

By default, Yii will use sessions to store the user identity. As with any use of sessions, this means that after a relatively short period of inactivity, the user will need to login again. If you'd like users to be recognized by your system for a longer duration, including after closing the browser and later returning, you can tell Yii to use cookies instead of sessions for storing the user identity.

{TIP} As a session's true expiration is partly based upon PHP's garbage collection mechanism, how quickly a session actually expires (once it becomes inactive) depends upon several factors, including how busy your site is.

The first thing you'll need to do is set the `allowAutoLogin` "user" component property to true in your configuration file:

```
# protected/config/main.php
// Lots of other stuff.
'components'=>array(
    'user'=>array(
        'allowAutoLogin'=>true,
    ),
    // More stuff
)
```

The default value for this property is false, but the code created by `yiic` configures it to true (i.e., that code is already in the generated configuration file).

That *allows* for cookies to be used. The next thing you need to do is tell Yii to actually use cookies for the user identity. To do that, provide a duration argument to the `login()` method of the `CWebUser` class (aka, the "user" component of the application). This value should be a number in seconds.

The code generated for you will already do this if the "remember me" checkbox is checked:

```
# protected/models/LoginForm.php::login()
$duration=$this->rememberMe ? 3600*24*30 : 0; // 30 days
Yii::app()->user->login($this->_identity,$duration);
```

By default, the cookie will last whatever duration you specified *from the time the cookie is first sent*. If the cookie is set to last for 30 days, that's 30 days from the original login, not from the point of last activity. If you'd like to change it so that the cookie is resent when the user is active, set the `autoRenewCookie` property to true:

```
# protected/config/main.php
// Lots of other stuff.
'components'=>array(
    'user'=>array(
        'allowAutoLogin'=>true,
        'autoRenewCookie'=>true,
    ),
    // More stuff
```

With that configuration, the cookie will automatically be resent with each user request, thereby continuing to push back the cookie's expiration. Yii will continue to use the original duration period for each cookie's expiration value. For that reason, you'd likely want to reduce the duration to a shorter period, such as a few days.

{WARNING} The `autoRenewCookie` feature can adversely affect performance.

If you want to otherwise customize the cookie, configure the `identityCookie` property of the `CWebUser` object, providing new values using the `CHCookie` properties (explained in Chapter 10, “[Maintaining State](#)”):

```
# protected/config/main.php
// Lots of other stuff.
'components'=>array(
    'user'=>array(
        'allowAutoLogin'=>true,
        'identityCookie' => array(
            'domain' => 'store.example.org',
            'secure' => true
        )
    ),
    // More stuff
```

I've just explained *how* you would use cookies instead of sessions to store the user's identity, but a secondary issue is *should you?*. Remember that cookies are less secure than sessions, as they are transmitted back and forth between the client and the server. As always, you must match the level of security to the site.

I would generally recommend that you use only sessions and disable `allowAutoLogin`, unless security is less of a concern for the site in question *and* it would be an unreasonable inconvenience to expect the user to frequently login. Good examples that meet both of these criteria are social media sites such as Facebook. Later in the chapter, though, you'll see how to store other information as part of the user's identity, and you must be careful about doing so when `allowAutoLogin` is enabled (I'll remind you about the security issues then, too, just for safe measure).

{TIP} If you're not allowing auto login (i.e., cookies storage for the user identity), then be certain to remove all references to the "remember me" checkbox from your model and view files.

If you are restricting the site to sessions for the user identity, you can make the system even more secure by restricting the authentication time period to an even smaller duration than the default session duration. To do that, set the `authTimeout` property to a time period, in seconds:

```
# protected/config/main.php
// Lots of other stuff.
'components'=>array(
    'user'=>array(
        // Next line not needed, as it's the default:
        'allowAutoLogin'=>false,
        'authTimeout' => (60*15) // 15 minutes
    ),
    // More stuff
```

## Logging Out

Finally, you also ought to know how to log out a user. That's accomplished by invoking the `logout()` method of the "user" component (the `CWebUser` class). This code comes from the "site" controller:

```
public function actionLogout() {
    Yii::app()->user->logout();
    $this->redirect(Yii::app()->homeUrl);
}
```

## Authentication Options

Now that you have an understanding of authentication in Yii (hopefully), let's start tweaking the default authentication system. To start, I'll explain how you'd change the static authentication. Then I'll explain how you would implement database authentication.

### Using Static Authentication

The first, and by far the easiest, option is to continue using the static authentication but change the login values. I've worked on a couple of projects built in Yii where only one user ever needed to login: a single administrator. In the rare situations where that's also true for you, then you can open **UserIdentity.php** and change this code:

```
$users=array(  
    // username => password  
    'demo'=>'demo',  
    'admin'=>'admin',  
) ;
```

to

```
$users=array(  
    'whateverName'=>'whateverPassword'  
) ;
```

At that's it!

*{TIP}* Don't forget to remove the hint paragraph from the view, as demo/demo and admin/admin will no longer work.

For security reasons, I prefer administrators to login for each session, though. In these situations, I would also disable the `allowAutoLogin` and remove references to the "remember me" attribute and checkbox.

### Implementing Database Authentication

If you can use static authentication, great, but more frequently, authentication will be performed against a database table. No matter what the source is for your authentication, you still need to:

- Create a login form that's associated with a model

- Create a `CUserIdentity` object that authenticates the user
- Register the `CUserIdentity` object with the “user” component

The process is the same regardless of the source; the actual implementation is mostly a matter of what classes you want to use.

For this example, let’s assume there’s a `user` table, and that the user will login by providing a combination of an email address and password. The password is hashed using the PHP `crypt()` function, as explained in Chapter 9, “[Working with Forms](#).“

Starting with the login form, there are two approaches you could try: using the `LoginForm` class (or your own class like it) or using the actual model associated with the underlying database table. Let’s walk through both options.

### Updating the `LoginForm` Class

The first way you can authenticate against a database table is to use the `LoginForm` class generated by `yic`, but tweak it to suit your needs. This approach is pretty easy. You’d start by changing the attributes in `LoginForm` to those you require: `email`, `password`, possibly `rememberMe` (depending upon the site, I’ll remove it), and `_identity`:

```
# protected/models/LoginForm.php
class LoginForm extends CFormModel {
    public $email;
    public $password;
    private $_identity;
    // Et cetera
```

Next, alter the rules accordingly. Instead of username and password being required, email and password are now required. Also, the email should be in a valid email address format. The application of the `authenticate()` method to validate the password remains:

```
# protected/models/LoginForm.php
public function rules() {
    return array(
        array('email, password', 'required'),
        array('email', 'email'),
        array('password', 'authenticate'),
    );
}
```

Next, if you want, update the `attributeLabels()` method for the email address and remove the label for `rememberMe`:

```
# protected/models/LoginForm.php
public function attributeLabels() {
    return array('email'=>'Email Address');
}
```

The next changes to the `LoginForm` model are in the `authenticate()` method. Two references to “username” must be changed to “email”. For example, this:

```
$this->_identity=new UserIdentity($this->username,
    $this->password);
```

becomes:

```
$this->_identity=new UserIdentity($this->email,
    $this->password);
```

The same change has to be made in the `login()` method.

The final edits to the `LoginForm` class are to remove references to `rememberMe` and `$duration` in the `login()` method. Here’s that subsection:

```
if ($this->_identity->errorCode==UserIdentity::ERROR_NONE) {
    Yii::app()->user->login($this->_identity);
```

And that takes care of edits to the `LoginForm` class.

Next, you need to edit `UserIdentity::authenticate()`, which is where the actual authentication against the database takes place. Replace the entire `authenticate()` method definition with the following (to be explained afterward):

```
1 # protected/components/UserIdentity.php::authenticate()
2 // Understand that email === username
3 $user = User::model()->findByAttributes(array(
4     'email'=>$this->username));
5 if ($user === null) {
6     // No user found!
7     $this->errorCode=self::ERROR_USERNAME_INVALID;
8 } else if ($user->pass !==
9     hash_hmac('sha256', $this->password,
10     Yii::app()->params['encryptionKey'])) {
11     // Invalid password!
12     $this->errorCode=self::ERROR_PASSWORD_INVALID;
13 } else { // Okay!
14     $this->errorCode=self::ERROR_NONE;
15 }
16 return !$this->errorCode;
```

The third line tries to retrieve a record from the database using the provided email address. You may be wondering why I refer to `$this->username` here. That's because the `CUserIdentity` class's constructor takes the provided email address and password (from `LoginForm`) and stores them in `$this->username` and `$this->password`. For that reason, I need to equate "username" with "email" here, which is better than editing the framework itself. You ought to leave a comment about this so that you won't be confused later when looking at the code.

Next the `authenticate()` method checks a series of possibilities and assigns constant values to the `errorCode` variable, just like the original version did. In the first conditional, if `$user` has no value, then no records were found, meaning that the email address was incorrect. In the second conditional, the stored password is compared against the `computeHMAC()` version of the submitted password. This assumes that Yii has been configured to use this method (see Chapter 9, “[Working with Forms](#)”) and that the same hashing configuration was used to register the user.

If neither of those two conditionals are true—`$user` is not `NULL` and the passwords match, then everything is okay.

Finally, the method returns a Boolean indicating whether or not an error exists. And that's it for changing the `UserIdentity` class.

The last remaining change is to the form itself. First, the hint paragraph needs to be removed, as `demo/demo` and `admin/admin` will no longer work. Then the code that displays the “remember me” checkbox should also be excised. Remember me functionality is only good for cookies, so it's useless here. Finally, the form should take an email address, not a username, so those two lines must be changed. The complete form code is now ([Figure 11.3](#)):

```
<div class="row">
    <?php echo $form->labelEx($model, 'email'); ?>
    <?php echo $form->textField($model, 'email'); ?>
    <?php echo $form->error($model, 'email'); ?>
</div>
<div class="row">
    <?php echo $form->labelEx($model, 'password'); ?>
    <?php echo $form->passwordField($model, 'password'); ?>
    <?php echo $form->error($model, 'password'); ?>
</div>
<div class="row buttons">
    <?php echo CHtml::submitButton('Login'); ?>
</div>
```

And that's it. You can now perform authentication against a database table.

That being said, if you do have problems with this, start by making sure that all passwords are hashed during the registration process (i.e., saved in the database) using the exact same algorithm and other particulars as are being used during the

# Login

Please fill out the following form with your login credentials:

Fields with \* are required.

Email Address \*

Password \*

**Login**

**Figure 11.3:** The updated login form.

login process. From there, I would next walk through the authentication process and confirm what is, or is not, working each step of the way.

## Using the Existing Model

An alternative approach to database-driven authentication is to use the `User` model directly. The argument for this approach is that it reduces the code redundancy. With the `LoginForm` model, you've duplicated two attributes—the email address and password—that are already in `User`. You've also duplicated the logic surrounding those attributes, such as the validation rules and the attribute labels. If you later want to make a simple change, such as making the email address label just “Email”, instead of “Email Address”, you'll need to remember to make that change in two places. This isn't a terrible thing, to be sure, but generally redundancies are to be avoided in any software.

Tapping into the `User` model for authentication is not that hard, so long as you remember your *scenarios*. Most of the validation rules would apply when a new user is created (i.e., registers) or when an existing user updates her information, but those rules would *not* apply during login. For example, the username would not be required upon login (if you're using the email address to login) and you wouldn't set the user's type then, either.

To make the model work for all situations, I would add new rules for the “login” scenario and exempt every other rule from that scenario. Here's part of that:

```
# protected/models/User.php::rules()
return array(
    // Always required fields:
    array('email', 'required'),
    // Only required when registering:
    array('username', 'required', 'on' => 'insert'),
    // Password must be authenticated when logging in:
    array('pass', 'authenticate', 'on' => 'login'),
    // And so on.
```

{NOTE} The `User` model uses `pass` as its password attribute name, not `password` as in `LoginForm`. You'll need to make sure all the code consistently uses the right attribute name.

Next, add to the `User` model the `authenticate()` and `login()` methods as previously explained for `LoginForm`. You'll also need to define the `UserIdentity` class as just explained in the `LoginForm` example.

Next, create the `actionLogin()` method of the `UserController` class. It needs to create and use an object of type `User` instead of `LoginForm`:

```
# protected/controllers/UserController.php::actionLogin()
$model=new User('login');
if(isset($_POST['User'])) {
    $model->attributes=$_POST['User'];
    // validate user input and redirect
    // to the previous page if valid:
    if($model->validate() && $model->login())
        $this->redirect(Yii::app()->user->returnUrl);
}
// display the login form
$this->render('login',array('model'=>$model));
}
```

And, finally, create the login form:

```
<div class="row">
    <?php echo $form->labelEx($model,'email'); ?>
    <?php echo $form->textField($model,'email'); ?>
    <?php echo $form->error($model,'email'); ?>
</div>
<div class="row">
    <?php echo $form->labelEx($model,'pass'); ?>
    <?php echo $form->passwordField($model,'pass'); ?>
```

```
<?php echo $form->error($model,'pass'); ?>
</div>
<div class="row buttons">
    <?php echo CHtml::submitButton('Login'); ?>
</div>
```

And that should do it!

Having seen the code required to use `User` for authentication, you can now appreciate the arguments against this approach. First, you'll end up adding two methods to the class that will only be used during the login process. Second, it makes the logic with the rules a bit more complicated, which could lead to bugs.

That being said, which approach you use—the `LoginForm` or the `User` class—is up to you.

## The `UserIdentity` State

The `IUserIdentity` interface class, and therefore `CUserIdentity` and `UserIdentity` (in the code created by `yiic`) has a concept called *state*. State is nothing more than stored data specific and unique to the authenticated user.

To start, by default, the user identity will store the user's name: the value provided to login. Using the default code, this value is automatically stored as part of the user when the authenticated user is registered with the “user” component. The value is therefore available through a reference to the “user” component. Specifically, `Yii::app()->user->name` will return the username value provided upon login. This allows you to greet the user by name (**Figure 11.4**):

```
<h2>Hello,
<?php # protected/views/someController/someView.php
echo (Yii::app()->user->isGuest) ? ' Guest' :
    CHtml::encode(Yii::app()->user->name);
?>
!</h2>
```

In a non-Yii site, you would store this information in a session (or possibly a cookie). You *could* do that in Yii, too, but since you've already authenticated the user, and the Yii application needs a reference to the user, it makes sense to store user-specific information in the “user” component.

Before going further, notice that if you change the login process to be based upon the user's email address and not a username, then `Yii::app()->user->name` will return the email address, as in Figure 11.4. This is because that value was associated with the internal `name` attribute during the login process (see the pages earlier in this chapter if this is still not clear).

# My Web Application

[Home](#)   [About](#)   [Contact](#)   [Logout \(text@example.com\)](#)

## Hello, text@example.com!

**Figure 11.4:** Greeting the user by login name.

Along with the `name` value, the user identity automatically also stores the user's ID: a unique identifier. There's a catch, though: by default, the unique identifier is the same as the username, and is therefore also returned by references to `id`. This code has the same result as the previous bit (and Figure 11.4):

```
<h2>Hello,  
<?php # protected/views/someController/someView.php  
echo (Yii::app()->user->isGuest()) ? ' Guest' :  
    CHtml::encode(Yii::app()->user->id);  
?  
!</h2>
```

This probably seems unnecessarily duplicitous, but with the default login scheme, based upon static values, the username and password are the only two pieces of information known about the user. Hence, while you need to be familiar with the user identity state and the role it plays, there are two other things you'll commonly want to do:

- Store other data as part of the user identity state
- Change the state's reference to the user's ID

### Adding to the State

To save other information in the user identity state, invoke the `setState()` method of the `CUserIdentity` class. Its first argument is a name and its second argument is a value (again, this is like storing a value in a session, but the storage is directly tied to the user).

As an example of this, the `User` model in the CMS example has a `type` property, which reflects the kind of user: public, author, or admin. Many pages will want to access this information to know, for example, if the current user should be allowed to create a new page of content or edit a specific page. Again, this value could be

stored in a session or cookie, but as it's particular to the user, it makes sense to store it in the "user" component. Here's the updated code in the `UserIdentity` class:

```
1 # protected/components/UserIdentity.php::authenticate()
2 $user = User::model()->findByAttributes(array(
3     'email'=>$this->username));
4 if ($user === null) {
5     $this->errorCode=self::ERROR_USERNAME_INVALID;
6 } else if ($user->pass !== hash_hmac('sha256',
7     $this->password, Yii::app()->params['encryptionKey'])) {
8     $this->errorCode=self::ERROR_PASSWORD_INVALID;
9 } else {
10     $this->errorCode=self::ERROR_NONE;
11     $this->setState('type', $user->type);
12 }
13 return !$this->errorCode;
```

There's only one new line of code there (line 11), which invokes `setState()`. With that in place, once the user has been authenticated, you can access the user's type value via `Yii::app()->user->type`.

## Storing the User's ID

Another thing you'll probably want to do in situations like this (authenticating against the database) is establish a proper reference to the user's ID. In the CMS site, the user's ID will be used to associate pages with users, files with users, and so forth.

One solution would be to just store the user ID as part of the identity using the `setState()` method. However, that solution could lead to confusion and bugs as references to `Yii::app()->user->id` would still return the user's name or email address, not her ID. The better solution is to change what value `Yii::app()->user->id` returns. That's accomplished by overriding the `getId()` method of the `CUserIdentity` class:

```
1 # protected/components/UserIdentity.php::authenticate()
2 class UserIdentity extends CUserIdentity {
3     private $_id;
4     public function authenticate() {
5         $user = User::model()->findByAttributes(array(
6             'email'=>$this->username));
7         if ($user === null) {
8             $this->errorCode=self::ERROR_USERNAME_INVALID;
9         } else if ($user->pass !== hash_hmac('sha256',
```

```
10     $this->password,
11     Yii::app()->params['encryptionKey']) ) {
12     $this->errorCode=self::ERROR_PASSWORD_INVALID;
13 } else { // Okay!
14     $this->errorCode=self::ERROR_NONE;
15     $this->setState('type', $user->type);
16     $this->_id = $user->id;
17 }
18 return !$this->errorCode;
}
20 public function getId() {
21     return $this->_id;
}
23 }
```

To start, a new private attribute is added to the class for storing the ID value (line 3). Then, in the `else` clause for successful authentication, the user's actual ID value is assigned to the class `$_id` attribute (line 16).

Finally, the `getId()` method is overwritten, now returning the class's `$_id` attribute instead of the user's name or email address.

## Where State Is Stored

The last thing you need to know about the user identity state is where the state information is stored. The answer is simple: it depends!

If cookie-based login is enabled (by setting `CWebUser::allowAutoLogin` to be true in the configuration file), the user identity information may also be saved in cookie. In such cases, you would never want to store the user's ID (i.e., primary key value).

{WARNING} Never store primary key values (from the database) in cookies!

If cookie-based login is disabled, then the user identity information will be stored in the session, by default, in which case it is safe to store the user ID and other important information as part of the user state.

## Authorization

Now that you (hopefully) have a firm grasp on authentication, it's time to turn to its sibling, *authorization*. As a reminder, authentication is a matter of verifying who the user is; authorization is a matter of confirming if the user has permission to perform a certain task.

One method of authorization was introduced in Chapter 7. There, you learned about “access rules” in controllers for basic access control. With the additional knowledge of user identity states, you can now learn one more way to define an access rule: using an expression. It’s also time to discuss a more sophisticated approach to authorization: Role-Based Access Control (RBAC). And, I’ll explain a couple of techniques for enforcing authorization and redirecting the user.

## Revisiting Access Control

The access control options discussed in Chapter 7 include dictating access based upon the user’s:

- Guest or non-guest status
- Specific username (i.e., that used to login)
- IP address

A fourth way to dictate access is to use an *expression*. An expression is simply PHP code that, when executed, returns a Boolean value. If the code returns true, the rule would apply. As a reminder, this is used in the `accessRules()` method of a controller:

```
# protected/controllers/SomeController.php::accessRules()
array(
    'allow',
    'actions' => array('index'),
    'users' => array('@'),
    'expression' => 'PHP code to be evaluated'
);
```

Two quick things to know about using expressions. First, you still need to use the “users” index in the array to dictate for what user types the expression should be evaluated. Second, you’ll probably want to restrict the rule to logged-in users (most likely).

Using the information presented in just this chapter, there’s already a good example of when you might want to use this. In the CMS example, only author and admin users should be able to create or update pages of content. Assuming that the user’s type has already been stored in the user identity state, that value can be used to fine-tune the access rules:

```
# protected/controllers/PageController.php::accessRules()
array('allow',
    'actions'=>array('create', 'update'),
    'users'=>array('@'),
```

```
'expression'=>'isset(Yii::app()->user->type) &&
((Yii::app()->user->type=="author") ||
(Yii::app()->user->type=="admin"))'
),
```

With that rule, users that aren't logged in, or users that are logged in but are the public type, won't be able to execute the "create" or "update" actions.

As another example, you could use a similar rule to restrict deleting of pages to only administrator types.

## Role-Based Access Control

When access rules are too simple, you can move on to the more custom and elaborate way to implement authorization in Yii: using Role-Based Access Control (RBAC). With RBAC, the goal is to identify who is allowed to do what, but RBAC uses a hierarchy which often better correlates to a site's users. For example, in a CMS site, an administrator has more power than an author who has more power than a public user. Of course, with this hierarchy comes more complexity.

The fundamental unit in RBAC is the *authorization item*, which is a permission to do something. Authorization items can be:

- Operations
- Tasks
- Roles

The hierarchy begins at the bottom with operations. An operation is a single atomic permission: edit a page, delete a user, post a comment, etc.

The next level of the hierarchy are tasks. Tasks commonly serve one of two purposes:

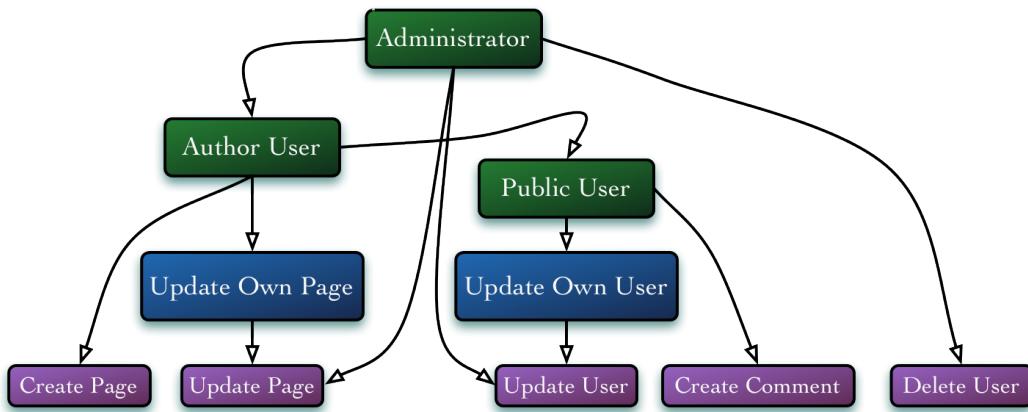
- As a modified permission (i.e., Can this user edit this page?)
- As a group of permissions (e.g., the managing files task might entail these operations: add a file, delete a file, edit a file, view a file)

At the top of the hierarchy are the roles. This is the final arbiter of who can do what. In the CMS example, perhaps an administrator can perform all the operations in all the tasks, but an author would not have authority to perform any of the user tasks, save those granted to the most basic user.

The hierarchies are very fluid in Yii's implementation of RBAC. For example, you can assign an operation directly to a role. Tasks can be parents of not just operations but also other tasks. One role can be subservient to another role.

To implement RBAC in your site, you should first start by sketching out the various roles, tasks, and operations involved. The end goal is to design your hierarchy as efficiently as possible. Understand that you only need to represent authorization items that are restricted. In the CMS example, any type of user, including non-authenticated guests, can view a page of content, so that does not need to be represented in the hierarchy.

**Figure 11.5** shows a subsection of authorization items for the CMS example. Understand that there's no one right answer here. Any two people could create slightly different hierarchies for the same site (in much the same way that two developers might come up with slightly different database schemes, both of which would work fine).



**Figure 11.5:** Some authorization items for the CMS example.

Once you've sketched out your items and hierarchy, you can begin defining the authorization items in Yii.

{TIP} A third way of enforcing authorization in Yii is to use proper Access Control Lists (ACL), a more formal and thorough implementation of Yii's built-in access lists. There's an [ACL extension](#) for this purpose.

### Establishing the Authorization Manager

RBAC requires the use of an *authorization manager* to function. The authorization manager is first used to define authorization items, and later performs the act of confirming the user's ability to perform a specific task.

Yii provides two classes of auth managers for you:

- **CPhpAuthManager**, which uses a PHP script
- **CDbAuthManager**, which uses the database

Normally you'll use the database, which is what I'll focus on here.

You tell your application which authorization manager to use by configuring the "authManager" application component:

```
# protected/config/main.php
// Other stuff.
'components'=>array(
    // Other stuff.
    'authManager'=>array(
        'class'=>'CDbAuthManager',
        'connectionID'=>'db',
    ),
),
// Other stuff.
```

As you can see in that code, when using the `CDbAuthManager` you have to also provide a connection identifier. In this case, that's the "db" application component. Yii can create the database tables for you but it's better if you do so yourself, using the SQL commands found in the framework's `**web/auth/schema-*.sql**` file.

When using `CPhpAuthManager`, you don't need to identify the database connection. Instead, you'd set the `authFile` property to the name of the PHP script that serves the same role. If you don't customize this property, the "authManager" component will use the file `protected/data/auth.php` by default.

Note that no matter what PHP script you use for the authorization manager, the script must be readable and writable by the Web server. That's because the authorization manager first acts as a storage mechanism for the authorization items. Also note that you normally shouldn't use the PHP script option (i.e., `CPhpAuthManager`). Using it for a complex set of rules will not be efficient, as reading from large text files is never as efficient as using an indexed database.

With the configuration in place, you can begin telling your Yii application what operations, tasks, and roles should exist, in that order.

## Defining Operations

To define an operation, you should first get a reference to the authorization manager:

```
$auth = Yii::app()->authManager;
```

A question you may have is: where do I put that code? You only need to establish the authentication items once for a site, so my recommendation would be to create a specific controller and/or action that performs the authorization initialization. For example, you might create an `actionSetup()` method in the "site" controller that

only administrators can execute. The administrator would then execute that action *once*, before the site is live. In fact, you may just create the authorization items once while developing the site, and then recreate them on the live site when you replicate the database.

{NOTE} A console script is a great choice for performing a site's setup. I'll discuss those in Chapter 16, "Leaving the Browser".

Next, invoke the `createOperation()` method to create each operation. Its first argument should be a unique name. Its second argument is an optional description. Here are a few operations based on Figure 11.5:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
$auth->createOperation('createPage');
$auth->createOperation('updatePage');
$auth->createOperation('updateUser');
$auth->createOperation('createComment');
$auth->createOperation('deleteUser');
```

Those five operations define a decent range of the kinds of things required by the CMS site. With the operations defined, you'd next define the tasks.

## Defining Tasks

Tasks are created via the `createTask()` method. Its first argument is a unique identifier and its second is an optional description. As a simple starting example (not represented in Figure 11.5), you might create a task that represents the creation of any site content (pages and files). Those two operations can go under one task:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
$auth->createOperation('createPage');
$auth->createOperation('createFile');
$task = $auth->createTask('createContent',
    'Allows users to create content on the site');
```

You would then associate the specific operations with that task, using the `addChild()` method, providing it with the name of the operation:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
$auth->createOperation('createPage');
```

```
$auth->createOperation('createFile');
$task = $auth->createTask('createContent',
    'Allows users to create content on the site');
$task->addChild('createPage');
$task->addChild('createFile');
```

{TIP} Operations can be assigned directly to roles (as in Figure 11.5), so tasks aren't always necessary, depending upon your structure.

Some tasks require a bit of business logic. For example, a user should be able to update her own record (e.g., change her password) or an author should be able to update a page he created. To add business logic to a task, provide a third argument to the `createTask()` method. This should be a string of PHP code whose returned Boolean value allows or denies the action:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
// Create operations.
$task = $auth->createTask('updateOwnUser',
    'Allows a user to update her record',
    'return $params["id"] == Yii::app()->user->id;');
$task->addChild('updateUser');
```

{TIP} Remember that references to `Yii::app()->user->id` will only return the user's primary key value if you've overwritten the `getId()` method in `UserIdentity`.

Perhaps that additional bit of code has left you confused. If so, that's understandable, and I'll go through it slowly. The business rule needs to return a Boolean. In this particular case, true should be returned if the current user's ID, represented by `Yii::app()->user->id` matches the `id` value of the user record being edited. Hopefully that side of the comparison makes sense. But where did `$params["id"]` come from?

The `$params` array will be populated (or, to be precise, *can* be populated) when the authorization is actually tested. In other words, when a user goes to update a user record, the auth manager will be used to confirm that the action is allowed. In doing so, the auth manager will be provided with the ID value of the user record being updated, which then gets assigned to `$params["id"]`. You'll see this in action shortly.

Here, though, is the most important tip I can give you regarding business rules, one that's not emphasized enough elsewhere: make sure your rule ends with a semicolon! If it does not, the business rule will not work as you expect it to.

{WARNING} End your business rules with semicolons or bugs will occur in your RBAC!

## Defining Roles

Finally, you should create roles and add tasks or operations to those roles:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
// Create operations.
// Create tasks.
$role = $auth->createRole('public');
$role->addChild('updateOwnUser');
$role->addChild('createComment');
// And so on.
```

That code assigns one task to the public user and one operation to the public user. Also remember that in the CMS example, “public” is the lowest level of authenticated user, different from an un-authenticated guest. Certain permissions, such as the viewing of a page, the creation of a user record (for registering), or the reading of a user record (for logging in) won’t be restricted at all.

To make the most of the hierarchical structure, you would add not only tasks to some roles but also roles to other roles. An author is just a basic (i.e., public) user with the added ability of creating pages and files and editing pages and files she created. An administrator is an author user with the added abilities of:

- Editing any page, file, or comment
- Deleting any page, file, or comment
- Editing any user
- Deleting any user

Here’s how that might play out:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
// Create operations.
// Create tasks.
$role = $auth->createRole('public');
$role->addChild('updateOwnUser');
$role->addChild('createComment');
// And so on.
$role = $auth->createRole('author');
$role->addChild('public');
```

```
$role->addChild('updateOwnPage');
$role->addChild('updateOwnFile');
// And so on.
$role = $auth->createRole('admin');
$role->addChild('author');
$role->addChild('updatePage');
$role->addChild('updateFile');
$role->addChild('updateComment');
```

Note that in the code I'm making reference to operations not in Figure 11.5 or previously discussed, just to give the code more bulk. You'll also notice that in my design, the "update page" operation is linked to the administrator twice: once directly and once through the "author" user. The "author" user doesn't actually have "update page" functionality, only "update own page", with the extra logic enforced. An administrator could update her own page, if she created it, but also needs to be able to update any page.

{TIP} To make this example a bit simpler, I did not create a "content" management task that is the parent of pages and files, although you certainly could.

Again, taking advantage of the hierarchical structure of users and permissions makes building up complex authorization structures much, much easier. The last step in the definition process is to assign roles to specific site users.

{TIP} If all this seems a bit complicated, there are Yii extensions that can simplify the process for you.

## Assigning Roles to Users

Finally, you need to associate authenticated users with authorization roles. This is done via the `assign()` method. Its first argument is the role being assigned and the second is an identifier of the user to which the role is assigned. If you are using static logging in (against an array of values, as in the default code), you would do this:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
// Create operations.
// Create tasks.
// Create roles.
$auth->assign('public', 'demo');
$auth->assign('admin', 'admin');
```

Now the “demo” user has been assigned certain tasks and the “admin” user has been assigned those tasks and more. Note that you only have to invoke the `assign()` method once (not each time the user logs in), as that creates the record in the database.

The last step is to save the actual rules you’ve established:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
// Create operations.
// Create tasks.
// Create roles.
// Assign users.
$auth->save();
```

All of the code to this point creates a slew of records in the database or text file, which will then be used to test authorization.

{TIP} If you need to tweak or re-define your authorization items, first clear the underlying database tables and then rerun your setup action.

In a few pages, I’ll explain how to tie the role assignments to database users, but let’s go with this for the time being, while I demonstrate how to enforce the authorization items you’ve declared.

{TIP} Roles can also be assigned using default roles, as explained in the [Guide](#).

## Enforcing Authorization

All of the code to this point establishes the authorization rules (as a text file or a database). Now you can make use of those rules to allow users to perform tasks. Understand that RBAC is better for defining what’s allowed, unlike the simpler access rules that can establish what’s allowed and what’s denied. In RBAC, if an action is not specifically allowed, then it’s denied by default.

There are two ways to enforce RBAC authorization: using access control roles or using the “user” component’s `checkAccess()` method. I normally end up using a combination of both.

To use RBAC with access control, make sure that access control is enabled as a filter:

```
# protected/controllers/PageController.php
public function filters() {
    return array(
        'accessControl',
        'postOnly + delete',
    );
}
```

Then you define your access rules, as you would when using simple access control, but this time also use the “roles” index, indicating the roles that should be allowed to execute certain actions:

```
# protected/controllers/PageController.php::accessRules()
return array(
    // Anyone can use "index" and "view":
    array('allow',
        'actions'=>array('index','view'),
        'users'=>array('*'),
    ),
    // Only admin roles can create and update content:
    array('allow',
        'actions'=>array('create','update'),
        'users'=>array('@'),
        'roles'=>array('admin')
    ),
    // And so on.
    array('deny', // deny all users
        'users'=>array('*'),
    ),
);
```

You’ll notice there’s a combination of basic access and RBAC there. This allows any guest to view pages, but only administrator (roles) to create or update them. (This particular example is limited, given only two users, but you’ll see a more dynamic version shortly.)

Sometimes you’ll want to check authorization within a specific controller action. That’s accomplished via the `CWebUser` class’s `checkAccess()` method. (As a reminder, `CWebUser` is the “user” component.) Provide to this method the name of the operation to be performed and it will return a Boolean indicating if the current user has that permission:

```
# protected/controllers/SomeController.php
public actionSomething() {
    if (Yii::app()->user->checkAccess('doThis')) {
```

```
        // Code for doing this.  
    } else {  
        // Throw an exception.  
    }  
}
```

Or, considering that exceptions stop the execution of a function, you could simplify the above to:

```
# protected/controllers/SomeController.php  
public actionSomething() {  
    if (!Yii::app()->user->checkAccess('doThis')) {  
        throw new CHttpException(403,  
            'You are not allowed to do this.');// Code for doing this.  
    }  
}
```

Another use of `checkAccess()` is to confirm that your RBAC is setup properly. Just do this in a view file to test all your permissions:

```
<?php  
echo '<p>Create comment: ' .  
    Yii::app()->user->checkAccess('createComment') .  
    '</p>';  
// Continue for the other permissions.  
?>
```

The output will be blank for lacking permission, and ones for granting permissions (**Figure 11.6**).

## Authorization with Database Users

The code to this point, and much that you'll find elsewhere, associate roles with static usernames. That's fine in those rare situations where you're using static users (in which case, you may not even need the complexity of RBAC), but most dynamic sites store users in a database table. How do you associate database users with RBAC roles?

The goal is to invoke the `assign()` method once *for each user*, as that's what the RBAC system will need in order to confirm permission.

The first thing you'll need to do is determine what user identifier counts. In other words: what table column and model attribute differentiates the different roles? Logically, this would be a property such as `user.type` in the CMS example. The goal, then, is to do this:

## Hello, author!

Create comment: 1

Create page: 1

Update page: 1

Delete user:

Update user:

**Figure 11.6:** Testing the permissions for an author user type.

```
if ($user->type === 'admin') {
    $auth->assign('admin', $user->id);
} elseif ($user->type === 'author') {
    $auth->assign('author', $user->id);
} elseif ($user->type === 'public') {
    $auth->assign('public', $user->id);
}
```

That code associates the user's ID with a specific RBAC role. As each `$user->type` value directly correlates to a role, that code can be condensed to:

```
$auth->assign($user->type, $user->id);
```

Second, you need to determine *when* it would make sense to invoke `assign()`. A logical time would be after the user registers. To do that, you could create an `afterSave()` method in the model class:

```
# protected/models/User.php
public function afterSave() {
    if (!Yii::app()->authManager->isAssigned(
        $this->type, $this->id)) {
        Yii::app()->authManager->assign($this->type,
            $this->id);
    }
    return parent::afterSave();
}
```

That code will be called after a model record is saved. This could be after a new record is created or after it is updated (like when the user changes her password). Because the second possibility exists, this code first checks that the assignment has not already taken place. If not, then the assignment is performed.

{TIP} If you have a situation where the user's permissions may be changed, you'd need to remove the existing role assignment and add the new one.

## Checking Parameters

Just a bit ago, I explained how to use the `checkAccess()` method within a controller action to confirm the ability to perform a task (as opposed to using an access rule). The most logical reason you'd check the authorization within an action is when a user is going to update a record that only that user should be allowed to update (e.g., the owner of a page can update a page). In those situations, the underlying business rule needs to know if the current user is the owner of the item in question. Here's the example of that from earlier:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
// Create operations.
$task = $auth->createTask('updateOwnUser',
    'Allows a user to update her record',
    'return $params["id"] == Yii::app()->user->id');
$task->addChild('updateUser');
```

Now you just need to know how to pass `$params` to the authorization item. To do that, provide a second argument to the `checkAccess()` method. The argument should be an array. Here's how that plays out in the "user" example:

```
# protected/controllers/UserController.php
public actionUpdate($id) {
    $model=$this->loadModel($id);
    if (!Yii::app()->user->checkAccess('updateUser',
        array('id' => $id))) {
        throw new CHttpException(403,
            'You are not allowed to do this.');
    }
    // Code for doing this.
}
```

That code passes the model's `id` property (i.e., the primary key) to the authorization item, giving it the index "id". When values are passed to an authorization item,

it receives them in a parameter named `$params`. Thus, `$params['id']` will be assigned the value of `$model->id` and the comparison can be made.

Alternatively, `$params['userId']` will automatically be added to the passed parameters, representing the value `Yii::app()->user->id`. So you could define the task's business logic to use it, if you'd prefer. Remember that this value represents the user's *name*, unless you overwrite the `getId()` method as previously explained.

Notice that the specific operation being checked is just “`updateUser`”, not “`updateOwnUser`”. This goes back to the hierarchy and how authorization is checked from the bottom up (see Figure 11.5). The functionality that's needed is the ability to update a user record. If an administrator goes to update the user, the hierarchy immediately shows that administrators have this ability, regardless of the user record in question (and authorization is allowed). If a non-administrator goes to update a record, the administrator path is blocked. The next path goes through “`update own user`”, so it's checked. If this is *not* the user's own record, then all paths have been blocked and permission is denied. If this is the user's own record, and the user is a public type *or an author*, then permission is allowed.

As another example, in which a foreign key value is checked, here's the code for only allowing an author of a page to edit that page:

```
# protected/controllers/SiteController.php::actionSetup()
$auth = Yii::app()->authManager;
// Create operations.
$task = $auth->createTask('updateOwnPage',
    'Allows a user to update pages she created',
    'return $params["ownerId"] == $params["userId"]');
$task->addChild('updatePage');
```

In that code I've switched to using `$params['userId']`, which is equivalent to `Yii::app()->user->id`.

And:

```
# protected/controllers/PageController.php
public actionUpdate($id) {
    $model=$this->loadModel($id);
    if (!Yii::app()->user->checkAccess('updatePage',
        array('ownerId' => $model->user_id))) {
        throw new CHttpException(403,
            'You are not allowed to do this.');
    }
    // Code for doing this.
}
```

The `user_id` value from the `Page` model record, which identifies the author of the page, gets passed to the task for comparison to the current user's ID. If they are

the same, the action is allowed. (The action will also be allowed to administrators, thanks to the direct assignment of the “update page” operation to the administrator type.)

## Handling Redirections

Switching gears a bit, Chapter 7 also mentioned what happens when Yii denies access to an action. If the user is not logged in and the access logic requires that she be logged in, the user will be redirected to the login page by default. After successfully logging in, the user will be redirected back to the page she had been trying to request. If the user *is* logged in but does not have permission to perform the task, an HTTP exception is thrown using the error code 403, which matches the “forbidden” HTTP status code value.

Looking at the other side of this equation, there are times when you’ll want to redirect the user upon a successful login. For example, if the user goes to a page and is denied (because the user was not logged in), it’d be nice if the site returned the user to the original intended destination upon successful login.

You can find out the URL of the user’s previous request via `Yii::app()->user->returnUrl`. You can redirect the browser using the `redirect()` method of the controller. Putting these two ideas together, the `actionLogin()` method of the “site” controller has an example of redirecting to the previously requested page:

```
if($model->validate() && $model->login())
    $this->redirect(Yii::app()->user->returnUrl);
```

## Working with Flash Messages

The last subject for this chapter is *flash messages*. Flash messages provide functionality that’s commonly needed by Web sites: an easy way to create and display errors or messages. Flash messages aren’t the same kind of storage mechanism as cookies, sessions, or user state, however:

- They are only available in the current request and the next request
- They are automatically cleared once used or a third request is made

Flash messages are most often used to convey the success or error of a recent user action.

You create a flash message by calling the `setFlash()` method of the `CWebUser` object, available in `Yii::app()->user`. This method should be provided with two arguments: an identifier and a value. This is normally done in a controller:

```
# protected/controllers/SomeController.php
public function actionSomething() {
    if (true) {
        Yii::app()->user->setFlash('success',
            'The thing you just did worked.');
    } else {
        Yii::app()->user->setFlash('error',
            'The thing you just did DID NOT work.');
    }
    $this->render('something');
}
```

{TIP} By default, flash messages are temporarily stored in the session.

As you'll see, the identifiers are just used to indicate whether a given type of flash message exists. It's the specific message that gets relayed to the user. Common identifiers are simple labels like "success" and "error", but the identifiers can be anything. One recommendation would be to align your flash message labels with your CSS classes, for reasons you'll soon see. As for the message itself, it must be a simple, scalar data type, such as a string.

{TIP} Flash messages are often used when redirection is also involved, as they provide a way to pass a message to be displayed on another page.

To use a flash message, invoke the `hasFlash()` method to see if a given flash message exists. Then use `getFlash()` to retrieve the actual message. This is most logically done in a view file:

```
<?php if(Yii::app()->user->hasFlash('success')):?>
    <div class="info">
        <?php echo Yii::app()->user->getFlash('success'); ?>
    </div>
<?php endif; ?>
```

Once the `getFlash()` method has been called to retrieve a flash message, it will be removed from user identity (i.e., you can't get a flash message twice without taking extra steps).

If you wanted to clear a flash message without using it, invoke `setFlash()`, providing the same identifier but no value:

```
// Whatever code.
Yii::app()->user->setFlash('success', null);
```

If it's possible that there would be more than one flash message, you can use `getFlashes()` to return them all and loop through them:

```
foreach (Yii::app()->user->getFlashes() as
    $key => $message) {
    echo '<div class="alert-' . $key . '">' .
        $message . '</div>';
}
```

In that particular bit of code, each flash message's identifier is used as part of the CSS class that wraps the message, letting you easily create one message with an "alert-info" class and another with an "alert-success" class (using the Twitter Bootstrap classnames).

## Chapter 12

# WORKING WITH WIDGETS

Widgets address a common concern with the MVC approach: you shouldn't put much programming logic in your view files, and yet, a decent amount of logic is required to render the proper output, particularly with Web sites. Once you factor in dynamic client-side behavior driven by JavaScript, the complexity of a view becomes even more elaborate. If you take as an example a navigable calendar or a dynamic table of data, you can appreciate how much HTML, JavaScript, and logic is required by one simple component on a page.

The focus in this chapter is on using widgets in general, and using the widgets defined within the Yii framework specifically. To start, you'll see how to add a widget to a view, and how to customize a widget's behavior. Then I'll walk through the usage and configuration of the most popular widgets.

## Using Widgets

If you're reading this book sequentially, then you will have already seen one use of a widget. By necessity, Chapter 9, “[Working with Forms](#),” used widgets, as the best way to create forms associated with models is to use the `CActiveForm` widget.

Widgets are, at their root, just an instance of a class. Specifically, the class must itself be `CWidget`, or, more commonly, a class that extends that. Yii has dozens of applicable widget classes defined for you, such as:

- `CActiveForm`
- `CListPage` and `CLinkPager`, which provides for data paging
- `CBreadcrumbs`, for creating breadcrumbs
- `CCaptcha`, for creating a CAPTCHA with a form
- `CJuiWidget`, for implementing jQuery User Interface components
- `CMenu`, for creating HTML navigation menus
- `CTabView`, for creating a tab interface

In this chapter, I'm going to focus on these predefined widget classes. In Part 3 of the book, you'll see how to create your own widget class.

Once you know what class you'll be using, you can create a widget in one of two ways. The first is to invoke the `widget()` method of the controller object. Its first argument is the name of the widget class to use and its second is for customizing that class instance (which is to say the widget).

```
# protected/views/thing/page.php
<?php $this->widget('ClassName',
    array(/* customization */)); ?>
```

Most widget classes are defined in Yii's `system` package (or a sub-package thereof), such as `CActiveForm` (**Figure 12.1**).

## CActiveForm



+1

&lt;

23

---

[All Packages](#) | [Properties](#) | [Methods](#)

<b>Package</b>	<a href="#">system.web.widgets</a>
<b>Inheritance</b>	class CActiveForm » <a href="#">CWidget</a> » <a href="#">CBaseController</a> » <a href="#">CComponent</a>
<b>Subclasses</b>	<a href="#">CCodeForm</a>
<b>Since</b>	1.1.1
<b>Source Code</b>	<a href="#">framework/web/widgets/CActiveForm.php</a>

**Figure 12.1:** The package and inheritance details for `CActiveForm`.

When a widget class is defined within `system`, you can just provide the class name when creating an instance of that widget:

```
<?php $this->widget('CCaptcha'); ?>
```

For classes not within `system`, you need to provide a complete reference to the class. All of the other classes you'll use in this chapter are in the `zii` family of packages, such as the jQuery UI accordion class:

```
<?php $this->widget('zii.widgets.jui.CJuiAccordion',
    array(/* customization */)); ?>
```

The alternative way to instantiate a widget is to use the `beginWidget()` method. This is to be followed by content that gets captured by the widget. And finally you invoke `endWidget()` to complete the widget creation. This is how the `CActiveForm` widget is used:

```
<?php $form=$this->beginWidget('CActiveForm',
    array(/* customization */)); ?>
<p class="note">Fields with <span class="required">*</span>
    are required.</p>
<?php echo $form->errorSummary($model); ?>
<div class="row">
    <?php echo $form->labelEx($model, 'name'); ?>
    <?php echo $form->textField($model, 'name'); ?>
    <?php echo $form->error($model, 'name'); ?>
</div>
<!-- And so on. -->
<?php $this->endWidget(); ?>
```

With that particular case, which is the most frequent use of `beginWidget()` and `endWidget()` that you'll see, those method calls end up creating the opening and closing FORM tags, with the form being written between them.

There are two challenges to using widgets in Yii:

- Knowing what widgets exist
- Customizing the widgets to function as you need them to

Over the rest of this chapter, I'll introduce what I think are the most important widgets. You can find others by searching online, searching the [Yii site](#), asking in the [Yii forums](#), or by looking in the [class docs](#) to see what classes extend `CWidget` and its children.

To customize the widgets—to know what to provide as an array to the `widget()` or `beginWidget()` method, you'll want to look at the public, writable attributes of the associated class. For example, `CActiveForm` has the following public, writable attributes (plus a couple more):

- `action` dictates the form's “action” attribute
- `enableAjaxValidation` turns Ajax validation on or off
- `enableClientValidation` turns client-side validation on or off
- `errorMessageCssClass` sets the CSS class used for errors
- `method` dictates the form's “method” attribute

Knowing this, you can use those attribute names for your configuration array's indexes. For the values, if it's not obvious what an appropriate value or value type would be, check out the class's documentation for those attributes. A simple customization:

```
<?php $form = $this->beginWidget('CActiveForm', array(
    'enableAjaxValidation' => true,
    'enableClientValidation' => true,
    'errorMessageCssClass' => 'error'
)); ?>
<!-- And so on. -->
<?php $this->endWidget(); ?>
```

The class docs also indicate the default properties, so you know whether customizations are even required.

## Basic Yii Widgets

To start, let's work with the non-jQuery UI widgets that are part of the Yii framework. These are all defined within the `system.web.widgets` package or the `zii.widgets` package. If you created a site using `yiic` and Gii, you'll already have several widgets in use:

- `CMenu` in the `main.php` layout file
- `CBreadcrumbs` in the `main.php` layout file
- `CPortlet` in the `column2.php` layout file
- `CActiveForm` for all the forms
- `CCaptcha` for implementing CAPTCHA on a form
- `CListView` on the `index.php` pages
- `CDetailView` on the `view.php` pages
- `CGridView` on the `admin.php` pages

I'll write a bit about each of these except for two: `CActiveForm` was covered in Chapter 9, and `CPortlet` really just provides a way to wrap the presentation of some content. There's not much to explain about it.

### Captcha

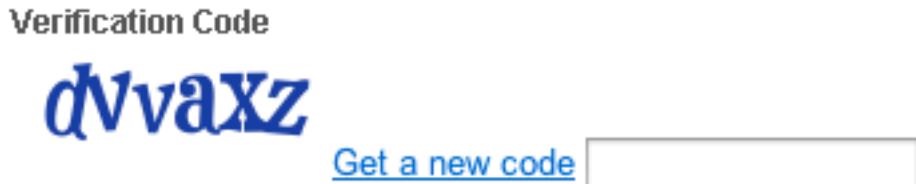
First up, let's take a look at how one implements *CAPTCHA*: Completely Automated Public Turing test to tell Computers and Humans Apart (how's that for an acronym?). I can actually explain how to use this widget very quickly, as the code generated by `yiic` implements CAPTCHA on the contact form already:

```
# protected/views/site/contact.php
<?php if(CCaptcha::checkRequirements()): ?>
<div class="row">
```

```
<?php echo $form->labelEx($model, 'verifyCode'); ?>
<div>
<?php $this->widget('CCaptcha'); ?>
<?php echo $form->textField($model, 'verifyCode'); ?>
</div>
<div class="hint">Please enter the letters as they are
    shown in the image above.
<br/>Letters are not case-sensitive.</div>
<?php echo $form->error($model, 'verifyCode'); ?>
</div>
<?php endif; ?>
```

The `CCaptcha` class requires the GD extension in order to work. As a safety measure, you can invoke the `checkRequirements()` method to confirm that the minimum requirements are met prior to attempting to use the class. If that method returns true, then you simply create the widget as you would any other: `$this->widget('CCaptcha')`.

The widget itself creates the HTML IMG tag that shows the CAPTCHA image. You can configure the CAPTCHA presentation a bit by setting the various `CCaptcha` class attributes, such as customizing how the user would refresh the CAPTCHA image (**Figure 12.2**).



**Figure 12.2:** The CAPTCHA widget displays the image and creates the “Get a new code” link.

As you can see in the form, you also need a text input where the user would enter what she thinks the CAPTCHA value is.

Turning to the model associated with the form, you need to apply the CAPTCHA validation rule to the text input. Again, this can be set to allow for an empty value if the PHP installation does not meet the minimum requirements:

```
# protected/models/ContactForm.php::rules()
array('verifyCode', 'captcha',
    'allowEmpty'=>!CCaptcha::checkRequirements()),
```

If the `CCaptcha::checkRequirements()` method returns false, meaning that the GD library is not available, then the “allowEmpty” option will be set to true (the opposite of false, which the method returns).

Finally, there’s the controller. Unlike most other widgets you’ll use, you need to tell the controller to do something when using CAPTCHA. Specifically, you need to have the controller create the CAPTCHA image (the widget itself just creates the IMG tag). Having the controller create the CAPTCHA image is done by adding an action to the controller:

```
# protected/controllers/SiteController.php
public function actions() {
    return array(
        'captcha' => array ('class' => 'CCaptchaAction')
    );
}
```

The `CCaptchaAction` class will actually create the image using the GD library.

And that’s all there is to it! If you ever need to use CAPTCHA on one of your forms, just copy and tweak the code already generated for you by `yiic`.

## CMenu

The `CMenu` class provides a way to display a hierarchical navigation menu using nested HTML lists. With the default code created by `yiic` and Gii, the `main.php` page ends up with this code:

```
<?php $this->widget('zii.widgets.CMenu',array(
    'items'=>array(
        array('label'=>'Home', 'url'=>array('/site/index')),
        array('label'=>'About', 'url'=>array('/site/page',
            'view'=>'about')),
        array('label'=>'Contact',
            'url'=>array('/site/contact')),
        array('label'=>'Login', 'url'=>array('/site/login'),
            'visible'=>Yii::app()->user->isGuest),
        array('label'=>'Logout ('.Yii::app()->user->name.')',
            'url'=>array('/site/logout'),
            'visible'=>!Yii::app()->user->isGuest)
    ),
)); ?>
```

That creates four menu items, as the Login/Logout items will only appear if the user is or is not logged in (**Figure 12.3**).



**Figure 12.3:** The default navigation menu.

Configuring the widget is a matter of assigning values to the writable properties of the `CMenu` class. There are only a few, such as `activeCssClass` for indicating the name of the CSS class to be applied to the currently active menu item. There are similar properties for setting the CSS class for the first and last item in the menu (or submenu).

{TIP} The `CMenu` widget automatically applies a class to highlight the active page, as demonstrated in Figure 12.3.

The most important property is `items`. It's used to establish the navigation items, and is declared as an array. Each item is represented by its own array, with any of the following indexes:

- `active`
- `itemOptions`
- `items`
- `label`
- `linkOptions`
- `submenuOptions`
- `template`
- `url`
- `visible`

The most important of these are “label” and “url”, as shown in the default code. The label is displayed to the user. The URL value is used for the link. For it, follow the same rules as for `normalizeUrl()` (covered in Chapter 7, “Working with Controllers”), with one exception: if you specify a controller, you must also specify the action (i.e., you cannot rely upon the default controller action). If you just specify an action value, the current controller will be used. If you don't specify a URL, the result will be an unlinked SPAN. That would be appropriate when working with submenus.

To create a submenu, provide an “items” subarray to an individual item. In theory. How well this works will depend upon your CSS, HTML, and such. The fact is that `CMenu` is better for simpler presentations of menus. More elaborate menus are best left to extensions.

Still `CMenu` is easy to use for basic menus and you should have no problems manipulating the code created by `yiic` for the main navigation menu. But the generated

code uses a second instance of `CMenu`, which may be a little confusing. Code created by Gii will have lines like this:

```
# protected/views/user/index.php
$this->menu=array(
    array('label'=>'Create User', 'url'=>array('create')),
    array('label'=>'Manage User', 'url'=>array('admin')),
);
```

What's going on there? It's actually a simpler concept than you might first imagine. All controllers in the generated code extend the `protected/components/Controller.php` class. That class creates a `menu` property. The above code is just assigning a value to that already declared class attribute.

During the rendering of the view file, this property is used by the `column2.php` layout file:

```
$this->widget('zii.widgets.CMenu', array(
    'items'=>$this->menu,
    'htmlOptions'=>array('class'=>'operations'),
));
```

That code says to use the controller's `menu` attribute value for the `CMenu` class's items. In short, this is an easy way to define a needed value in one place (a view file) and use it in another (the layout file).

## CBreadcrumbs

Another widget created by the `yiic` command is `CBreadcrumbs`. It's used to create the breadcrumbs effect at the top of the page, which users and search engines alike both appreciate (**Figure 12.4**).



**Figure 12.4:** A rather short trail of breadcrumbs.

The most important property is `links`, which should be an array. If an array element has an index and a value, that will be used for the link label and URL. If just provided with a value, the value will be the label and no link will be created. This structure allows you to link to items higher up the breadcrumb trail but only show, not link, the current page. For example, Figure 12.4 shows the (current) “About” page unlinked, with the “Home” page linked:

```
<?php $this->widget('zii.widgets.CBreadcrumbs', array(
    'links'=>array(
        'About'
    ),
)); ?>
```

Note that the home page is always assumed and you never have to specify it. Here's another example (**Figure 12.5**):

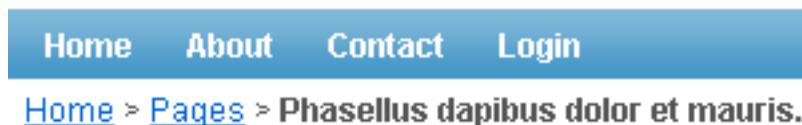


**Figure 12.5:** The breadcrumb trail to a specific page being viewed.

```
# protected/views/page/view.php (in theory)
<?php $this->widget('zii.widgets.CBreadcrumbs', array(
    'links'=>array(
        'Pages'=>array('index'),
        $model->title,
    ),
)); ?>
```

The order the items are listed in the array is the order in which they will be listed (from left to right) in the breadcrumbs.

If you look at the documentation for the `CBreadcrumbs` class, you'll find the attributes that are public and writable, such as `separator`, for defining the character(s) placed between items (**Figure 12.6**):



**Figure 12.6:** The same trail with a different separator.

```
# protected/views/page/view.php (in theory)
<?php $this->widget('zii.widgets.CBreadcrumbs', array(
    'links'=>array(
        'Pages'=>array('index'),
        $model->title,
    ),
)); ?>
```

```
'separator' => ' > '
)); ?>
```

The only other thing to know about breadcrumbs is that the default generated code uses a trick similar to the `CMenu` trick for defining the breadcrumb items in the view file but creating the `CBreadcrumb` widget instance in the layout:

```
# protected/views/page/view.php
$this->breadcrumbs=array(
    'Pages'=>array('index'),
    $model->title,
);
# protected/layouts/main.php
<?php if(isset($this->breadcrumbs)):>
    <?php $this->widget('zii.widgets.CBreadcrumb', array(
        'links'=>$this->breadcrumbs,
    )); ?><!-- breadcrumbs -->
<?php endif?>
```

The value of `$this->breadcrumbs` provided in the controller will be used for the array of links in the main layout file. If you want to customize the breadcrumbs behavior, you'd do that in the main layout file.

## Presenting Data

The next group of widgets to be covered are all similar to each other in that they present data. Some widgets present multiple records at once, offering great features like sorting, pagination, and filtering, while others present just a single record. Therefore, before I get into the specifics of each widget, I must first discuss how you provide data to them. For some of these widgets, you don't just provide an array of objects, but rather a specific kind of data type, one that supports features such as sorting, pagination, and filtering. Let's look at the data types first.

### Data Formats

The data formats you'll use with some of the Yii widgets are defined as classes that implement the `IDataProvider` interface. The base class is `CDataProvider`, which is then extended by three child classes. All three class types can be used in the same way as a data source for a widget. They differ in where they get their data from:

- `CActiveDataProvider`, uses an Active Record model
- `CArrayDataProvider`, uses an array

- `CSqlDataProvider`, uses an SQL query

Put another way, each of these classes is a wrapper that can take a different data source and make it universally usable by the various widgets.

You create an object of one of these three types using this syntax:

```
$dp = new CClassType(<source>, <configuration>);
```

The class types have already been mentioned. The source will be a model name (for `CActiveDataProvider`), an array (for `CArrayDataProvider`), or an SQL query (for `CSqlDataProvider`). For example, here is how you would create a data source from all of the `User` records:

```
$dp = new CActiveDataProvider('User');
```

That code executes a `User::model()->fetchAll()` query and returns the results as a `CActiveDataProvider` object, usable by a widget.

Here's how you'd accomplish the same thing using a direct query:

```
$dp = new CSqlDataProvider('SELECT * FROM user');
```

That code will run the query on the database and return the results as a `CSqlDataProvider` object, usable by a widget. Understand that this particular query is more simple than you'd normally use for `CSqlDataProvider`, but it works just the same.

{NOTE} Because using an array as a data source is less common when working with a database, I won't discuss it in this chapter. But you'll see an example in Chapter 16, “[Leaving the Browser](#),” in which I explain how to work with Web services.

Configuring the class is more involved, and more interesting. The second argument to each class's constructor can be an array of name=>value pairs for assigning values to the class's public, writable properties. What properties exist will depend upon the class, although there are many common ones.

When working with `CActiveDataProvider`, an important property is `criteria`, used to configure how the Active Record model fetches the records. These are the same `CDbCriteria` options explained in Chapter 8, “[Working with Databases](#)”. For example, say you wanted to retrieve only the “author” type users, in alphabetical order by username:

```
$dp = new CActiveDataProvider('User', array(
    'criteria' => array(
        'condition' => 'type="author"',
        'order' => 'username ASC'
    )
));
```

If you needed to fetch associated data from related models, you'd just add a "with" item to the array. For all the nitty gritty on `CDbCriteria`, see Chapter 8.

Another common configuration option for the various sources is "pagination". You can use it to dictate how pagination is accomplished with the data set. The most common configuration setting is how many items should be in a page of results. To set that, assign a value to the pagination's "pageSize":

```
$dp = new CActiveDataProvider('User', array(
    'criteria' => array(
        'condition' => 'type="author"',
        'order' => 'username ASC'
    ),
    'pagination' => array(
        'pageSize' => 10
    )
));
```

Pagination works the same when using an SQL command, except that you need to tell the class how many total records exist in that situation:

```
$q = 'SELECT COUNT(id) FROM user WHERE type="author"';
$count = Yii::app()->db->createCommand($q)->queryScalar();
$dp = new CSqlDataProvider('SELECT * FROM user
    WHERE type="author"',
    array(
        'totalItemCount' => $count,
        'pagination' => array(
            'pageSize' => 10
        )
    )
);
```

{WARNING} If your primary query uses a conditional to limit the results, the count query must use that same condition or else the number of pages won't match the number of records to display.

Adding sorting to the process is a bit more complicated, too. You should indicate the columns used for sorting as one configuration item. Then you set the default sorting order for those columns as true for DESC and false for ASC:

```
$q = 'SELECT COUNT(id) FROM user WHERE type="author"';  
$count = Yii::app()->db->createCommand($q)->queryScalar();  
$dp = new CSqlDataProvider('SELECT * FROM user  
    WHERE type="author"',  
    array(  
        'totalItemCount' => $count,  
        'pagination' => array(  
            'pageSize' => 10  
        ),  
        'sort' => array(  
            'attributes' => array('username'),  
            'defaultOrder' => array('username' => false)  
        )  
    )  
);
```

Once you've created a data source, most widgets will take that object as the value for its "dataProvider" property.

*{NOTE}* The data providers assume that "id" is the name of the primary key in the table. If not, assign the primary key column name to the "keyField" property.

## CLListView

Now I'm going to move into a series of widgets that make the presentation of data much, much easier. The first of these is **CLListView**. The **CLListView** class presents multiple records of data in a list (as opposed to a table). You can see an example of it on the **index.php** page created by Gii (**Figure 12.7**).

That output is created by the following code:

```
<?php $this->widget('zii.widgets.CListView', array(  
    'dataProvider'=>$dataProvider,  
    'itemView'=>'_view',  
)); ?>
```

You'll notice the data provider there, which can be any of the three class types already explained.

## Users

		Displaying 1-10 of 16 results.
		Sort by: <a href="#">Username</a> <a href="#">Email</a>
ID:	<a href="#">5</a>	<b>Username:</b> Administrator <b>Email:</b> admin@example.com <b>Password:</b> 21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6 <b>Type:</b> admin <b>Date Entered:</b> 2013-02-17 15:59:24
ID:	<a href="#">6</a>	<b>Username:</b> Another Admin <b>Email:</b> admin2@example.net <b>Password:</b> 21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6 <b>Type:</b> admin <b>Date Entered:</b> 2013-02-10 15:59:24
ID:	<a href="#">4</a>	<b>Username:</b> Another Author <b>Email:</b> blah@example.org <b>Password:</b> 21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6 <b>Type:</b> author <b>Date Entered:</b> 2013-01-27 15:59:24

**Figure 12.7:** A list of users.

The “itemView” index is used to identify the view file that will present each record being displayed. The above code specifies the individual view file as `_view.php` (in the same `views` subfolder). Here’s a snippet of that code:

```
<div class="view">
<b>
<?php echo CHtml::encode($data->getAttributeLabel('id')) ;
?></b>
<?php echo CHtml::link(CHtml::encode($data->id) ,
    array('view', 'id'=>$data->id)); ?>
<br />
<b>
<?php echo CHtml::encode($data->getAttributeLabel('username')) ;
?></b>
<?php echo CHtml::encode($data->username); ?>
<br />
// And so on.
```

For each item shown, the individual view file is passed a specific item as the `$data` array. If the data provider object contains Active Model `User` objects, then `$data` in the file will be a `User` instance. For that reason, you can reference the model’s methods and attributes accordingly. To change the presentation of the information in `CListView`, either edit the individual view file or use one of your own creation.

{TIP} Within the individual view file, the `$index` variable will represent the index number of the item being displayed, starting at 0.

Pagination of the records will automatically be applied. To enable sorting, set the “sortableAttributes” property of the `CListView` class:

```
# protected/views/user/index.php
<?php $this->widget('zii.widgets.CListView', array(
    'dataProvider'=>$dataProvider,
    'itemView'=>'_view',
    'sortableAttributes'=>array('username', 'email')
)); ?>
```

That will prompt Yii to create sorting links above the list (as in Figure 12.7). Clicking the links changes the order of the displayed items.

Not only does Yii create, and handle, the pagination and sorting features automatically, but it will do so using JavaScript, if enabled, or HTML, if not. To test this, click the pagination or sorting links and notice that no browser refreshes are required (if JavaScript is enabled). Then disable JavaScript in your browser and test it again: still works!

As you can see in Figure 12.7, the default layout of the entire list view is: summary (e.g., *Displaying x-y of z results*), followed by the sorting links, followed by the actual results, followed by the pagination links. To change the layout, set the `template` attribute. Use `{summary}`, `{sorter}`, `{items}`, and `{pager}` placeholders to insert those values dynamically:

```
# protected/views/user/index.php
<?php $this->widget('zii.widgets.CListView', array(
    'dataProvider'=>$dataProvider,
    'itemView'=>'_view',
    'sortableAttributes'=>array('username', 'email'),
    'template' => '{sorter}{items}<hr>{summary}{pager}'
)); ?>
```

There are more attributes you can set, too, that will do things like set the text before and after the sorting links, change the CSS classes used, and so forth. These are all explained in the [Yii class docs](#).

## CDetailView

The `CDetailView` widget is used to display information about a single record. An example of its usage is written into the `view.php` file by Gii ([Figure 12.8](#)).

# View Page #1

<b>ID</b>	1
<b>User</b>	3
<b>Live</b>	1
<b>Title</b>	Aliquam malesuada, ligula sit amet.
<b>Content</b>	<p> Lorem ipsum dolor sit amet, consectetur ad litora torquent per conubia nostra, p hendrerit odio porta non. Donec eu me elit rutrum at porta lacus aliquet. Pelle nascetur ridiculus mus.</p>

Figure 12.8: The view display for a single Page record.

Unlike CListView and CGridView, CDetailView expects as its data source either a single model instance or a single associative array. This is assigned to the CDetailView class's `data` property.

The second most important property is `attributes`. This is how you dictate which values in the model or array are displayed:

```
# protected/views/page/view.php:  
<?php $this->widget('zii.widgets.CDetailView', array(  
    'data'=>$model,  
    'attributes'=>array(  
        'id',  
        'user_id',  
        'live',  
        'title',  
        'content',  
        'date_updated',  
        'date_published',  
    ),  
)); ?>
```

{TIP} The order in which the attributes are listed are the order in which they will be displayed (from top to bottom).

If you don't want to display a value, just remove it from that list. If you're using an Active Record model instance and you want to show an attribute from a related model, use `relationName.attribute`. For example, `Page` is related to `User` via the `user_id` column. The defined relationship (in `Page`) is:

```
'pageUser' => array(self::BELONGS_TO, 'User', 'user_id'),
```

As the detail view should probably show the associated user's username, not the `user_id`, replace `user_id` in that list with `pageUser.username` (**Figure 12.9**).

## View Page #1

ID	1
Username	Some Author
Live	1
Title	Aliquam malesuada, ligula sit amet.
Content	<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ipsum tincidunt lacinia. Integer lacinia semper varius. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla lobortis nulla et leo egestas at luctus tellus tempor. Nulla faucibus pulvinar metus, quis hendrerit odio porta non. Donec eu metus tincidunt tellus convallis tincidunt a ac quam. Quisque a mollis lectus. Fusce nec pretium libero. Ut rhoncus augue eu elit rutrum at porta lacus aliquet. Pellentesque molestie viverra purus et lacinia. Nulla facilisi. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. </p><hr>Red hihenaeos lirula ac urna aeneac eulement. Nunc eu malectio

**Figure 12.9:** The same page with the author's name now displayed.

The general format for specifying how something is displayed is `Name>Type:Label`, with the last two being optional. The “type” value dictates how the value is formatted, with plain text being the default. Other possible values are:

- “raw” does not change the value
- “text” HTML-encodes the value
- “ntext” HTML-encodes the value and applies `nl2br()`
- “html” purifies and returns the value as HTML
- “date” formats the value as a date
- “time” formats the value as a time
- “date time” formats the value as a date and time
- “boolean” displays the value as a Boolean
- “number” formats the value as a number
- “email” wraps the value in a “mailto” link
- “image” creates the proper IMG tag to show the value
- “url” formats the value as a hyperlink

These options come from the `CFormatter` class, which is used to format the presentation of data. If you use data formatting a lot, you'll want to spend some time reading [its documentation](#).

By default, all values are shown as encoded text. With the page example, you may want to make a few changes (**Figure 12.10**):

```
# protected/views/page/view.php:  
<?php $this->widget('zii.widgets.CDetailView', array(  
    'data'=>$model,  
    'attributes'=>array(  
        'id',  
        'pageUser.username',  
        'live:boolean',  
        'title',  
        'content:html',  
        'date_updated',  
        'date_published',  
    ),  
)); ?>
```

## View Page #1

ID	1
Username	Some Author
Live	Yes
Title	Aliquam malesuada, ligula sit amet.
Content	<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ipsum tincidunt lacinia. Integer lacinia semper varius. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla lobortis nulla et leo egestas at luctus tellus tempor. Nulla faucibus pulvinar metus, quis hendrerit odio porta non. Donec eu metus tincidunt tellus convallis tincidunt a ac quam. Quisque a mollis lectus. Fusce nec prelum libero. Ut rhoncus augue eu elit rutrum at porta lacus aliquet. Pellentesque molestie viverra purus et lacinia. Nulla facilisi. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.</p>

Figure 12.10: The same page again, with a better display.

If you don't specify a label, the model's attribute label value will be used.

Instead of the “Name>Type:Label” structure for each attribute, you can format each attribute as an array. This flexibility allows you to further customize the output. The following code will change the displayed value to be the author’s username, linked to that author’s view page:

```
# protected/views/page/view.php:  
<?php $this->widget('zii.widgets.CDetailView', array(  
    'data'=>$model,  
    'attributes'=>array(  
        'id',  
        array(  
            'label' => 'Author',  
            'value' => CHtml::link(CHtml::encode(  
                $model->pageUser->username),  
                array('user/view', 'id'=>$model->user_id)),  
            'type' => 'raw'
```

```
        ),
        'live:boolean',
        'title',
        'content:html',
        'date_updated',
        'date_published',
    ),
)); ?>
```

{TIP} When you use a “value” element in the array, “name” will be ignored.

CDetailView uses a table row to display each item. Naturally, this is also customizable. To use, say paragraphs instead of a table, you would set the `tagName` property to NULL (or DIV, as a parent), then assign a value to the `itemTemplate` property. Use the placeholders `{class}`, `{label}`, and `{value}`:

```
# protected/views/page/view.php:
<?php $this->widget('zii.widgets.CDetailView', array(
    'data'=>$model,
    'tagName' => 'div',
    'itemTemplate' => '<p><b>{label}</b>: {value}</p>',
    'attributes'=>array(
        'id',
        array(
            'label' => 'Author',
            'value' => CHtml::link(CHtml::encode(
                $model->pageUser->username),
                array('user/view','id'=>$model->user_id)),
            'type' => 'raw'
        ),
        'live:boolean',
        'title',
        'content:html',
        'date_updated',
        'date_published',
    ),
)); ?>
```

## CGridView

The `CGridView` class is the true workhorse of the bunch. You can see it in action on the `admin.php` page (**Figure 12.11**).

Displaying 1-10 of 16 results.						
ID	Username	Email		Password	Type	Date Entered
1	Test	test@example.com		21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6	public	2013-01-01 00:00:00
2	Someones	me@example.org		21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6	public	2013-01-21 11:32:51
3	Some Author	auth@example.net		21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6	author	2013-03-03 15:59:24

**Figure 12.11:** The admin grid of users.

The `CGridView` presents a series of records in a table and provides the following functionality:

- Pagination
- Sorting
- Links to view, edit, or delete a record
- Basic searching by field
- Advanced searching
- Live search results via Ajax

If you've spent any time creating similar functionality, normally for the administration of a site, you can appreciate just how much work goes into implementing all that capability.

And here's the code that creates the widget:

```
# protected/views/user/admin.php
<?php $this->widget('zii.widgets.grid.CGridView', array(
    'id'=>'user-grid',
    'dataProvider'=>$model->search(),
    'filter'=>$model,
    'columns'=>array(
        'id',
        'username',
        'email',
        'pass',
        'type',
        'date_entered',
        array(
            'class'=>'CButtonColumn',
        ),
    ),
)); ?>
```

This is a great “bang for your buck” example: just a bit of code delivers tons of functionality. But, how, exactly does it work?

{NOTE} The dynamic display and form submission handling of the advanced search is accomplished via some JavaScript and jQuery, to be explained in Chapter 14, “[JavaScript and jQuery](#).”

Four `CGridView` attributes are set in that code: `id`, `dataProvider`, `filter`, and `columns`. The `id` value is only necessary because some JavaScript (for the advanced search) needs a quick reference to the grid. The `dataProvider` should be familiar to you, although its value—`$model->search()`—won’t be. The `filter` attribute is also a new one. And `columns` is how you dictate what columns are shown. Let’s look at each of these properties in detail. But first, let’s look at the controller that renders this view.

{NOTE} I could easily write an entire chapter on just this widget, considering all the permutations and configurations people may want to make to a `CGridView` widget. As one chapter on just this would be impractical (and still not exhaustive), I’ll cover the absolute fundamentals and rely upon you to search online for more and more examples as you need them.

## The `CGridView` Controller and Filter

The code generated by Gii creates the following in the controllers:

```
# protected/controllers/UserController.php
public function actionAdmin() {
    $model=new User('search');
    $model->unsetAttributes();
    if(isset($_GET['User']))
        $model->attributes=$_GET['User'];
    $this->render('admin',array(
        'model'=>$model,
    ));
}
```

As you can see, `$model`, which gets passed to the view, is an instance of the `User` class, but more specifically, it’s an instance of the “search” scenario. Scenarios were explained in Chapter 5, “[Working with Models](#),” but the short explanation is that you can set different validation rules for different situations. The default rules for the “search” scenario is to make every attribute safe:

```
# protected/models/User.php::rules()
array('id', 'username', 'email', 'pass', 'type',
    'date_entered', 'safe', 'on'=>'search'),
```

Thanks to that line, when code uses `$model=new User('search')`, all of the attributes are considered safe without passing any rules. This means that “varmit” will be accepted as an email address or an ID value! To know why this is *correct* requires an understanding of how model rules are used.

Model rules will only allow values to be assigned to model attributes if the values pass the validation rules. For example, only a syntactically valid email address can be assigned to the `User` model’s `email` attribute. That’s good, right? But the grid has a search component that will allow the user to look up records by attributes. A user, when performing a search, may only provide *part* of an email address for searching: instead of “test@example.com”, the user might search for just email addresses that start with “test” or use the “@example.com” domain. If you don’t make the `email` attribute safe without passing the email validation rule, the search functionality will be too limited. On the other hand, if you have model attributes that would never be used for searching, you should remove those from the rule, just to be more secure.

Returning to the controller, the next line is `$model->unsetAttributes();`. As a comment there indicates, this clears out any default values that might be in the model’s attributes. This way, what the user is searching for won’t be polluted by default model values.

Next, the model attributes are assigned values from the form, when submitted:

```
if(isset($_GET['User']))
    $model->attributes=$_GET['User'];
```

This is similar to how the `actionCreate()` method works, but this time GET is used instead of POST. You want to use GET here because the grid form submissions will be transmitted via Ajax using the GET method (as does the form itself, in case JavaScript is disabled).

By this line in the controller, if the user entered “test” in the email input, then `$model->email` would have a value of “test” and no other model attribute would have a value. This model instance will be used by the grid’s filter to limit what records are shown:

```
# protected/views/user/admin.php
<?php $this->widget('zii.widgets.grid.CGridView', array(
    'id'=>'user-grid',
    'dataProvider'=>$model->search(),
    'filter'=>$model,
    // And so on.
```

The `CGridView` class's `filter` attribute is optional, but accepts a model instance as its value. If no `filter` attribute value is set, then the filtering boxes above the grid would not be shown. You can set the `filterPosition` attribute of the widget to "header", "body", or "footer" to change where the boxes appear: just above the column headings, just below the column headings ("body", the default), or below the final record.

## The `CGridView` Data Provider

For the `dataProvider` attribute of the grid widget, the value is `$model->search()`. In other words, the data for the grid will be returned by the `search()` method of the model instance. Here's what that method looks like:

```
#protected/models/User.php
public function search() {
    $criteria=new CDbCriteria;
    $criteria->compare('id',$this->id,true);
    $criteria->compare('username',$this->username,true);
    $criteria->compare('email',$this->email,true);
    $criteria->compare('pass',$this->pass,true);
    $criteria->compare('type',$this->type,true);
    $criteria->compare('date_entered',
        $this->date_entered,true);
    return new CActiveDataProvider($this, array(
        'criteria'=>$criteria,
    ));
}
```

At the end of the method, you can see that a `CActiveDataProvider` object is returned. As already explained, its first argument is the class being used, which is represented by the magical keyword `$this`.

And, as also explained earlier in the chapter, the second argument to the `CActiveDataProvider` constructor can be used to configure how the records are returned. In this case, that's a matter of setting criteria for what records are selected. Logically, the records will be selected based upon the search criteria provided by the user. The desired result is to create some number of conditions in the WHERE clause that would be added to the SELECT query, equivalent to `SELECT * FROM user WHERE email LIKE '%@example.com%' AND type='public'`, as an example. Those conditions are added to the criteria by the `compare()` method, which I've not previously discussed.

The `CDbCriteria` class's `compare()` method is used to add comparison expressions to a criteria. In situations where the condition may be built up dynamically, `compare()` is a much better solution than trying to create your own complex "condition" value. The `compare()` method takes up to five arguments:

- The name of the column to be used in the comparison
- The comparison value
- Whether a full or partial match should be made (i.e., an equality comparison or a LIKE comparison, with the default being a full equality match)
- How this condition should be appended to any existing condition, with the default being AND
- Whether the value needs to be escaped (which you'd want to do when allowing for partial matches, if % or \_ might be in the value)

As you can see in the code, the current model instance's values are used for the values of the comparisons. When a user enters "test" in the username box, `$model->username` gets a value of "test" in the controller, meaning it will have that value in the view and in this method when it's called. If a model has an empty value for any attribute, then that condition is *not* added to the query.

The default code results in partial match conditions separated by AND. If the user enters "@example.com" for the email address and "public" for the type, the result will be a query like `SELECT * FROM user WHERE email LIKE '%@example.com%' AND type LIKE '%public%'`. Understanding how this works, there are some edits you'll likely want to make to the `search()` method.

First, remove any column that should not be searchable. Also remove that column from the "search" scenario rule. You may even want to remove that column from the displayed list in the grid (that's up to you).

Second, see if you can't change any comparison from a partial match to a full match. LIKE conditionals are much less efficient to run on the database than equality conditionals. In a situation like an email address or a username, you'd probably need to allow for partial matches. For numeric columns, however, partial matches often don't make sense. For example, if you were to allow the results to be searched by primary key, you wouldn't want a partial match there, as the primary key 23 should not also bring up 123, 238, and 4231. Similarly, ENUM or SET columns can be set to full matches if you also edit the filtering so that the user can only select from appropriate values (more on that shortly).

Third, change the final parameter to false if a partial match is allowed but it's not logical for a value to contain an underscore or a percent sign (i.e., those two characters with special meaning in a LIKE conditional). The most common example would be an email address.

You can also change the conditional operator from AND to OR using the fourth argument, if you prefer. If you do so, make sure you change them all to be consistent. And, more importantly, add some text to the view to notify the user that multiple search criteria expands the search results, not limits them.

## Customizing the Display

The data provider and the filter dictate which rows of records are displayed. You can also change what columns are displayed, or how they are displayed. The first way of doing so is to change the values listed for the `CGridView` class's `columns` attribute. This is an array of attribute names by default. To start customizing this aspect, remove any attributes you don't need to show, such as a user's password.

{TIP} The order of the column listings dictates the order of the displayed columns in the grid, from left to right.

From there, you can customize the column values the same way you customize the `attributes` property in the `CDetailView`. For example, the `Page` model's `live` attribute is a 1 or a 0. It would make more sense to display that as "Live" or "Draft" (or something like that). Just change the value displayed accordingly:

```
# protected/views/page/admin.php
<?php $this->widget('zii.widgets.grid.CGridView', array(
    'id'=>'page-grid',
    'dataProvider'=>$model->search(),
    'filter'=>$model,
    'columns'=>array(
        'id',
        'pageUser.username',
        array (
            'header' => 'Live?',
            'value'=>'($data->live == 1) ?
                "Live" : "Draft"',
        ),
        'title',
        'date_updated',
        array(
            'class'=>'CButtonColumn',
        ),
    ),
)); ?>
```

For the `user_id` value, you probably instead want to change the displayed value from the author's ID to the actual author name. Just use `relationName.attribute` as explained for `CDetailView`. The above code already does this (**Figure 12.12**).

{NOTE} Changing the displayed values will probably cause problems with the filtering/search functionality as in Figure 12.12. I'll explain why, and the fix, shortly.

Displaying 1-10 of 22 results.						
ID	Username	Live?	Title	Date Published		
1	Some Author	Live	Aliquam malesuada, ligula sit amet.	2013-02-17		
2	Another Author	Live	Ten years ago a...	2013-02-22		
3	Moe	Draft	Phasellus dapibus dolor et mauris.			
4	Another Author	Live	Thunder, thunder, thundercats, Ho!	2013-02-01		

Figure 12.12: The updated page grid view.

Sometimes columns may have NULL or empty values and you'll want to display something to indicate that status. To do so, set the `nullDisplay` and/or `blankDisplay` values:

```
# protected/views/page/admin.php
<?php $this->widget('zii.widgets.grid.CGridView', array(
    'id'=>'page-grid',
    'dataProvider'=>$model->search(),
    'filter'=>$model,
    'nullDisplay' => 'N/A',
    'blankDisplay' => 'N/A',
    'columns'=>array(
        // Actual columns.
    ),
)); ?>
```

{TIP} There are multiple properties for changing the CSS classes involved, too. See the [Yii docs](#) for specifics.

## Customizing the Buttons

With the default code, in the far right column of the grid, three buttons are displayed: view, update, and delete. If you want to change these, you need to configure the `CButtonColumn` class. The two most important properties are `buttons` and `template`. The `template` property lays out the buttons. You can use the `{view}`, `{update}`, and `{delete}` placeholders to reference default buttons.

This code removes the delete option and swaps the order of the other two:

```
<?php $this->widget('zii.widgets.grid.CGridView', array(
    'id'=>'page-grid',
    'dataProvider'=>$model->search(),
    'filter'=>$model,
    'columns'=>array(
```

```
// Other columns,
array(
    'class'=>'CButtonColumn',
    'template'=>'{update} {view}'
),
),
));
?>
```

If you want to change the images used for the buttons, assign new values to the `deleteButtonImageUrl`, `updateButtonImageUrl`, and `viewButtonImageUrl` properties.

You can also define your own buttons via the `buttons` property. This is explained in the Yii class docs for [CButtonColumn](#).

## More Complex Searches

The searching and filtering built into the grid is a wonderful start, but can be improved. For example, the available user types are only “public”, “author”, and “admin”. There’s no point in allowing the user to search by *any* user type value, and it would be far more accurate to pre-set those values. To accomplish that, you’d need to change the filter box for the user type column from a text input to a drop down menu. That’s done by setting the “filter” index of a column:

```
# protected/views/user/admin.php:
<?php $this->widget('zii.widgets.grid.CGridView', array(
'id'=>'user-grid',
'dataProvider'=>$model->search(),
'filter'=>$model,
'columns'=>array(
    'id',
    'username',
    'email',
    array(
        'value' => 'ucfirst($data->type)',
        'filter' => CHtml::dropDownList('User[type]', $model->type, array('public' => 'Public',
            'author' => 'Author', 'admin' => 'Admin'),
            array('empty' => '(Select)'))
    ),
    'date_entered',
    array(
        'class'=>'CButtonColumn',
    ),
),
```

```
),
)); ?>
```

And that will do that. Displaying the drop down list is fairly easy. To get the filtering of the grid to work, you just need to associate the model attribute—`User[type]`—with the drop down menu by providing it as the first argument as in the above code (**Figure 12.13**).

The screenshot shows a Yii application interface. At the top, there's a blue header bar with the word 'Email'. Below it is a table with three columns: 'example.net', 'example.org', and 'example.org'. The second column contains the values 'Author' and 'Admin'. A dropdown menu is open over the second row, specifically over the value 'Author'. The dropdown has a white background and a thin black border. It contains four options: '(Select)', 'Public', 'Author' (which is highlighted with a blue background and a checked checkmark), and 'Admin'. A mouse cursor is visible, pointing at the 'Author' option. The table rows have alternating light blue and white backgrounds.

**Figure 12.13:** Users can now be filtered by specific type values.

Because the values returned by the drop down exactly match those used in the database, no further customization would be required. As you can see in the figure and code, I've also changed the displayed value to capitalize the type, but that only impacts the display.

Another good example would be to use a drop down list to filter pages by those that are live or not. The grid may display the words “Live” and “Draft” for those values, but the drop down list should use the corresponding database values:

```
# protected/views/page/admin.php
<?php $this->widget('zii.widgets.grid.CGridView', array(
'id'=>'page-grid',
'dataProvider'=>$model->search(),
'filter'=>$model,
'columns'=>array(
    'id',
    'pageUser.username',
    array (
        'header' => 'Live?',
        'value'=>($data->live == 1) ? "Live" : "Draft",
        'filter' => CHtml::dropDownList('Page[live]',
            $model->live, array('1' => 'Live',
            '0' => 'Draft'), array('empty' => '(select)'))
```

```
        ),
        'title',
        'date_published',
        array(
            'class'=>'CButtonColumn',
        ),
),
)); ?>
```

You can see the results in **Figure 12.14**.

The screenshot shows a Yii2 application's admin interface. A grid view displays three columns: 'Username', 'Status', and 'Phase'. The 'Status' column contains three items: 'Live', 'Draft', and 'Pending'. A dropdown menu is open over the 'Draft' item, showing options '(select)', 'Live', and 'Draft'. The 'Draft' option is highlighted with a blue background and a checked checkbox icon. The grid data is as follows:

Username	Status	Phase
Moe	Draft	Phase
Some Author	Draft	I never
Another Author	Draft	Barnak

**Figure 12.14:** The drop down filter for the page's status.

When you're working with a single model, you can customize the filtering pretty easily. When you have related models, it becomes a bit trickier. Take, for example, a grid for pages that shows the username of each page author (see the above code and Figure 12.14). That value comes from the `user` table. As it stands, the above code will display the username, but won't do proper filtering by username as the underlying `Page` attribute is `user_id`. As you can see in Figure 12.14, no filter box is provided anyway. But even if a text input were present, the person using the grid could enter "test", but that will never match a `user_id` value.

Two steps are required to solve this riddle. First, a text input must be displayed for the column. The filters are based upon the model, and as the model has no `pageUser.username` attribute, no text input shows. The fix for that is to name the column `user_id` for the filters, but still use `pageUser.username` as the value of the column:

```
# protected/view/page/admin.php
<?php $this->widget('zii.widgets.grid.CGridView', array(
    'id'=>'page-grid',
    'dataProvider'=>$model->search(),
    'filter'=>$model,
```

```
'columns'=>array(
    'id',
    array (
        'header' => 'Author',
        'name' => 'user_id',
        'value' => '$data->pageUser->username'
    ),
// And so on.
```

Now the grid shows an input for the column (test it for yourself to see).

Next, the `search()` method must be changed so that the query uses the supplied `user_id` value to compare against the `username` column in the `user` table.

The default `Page::search()` method looks like this:

```
# protected/models/Page.php
public function search() {
    $criteria=new CDbCriteria;
    $criteria->compare('id',$this->id,true);
    $criteria->compare('user_id',$this->user_id,true);
    // Other comparisons
    return new CActiveDataProvider($this, array(
        'criteria'=>$criteria,
    ));
}
```

First, you'll want to add a `with` clause to change from lazy loading of the user records to eager loading. Then, change the comparison to use the `pageUser.username` field instead of `user_id`:

```
# protected/models/Page.php
public function search() {
    $criteria=new CDbCriteria;
    $criteria->with = 'pageUser';
    $criteria->compare('id',$this->id);
    $criteria->compare('pageUser.username',
        $this->user_id,true);
    // Other comparisons
    return new CActiveDataProvider($this, array(
        'criteria'=>$criteria,
    ));
}
```

And now the grid of pages can be filtered by author name (**Figure 12.15**).

Author	Live?	
Moe	(select)	
Moe	Draft	Phasellus dapibus.
Moe	Live	Test Title
Moe	Live	There's a voice in my head. me.

**Figure 12.15:** Only pages by “Moe” are now shown.

Working with dates brings its own problems, as you may know from your experiences interacting with a database. How you address this issue depends greatly upon how the dates are stored in the database and how you would imagine a user would filter by dates. For an example to work with, let’s imagine you want to be able to filter users by the date they registered. This value is stored in the `user.date_entered` column, which is a timestamp.

A timestamp is a perfect way of marking when records are created, but it’s generally impractical to filter by the date *and* time (down to the seconds) unless you’re looking at ranges. But a reasonable alternative for filtering the grid would be to expect a date to be provided and then to find all records created that day.

In that situation, you would probably want to first only show the registration dates as YYYY-MM-DD, which also provides a sense of how filtering would be accomplished. I would change the `search()` method to pull out dates in that format:

```
# protected/models/User.php::search()
$criteria=new CDbCriteria;
// Select the date in a formatted way:
$criteria->select = array('*' , new
    CDbExpression('DATE_FORMAT(t.date_entered,
        "%Y-%m-%d") AS date_entered')
);
$criteria->compare('id',$this->id,true);
// And so on.
return new CActiveDataProvider($this, array(
    'criteria'=>$criteria,
));
```

That code change uses a `CDbExpression` to format the selected data. Thanks to an alias, the formatted date is still returned as `date_entered`. The end result is that the formatted date is used for `date_entered` in the model and in the grid.

Next, you need to change the `search()` method to perform a different comparison of the `date_entered` column value against the provided `date_entered` value. I would do that using a *condition*:

```
# protected/models/User.php::search()
$criteria=new CDbCriteria;
// Select the date in a formatted way:
$criteria->select = array('*' , new
    CDbExpression('DATE_FORMAT(t.date_entered,
        "%Y-%m-%d") AS date_entered')
);
// Check for a date:
if (isset($this->date_entered) &&
    preg_match('/^([0-9]{4})-([0-9]{2})-([0-9]{2})$/' ,
        $this->date_entered)) {
    $criteria->condition = 'DATE_FORMAT(date_entered,
        "%Y-%m-%d") = :de';
    $criteria->params = array(':de' => $this->date_entered);
}
$criteria->compare('id',$this->id,true);
// And so on.
return new CActiveDataProvider($this, array(
    'criteria'=>$criteria,
));
```

That code confirms that `$this->date_entered` has a value and that the value matches the pattern `####-##-##`. If so, then a condition is added to the criteria, checking for an equality match between that provide value and the formatted date.

Note that the `condition` property must be set prior to any `compare()` calls or else the query's logic will get messed up. Also be certain to remove the `compare()` use of `date_entered` that the `search()` method also has.

This is a simple and effective way to filter the records by the date (**Figure 12.16**), but you should make it clear to the user in what format the date criteria must be provided.

## The jQuery UI Widgets

One of the features of Yii that I always appreciated is that it has support for jQuery built-in (the Zend Framework, by comparison, took years to add a jQuery component). Chapter 14 goes into JavaScript and jQuery in Yii in more detail, but while I'm talking about widgets, I'll go ahead and mention the [jQuery User Interface](#) (jQuery UI) widgets now.

Displaying 1-3 of 3 results.		
	Date Entered	
(Select)	2013-02-17	
Admin	2013-02-17	
Public	2013-02-17	
Author	2013-02-17	

**Figure 12.16:** The grid can now be filtered by date.

The jQuery User Interface is a package of useful components built on top of jQuery. jQuery UI includes:

- Functionality such as dragging, dropping, sorting, and resizing
- Widgets
- Effects such as hiding, showing, color animation, and so on
- A couple of utilities

The jQuery UI widgets include much of the functionality common in today's Web sites:

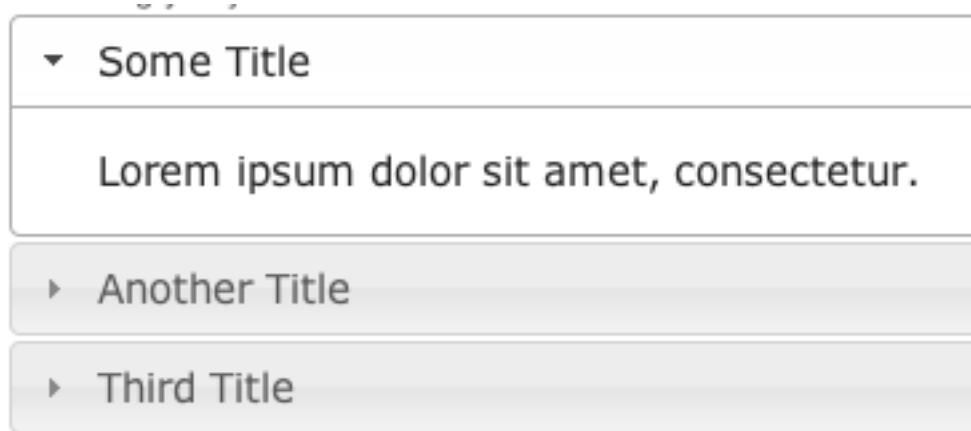
- Accordion
- Autocomplete
- Datepicker
- Dialog
- Menu
- Slider
- Spinner
- Tabs
- Tooltips

As jQuery is built-into Yii, it was only natural to have parts of jQuery UI ported into Yii as well. About a dozen jQuery UI components have been recreated in Yii as widgets, found within the **zii.widgets.jui** package. Most of these are pretty easy to use just by looking up the corresponding Yii class documentation.

In this chapter, I'll demonstrate a couple that are the easiest to use, most necessary, and do not require additional JavaScript (such as an Ajax component). Chapter 14 will discuss a couple of others.

## Accordions

As a first example, the `CJuiAccordion` creates an accordion display of content (**Figure 12.17**).



**Figure 12.17:** A *jQuery UI accordion*.

That output is created by this Yii code:

```
<?php
$this->widget('zii.widgets.jui.CJuiAccordion',array(
'panels'=>array(
    'Some Title'=>'Lorem ipsum dolor sit amet.',
    'Another Title'=>'Mauris pharetra viverra lacinia.',
    'Third Title'=>'Morbi iaculis fermentum lorem eu.',
),
));
?>
```

And that will do it! Of course, it's not truly reasonable to hardcode the different content into the widget. Normally the content will be dynamically generated. When that's the case, there are a couple of ways you can approach the issue. As an example of this, let's say the controller is passing the accordion view page an array of information:

```
<?php
# protected/controllers/SomeController.php::someAction()
$data = array(
    array('title' => 'Some Title',
          'content' =>'Lorem ipsum dolor sit amet.'
    ),
    array('title' => 'Another Title',
          'content' =>'Mauris pharetra viverra lacinia.'
    )
);
```

```
        'content' =>'Mauris pharetra viverra lacinia.'
    ),
    array('title' => 'Third Title',
        'content' =>'Morbi iaculis fermentum lorem eu.' )
);
$this->render('accordionView', array('data' => $data));
```

Then, in the **accordionView.php** page, you would use the received data:

```
<?php
$this->widget('zii.widgets.jui.CJuiAccordion',array(
    'panels'=>array(
        $data[0] ['title'] => $data[0] ['content'],
        $data[1] ['title'] => $data[1] ['content'],
        $data[2] ['title'] => $data[2] ['content']
    ),
));
?>
```

{NOTE} There are other, more automated ways to create the data to be used for the accordion, but I'm trying to keep things more simple.

If the content was much more complex or dynamically generated, you could use `renderPartial()` to assign another view file as the content. You'd just need to set the method's third argument to true to have the rendered content returned:

```
<?php
$this->widget('zii.widgets.jui.CJuiAccordion',array(
    'panels'=>array(
        'Actual Title' => $this->renderPartial('_accordionItem',
            array(/* pass data */), true);
    // Etc.
));
```

## Tabs

The **CJuiTabs** widget works exactly the same way as **CJuiAccordion**, but uses the `tabs` property and lays out the content in tabs:

```
<?php
$this->widget('zii.widgets.jui.CJuiTabs',array(
    'tabs'=>array(
        $data[0] ['title'] => $data[0] ['content'],
        $data[1] ['title'] => $data[1] ['content'],
    ));
```

```
$data[2]['title'] => $data[2]['content']
),
));
?>
```

Again, you could render partial views for the content when needed.

## Datepicker

Another great and useful widget is the Datepicker (**Figure 12.18**).



**Figure 12.18:** A jQuery UI datepicker.

That's created by this code:

```
<?php
$this->widget('zii.widgets.jui.CJuiDatePicker',array(
    'attribute'=>'date_published',
    'model' => $model
));
?>
```

Because this is a form element, you can associate it with a model, as the code shows.

Although the Datepicker is easy to use, there is a catch when it's associated with a model and an underlying database that expects dates to be in a particular format. I'll explain that in the next section.

## Customizing jQuery UI Widgets

As with any widget, the available properties of the associated class can be used to customize its behavior. All jQuery UI widgets support `theme` and `themeUrl` properties, for example, if you're using a jQuery UI theme.

The most important property for the jQuery UI widgets is `options`. Through the `options` property you can configure the widget's behavior. For the `option` indexes and values, turn to the corresponding [jQuery UI documentation](#). For example, the jQuery UI accordion has the “animate”, “collapsible”, “heightSize” and other properties. To set the accordion as collapsible, which means that every section can be closed at the same time, set that property to true:

```
<?php
$this->widget('zii.widgets.jui.CJuiAccordion',array(
    'panels'=>array(/* values */),
    'options'=>array(
        'collapsible'=>true
    )
));
?>
```

The tabs widget has properties for dictating how tabs are hidden and shown (among others);

```
<?php
$this->widget('zii.widgets.jui.CJuiTabs',array(
    'tabs'=>array(/* values */),
    'options'=>array(
        'hide'=>'fade',
        'show'=>'highlight'
    )
));
?>
```

The Datepicker has a [ton of options](#). The “dateFormat” is used to set the format for the selected and displayed dates. You can set the latest date that can be selected via “maxDate” and the earliest via “minDate”.

```
<?php
$this->widget('zii.widgets.jui.CJuiDatePicker',array(
    'attribute'=>'starting_date',
    'model' => $model,
    'options'=>array(
        'dateFormat'=>'yy-mm-dd',
        'maxDate'=>'+1m', // One month ahead
        'minDate'=>'new Date()', // Today
    )
));
```

To find what options are available, and what an appropriate value would be, check the jQuery UI documentation.

As for the trick mentioned earlier that may be required when using Datepickers with models, if the model attribute correlates to a database column, updates and inserts will only work properly if the submitted date is in a format that MySQL accepts. The default format for the date picker is mm/dd/yy, which will fail when used to update or insert a record in the database. One solution is to customize the widget to use a better format, as in the code above.

## Chapter 13

# USING EXTENSIONS

When I first started using Yii, I thought of extensions as being similar to third-party libraries. Basically, I imagined them as entirely separate components that were used to add large amounts of functionality, like widgets on steroids. Over time, I realized that extensions are quite literally that: extensions of the core Yii framework. Some extensions *are* separate components, while others add very specific functionality, and yet others actually define widgets.

This chapter starts by explaining the concept of extensions in Yii. Next, you'll learn some basic recommendations as to how to select an extension for your needs.

The bulk of the chapter highlights a few notable extensions, from the hundreds that are available. To generate this list, I looked at the extensions that:

- I've personally used
- Are the most downloaded
- Are the highest rated
- Are newer (at the time of this writing)
- Are the easiest to get started with

Obviously, this chapter cannot be exhaustive in terms of the extensions covered, or the coverage of individual extensions. But by the end of the chapter, you should better understand the range of what extensions have to offer, and how you go about using them.

{NOTE} In Chapter 19, “Extending Yii,” you’ll learn how to write your own extensions of the framework.

### The Basics of Extensions

The first fact to know about extensions, if you don’t already, is the one that took me a while to learn: extensions can serve many different roles. They can act as

application components, add new behaviors, be used as widgets, create filters and validators, be used as stand-alone modules, and more.

In fact, you may be surprised to find out that you've probably already used two extensions. First, there's the Gii extension, which is an application component. Second, there's Zii, which is an extension that defines several widgets, among other things. Both of these extensions are unlike many of those you'll deal with in Yii in that they are automatically installed as part of the framework itself.

For all the other available extensions, head to the [Yii framework extensions](#) page. At the time of this writing, there are over 1,100 extensions available, sorted into 15 categories. Once there, you'll need to choose what extensions you'll want to use for your project. Obviously the primary criteria will be your application's needs, such as:

- A WYSIWYG editor
- Easy and powerful authorization management
- Excellent debugging tools
- Cache management
- The ability to send HTML email

Whatever the need, there's a good chance there's already an extension that will do the job.

Once you've identified your criteria, and have used the extensions page to find options that *may* do the job, I would make a specific decision based upon (in this order):

- **What versions of Yii it requires**

This could be even more of an issue when Yii 2 comes out.

- **What version the extension is currently in**

You probably don't want to use a beta version of an extension on a production site. On the other hand, if the extension looks to be perfect for your needs, using the beta version while you develop the site, and helping the developer find and fix bugs, could be a symbiotic relationship. Moreover, some extension developers mark their releases as betas just to cover themselves.

- **How well maintained it is**

To me, one of the most important criteria is how well maintained an extension (or any software/code you use) is. It's best not to rely upon an extension that will become too outdated to be useful. Look at the extension's version number to tell how well maintained an extension is. Also note how recently updated the extension is. And check out how active the developer is in replying to comments.

- **How well documented it is**

There's no point in attempting to use an extension that you won't be able to figure out how to use. And, as a writer, I particularly value good documentation.

- **How popular it is (in terms of both downloads and rating)**

Popularity isn't always a good thing, and it's certainly not the most important criteria, but can be useful when making a final decision.

{TIP} You can also learn a lot about an extension—how useful it is, how well maintained it is, etc.—by searching for the extension in the Yii forums.

Once you've identified the extension to use, the installation process goes like so:

1. Download the code (from its extension page).
2. Expand the downloaded code (from a **.zip** or other file type to a folder).
3. Move or copy the resulting folder to the **protected/extensions** directory.

These are generic instructions. Some extensions will require that you rename the resulting folder (from, say, “yii-bootstrap-2.0.3.r329” to just “bootstrap”). Other extensions might expect you to move a subdirectory from the resulting folder to your **extensions** directory. Just read and follow the installation instructions that the extension provides. (If it doesn't have installation instructions, then you don't want that extension.)

How you import, configure, and use the extension will also differ from one extension type to the next. Some are configured as application components, others just need to be referenced where you're using it (e.g., a widget).

{TIP} The path alias **ext.name** refers to the extension's base directory, where “name” is the name of the folder in which the extension resides.

Before moving on, I have two specific recommendations. First, if your permissions are not properly set (if the Web server cannot read everything within the **extensions** directory), you'll get errors and unusual results when you go to use any extension. I found (when using Mac OS X), that any time I had unusual results when first using an extension, I would have to fix the permissions on the applicable directories to get the problem sorted.

Second, trying to use any extension for the first time can be quite frustrating. In writing this chapter, I ran into many hurdles with extensions that I would have liked to cover (not insurmountable hurdles, necessarily, but too many hurdles to reasonably still use the extension in a book). As this will likely be the case for you as well, I would recommend first installing and testing new extensions on a practice project. By using a demo site, you won't run the risk of cluttering up, or worse yet, breaking, your actual project.

With that general introduction in place, let's look at some specific extensions.

## The bootstrap Extension

[Twitter Bootstrap](#) is a framework of HTML, CSS, and JavaScript, designed to make front-end Web development quick, reliable, and painless. A highlight of Bootstrap's features include:

*{NOTE}* At the time of this writing Twitter Bootstrap 3 is in beta, with some differences from version 2.

- A grid system for easy layout
- Responsive design
- Basic typography
- Table, form, and image styling
- CSS buttons
- Common components, such as navigation menus, drop down menus, alerts, etc.
- JavaScript-dependent components like modal windows, tabs, tooltips, carousels, and so forth

*{NOTE}* Twitter Bootstrap does require that you use HTML5, which you ought be to using anyway.

To use Twitter Bootstrap in a Yii-based site, you have a couple of options. The first and most obvious would be to download the Bootstrap framework, install it on your site, and then have Yii create the necessary HTML and JavaScript to create the various elements. That approach would get tedious fast.

An alternative, then, is to use one of the available Twitter Bootstrap extensions for Yii. The two likely candidates are:

- [Yii-Bootstrap](#)
- [YiiBooster](#)

Both are great, with YiiBooster being an extension of Yii-Bootstrap. I'll quickly walk through its installation and usage.

To install YiiBooster, download it from <http://yii-booster.clevertch.biz/index.html>. Then expand the downloaded file to create a folder called something like *clevertch-YiiBooster-bf8ace0*. Rename this folder as *bootstrap* and place it in your extensions directory.

To enable this extension, add it to the “components” section of the configuration file:

```
# protected/config/main.php
// Other stuff.
'components'=>array(
    'bootstrap' => array(
        'class' => 'ext.bootstrap.components.Bootstrap',
        'responsiveCss'=>true,
    ),
// More other stuff.
```

Then, you also need to tell Yii to preload this extension:

```
# protected/config/main.php
// Other stuff.
'preload'=>array('log', 'bootstrap'),
// More other stuff.
```

Once enabled, you'll immediately see some aesthetic changes, even to the default Yii-generated site. And the layout will be somewhat responsive if you've set “responsiveCss” to true (as in the above).

*{TIP}* There's also a series of Gii templates you can enable when using YiiBooster.

To create a site layout that uses Twitter Bootstrap, I'd begin with one of the [examples](#) that Twitter Bootstrap provides, such as the starter template. Copy that example page's HTML to a new layout file (perhaps called “bootstrap.php”).

*{NOTE}* Also see Chapter 6, “[Working with Views](#),” for instructions on creating and switching layouts in general.

Next, you'll want to change the TITLE tag to have it be populated by Yii:

```
<title><?php echo CHtml::encode($this->pageTitle); ?>
</title>
```

And don't forget the most important step: having Yii insert the page-specific content in the right place:

```
<?php echo $content; ?>
```

To convert the default Yii main menu to a Twitter Bootstrap menu, just add the "nav" class as an HTML option:

```
<div class="nav-collapse collapse">
<?php $this->widget('zii.widgets.CMenu', array(
    'items'=>array(
        array('label'=>'Home', 'url'=>array('/site/index')),
        array('label'=>'About',
            'url'=>array('/site/page', 'view'=>'about')),
        array('label'=>'Contact',
            'url'=>array('/site/contact')),
        array('label'=>'Login',
            'url'=>array('/site/login'),
            'visible'=>Yii::app()->user->isGuest),
        array('label'=>'Logout ('.Yii::app()->user->name.')',
            'url'=>array('/site/logout'),
            'visible'=>!Yii::app()->user->isGuest),
    ),
    'htmlOptions' => array('class'=>'nav')
)); ?>
</div><!-- .nav-collapse -->
```

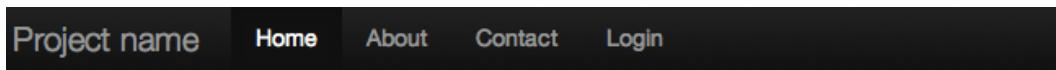
And now you have a basic Twitter Bootstrap implemented in Yii (**Figure 13.1**).

Note that you don't have to install Twitter Bootstrap yourself. Nor do you have to add references to the Twitter Bootstrap CSS and other files to your layout. The extension's assets manager will take care of all that for you.

Using YiiBooster, you can also make use of any of the formatting options that Twitter Bootstrap provides. For example, Twitter Bootstrap has labels for making text prominent (**Figure 13.2**):

```
<span class="label label-important">WARNING!</span>
```

With YiiBooster, you can create the HTML as in the above, or dynamically create it using a widget:



# Welcome to My Web Application

Congratulations! You have successfully created your Yii application.

You may change the content of this page by modifying the following two files:

- View file: `/Users/larryullman/Sites/htdocs/protected/views/site/index.php`
- Layout file: `/Users/larryullman/Sites/htdocs/protected/views/layouts/main.php`

For more details on how to further develop this application, please read the [documentation](#). Feel free to ask questions on the [forum](#).

**Figure 13.1:** The Bootstrap version of the basic site design.

**WARNING!** Do not do whatever it is you are about to do!

**Figure 13.2:** A Bootstrap label.

```
<?php
$this->widget('bootstrap.widgets.TbLabel', array(
    'type'=>'important',
    'label'=>'WARNING!',
));
?>
```

There are plenty more widgets defined in YiiBooster. For example, here's how you would implement the breadcrumbs widget (**Figure 13.3**):

```
# protected/views/layouts/main.php
<?php if(isset($this->breadcrumbs)) :?>
    <?php
        $this->widget('bootstrap.widgets.TbBreadcrumbs', array(
            'links'=>$this->breadcrumbs
        ));
    ?>
<?php endif?>
```

Home / Pages / Aliquam malesuada, ligula sit amet.

**Figure 13.3:** The Bootstrap version of the breadcrumbs.

Chapter 12, “[Working with Widgets](#),” spends a decent amount of time discussing the CGridView widget. YiiBooster extends this widget in the TbExtendedGridView class ([Figure 13.4](#)):

```
# protected/views/page/admin.php
$this->widget('bootstrap.widgets.TbExtendedGridView', array(
    'dataProvider' => $model->search(),
    'filter' => $model,
    'type' => 'striped bordered',
    'columns' => array(
        'id',
        array (
            'header' => 'Author',
            'name' => 'user_id',
            'value' => '$data->pageUser->username'
        ),
        array (
            'header' => 'Live?',
            'value'=>'($data->live == 1) ?
                "Live" : "Draft"',
            'filter' => CHtml::dropDownList('Page[live]',
                $model->live, array(
                    '1' => 'Live',
                    '0' => 'Draft'),
                array('empty' => '(select)'))
        ),
        'title',
        'date_published',
        array(
            'header' => Yii::t('ses', 'Edit'),
            'class' => 'bootstrap.widgets.TbButtonColumn',
            'template' => '{view} {delete}',
        ),
    ),
));
```

You can configure the YiiBooster grid view to add many great features:

- A fixed table header (that stays visible as you scroll down the table)
- Inline editing
- Bulk edits
- Table summaries
- Graphs
- Groupings

ID	Author	Live?	Title	Date Published	Edit
		(select) ▾			
1	Some Author	Live	Aliquam malesuada, ligula sit amet.	2013-02-17	🕒 🗑
2	Another Author	Live	Ten years ago a...	2013-02-22	🕒 🗑
3	Moe	Draft	Phasellus dapibus dolor et mauris.		🕒 🗑
4	Another Author	Live	Thunder, thunder, thundercats, Ho!	2013-02-01	🕒 🗑
5	Another Author	Live	Michael Knight, a young loner	2013-03-15	🕒 🗑
6	Moe	Live	Test Title	2013-01-03	🕒 🗑

Figure 13.4: The Bootstrap version of the grid view.

- Advanced filtering

Just check out the YiiBooster documentation for how to implement these features, and to see what else is possible with the extension.

{TIP} The creators of YiiBooster also created [YiiBoilerplate](#), an advanced template for an entire Yii project.

## The giix Extension

Another extension I'd like to highlight is [giix](#), short for *gii Extended*. giix is a version of Gii with extra features:

- Better handling of complex relations amongst models
- Support for internationalization (i18n) out-of-the-box
- Generation of form elements more specific to model attributes
- Generation of forms that reflect model relations
- Creation of base model classes that are extended
- Creation of new model methods

To enable giix, first download and expand the extension. Then move or copy its **giix-components** and **giix-core** folders to your **extensions** directory. Next, enable the extension in your configuration file:

```
# protected/config/main.php
// Other stuff.
'modules'=>array(
    'gii' => array(
        'class' => 'system.gii.GiiModule',
        'password'=>'1234',
```

```
'generatorPaths' => array(
    'ext.giix-core',
),
),
// Lots more other stuff.
```

As you can see in that code, you're just telling Gii to use the giix-core files for the code generators.

You should also import the giix components:

```
# protected/config/main.php
// Other stuff.
'import'=>array(
    'application.models.*',
    'application.components.*',
    'ext.giix-components.*',
),
// Lots more other stuff.
```

With that configured, you can head to Gii as you usually would.

giix adds two new options to Gii:

- GiixCrud Generator
- GiixModel Generator

Both of these are used just like the standard Gii tools, with the difference being in the code they generate. For example, when using giix to model and CRUD the `page` table in the CMS example, the model generator creates a `BasePage` class which is then extended by `Page`. This allows you to make edits to `Page` that won't be overwritten, even if you later need to re-generate `BasePage` (e.g., after changing the database table).

Where giix really shines is in the view files, however. Using the CMS page example, giix will do a few nice things for you, such as create a drop down list of authors for searching, filtering, adding, and updating page records (**Figure 13.5**).

Similarly, the author's name is also automatically displayed in the list and detail views. Further, on the view page, the author's name is automatically linked to the author's view page (**Figure 13.6**).

The strong suit of giix is really all the ways that related models are automatically used. As another example, the individual view page also immediately lists all the comments and files associated with that page, which is something you'd likely want to implement anyway (**Figure 13.7**).

ID	User	Live	Title	Content
	✓ Administrator			
	Another Admin			
	Another Author			
	Curly			<p>Lorem ipsum dolor sit amet, consectetur porta lectus sed ipsum tincidunt lacinia. Inte varius. Class aptent taciti sociosqu ad litora nostra, per inceptos himenaeos. Nulla lobor

Figure 13.5: The author username drop down is automatically added to the grid view filter.

## View Page Aliquam malesuada, ligula

ID	1
User	<a href="#">Some Author</a>
Live	1
Title	Aliquam malesuada, ligula sit amet.

Figure 13.6: The default view for an individual page.

Date Updated	2013-02-17 16:10:18
Date Published	2013-02-17

### Comments

- [Sed bibendum ligula ac urna egestas euismod. Nunc eu molestie purus.](#)
- [Sed bibendum ligula ac urna egestas euismod. Nunc eu molestie purus.](#)
- [Sed bibendum ligula ac urna egestas euismod. Nunc eu molestie purus.](#)

### Files

Figure 13.7: The comments and files associated with the page.

Similarly, giix implements “save related” functionality into the controllers and models. With the CMS example, if you were to select files associated with a page (all the files are automatically listed as checkboxes on the page form), the giix code will save the file relations, too.

Although giix does add some great functionality, there’s another reason why I’m highlighting it in this chapter. If you use Yii a lot, and have Gii generate a lot of code for you, there’s a strong argument to be made for customizing the generated code so that it closely matches what you want and prefer. That’s exactly what giix does.

## Validator Extensions

Taking a look at another type of extension, there are many Yii extensions written expressly for acting as validators. As explained in Chapter 5, “[Working with Models](#),” Yii has a slew of built-in validators for use in your models. But there are ways you’ll need to validate model attributes that aren’t sufficiently covered by the built-in options.

One way of creating additional validators, as also demonstrated in Chapter 5, is to define a validator as a method in the model. The downside to this approach is the code ends up being less reusable. A more flexible solution would be to write the validator as its own class. When you’ve done that, you’ve created an extension.

On Yii’s extensions page, there are oodles of validator extensions available. For this example, I’ll demonstrate [eccvalidator](#), which confirms that a value is a syntactically valid credit card number. To start, download the extension.

Next, I would create a **validators** folder within **extensions**, into which you’ll put any validator class you use. Copy the downloaded code—**ECCValidator.php**—to this directory.

To use the validator, you’d reference it in the **rules()** method of a model. Let’s say there’s a **PaymentForm** model, that has a **ccNum** attribute:

```
# protected/models/PaymentForm.php
class PaymentForm extends CFormModel {
    public $ccNum;
```

Here’s how the **rules()** method would partially look:

```
Yii::import('ext.validators.ECCValidator');
return array(
    // ccNum is required:
    array('ccNum', 'required'),
    // ccNum needs to be a valid number:
```

```
    array('ccNum', 'ext.validators.ECCValidator'),  
);
```

First the validator class file is imported so that it may be used. Then the credit card number is marked as required, using the standard Yii validators. Finally, the credit card number is validated using the `ECCValidator` class.

And that is all you need to do! Now the credit card number in the form will be tested, with errors reported (**Figure 13.8**).

The screenshot shows a web form titled "Payment". A field labeled "Credit Card Number \*" contains the value "4485587139555400". Below the field, a red error message reads "Credit Card Number is not a valid Credit Card number." The entire input field and its error message are highlighted with a red border.

**Figure 13.8:** A syntactically invalid credit card number is reported.

The `eccvalidator` supports over a dozen credit card types. To limit the allowed types, use the “format” index and the constants defined in the class:

```
array('ccNum', 'ext.validators.ECCValidator',  
    'format' => array(  
        ECCValidator::MASTERCARD,  
        ECCValidator::VISA,  
        ECCValidator::AMERICAN_EXPRESS,  
        ECCValidator::DISCOVER  
    ),  
,
```

If you look at the `ECCValidator` class code, you’ll find all the available public properties that you can configure. This list includes “allowEmpty”, meaning you can omit the “required” rule and declare the requirement as part of the `ECCValidator` rule:

```
array('ccNum', 'ext.validators.ECCValidator',  
    'allowEmpty' => false,  
    'format' => array(  
        ECCValidator::MASTERCARD,  
        ECCValidator::VISA,  
        ECCValidator::AMERICAN_EXPRESS,  
        ECCValidator::DISCOVER  
    ),  
,
```

And that's an example of how you use extensions as validators. Just search or browse through the Yii extensions pages to find others that you might need.

## Auto-Setting Timestamps

Also in Chapter 5, I explained two different ways of setting a model's attributes to the current timestamp:

- Using a default value rule
- Using a `beforeSave()` event handler

A third option is to use a *behavior* added to the framework via an extension. Behaviors, in general, allow you to extend a model by adding additional members at runtime. In OOP terms, behaviors are like *mixins*, and atone for some of the limitations of inheritance in PHP.

Behaviors are most useful when:

- The same functionality might be needed by multiple classes (and, therefore, defining the functionality in a separate class makes it more portable)
- Functionality may only be needed in certain circumstances

There are extensions you can download to add behaviors to a site, but one is already available to you via the Zii extension that comes with the framework. The `CTimestampBehavior` class can be used to automatically populate date and time attributes. To use it, create a `behaviors()` method in your model. Within the method, return an array that specifies `CTimestampBehavior` and the attributes to be populated upon creating and updating model instances:

```
# protected/models/SomeModel.php
public function behaviors() {
    return array (
        'CTimestampBehavior' => array(
            'class' => 'zii.behaviors.CTimestampBehavior',
            'createAttribute' => 'date_entered',
            'updateAttribute' => 'date_updated'
        )
    );
}
```

The behavior will automatically determine how best to set the timestamp, but you can specify where the timestamp value should come from if you want. See the [CTimestampBehavior class docs](#) for details.

Because the `CTimestampBehavior` is added to the `behaviors()` method, it's always attached to the model. You can attach a behavior only under certain circumstances using the `attachBehavior()` method. I may discuss this more later in the book.

## Using a WYSIWYG Editor

Often, a site like the CMS example will require an administrative area to dynamically manage the site's content. Much of the content can contain some HTML, including media (images, videos, etc.), typography, lists, and so forth. So that non-technical people can create nice-looking HTML, I normally turn to a Web-based WYSIWYG editor like [CKEditor](#) or [TinyMCE](#). Getting either to work within the Yii environment isn't too hard, once you know what to do. However, the process can be greatly simplified thanks to the right extension.

If you want to use CKEditor, the `editMe` extension is brilliantly easy to use. It creates a widget that you can drop into your view files wherever you need an instance of the CKEditor.

Start by downloading the `editMe` extension, and expanding the downloaded file. Copy the resulting folder to the `protected/extensions` directory. There's nothing you need to do to enable this extension.

To use the CKEditor in a form, you'll need to edit the `_form.php` file for the particular view, such as for "page" in the CMS example. By default the form will have a standard textarea for every TEXT type:

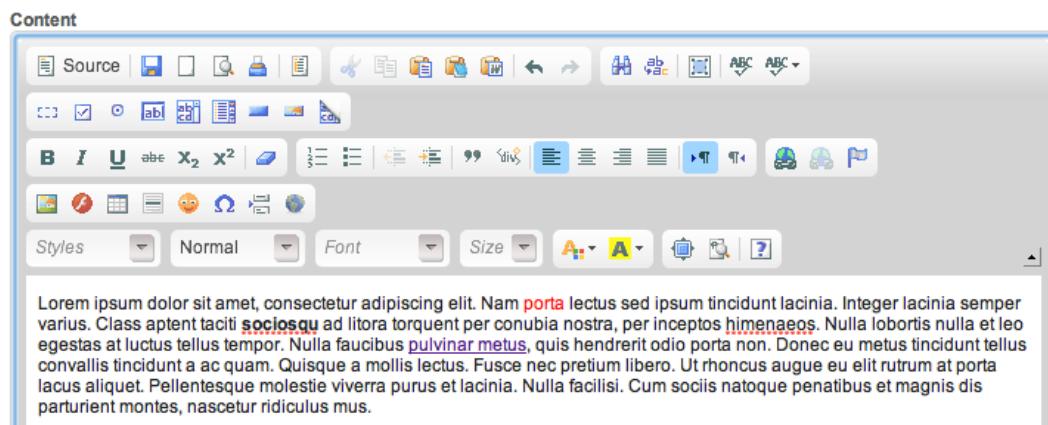
```
# protected/views/page/_form.php
<div class="row">
<?php echo $form->labelEx($model, 'content'); ?>
<?php echo $form->textArea($model, 'content'); ?>
<?php echo $form->error($model, 'content'); ?>
</div><!-- row -->
```

To use CKEditor instead of the textarea, invoke the `editMe` widget by replacing that code with this:

```
# protected/views/page/_form.php
<div class="row">
<?php echo $form->labelEx($model, 'content'); ?>
<?php $this->widget('ext.editMe.widgets.ExtEditMe', array(
    'model'=>$model,
    'attribute'=>'content'
)); ?>
<?php echo $form->error($model, 'content'); ?>
</div><!-- row -->
```

That code starts by saying a widget should be rendered in this place, specifically the editMe widget. That widget gets passed an array of values to configure it. To start, pass the model instance, which Yii stores in the \$model variable by default. The attribute element is the name of the associated model attribute.

If you load page/create in your browser, you should now see a lovely WYSIWYG editor in lieu of the text area (**Figure 13.9**).



**Figure 13.9:** The CKEditor instance used to create and edit a Page record.

There are further configurations explained in the [extension's wiki](#). For example, “height” and “width” change the size of the text area:

```
# protected/views/page/_form.php
<div class="row">
<?php echo $form->labelEx($model, 'content'); ?>
<?php $this->widget('ext.editMe.widgets.ExtEditMe', array(
    'model'=>$model,
    'attribute'=>'content',
    'height'=>'400'
)); ?>
<?php echo $form->error($model, 'content'); ?>
</div><!-- row -->
```

Those steps are pretty easy to understand and will get you a working WYSIWYG editor in no time. But you’ll likely need to tweak how the CKEditor behaves, too, which is much more complicated.

For starters, if you want to allow the admin to upload files to the server, like images or videos, you’ll need to enable the file manager. This gets complicated, as CKEditor does not come with this functionality built-in. You’ll need to either buy the commercial CKFinder, or find a third-party alternative. Once you’ve done that, you’d set the various editMe properties that begin with “filebrowser”. (See the editMe docs for details.)

Next, you'll likely want to configure the CKEditor as it'll behave in the Web browser. To do so, you can first set the "toolbar" index. This is an array of options that should appear in the toolbar. The values themselves come from the CKEditor documentation:

```
# protected/views/page/_form.php
<?php $this->widget('ext.editMe.widgets.ExtEditMe', array(
    'model'=>$model,
    'attribute'=>'content',
    'height'=>'400'
    'toolbar'=>array(
        array(
            'Bold', 'Italic', 'Underline', 'Strike',
            'Subscript', 'Superscript', 'RemoveFormat'
        ),
        '-',
        array('Link', 'Unlink', 'Anchor'),
        '/',
        array('NumberedList', 'BulletedList', '-',
            'Outdent', 'Indent', '-',
            'Blockquote', 'CreateDiv', '-',
            'JustifyLeft', 'JustifyCenter', 'JustifyRight',
        )
    )
)); ?>
```

Within the array, each subarray is a grouping (**Figure 13.10**). Visual separators are created by a hyphen, and a slash creates a line break (i.e., to start the next sequence of toolbar options on the next row). If you attempt this configuration and the result is a blank toolbar, or no WYSIWYG editor at all, then you've probably created a syntax error of some type.

Finally, as a reminder, consider that the views, by default, echo out model data using the `CHtml::encode()` method. This is a logical and necessary security feature, as it prevents cross-site scripting attacks (XSS). However, it also makes the CKEditor-generated HTML completely useless. Therefore, whenever you use a WYSIWYG editor for site content, be certain to change the corresponding view file (that displays the content) so that it does not use `CHtml::encode()` for the HTML content.

To still have some security measures in place, and to make sure that the administrator does not do anything to mess up the site's presentation, you can apply PHP's `strip_tags()` function to submitted content. This function has an optional second argument wherein you can specify the *allowed* tags:

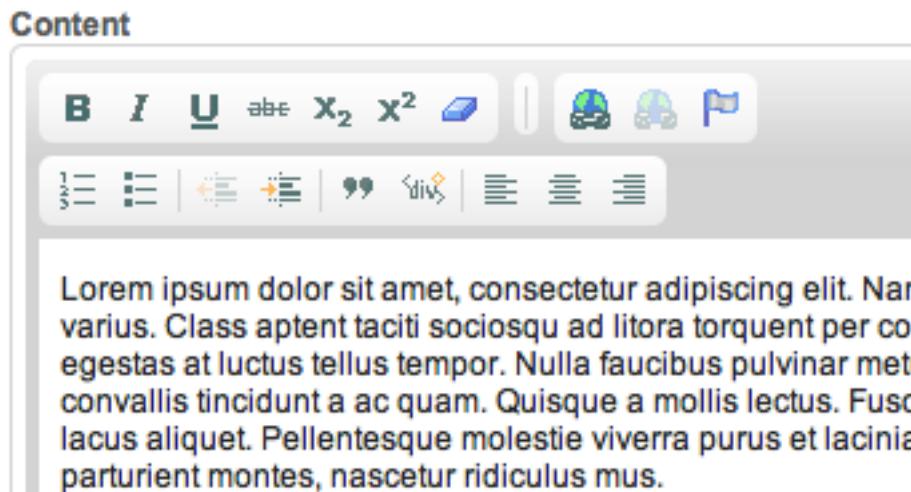


Figure 13.10: A customized toolbar.

```
# protected/views/page/view.php
<?php echo strip_tags($model->content,
'<p><a><div><ul><ol><li><span>');
?>
```

## Chapter 14

# JAVASCRIPT AND JQUERY

I've been very fortunate in my career in many regards, including this one: the first two languages I learned as a professional were PHP and JavaScript (I forget in which order). That was in 1999, and while there are plenty of dynamic Web sites built today that don't use PHP, there are very, very few that don't make use of JavaScript. I wouldn't go so far as to say that if you were to learn only one Web development language, it should be JavaScript, but I can definitively say that you need to learn at least two, and one must be JavaScript.

As Yii is used to create dynamic Web sites, being able to apply JavaScript to a Yii-based site is a critical skill. That is exactly the goal of this chapter, covering the three core concepts:

- Adding raw JavaScript to a page (as opposed to jQuery)
- Using jQuery on a page
- Implementing Ajax

And, at the end of the chapter, I'll explain how to implement a couple of common needs.

**Note that this chapter does assume comfort with JavaScript and jQuery.** It's just impossible to try to teach either the JavaScript language or the jQuery JavaScript framework in this book, let alone in this chapter. If you aren't already comfortable with JavaScript, might I (selfishly) suggest you read my "[Modern JavaScript: Develop and Design](#)" book, which teaches JavaScript for beginners, and introduces the jQuery library.

### What You Must Know

Despite the fact that I just said you need to know JavaScript and jQuery in order to make the most of this chapter, history would suggest some of you will continue

through this chapter regardless. As a precaution, I want to start with a few fundamentals of JavaScript, jQuery, and Yii that everyone needs to understand and appreciate.

First, *jQuery is JavaScript*. This should be obvious, but some don't appreciate the significance of this fact. When you're programming in jQuery, you're actually programming in JavaScript (just as when you're using Yii, you're programming in PHP). jQuery is extremely reliable and easy to use, which has a negative consequence: many people will implement jQuery without actually knowing JavaScript. That, to me, is a problem. Before attempting to use jQuery, learn JavaScript, because jQuery is JavaScript!

Second, *you will inevitably have issues due to how browsers load the Document Object Model (DOM)*. Trust me on this one. The DOM provides a way for browsers to represent and interact with elements in a Web page. But a browser does not have access to *any* page element until it has loaded *every* page element. I'm simplifying this discussion some; but programming as if that last sentence is exactly the case is most foolproof. This trips up JavaScript programmers that attempt to make immediate reference to DOM elements. The solution is to only reference DOM elements when the window's contents have been loaded, or when jQuery's "ready" event has occurred. If you know JavaScript, you know what I'm talking about, but I'm reminding you of this issue here as you'll inevitably make this common mistake in your Yii sites, too.

Third, *identify, install, and familiarize yourself with some good JavaScript debugging tools*. As JavaScript runs in the browser, you'll need to use your browser debugging tools to identify and fix any problems. Again, if you know JavaScript, you know this already.

## Adding JavaScript to a Page

The first thing you'll need to know to use JavaScript and jQuery in Yii is how to add JavaScript to a Web page. As with any standard Web page, there are two primary options:

- Link to an external file that contains the JavaScript code
- Place the JavaScript code directly in the page using SCRIPT tags

Just as I assume you're already comfortable with JavaScript, I'll also assume you know the arguments for and against both approaches. (Technically, there's a third option: place the JavaScript inline within an HTML tag. This is not a recommended approach in modern Web sites, however, and I won't demonstrate it here.)

### Linking to JavaScript Files

External JavaScript files are linked to a page using the SCRIPT tag:

```
<script src="/path/to/file.js"></script>
```

The contemporary approach is to link external files at the end of the HTML BODY, although some JavaScript libraries must be included in the HEAD.

If you need to include a JavaScript file on every page of your site, an option is to just add the reference to your layout file:

```
<script src="=Yii::app()-request->baseUrl;
?>/path/to/file.js"></script>
```

Do be certain to use an *absolute reference* to the file, for reasons explained in Chapter 6, “[Working with Views](#).”

Alternatively, you can use Yii’s `CHtml::scriptFile()` method to create the entire HTML tag:

```
<?php echo CHtml::scriptFile(Yii::app()->request->baseUrl .
 '/path/to/file.js');
?>
```

The end result is the same.

Sometimes, you’ll have external JavaScript files that should only be included on specific pages. In theory, you could just add the appropriate SCRIPT tag to the corresponding view files, but that’s less than ideal for a couple of reasons. For one, the final page will end up with SCRIPT tags in the middle of the page BODY, which is sloppy. Another reason why you don’t want to take this approach is that it gives you no vehicle for putting the script in the HTML HEAD, should that be necessary.

The better way to add external files to a page from within the view file is to use Yii’s “clientScript” component, and its `registerScriptFile()` method in particular:

```
# protected/views/foo/bar.php
<?php Yii::app()->clientScript
    ->registerScriptFile('/path/to/file.js'); ?>
```

The “clientScript” application component manages JavaScript and CSS resources used by a site. By calling that line anywhere in a view file (or a controller), Yii will automatically include a link to the named JavaScript file in the complete rendered HTML. Further, if, for whatever reason, you register the same JavaScript file more than once, Yii will still only create a single SCRIPT tag for that file.

By default, Yii will link the registered script in the HTML HEAD. To change the destination, add a second argument to `registerScriptFile()`. This argument should be a constant that indicates the proper position in the HTML page for the JavaScript file reference:

- CClientScript::POS\_HEAD, in the HEAD before the TITLE (the default)
- CClientScript::POS\_BEGIN, at the beginning of the BODY
- CClientScript::POS\_END, at the end of the BODY

To have the SCRIPT tag added at the end of the body, you would do this:

```
# protected/views/foo/bar.php
<?php Yii::app()->clientScript
    ->registerScriptFile('/path/to/file.js',
    CClientScript::POS_END); ?>
```

## Linking jQuery

The previous section explained how to link external JavaScript files from a Web page. There's a subset of external JavaScript files that are treated differently, however. I'm referring to JavaScript files that come with the Yii framework, such as the jQuery library.

To incorporate jQuery into a Web page, invoke the `registerCoreScript()` method, providing it with the value "jquery":

```
<?php Yii::app()->clientScript->registerCoreScript('jquery'); ?>
```

That line results in the jQuery library being linked to the page via a SCRIPT tag (**Figure 14.1**).

```
<link rel="stylesheet" type="text/css" href="/yii-test/css/main.css" />
<link rel="stylesheet" type="text/css" href="/yii-test/css/form.css" />

<script type="text/javascript" src="/yii-test/assets/8479529c/jquery.js"></script>
<title>My Web Application</title>
</head>
```

**Figure 14.1:** The resulting link to the jQuery library.

As you can see in the figure, the jQuery library is referenced within the Web directory's `assets` folder. Copies of the library will be placed in that folder by Yii's assets manager.

By default, `registerCoreScript()` will place SCRIPT tags within the HTML HEAD (as in Figure 14.1). To have Yii place the tags elsewhere, change the `coreScriptPosition` attribute value to one of the constants already named:

```
<?php
Yii::app()->clientScript->coreScriptPosition =
    CClientScript::POS_END;
Yii::app()->clientScript->registerCoreScript('jquery');
?>
```

You can also globally change this setting in your primary configuration file (by assigning a value to the “coreScriptPosition” index of the “clientScript” component).

Some pages, like those that use certain widgets or that have Ajax form validation enabled, will already include jQuery. Fortunately, you don’t have to worry about possible duplication, as the Yii assets manager will only incorporate the library once.

If you’re curious as to what other core scripts are available, check out the **web/js/packages.php** file found in your Yii framework directory. As of this writing, some of the core scripts are:

- jquery
- yii, a Yii extension of the jQuery library
- yiiactiveform, which provides code for client-side form validation
- jquery.ui
- cookie, for cookie management
- history, for client-side history management

This means, for example, to incorporate the jQuery User Interface library, you would do this:

```
<?php  
Yii::app()->clientScript->registerCoreScript('jquery.ui');  
?>
```

For all of the options, and details on the associated scripts, see the **web/js/sources** directory (within the Yii framework folder).

## Adding JavaScript Code

When you have short snippets of JavaScript code, or when the code only pertains to a single file, it’s common to write that code directly between the HTML SCRIPT tags. Again, you *could* do this in your view files:

```
<script>  
/* Actual JavaScript code. */  
</script>
```

You could also use the Yii **CHtml::script()** method to create the SCRIPT tag for you:

```
<?php echo CHtml::script('/* Actual JavaScript code. */'); ?>
```

You *could* add SCRIPT tags in either of those ways, but there's a better approach: the “clientScript” component’s `registerScript()` method. This is the companion to `registerScriptFile()`, but instead of linking to an external JavaScript file, it’s used to add JavaScript code directly to the page.

The method’s first argument is a unique identifier you should give to the code snippet. The second is the JavaScript code itself. This code is generated by Gii on the “admin” view pages:

```
1 # protected/views/page/admin.php
2 <?php
3 Yii::app()->clientScript->registerScript('search', "
4     $('.search-button').click(function(){
5         $('.search-form').toggle();
6         return false;
7     });
8     $('.search-form form').submit(function(){
9         $.fn.yiiGridView.update('post-grid', {
10             data: $(this).serialize()
11         });
12         return false;
13     });
14 ");
15 ?>
```

The first bit of JavaScript (lines 4-7), shows and hides the advanced search form that can appear above the grid. The form’s visibility is toggled when the user clicks on the search button. The second bit (lines 8-13) invokes the `yiiGridView.update()` method when the search form is submitted. Both sections use jQuery.

As you can see in that example, the combination of PHP and JavaScript can easily lead to syntax errors. You should use one set of quotation types to *encapsulate* the JavaScript (passed to `registerScript()`) and another type *within* the JavaScript. Also be certain to terminate JavaScript commands with semicolons, and terminate the PHP command, too. If the JavaScript you write doesn’t work, start by confirming that the resulting JavaScript code (in the browser’s source) is syntactically correct.

One of the benefits of providing a unique identifier is that the “clientScript” component will manage the code bits so that even if the same code (by identifier) is registered multiple times, it will still only be placed on the page once.

As with `registerScriptFile()`, `registerScript()` takes a final argument to indicate where, in the HTML page, the JavaScript should be added:

- `CClientScript::POS_HEAD`, in the HEAD before the TITLE (the default)
- `CClientScript::POS_BEGIN`, at the beginning of the BODY.

- `CClentScript::POS_END`, at the end of the BODY.
- `CClentScript::POS_LOAD`, within a `window.onload` event handler
- `CClentScript::POS_READY`, within a jQuery “ready” event handler

The two additional options are necessary because of the way the browser loads the DOM. If you are using jQuery and you want to execute some JavaScript when the document is ready, use `CClentScript::POS_READY`. It’s slightly faster than the standard JavaScript `window.onload` option. If you’re not using jQuery, then use `CClentScript::POS_LOAD`.

## Using JavaScript with CActiveForm

One of the absolutely most critical uses of JavaScript in today’s Web sites is for form validation. This is no less true when using Yii, although Yii can do much of the work for you, as is the case with so many things. Let’s quickly look at how JavaScript is used with forms in Yii, specifically when using the `CActiveForm` widget.

### Client-Side Validation

When you create a new `CActiveForm` widget instance, you can configure how it behaves, as is the case with most widgets. Configuration is performed by passing an array of name=>value pairs as the second argument to the `beginWidget()` method (see Chapter 12, “[Working with Widgets](#)”).

To enable client-side JavaScript form validation, set the “enableClientValidation” property to true:

```
# protected/views/page/_form.php
<?php $form=$this->beginWidget(' CActiveForm', array(
    'id'=>'page-form',
    'enableClientValidation'=>true,
)); ?>
```

By setting this property to true, Yii will add the appropriate JavaScript to the page to perform client-side validation. The validation will use the same rules as defined in the associated model, assuming that the validator is supported on the JavaScript side. At the time of this writing, all of these validators can also be used on the client side:

- `CBooleanValidator`
- `CCaptchaValidator`
- `CCompareValidator`
- `CEmailValidator`

- CNumberValidator
- CRangeValidator
- CRegularExpressionValidator
- CRequiredValidator
- CStringValidator
- CUrlValidator

To be perfectly clear, this means that if you have an `email` attribute in a model that's associated with an "email" form input and that has an "email" validator in the model's rules, that validation can be performed client-side, too. On the other hand, attributes that have the "default", "date", "exist", and other validators not listed above applied to them cannot be validated in the client.

*{TIP}* You may not be able to see, or appreciate, the effects of client-side validation until you configure the "clientOptions" as well. I'll explain those in just a couple of pages.

Not only will your existing model validation rules be used when you enable client-side validation, but so will the existing error messages for reporting problems. If you customize an error message in a validation rule, the client-side validation will use that when the data does not pass.

If the user does not have JavaScript enabled, then client-side validation cannot occur (of course). In those cases, the server-side validation will still be used (in the controller that handles the form submission). In fact, for security purposes, your controllers should always be written to perform server-side validation. Client-side validation is a convenience to the user; not a security technique.

*{WARNING}* Always use server-side validation!

## Ajax Validation

Another way to validate a form using JavaScript is via Ajax. Ajax validation makes an actual request of the server to validate the form data. For this reason, Ajax validation can be used to validate form elements that cannot be validated via client-side JavaScript alone, such as:

- The availability of a username
- Confirming that a value exists in a related table
- If a value is unique in the database

There are limits to Ajax validation, however. First, Ajax cannot validate uploaded files. This is a restriction on Ajax in general, not in Yii. Second, Ajax in Yii cannot validate "tabular" data: multiple records of data at a single time.

To enable Ajax validation, set “enableAjaxValidation” to true when configuring the CActiveForm widget:

```
# protected/views/page/_form.php
<?php $form=$this->beginWidget(' CActiveForm', array(
    'id'=>'page-form',
    'enableAjaxValidation'=>true,
)); ?>
```

{TIP} You may not be able to see, or appreciate, the effects of Ajax validation until you configure the “clientOptions” as well. I’ll explain those in just a couple of pages.

There are two sides to Ajax, of course: the client-side JavaScript and the server-side code that handles the JavaScript request. For the Ajax validation to work, you must create the appropriate server-side code, too. That is easily done, though, and Yii provides a template for you:

```
# protected/controllers/PageController.php
public function actionCreate() {
    $model=new Page;
    if(isset($_POST['ajax'])
    && $_POST['ajax']=='page-form') {
        echo CActiveForm::validate($model);
        Yii::app()->end();
    }
    // Rest of the action.
}
```

The code first checks if `$_POST['ajax']` is set and if it has the value of “*modelName*-form”. If so, the controller prints out the result returned by calling the `CActiveForm::validate()` method. The `validate()` method returns the results as JSON data (**Figure 14.2**), so that’s what the JavaScript in the browser will receive.

```
{"User_username":["Username cannot be blank."],"User_email":["Email cannot be blank."],"User_pass":["Password cannot be blank."],"User_acceptTerms":["You must accept the terms to register."]}
```

**Figure 14.2:** The JSON reporting for the validation of a user.

Next, the code terminates the application so that nothing else will be sent back to the JavaScript that made the Ajax request.

In the code created by Gii, this same process is handled slightly differently. Gii creates a controller method that performs the Ajax validation:

```
# protected/controllers/UserController.php
protected function performAjaxValidation($model) {
    if (isset($_POST['ajax']))
        && $_POST['ajax']=='page-form') {
        echo CActiveForm::validate($model);
        Yii::app()->end();
    }
}
```

Then both the “create” and “update” actions can make use of that method:

```
# protected/controllers/PageController.php
// Uncomment the following line if AJAX validation is needed
// $this->performAjaxValidation($model);
```

The end result is the same as if that validation code were in the individual action methods, however.

## Setting clientOptions

The third configuration option when working with CActiveForm with which you should be familiar is “clientOptions”. This is an array of values that can be used to further customize the JavaScript validation:

```
# protected/views/page/_form.php
<?php $form=$this->beginWidget('CActiveForm', array(
    'id'=>'page-form',
    'enableClientValidation'=>true,
    'clientOptions'=>array(
        /* name=>value pairs */
    )
)); ?>
```

For example, you can identify a different URL to use for validation purposes by assigning a value to “validationURL” (by default, the validation URL is the same as the form’s “action” attribute). Or you can change when validation is performed. By default, validation is performed when any form element’s value changes, but you can set the validation to occur upon submission instead:

```
# protected/views/page/_form.php
<?php $form=$this->beginWidget('CActiveForm', array(
    'id'=>'page-form',
    'enableClientValidation'=>true,
    'clientOptions'=>array(
        'validateOnSubmit'=>true,
        'validateOnChange'=>false,
    )
)); ?>
```

As another example, you can change the CSS classes associated with validation by changing the corresponding property:

- `errorCssClass`, which styles the container in which the error occurred (defaults to “error”)
- `successCssClass`
- `errorMessageCssClass`, which styles the error message itself (defaults to “errorMessage”)

For all the possibilities, see the [CActiveForm::clientOptions](#) documentation in the Yii API. Note that these settings can impact both types of client-side validation: only JavaScript or also Ajax.

## Implementing Ajax

Ajax is one of the reasons why JavaScript is so critical to today’s Web sites. Ajax has been around for more than a decade now, and the features Ajax can add to a Web site are pretty much expected by most users anymore (whether they know it or not). I’ve already mentioned Ajax once in this chapter (for validation purposes), and assume you do know the fundamentals of this vital technology. But let’s quickly look at a couple more ways to implement Ajax in a Yii-based site.

### Ajax in Controllers

Ajax blends the two sides of Web development: the client-side (aka, the browser) and the server-side. When developing Ajax processes, I like to start on the server-side of things so that I know what to do and expect on the client-side. (Also, I’m a server-side developer first.)

Server-side Ajax resources in Yii are represented as controller actions, just like regular Web pages. But there are a few major differences between an Ajax action and a standard one. The first key difference is that an Ajax response is almost always made up of the most minimal amount of data:

- Short, plain text
- A snippet of HTML
- More complex data as JSON
- More complex data as XML

For this reason, Ajax controller actions almost never use the `render()` method to create the output. Instead, there are two common approaches:

1. Directly print the desired output from the controller
2. Use `renderPartial()` to have a view represent the output (but without the primary layout file)

{NOTE} In Chapter 16, “[Leaving the Browser](#),” I present a situation in which you would use `render()`: You could output XML, with the primary layout file representing the beginning and end of the XML document.

Ajax actions are also different in that they’re not meant to be accessed directly by users in the browser. It’s not a big deal, normally, but at the very least, the user will have an unappealing, if not confusing, experience if she ends up directly requesting an Ajax resource. There are a couple of ways in Yii that you can limit access to an action to an Ajax request. One option is to set a filter:

```
# protected/controllers/SomeController.php
public function filters() {
    return array(
        'ajaxOnly + ajaxActionId'
    );
}
```

Just replace `ajaxActionId` with the actual ID value(s) of the action(s) that must be used via Ajax. Non-Ajax requests of the named action(s) will result in 400 (Bad Request) exceptions being thrown.

Some actions are written to be accessed via Ajax and non-Ajax alike, reacting slightly differently in each case. The “create” and “update” actions generated by Gii are examples. In such situations, you can test if an Ajax request is being made via the “request” application component:

```
# protected/controllers/SomeController.php
public function actionSomething() {
    // React different based upon Ajax request status:
    if (Yii::app()->request->isAjaxRequest) {
        // Do things this way.
    }
}
```

```
    } else {
        // Do things this other way.
    }
}
```

In situations where the same action may be used by Ajax and non-Ajax requests, for the Ajax portion, you'll also need to invoke the application's `end()` method to terminate the output immediately. This prevents the non-Ajax output from being added to the result. Here's an example of what I mean:

```
# protected/controllers/PageController.php
public function actionCreate() {
    $model=new Page;
    if(isset($_POST['ajax'])
    && $_POST['ajax']=='page-form') {
        echo CActiveForm::validate($model);
        Yii::app()->end();
    }
    // Rest of the action.
    // Including rendering the form.
}
```

The last thing to keep in mind with Ajax processes is to set the proper controller permissions. It's not obvious to many developers, but when an Ajax request is performed, it's as if the user himself requested the resource directly. In other words, an Ajax request made from a user's browser is still being made by that user. This means that Yii's permissions apply to Ajax requests just the same as they do to other requests. Keep this in mind when creating actions and setting permissions.

As a rule of thumb, if the "foo" page makes an Ajax request of the "bar" action, then, logically, both "foo" and "bar" need to have the same permissions in the controller. Still, very rarely will an Ajax action need protection at all, so you can normally make them publicly accessible. Do so if you'd rather not run the risk of having Ajax request failures due to permission issues.

Another approach for the Ajax permissions is to create a controller explicitly for all Ajax requests. That controller would have open permissions, like the "site" controller does. I'll demonstrate that concept next.

## Sample Ajax Actions

In order to test Ajax processes, at least within the confines of this book, it may help to have a couple of test Ajax processes that can be used for experimentation. To be clear, I'm talking about making sample PHP resources that client-side JavaScript can request. To do so, let's create a new controller for this purpose:

```
# protected/controllers/AjaxController.php
<?php
class AjaxController extends Controller {
}
```

Within that controller, I'll define three actions:

- One that returns (or prints) a simple string
- One that returns some HTML
- One that returns dynamic HTML (in theory)

In just a few pages, I'll add another action that returns data in JSON format. Note that all of these actions as defined will be rather static, but they are all easy enough to update to being truly dynamic in a real-world site. Also, there are no filters in this controller, such as the access control filter, so every action will be executable by any user. The “ajaxOnly” filter is not used either, so you can test these directly in your browser.

The first action only returns a simple text message:

```
# protected/controllers/AjaxController.php
public function actionSimple() {
    echo 'true';
}
```

This simple action might be used to verify that an username is available or that an email address has not yet been registered. In a real-world site, the action would perform the necessary logic, and then print “true” or “false” accordingly. Note that the Ajax action must print *strings*, not Booleans.

Next, there's an action that returns a bit of HTML. The premise is the same, but the text is actually HTML:

```
# protected/controllers/AjaxController.php
public function actionHtml() {
    echo '<p>Lorem ipsum <em>dolor</em>...</p>';
}
```

Finally, the third action returns more dynamic HTML, using a variable and a view file:

```
# protected/controllers/AjaxController.php
public function actionDynamicHtml() {
    // Dynamic data:
```

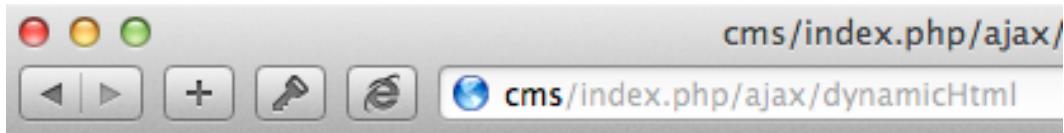
```
$data = array(
    'title'=>'Dynamic!',
    'content'=>'<p>Lorem ipsum <em>dolor</em>...</p>'
);
// Render the page:
$this->renderPartial('dynamic', array('data'=>$data));
}
```

Obviously, in the real-world, the data itself might be pulled from the database.

The view file looks like this:

```
# protected/view/ajax/dynamic.php
<?php
/* @var $this AjaxController */
/* @var $data array */
?>
<article>
    <h3><?php echo $data['title']; ?></h3>
    <div><?php echo $data['content']; ?></div>
</article>
```

The received `$data` array's pieces are placed within a context of HTML (**Figure 14.3**). Changing the data values in the controller therefore changes the output.



## Dynamic!

**L**orem ipsum **dolor** sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit animi.

**Figure 14.3:** The “dynamic” HTML response.

Now that three sample Ajax processes have been defined, you can test them in your browser by going to:

- ajax/simple

- **ajax/html**
- **ajax/dynamicHtml**

{TIP} I always recommend testing server-side Ajax resources directly first, to confirm they are working, before connecting them to the JavaScript.

Once you have those working as you would hope, it's time to turn to the JavaScript. There are four ways you can perform an Ajax request using Yii:

- Via an immediate Ajax request
- Via a link
- Via a button
- By tying an Ajax request to another DOM element

I'll explain all three, saving the last example for later in the chapter.

## Making Direct Ajax Calls

To start, let's look at how to make a direct and immediate Ajax call. This is accomplished via the `CHtml::ajax()` method. It takes one argument: an array of options. As soon as this method is invoked, the Ajax request is begun. In my experience, one does not frequently use this approach (as opposed to an Ajax request triggered by a user action), but as this method is the foundation for the alternatives, I'll begin with it.

The syntax is:

```
# protected/views/foo/bar.php
<?php echo CHtml::ajax(/* options */); ?>
```

That method call creates the JavaScript required to perform an Ajax request. The JavaScript itself uses the jQuery `ajax()` method. For the options, you can start with the possible settings outlined for the jQuery `ajax()` method in the [jQuery documentation](#). If you do Ajax in Yii (using jQuery), you absolutely must familiarize yourself with jQuery's Ajax options.

The most important of the configuration options are:

- “data”, which is data to be sent as part of the request
- “dataType”, the type of data expected in return (“text”, “html”, “json”, etc.)
- “url”, the URL to request
- “type”, the request, or method, type (i.e., “get” or “post”)

- “success”, the JavaScript function to call upon a successful request being made

Here is how you might perform an Ajax request of the “simple” controller action:

```
# protected/views/foo/bar.php
<?php echo CHtml::ajax(array(
    'dataType' => 'text',
    'url' => Yii::app()->createUrl('ajax/simple'),
    'type' => 'get',
    'success' => 'function(result) {
        alert(result);
    }'
)); ?>
```

There are a couple of things to notice there. First, for the URL, use Yii to create a proper URL (e.g., using `createUrl()`). Not using an accurate URL is a common cause of problems. Second, the “success” item takes a JavaScript function that will be invoked when the request is successfully completed. This can be the name of an existing JavaScript function, or an anonymous function as in the above. Per how jQuery’s `ajax()` method works, this function can be written to take up to three arguments, the first being the actual response.

That line of Yii code generates this output:

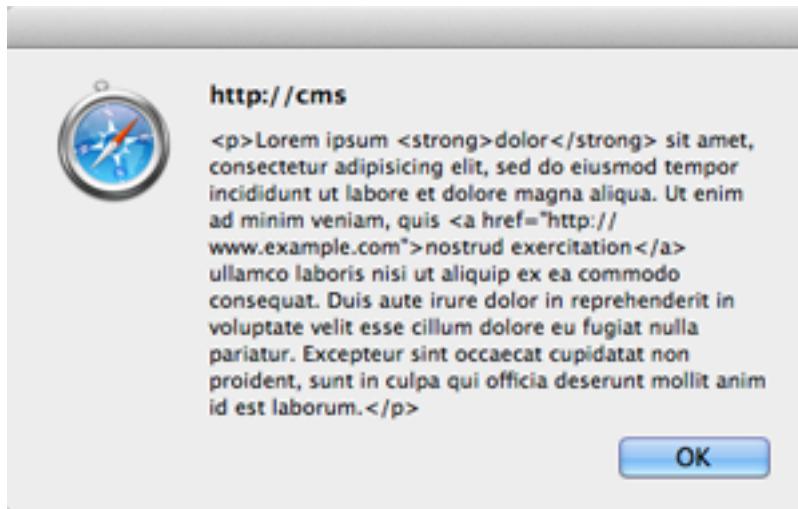
```
jQuery.ajax({
    'dataType':'text',
    'url':'/index.php/ajax/simple',
    'type':'get',
    'success':function(result) { alert(result); },
    'cache':false,
    'data':jQuery(this).parents("form").serialize()
});
```

Understand that `CHtml::ajax()` only *returns* that code. In order for it to be actually executed, it must be added to the page as JavaScript:

```
<?php echo CHtml::script(
    CHtml::ajax(array(
        'dataType' => 'text',
        'url' => Yii::app()->createUrl('ajax/simple'),
        'type' => 'get',
        'success' => 'function(result) {
            alert(result);
        }'
    )) // ajax
```

```
); // script  
?>
```

Now, when the page is loaded, the Ajax request will be made and the response alerted. Merely change the URL being requested to get different responses (**Figure 14.4**).



**Figure 14.4:** The response from the HTML action.

Of course, you don't want to just alert the Ajax response. Normally, you'll update the DOM in some way, perhaps based upon what the response was:

```
<?php echo CHtml::script(  
CHtml::ajax(array(  
    'dataType' => 'text',  
    'url' => Yii::app()->createUrl('ajax/simple'),  
    'type' => 'get',  
    'success' => 'function(result) {  
        if (result === "true") {  
            $("#response").text("The username is available.");  
        } else {  
            $("#response").text("The username has been taken.");  
        }  
    }'  
) // ajax  
); // script  
?>
```

(A quick reminder: I assume you're comfortable with JavaScript and jQuery. If not, learn them now!)

Or, you may add the response itself to the page:

```
<?php echo CHtml::script(
CHtml::ajax(array(
    'dataType' => 'text',
    'url' => Yii::app()->createUrl('ajax/html'),
    'type' => 'get',
    'success' => 'function(result) {
        $("#destination").html(result);
    }'
)) // ajax
); // script
?>
```

When you're doing a simple update of the page using the result, Yii has created a shortcut for you. Assign the jQuery selector (the element(s) being updated with the result) to the "update" index. This is the equivalent to the above:

```
<?php echo CHtml::script(
CHtml::ajax(array(
    'dataType' => 'text',
    'url' => Yii::app()->createUrl('ajax/html'),
    'type' => 'get',
    'update' => '#destination'
)) // ajax
); // script
?>
```

Again, that code is the same as the previous example. jQuery is used to select the element with an ID value of "destination", and the jQuery `html()` method is invoked upon that selection, which replaces its HTML content with the new content provided.

*{TIP}* Yii also has a "replace" option, which is used like "update", but invokes the jQuery `replaceWith()` method instead of `html()`.

Understand that if you use the jQuery "success" option, then any "update" or "replace" value will be ignored.

## Links and Buttons

Generally speaking, you won't want to use `CHtml::ajax()` itself very often. But you need to understand the `ajax()` method in order to use two Yii methods that make use of the same jQuery `ajax()`:

- CHtml::ajaxLink()
- CHtml::ajaxButton()

The only difference in the two is that one creates a link and the other creates a button. Both, when clicked, will perform the Ajax request. The arguments to both methods are essentially the same (**Figure 14.5**).

#### ajaxButton() method

public static string ajaxButton(string \$label, mixed \$url, array \$ajaxOptions=array( ), array \$htmlOptions=array( ))		
\$label	string	the button label
\$url	mixed	the URL for the AJAX request. If empty, it is assumed to be the current URL. See <a href="#">normalizeUrl</a> for more details.
\$ajaxOptions	array	AJAX options (see <a href="#">ajax</a> )
\$htmlOptions	array	additional HTML attributes. Besides normal HTML attributes, a few special attributes are also recognized (see <a href="#">clientChange</a> and <a href="#">tag</a> for more details.)
{return}	string	the generated button

**Figure 14.5:** The description of the ajaxButton() method.

Unlike ajax(), the URL to request is the second argument. The third argument to both methods is how you set any of the other jQuery ajax() settings, plus the two additional Yii options: “update” and “replace”. Here’s an example:

```
# protected/views/foo/bar.php
<div id="destination"></div>
<?php echo CHtml::ajaxButton('Get Content',
Yii::app()->createUrl('ajax/dynamicHtml'),
array(
    'dataType' => 'html',
    'type' => 'get',
    'update' => '#destination'
) // ajax
); // script
?>
```

**Figures 14.6** and **14.7** show this in action. (Well, in printed, not live, action.) Also notice that this method can be called on its own directly, not fed to CHtml::script().

## Working with JSON

The three Ajax controller actions defined offer a range of possibilities, but there’s one more example to implement. When you need to return more complex data from

# Testing Ajax

[Get Content](#)

**Figure 14.6:** The page when the user first sees it.

# Testing Ajax

## Dynamic!

**Get Content**

**Dynamic!**  
Lorem ipsum dolor sit amet, consectetur ad  
ullamco laboris nisi ut aliquip ex ea commo  
cupidatat non proident, sunt in culpa qui off

**Figure 14.7:** The same page after the user has clicked the button.

the server to the client, plain-text and HTML formats are insufficient. Originally, eXtensible Markup Language (XML) was used as the data format (“Ajax” either is or is not an acronym for “Asynchronous JavaScript and XML”, depending upon whom you ask). These days, JSON (JavaScript Object Notation) is the norm. The JSON format is compact, resulting in faster response times, and readily usable by JavaScript in the client.

The downside to JSON is that its syntax is particular and can be difficult to get right. Fortunately, Yii has the `CJSON` class, and its `encode()` method, for creating JSON data. You can feed it almost any data type and it will output proper JSON. Let’s add another demo action to the “ajax” controller:

{NOTE} `CJSON` is one of the rare classes in Yii whose name is entirely in uppercase letters.

```
# protected/controllers/AjaxController.php
public function actionJson() {
    $data = array(
        'title'=>'Dynamic!',
        'content'=><p>Lorem ipsum <em>dolor</em>...</p>
    );
    echo CJSON::encode($data);
}
```

And here’s how that might be used in the view file:

```
<h3 id="updateTitle"></h3>
<div id="updateContent"></div>

<?php echo CHtml::ajaxButton('Get Content',
Yii::app()->createUrl('ajax/json'), array(
    'dataType' => 'json',
    'type' => 'get',
    'success' => 'function(result) {
        $("#updateTitle").html(result.title);
        $("#updateContent").html(result.content);
    }'
) // ajax
); // script
?>
```

Figures 14.8 and 14.9 show this in action.

# Testing Ajax

[Get Content](#)

**Figure 14.8:** The page when the user first sees it.

# Testing Ajax

## Dynamic!

**LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISICUM LOR**  
ullamco laboris nisi ut aliquip ex ea commodo consequat. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

[Get Content](#)

**Figure 14.9:** The same page after the user has clicked the button.

## Common Needs

To wrap up this chapter, I'll cover a few common needs and points of confusion when it comes to JavaScript and jQuery in Yii. As with the entire book, if you're still confused or curious about an issue after completing this chapter, let me know, and I'll see about addressing it in another release of the book or online.

### Setting Focus

If you want to set the focus on a particular form element (e.g., have the user's cursor begin in that element), there are a couple of options. The first is available if you're using HTML5: set the "autofocus" property on the element. Here's how that would look in straight-up HTML:

```
<input type="email" name="email" autofocus>
```

When you're using Yii to create form elements, just add this as an additional HTML attribute:

```
<?php echo $form->textField($model, 'attribute',
    array('autofocus'=>'autofocus')); ?>
```

The HTML5 “autofocus” property is supported by most modern browsers, but not in Internet Explorer until version 10. As a backup, you can also use JavaScript to set the focus. This is done by configuring the `CActiveForm` widget’s `focus` property. You can assign to this property a value in many different formats. Normally, the most direct option is to set it to the element associated with a specific model attribute:

```
# protected/views/site/login.php
<?php $form=$this->beginWidget('CActiveForm', array(
    'id'=>'login-form',
    'enableClientValidation'=>true,
    'clientOptions'=>array(
        'validateOnSubmit'=>true,
    ),
    'focus'=>array($model, 'username')
)); ?>
```

Other possible values for “focus” are listed in the documentation for the `CActiveForm`.

## Implementing Autocomplete

A common use of JavaScript and Ajax is *autocomplete* functionality. First popularized as Google’s Suggest tool, autocomplete is now a Web standard, including in the [Yii class reference](#) (**Figure 14.10**).

Thanks to the jQuery UI autocomplete widget, and the Yii `CJuiAutoComplete` class (defined in the Zii extension), it’s pretty easy to implement autocomplete on your Web site. As an example of this, let’s create the ability to autocomplete pages in the CMS site by title (**Figure 14.11**).

To start, in the view file, create an instance of the `CJuiAutoComplete` widget:

```
1 <?php
2 $this->widget('zii.widgets.jui.CJuiAutoComplete', array(
3     'name'=>'title',
4     'sourceUrl'=>Yii::app()->createUrl('ajax/getPageTitles'),
5     'options'=>array(
6         'minLength'=>'2',
7         'type'=>'get',
8         'select'=>'js:function(event, ui) {
```

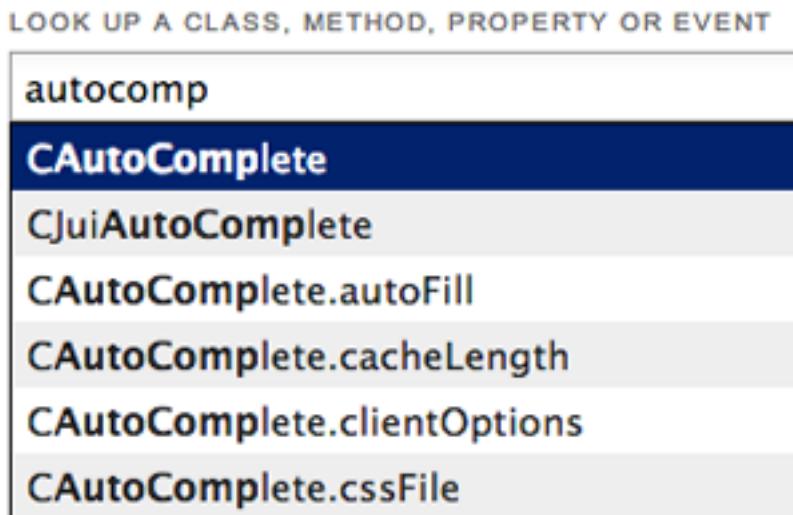


Figure 14.10: Autocomplete functionality in the Yii class reference.

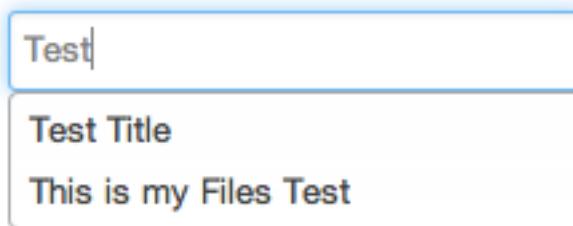


Figure 14.11: Autocompletion of page titles.

```
9         $("#" + selectedTitle).text(ui.item.value);
10    }'
11  ),
12 );
13 ?>
14 <span id="selectedTitle"></span>
```

This widget will, by default, create a text input with the “name” value you provide (line 3). When using dynamic data returned by an Ajax request, the “sourceUrl” value needs to point to the controller action that will return the results (line 4).

The “options” is where you configure the jQuery UI options, found in the [jQuery UI documentation for the autocomplete widget](#). In those options, I’ve set the “minLength” to 2, so that no results are returned until at least 2 characters are entered. I’ve also created a function that will be called when a selection is made from the list of options (lines 8-10). Let’s look at that function in detail...

The anonymous function takes two arguments: an event and an object. This object is conventionally, in jQuery UI, called “ui”, and its `item` property will represent the selected item. To make it obvious which value was selected, a SPAN is updated upon selection.

If you’re paying close attention, you’ll notice that this function definition is prefaced with “js:”. This is required by Yii (in some situations). The need for this preface is that Yii will often escape values to prevent them from being executed code, which could be insecure. Thus, if you were to write `function(event, ui) { ... }` in the above code, it wouldn’t result in a usable JavaScript function. The fix is to preface the function definition with “js:” to tell Yii that this is proper JavaScript, not to be escaped (i.e., creating some executable JavaScript code is your intent).

With the widget in place in the view, it’s time to create the controller action that provides the source data for the widget. Per the widget configuration, the source URL is “`ajax/getPageTitles`”, which means that there needs to be a “`getPageTitles`” action in the “`ajax`” controller. This action should use the submitted term—what the user typed—and return an array of values in JSON format:

```
public function actiongetPageTitles() {
    $q = 'SELECT id, title AS value FROM page
          WHERE title LIKE ?';
    $cmd = Yii::app()->db->createCommand($q);
    $result = $cmd->query(array('%' . $_GET['term'] . '%'));
    $data = array();
    foreach ($result as $row) {
        $data[] = $row;
    }
    echo CJSON::encode($data);
```

```
    Yii::app()->end();  
}
```

The specific query is supposed to fetch the page ID and title for every page whose title is similar to the provided input. To accomplish that, I'm using Data Access Objects (DAO), explained in Chapter 8, “[Working with Databases](#).” I've chosen to make the LIKE condition extremely flexible (i.e., LIKE %term%), but you could change it to just LIKE term% to be less so. The jQuery autocomplete widget will provide what the user typed as “term”, so that's available in \$\_GET['term'].

Finally, notice that I've chosen to select the page title aliased (in the query) as “value”. This makes it easy to use in the generated drop-down list of autocomplete matches. This is also why the “select” JavaScript function in the widget refers to ui.item.value. If the page titles were selected as “title”, you would also have to configure how the matches are rendered by jQuery UI.

And that's enough to implement autocomplete in a Yii-based site. To properly use the selected value, just change the contents of the “select” function to suit your needs.

If you have any problems in implementing this, begin by confirming the results of your Ajax request, as that's the most likely cause of problems. Also familiarize yourself with the jQuery UI autocomplete widget, as the CJuiAutoComplete class is just a wrapper to it.

## Using the clientChange Options

The last topic I want to discuss has a broad range of influence. Many of the methods in CHtml that create form elements, take an “htmlOptions” parameter. This is an array of name=>value pairs that can configure the resulting element's HTML. For example, you can use “htmlOptions” to apply a class to an element or size a text area. Along with the expected HTML attributes that you can configure through “htmlOptions”, there are also “clientChange” options.

The “clientChange” options stem from the CHtml::clientChange() method, which is used to create JavaScript to be associated with changes in the browser. For example, there's a “confirm” “clientChange” option which allows you to add a JavaScript confirmation window to a form element:

```
# protected/views/foo/bar.php  
<?php echo CHtml::submitButton($model->isNewRecord ?  
    'Create' : 'Save',  
    array(  
        'confirm'=>'Are you sure?'  
    )  
); ?>
```

In fact, the `CGridView` widget uses this same functionality to confirm the deletion of records.

Another useful “clientChange” option is “ajax”. If you create a form element with an “htmlOption” of “ajax”, you’ll tie changes in that form element to an Ajax request (using the jQuery `ajax()` method already explained). Here, then, is how you could validate that a username is available via Ajax (without applying Ajax validation to the entire form):

```
# protected/views/foo/bar.php
<?php echo $form->textField($model, 'username', array(
    'ajax' => array(
        'dataType' => 'text',
        'data'=> array('username'=>'js:$(this).val()'),
        'url' => Yii::app()->createUrl('user/checkUsername'),
        'type' => 'get',
        'success' => 'function(result) {
            if (result === "true") {
                $("#response").text("The username is available.");
            } else {
                $("#response").text("The username has been taken.");
            }
        }'
    )
)); ?>
```

With that code in place, changes to the username text input results in an Ajax request of the “checkUsername” action in the “user” controller. That action would receive the entered value in `$_GET['username']`. The action would then return just “true” or “false” depending upon whether or not that username is available.

This ability to tie Ajax requests to specific form elements allows for tons of functionality, such as dependent dropdown lists, as explained in [this Yii wiki article](#). Once you understand the role that the “clientChange” option plays, the potential uses are only limited to your needs and imagination.

## Chapter 15

# INTERNATIONALIZATION

Adoption of *internationalization*, commonly abbreviated *i18n*, is an acknowledgement that the World Wide Web is indeed global. A Web site is available to anyone anywhere in the world, as long as they have a browser connected to the Internet, including on a mobile device. In order to support as wide of an audience as possible, your site should embrace internationalization.

In this chapter, I'll explain what i18n is, and how you implement it in a Yii-based site.

{TIP} The abbreviation “i18n” simply stands for the initial “i”, and terminating “n”, plus the 18 letters that come between them.

### What is i18n?

Internationalization is the act of writing software so that it can adapt to the different languages and customs of the software's various users. The most obvious example of customization for the user is the language used by the site. By applying internationalization, a site could present all of its navigation and interface items in English for some users, in French for others, in Russian for others, and so on. Behind the scenes, the core functionality would be the same, but the user experience is greatly enhanced.

The differences between any two languages can vary in many ways:

- The characters used (Latin-based languages share many common characters, Cyrillic languages use entirely different ones)
- The characters used to represent numbers specifically
- The direction in which words and sentences are written (left-to-right, right-to-left, or even top-down)
- How words are capitalized

- How words are sorted (as in an alphabetical list)

{TIP} In order to support any possible language, be certain that the Web page and the database use UTF-8 encoding.

Along with the primary issue of the language used, two other topics are commonly associated with internationalization:

- Cultural habits
- Writing conventions

Cultural differences range from the very simple to the highly complex (and sensitive). Simple differences include the:

- Formatting of telephone numbers
- Formatting of addresses and postal codes
- Currency used
- Weights and measures used

If you really want to focus on cultural differences, you'll get into issues such as the significance given to certain colors, images, words, and so forth. For example, in the Western world, brides wear the color white, but in China and parts of Africa, white is a mourning color. If you're creating a truly international site, those are the kinds of things you'll want to get right. That being said, the high end of cultural issues are well beyond the scope of this chapter.

Next, there's the issue of *writing conventions*, which is somewhat different than the language issue. Here, I'm not talking about the words or characters used, but rather in what format. The two most obvious examples are how one writes dates/times and numbers.

With dates, the issue is most problematic when using two-digit representations: 08/02/06. In the United States, that would be read as August 2nd, 2006. In Europe, that would be read as February 8th, 2006. It could also be interpreted as February 6th, 2008. Until we can all agree on the proper, standardized way of representing dates (which will never happen), you may want to have your site adjust accordingly.

With numbers, the differences are mostly a matter of what character is used for separating thousands and for separating decimals. Should it be 1,000.78 or 1.000,78?

Internationalization starts with the biggest, global differences, such as the character set and language used. This is further customized via *localization* (L10n), which takes into account *locale* issues. A user's locale consists of her preferred language, region, and sometimes other cultural preferences. When you install a new operating system for the first time, and it asks you about your physical location, preferred

language (e.g., English US, English UK, etc.), time zone, and so forth, those answers all go into your *locale*.

With an understanding of i18n in place, the next issue is: should you use internationalization? As with almost everything, the answer is “it depends”. If you’re creating a Web site for a local restaurant, you probably don’t need to worry about internationalization at all. An exception would be if there’s a large population in the area that speaks another language or represents a significantly different culture. On the other hand, if you’re creating a site that you intend to be a global resource or e-commerce marketplace, embracing some internationalization would certainly help.

To implement internationalization in a Yii-based site, you’ll need to use a combination of the framework and some of your own logic and effort. Yii itself provides:

- Locale data
- A tool for displaying text in different languages
- Locale-specific date, time, and number formatting

Let’s look at these features of Yii in more detail.

{NOTE} You’ll need to add your own logic to your site to display dates and times in the user’s time zone.

## Setting the Locale

To use internationalization, you have to establish the locales involved. Specifically, you must set two:

- The application’s (i.e., the language in which you designed and wrote the application)
- The user’s

For a simple example, say that I create a Web site, and my locale is English (US). For someone in the United Kingdom, I may want to present the site customized to her locale, English (UK).

{NOTE} If the application and the user share a single locale, no internationalization is necessary, as no changes would be required.

Locales can be set by assigning values to two properties of the `CApplication` object:

- `sourceLanguage` is the locale of the application (i.e., the “from” language)

- `language` is the user's locale, which the [Guide](#) also calls the “target language” (i.e., the “to” or destination language)

As these are public, writable properties of the `CApplication` object, you can assign values to them in your configuration file. Locale values are represented as *LanguageID\_RegionID*. For example, my locale is `en_US`, which stands for English in the United States. For consistency sake, Yii uses all lowercase letters for locales, making mine `en_us`.

```
# protected/config/main.php
return array(
    'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'...',
    'name'=>'My Web Application',
    'sourceLanguage' => 'en_us',
    'language'=>'en_uk',
    // Other stuff.
);
```

English (US) is the default, so you only need to provide a value if you'll be using a locale other than the default. Second, it's not often that you'd set the target locale in the primary configuration file (i.e., identify two separate locales on a site-wide basis). Normally you'll want to set the user's locale dynamically:

```
Yii::app()->language = 'en_uk';
```

How you determine the user's locale will be covered next.

*{TIP}* The locale represents more than just the user's preferred language, but the term is commonly used to refer to the user's language.

## Detecting the User's Locale

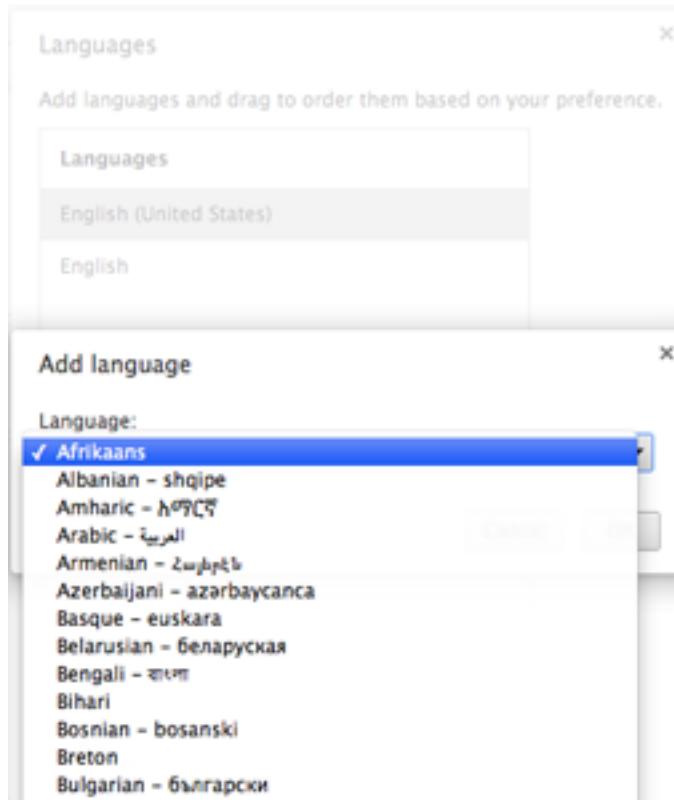
To identify the user's preferred locale, you'll need to get that information from the user. This can be done either overtly or secretly. The overt option is to present the user with an interface element through which he can set his locale. This might be a drop-down menu of languages or a series of flag icons. When the user makes a selection in the drop-down menu, or clicks on a flag icon, the site would then set the locale to that selection and update the page to that locale.

*{TIP}* When using the drop-down menu route, be sure the options are in the native languages. Showing an English user the word “English” written in Chinese characters will do little good.

A second option is to get the user's preferred locale from the browser itself. This value is available in the `preferredLanguage` property of the `CHttpRequest` object:

```
if (!empty(Yii::app()->request->preferredLanguage)) {  
    Yii::app()->language = Yii::app()->request->preferredLanguage;  
}
```

The browser will have this value by either getting it from the computer itself, or by having it be set within the browser's preferences (**Figure 15.1**). If set, the browser may provide this to Web sites as part of the request (in an "Accept-Language" header), which is why the value could be found in the `CHttpRequest` object in Yii.



**Figure 15.1:** Chrome's interface for managing languages.

If multiple acceptable languages are provided, you can use the `preferredLanguages` property of `CHttpRequest` to find those (not the pluralized form of "Language"). The property will be an array of values, in order from most preferred to least. Using that information, you could loop through the acceptable languages until one is found that matches a locale your application is set to handle.

{NOTE} The Yii framework includes locale data for almost every language and region, thanks to the [Common Locale Data Repository](#) (CLDR).

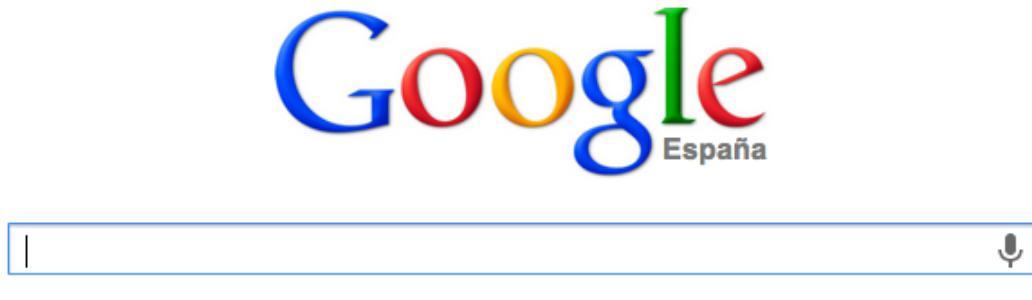
The problem with using the user’s provided locale (i.e., that provided by the browser) is that it assumes your site is setup to handle any possible language value. Of course, that will never be the case. I would recommend that you setup your system so that it checks the user’s preferred language, and uses that if possible. If the preferred language is not supported, you would then prompt the user to select her preferred language.

Moreover, there is a strong argument to supporting both methods of user locale selection regardless: both overt and implied. For example, I might be on vacation in Greece and stop in at an Internet café. The locale on those computers could logically be *el\_GR* (or *el\_gr* in Yii), which is what the browser would provide to a site I visit. In order for me to use the site, however, I’d need a way to manually set the locale to my preferred *en\_US*.

With the target (user) locale set, you can now perform internationalization: customize the experience to the user.

## Providing Language-Appropriate Text

The biggest role i18n plays is providing language-appropriate text. For example, standard Google has a “Google Search” button, but Spanish Google has “Buscar con Google” (**Figure 15.2**).



**Figure 15.2:** The Spanish Google interface.

The Yii Guide refers to the ability to changing text based upon the language in use as “translation”. However, I tend to think of translating as an active, thoughtful act, whereas what you’ll do in Yii is really string replacement. That being said, certainly “translation” is a pithier label, and I’ll continue to use it in this chapter to be consistent with the Yii documentation.

Translating is a two-step process:

1. Define in your application the translations for various words and strings.
2. Invoke a specific method to have Yii retrieve the string in the desired language.

I'll explain these steps over the next several pages.

## Defining Translations

The reason I don't care for the term "translation" is that Yii is not actively translating anything. What *is* happening is that you define the strings you'll need to use in the languages you want to support. Then, when the time comes to present that string, instead of printing the string itself, you invoke a function that will retrieve the correct version of the saved string in the destination language.

The first step, then, is defining the "translations". The translations are stored in one of three places:

- A PHP file
- The database
- [GNU gettext](#) files

These are known as *message sources*. Behind-the-scenes, the `CMessagesource` class defines the necessary storage and retrieval functionality. The three listed storage options all use classes that extend `CMessagesource`. You can also extend it yourself to create your own message source. For simplicity sake here, let's use a PHP file.

The name and location of the PHP file will depend upon two factors:

1. The locale ID
2. A category name

The locale ID is the Yii format version, such as `en_us` or `el_gr`.

By default, PHP message files go in **protected/messages/LocaleID**. If your application is going to support three languages, besides the source/application language, then you'll need to have three subdirectories within **protected/messages**.

*{NOTE}* You do not need to provide a message source for the application's primary language, as no translations are required when the primary and target languages are the same.

Within each locale directory, you'll have one PHP file for each message category. Categories are simply an organizational scheme that makes creating, maintaining, and using translations easier and faster. Instead of having, say, 200 strings defined in one file, those 200 can be broken up into 5 or 10 categories.

The categories are of your own creation, organized as you see fit. As a simple starting point, you might want a category for each model in your application, plus a category for the application as a whole (the most universal category). In a CMS example, that would mean you'd create these files:

- `app.php`
- `comment.php`
- `file.php`
- `page.php`
- `user.php`

Understand that you'll need to create each of these files for each locale you'll be supporting.

{NOTE} You cannot create a category named “yii”, as that's reserved by the framework already.

Within each file, you'll need to return an array. The array should use the message in your source language as its key and the same message in the target (aka user's) language as its value. For example, here's what part of the “app” category would look like with French translations:

```
# protected/messages/fr_fr/app.php
<?php
return array(
    'Home' => 'Accueil',
    'Register' => 'S\'enregistrer',
    'Login' => 'Se connecter',
    'Logout' => 'Déconnexion',
    'Subject' => 'Sujet',
    'Body' => 'Contenu',
    'Submit' => 'Soumettez'
);
```

The success of the translation system is dependent upon the index there. This is case sensitive and must exactly match the string in the source language.

Here's how the same words would be represented in Norwegian:

```
# protected/messages/nb_no/app.php
<?php
return array(
    'Home' => 'Hjem',
    'Register' => 'Registrer deg',
    'Login' => 'Logg inn',
    'Logout' => 'Logg ut',
    'Subject' => 'Emne',
    'Body' => 'Melding',
    'Submit' => 'Send'
);
```

For now, I'm going to work with those simple translations. In just a few pages, you'll learn how to make translations more flexible.

{TIP} When using translations, for optimal performance, you'll want to set a `cachingDuration` in the application's configuration which will tell Yii to cache the messages. You'll learn more about caching in Chapter 17, "Improving Performance".

## Using Translations

Once you've defined the translations, using them is remarkably easy. The static `t()` method in the `YiiBase` class serves this purpose. Its first argument is the category (which, therefore, identifies the message source) and its second is the string to be translated. This needs to match the indexes used in the PHP array, and should be in the application's native language.

Because `t()` is a *static* method, it's invoked without an object instance:

```
echo Yii::t('app', 'Home');
```

And that's all there is to it! If the target (user) locale is `fr_fr`, then "Accueil" will be printed by that line. If the target locale is `nb_no`, "Hjem" will be printed instead. If the target locale is the same as the application's, then no translation occurs at all and the provided string is printed.

{TIP} If an extension, such as a module or a widget, supports i18n, you can access those translation strings using `extName.category`.

## Using Placeholders and Parameters

There are plenty of strings that can be translated directly, without any dynamic functionality at all. For example, navigation elements, such as "Home", "Search", and so forth, will all be treated literally without modification. Other times, the translations need to be more dynamic. For example, a message that indicates an email address was not found in the system may want to display the actual email address, too. For these situations, you can use placeholders and parameters.

The placeholders go in the string: both the index/original language version and in the translated version. Placeholders are words wrapped in curly brackets: `{placeholder}`. Note that these are not variables.

Here is an example definition:

```
# protected/messages/es_co/app.php
<?php
return array(
    // Other stuff.
    'The username {username} is not available.' => 'El
nombre de usuario {username} no está disponible.',
    // More other stuff.
);
```

Again, note that the placeholder appears in both the index (the application language version) and the translation.

To use placeholders in a translated string, provide it as a third argument to the `t()` method:

```
echo Yii::t('app', 'The username {username} is not
available.', array('{username}' => $username));
```

Now the translated text will dynamically insert the value of the `$username` variable into the target language's version of the string (**Figure 15.3**).

**El nombre de usuario ACTUAL USERNAME no está disponible.**

**Figure 15.3:** *The localized version of the message also displays the value of the `username` variable.*

Thanks to Twitter (well, helpful people on Twitter), here are localized versions of that same message in other languages if you want to practice with this:

- Il nome utente {username} non è disponibile. (it\_it)
- De gebruikersnaam {username} is niet beschikbaar. (nl\_nl)
- Nazwa użytkownika {username} jest już zajęta. (pl\_pl)
- Der Benutzername {username} ist nicht verfügbar. (de\_de)
- O usuário {username} não está disponível. (pt\_br)
- O utilizador {username} não está disponível. (pt\_pt)
- Lietotājs {username} nav pieejams. (lv\_lv)

{TIP} You can use multiple placeholders in a string, too.

## Handling Plural Forms

Some strings will refer to a noun that may be singular or plural, such as:

- The item has been added to your cart.
- The items have been added to your cart.

You *could* create separate translation strings for each possibility, but Yii is prepared for this scenario and lets you create more flexible translations. This concept is called either *plural forms* or *choice format*.

To use choice format, the message should be defined as more than one *expression#message* combination, each separated by the pipe character (|). If an expression is true, the corresponding message is used. Within the expression, the letter “n” is used to identify the number involved (this is not a variable \$n). Through the third argument to the t() method you can pass the associated number.

This probably sounds more confusing than it is, so let’s look at an example. Here is how you would have Yii “translate” the above message based upon the number of items added to the cart:

```
echo 'The ' .
Yii::t('app', 'n==1#item has|n>1#items have', $num) .
' been added to your cart.';
```

Here’s another example, treating “n” like a string:

```
echo Yii::t('app', 'n=="female"#her|n=="male"#his',
$gender);
```

Understand that this works even when both the application and target languages are the same. If the two languages are different, and if the target language has defined that same message, then the translated version will be used. This is a better, albeit wordier, version:

```
echo 'The ' .
Yii::t('app', 'n==1#The item has been added to your cart.|'
n>1#The items have been added to your cart.', $num) .
```

And if you have this:

```
# protected/messages/es_mx/app.php
<?php
return array(
    'n==1#The item has been added to your cart.|'
n>1#The items have been added to your cart.' => 'El artículo
ha sido añadido al carrito de compras.|Los artículos han
sido añadidos al carrito de compras.',
);
```

Then the message will be both translated into another language and reflect the singular or plural format.

Because plural forms are a common issue, there is an abbreviated syntax for situations where you need to differentiate just between one and more than one:

```
echo 'The ' .  
Yii::t('app', 'The item has been added to your cart.|  
The items have been added to your cart.', $num) .
```

If you want to use the actual number in your message, use the placeholder `{n}`:

```
echo  
Yii::t('app', 'The item has been added to your cart.|  
The {n} items have been added to your cart.', $num);
```

You can add in other placeholders, too, so long as the “n” value is passed first:

```
echo  
Yii::t('app', 'The {product} has been added to your cart.|  
The {n} {product} have been added to your cart.',  
array($num, '{product}' => $product));
```

To be clear, the actual rules for plural forms comes from the CLDR database, and vary from one language to the next. Russian, for example, has more complex plural form rules than simply the singular or plural distinction that exists in English.

## Other Concepts

There's a fair amount of complexity to internationalization, just within the language area alone, but I do want to mention a couple more items that you may want to investigate further.

### Translating Files

In the same way that Yii can create different versions of a string in different languages, it can also render different language-specific view files, too. To use this, create locale-specific directories within each of your **views/ControllerID** directories. Then render the view file as you normally would.

For example, say the “user” controller’s “create” view file should be localized to Spanish. To do that, make the **create.php** file, using all the Spanish you want. Store this file in **protected/views/user/es\_es**. Then your controller simply has to do:

```
# protected/controllers/UserController.php
$this->render('create');
```

If the target (user's) locale is "es\_es", the Spanish version of the view file will be rendered instead of the default one.

If no translation file is available for the view, the default, untranslated version will be used instead.

## i18n Extensions

If you use translations a lot, or extensively on one site, it may be worth your while to investigate the available [i18n](#) extensions. A particularly good one is [yii-language-behavior](#). This is a relatively new extension, but is actively maintained so far.

That extension uses a database table as the message source, which is easier to work with for larger sites and should perform better than using text files. With the database defined, the extension is enabled as a behavior. For an in-depth discussion of this extension, see [this post](#) by [Matti Ressler](#).

## Formatting Dates

Next up, you'll want to know how to format dates and times in a manner that's consistent with the user's locale. Formatting of dates in times in a locale-sensitive way is done via the `CDateFormatter` class. An instance of `CDateFormatter`, specific to the target (aka, user) locale is available through the application's `dateFormatter` property:

```
$df = Yii::app()->dateFormatter;
```

Once you have that instance, you'll call one of two of its methods:

- `format()`
- `formatDateTime()`

The latter is easiest to use. Its three arguments are: the date/time, the "width" of the date to be displayed, and the "width" of the time to be displayed. The "width" refers to how verbose the output is, with the options being: "full", "long", "medium", and "short". The default value is "medium". You can also provide null as a value, to not have that part of the date and time returned. The date/time can either be a Unix timestamp or a string formatted in a way that the PHP `strtotime()` function understands.

Here are some samples, with the resulting output shown in [Figure 15.4](#):

```
<?php
$time = time();
$df = Yii::app()->dateFormatter;
echo '<p>English (long): ' .
    $df->formatDateTime($time, 'long', 'long') . '</p>';
echo '<p>English (medium): ' .
    $df->formatDateTime($time, 'medium', 'medium') . '</p>';
echo '<p>English (short): ' .
    $df->formatDateTime($time, 'short', 'short') . '</p>';
echo '<p>English (short, date only): ' .
    $df->formatDateTime($time, 'short', null) . '</p>';

// Change the target language:
Yii::app()->language = 'es_es';

// Get the formatter anew:
$df = Yii::app()->dateFormatter;

// Show it in Spanish:
echo '<p>Spanish (long): ' .
    $df->formatDateTime($time, 'long', 'long') . '</p>';
echo '<p>Spanish (short, date only): ' .
    $df->formatDateTime($time, 'short', null) . '</p>';
```

English (long): June 4, 2013 11:37:47 AM EDT

English (medium): Jun 4, 2013 11:37:47 AM

English (short): 6/4/13 11:37 AM

English (short, date only): 6/4/13

Spanish (long): 4 de junio de 2013 11:37:47 EDT

Spanish (short, date only): 04/06/13

**Figure 15.4:** The date and time formatted in different ways, including Spanish.

{WARNING} You must fetch an instance of the date formatter after setting or changing the target language.

The `format()` method takes a pattern as its first argument and a date/time as its second. Again, the date/time can be either a Unix timestamp or a string formatted in a manner that PHP's `strtotime()` function can understand. For the pattern, you'll normally use combinations of "y", "M", "d", "h", and "m" (note all the capitalizations). These come from [Unicode's date format patterns](#). Some examples (using 1:15PM on April 1st, 2013):

- *YYYY-MM-dd* would be 2013-04-01
- *MM/dd/YY HH:mm* would be 04/01/13 13:15
- *MMM d, YYYY* would be Apr 1, 2013
- *MMMM d, YYYY h:mm a* would be April 1, 2013 1:15 PM

The great feature is that although you specify the format to be used, the `CDateFormatter` class will still output the result in a way that's most appropriate to the target locale.

## Representing Time Zones

Another way you might want to customize the look and behavior for the user is to take into account the user's time zone. This is especially true on sites where time-sensitive events occur. For example, if your site sells tickets to an event, and the tickets do not go on sale until 10am on a certain day, whose 10am is that?

Yii itself does not offer a solution for this problem, but some logic and good use of your database application can do the trick. This is something I've explained in detail in my "[PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide](#)" book. Unfortunately, because I've already published that material, I cannot provide too many details here (for legal reasons). But in case you don't have that book, the ideas are these:

1. Enable time zone support in MySQL (assuming you're using MySQL)
2. Store all dates and times using UTC (e.g., the MySQL `UTC_TIMESTAMP()` function returns the current timestamp in UTC)
3. Determine the user's time zone, either automatically (e.g., using the IP geolocation) or by asking for it
4. Store the user's time zone in the user state
5. Use the MySQL `CONVERT_TZ()` function to convert dates and times from UTC to a specific time zone

That's the premise, which is not too difficult to implement. From my experience with readers of my PHP and MySQL book, the most common problem is not having time zone support enabled in MySQL (see the MySQL manual for an explanation of how one does that).

Of course when using Yii, there's an added complication. In order to use `CONVERT_TZ()`, you must have some control over the `SELECT` query being executed. If you're using Query Builder or Direct Access Objects (see Chapter 8, “[Working with Databases](#)”), that's not a problem. For models using Active Record, I'd recommend creating a scope (again, see Chapter 8) that uses `CONVERT_TZ()` as part of the selection.

It's generally best to push as much work onto the database application as you can, as in the above sequence. But if you have dates and/or times that aren't in the database that must be converted to the user's time zone, familiarize yourself with the PHP `DateTime` class. It was added in PHP 5.2, is easy to use, and is very powerful.

## Formatting Numbers

To format numbers and currency in Yii, use the `CNumberFormatter` class. An instance of it, set to the target locale, is available through `Yii::app()->numberFormatter`:

```
$nf = Yii::app()->numberFormatter;
```

Once you have that class instance, there are four methods you'll use:

- `format()`
- `formatDecimal()`
- `formatCurrency()`
- `formatPercentage()`

To output a formatted percentage, you would just provide the method with the value to be formatted:

```
echo $nf->formatPercentage($value);
```

To output a formatted currency, provide that method with the value to be formatted and the currency code:

```
echo $nf->formatCurrency($value, 'USD');
```

The currency code is a three-letter code defined by [ISO 4217](#).

The `formatDecimal()` method just takes the value to be formatted:

```
echo $nf->formatDecimal($value);
```

The fourth method, `format()`, is the most flexible. It takes up to three arguments: a formatting pattern, the value to format, and an optional currency indicator. This last argument would allow you to use `format()` to format currencies directly. This may seem redundant, but behind-the-scenes, `format()` is actually used by the other three methods, but with pre-defined patterns.

Patterns are created using meaningful characters (**Table 15.1**).

Character	Meaning
.	decimal point
,	group
0	required digit (padding)
#	optional digit
¤	currency
%	percentage
%00	permillage
:	negatives/positives

“Meaning” in that table references what locale-specific character will be used in place of the character listed in the first column.

Here are some samples, with the resulting output shown in **Figure 15.5**:

```
<?php
// Set the number:
$number = 23049.59;

// Start in English (US):
Yii::app()->language = 'en_us';
$nf = Yii::app()->numberFormatter;
echo '<p>English (decimal): ' . $nf->formatDecimal($number)
. '</p>';
echo '<p>English (currency): ' . $nf->formatCurrency($number, 'USD')
. '</p>';

// Change to Spanish:
Yii::app()->language = 'es_es';
$nf = Yii::app()->numberFormatter;
echo '<p>Spanish (decimal): ' . $nf->formatDecimal($number)
. '</p>';
echo '<p>Spanish (currency): ' . $nf->formatCurrency($number, 'EUR')
. '</p>';
```

English (decimal): 23,049.59  
English (currency): \$23,049.59  
Spanish (decimal): 23.049,59  
Spanish (currency): 23.049,59 €

**Figure 15.5:** Numbers and currencies formatted in different ways, including Spanish standards.

## i18n and Your Models

So much of a site depends upon its models, so let's conclude this chapter by revisiting the topic of models with respect to internationalization. For example, if your site has an international audience and a “user” model, it would make sense for model references to be localized. In other words, have the words “username”, “password”, and so forth be displayed in the user's preferred language.

This is easily accomplished by changing the values returned by the model's `attributeLabels()` method. Instead of having it return hard-coded strings in one language, have it return localized strings via the `t()` method:

```
# protected/models/AnyModel.php
public function attributeLabels() {
    return array(
        'username' => Yii::t('user','Username'),
        'email' => Yii::t('user','Email Address'),
        'password' => Yii::t('user','Password')
    );
}
```

For each language the site would support, you would then create a `protected/messages/LocaleId/user.php` file that returns an array of those values with the proper translations. (Or use a database that does the same.)

Similarly, you may want to automatically display dates formatted for the user's locale. I earlier provided some tips on converting a date and time to a user's time zone. That would be done within the database. In models based upon Active Record, you can use a scope to modify the SELECT query to convert the retrieved date and time.

Taking this a step further, not only might you want to convert dates and times to the user's time zone, you might also want to display them formatted to the user's

locale. One way of doing so is to pass model values to the date formatter object. Assuming that the `$model` is an instance of the `User` class and has a `date_entered` attribute, you could display the date that the user registered, formatted to the user's locale:

```
<?php
$df = Yii::app()->dateFormatter;
echo '<p>Member since: ' .
    $df->formatDateTime($model->date_entered, 'medium', null) . '</p>';
```

(Obviously you'd want to translate the “Member since” part, too.)

An alternative approach would be to embed the date and time formatting within the model itself:

```
# protected/models/User.php
public function getDateRegistered() {
    return Yii::app()->dateFormatter->formatDateTime(
        $this->date_entered, 'medium', null);
}
```

Now a view file can use `$model->getDateRegistered()` to get the `date_entered` value formatted to the user's locale. The primary downside to this approach is that the formatting style—long, medium, or short—is embedded into the method.

To explain a slightly more advanced option, you can also use `$model->dateRegistered` in your view files to get that same value. In other words, `$model->dateRegistered` and `$model->getDateRegistered()` are the same. I'll explain this in more detail in Chapter 19, “Extending Yii,” but the simple reason is that models extend the `CComponent` class, and one of the features that `CComponent` creates is the ability to get and set attributes. This was mentioned in Chapter 5, “[Working with Models](#).”

# Chapter 16

## LEAVING THE BROWSER

Somewhat ironically, not everything that pertains to a Web site makes use of a Web browser. Although what is normally considered to be a “Web site” is the combination of HTML, JavaScript, CSS, and media that the user sees, today’s sites often other resources that are never intended to be accessed by a browser.

In this chapter, I’ll explain how to use the Yii framework in ways that *don’t* make use of the browser. There are three specific topics to be covered:

- Proxy scripts
- Web services
- Console applications

Now, in truth, the first two of these *can* be accessed via the Web browser in that they’ll be available over HTTP. But the distinction I’m making is that all three uses are not intended to be directly accessed by end users with their browsers. Put another way, this chapter explains situations in which you’d use the Yii framework but generally *not* use the primary layout file to render a complete HTML page.

### Writing a Proxy Script

*Proxy scripts*, in case you’re not familiar with them, are agents in a Web site, standing in for other resources. Proxy scripts are commonly used to:

- Provide access to restricted resources
- Hide direct access to resources
- Track usage of materials, such as the number of times a file has been downloaded

For example, if you have files uploaded outside of the Web directory, a proxy script would be required to provide those files to the browser. Or if only logged-in users could view a PDF, a proxy script could enforce that restriction.

{TIP} In the Web site I created for [selling this book](#), a proxy script prevents un-paid users from downloading the book. The same proxy script also tracks the number of downloads.

The most important distinction between a proxy script and a standard PHP page is that proxy scripts generally don't output *any* HTML. When it comes to using the Yii framework, this means that your controllers will use `renderPartial()` instead of `render()`, so as to omit the layout file:

```
# protected/controllers/SomeController.php
public function someAction() {
    // Do whatever.
    $this->renderPartial('thing');
}
```

Of course, what, exactly, the `thing.php` view file does depends entirely upon the specific situation. I'll return to this shortly.

Proxy scripts run through the Yii framework can easily take advantage of the same access control that any other page can. If part of a proxy script's role is to limit access, just apply the information covered in Chapter 11, "[User Authentication and Authorization](#)."

In certain situations, it's useful to hide direct access to a resource. Maybe a page or file is directly available, but you'd rather not make the URL obvious in the browser. If it's a standard HTML page, you can use `render()` but render a page from another controller or view:

```
# protected/controllers/SomeController.php
public function someAction() {
    // Do whatever.
    $this->render('//secret/thing');
}
```

Other times, the resource being "hidden" is a file. In standard PHP, you'd use several `header()` function calls, plus `readfile()` to send the file to the browser. Yii has simplified that process with the `sendFile()` method of the `CHttpRequest` class. An instance of that class is available through the application's `getRequest()` object:

```
Yii::app()->getRequest()->sendFile($filename, $content);
```

The first argument needs to be the path to the file on the server. The second is the file's content. Provided with these two pieces of information, this method will automatically:

- Determine the proper MIME type
- Send the proper headers
- Send the content itself (i.e., the file)
- Terminate the application

The simplest way to use this method is like so:

```
# protected/controllers/SomeController.php
public function someAction() {
    // Do whatever.
    $file = '/path/to/file.ext';
    $content = file_get_contents($file);
    Yii::app()->getRequest()->sendFile($file, $content);
}
```

This method does a lot of the work for you, but has its downsides, too. For starters, the provided file name is directly given to the browser. If the file name also includes a path to the file's directory, sending all of that to the browser is illogical at best, and a security concern at worst.

A second problem is that arguably one should rename uploaded files for security purposes. They might be renamed as a simple integer or a hash. In such situations, you'd want the browser to be provided with the original file name (so that the original file name would be the default file name when the user downloads the file).

The solution is to provide the *actual* file name to the `sendFile()` method as the first argument, the file's content as the second, and the file's MIME type as the third. For a hypothetical example, let's assume that the files are represented by the `File` class. The controller method will load an instance of that in order to get all of the file's information. Then the method will call `sendFile()`, passing along three arguments:

```
# protected/controllers/FileController.php
public function downloadFile($id) {
    $file = File::model->findByPK($id);
    if ($file === null) {
        throw new Exception('The file could not be found.');
}
```

```
$content = file_get_contents($file->name);
Yii::app->getRequest->sendFile($file->file_name,
    $content, $file->type);
}
```

Passing along the MIME type is necessary when providing a file name that doesn't directly map to a file on the server. This is because the `sendFile()` method ordinarily uses the file name to reference the file in order to get the file's MIME type.

Another possible issue with the `sendFile()` method is that it always sends files using the "attachment" Content-Disposition header. The actual impact will depend upon the browser and the file, but this normally results in the file being downloaded. When serving images, that may not be the desired intent (i.e., you'd want to use the "inline" Content-Disposition). In that case, you'd be best off foregoing `sendFile()` entirely and just manually setting all the headers and doing all the work.

## Creating Web Services

A "Web service", in case you're unfamiliar with the term, is a defined method of communication that occurs over the Web. A Web service can be so simple that it merely returns the current time, or so complex that it expects multiple parameters (e.g., a list of stock abbreviations, a format designator, and so forth) and returns an array of objects in a class understood by both parties.

In terms of technologies involved, Web services encompasses a wide variety, and quite a lot of acronyms. In whatever format, though, tapping into Web services makes for very powerful sites.

Web services can be implemented in many ways. In this chapter, I'll explain two types: SOAP (Simple Object Access Protocol) and RESTful (Representational State Transfer). You'll see what code you should use to create both types of services using Yii.

Once defined, Web services can also be used in two generic ways:

- Directly in the browser, via Ajax
- Directly from the server, via cURL or the like

For example, you might create a service on your site that's intended to be invoked via Ajax from a page within your own site. Chapter 14, "[JavaScript and jQuery](#)," already demonstrated that. Or, within a controller on your site, you might invoke a service on another site in order to obtain information to be passed to the view file (e.g., a stock quote).

## SOAP Services

The older, more formal standard for Web services is Simple Object Access Protocol (SOAP). SOAP is relatively complex (especially when compared with REST), but has a few advantages:

- SOAP can be used over any protocol
- SOAP services can be discoverable (when used with Web Services Design Language [WSDL])
- SOAP can transfer meaningful data, such as objects of a known class type

These advantages come at a price, of course. SOAP is more complicated to implement than some alternatives, requires a lot more back and forth between the client and the server, and may also require special libraries and extensions to be installed on both ends of the transaction.

*{NOTE}* Technically speaking, SOAP no longer stands for Simple Object Access Protocol, but it annoys me when people try to un-acronym an obvious acronym.

You can create Web services in Yii manually, which is to say write all the code yourself, but the Yii framework nicely provides a couple of useful classes for you.

For a working example of this, let's take the CMS project I've been discussing throughout the book and create a SOAP service that returns a page's information when provided with a page ID.

Before continuing, understand that using Yii to create a SOAP service requires PHP's [SOAP extension](#). You should probably run a `phpinfo()` script to confirm this extension is enabled prior to continuing.

*{WARNING}* Support for SOAP will be dropped in Yii2, as it's becoming less commonly used.

### Creating a Service Provider

To start, you must first create a "service provider". A service provider is a declared group of Application Programming Interfaces (APIs), defined within a class. Normally the class will extend `CController` (or `Controller`), although that's not required. You can create new classes to act as your services, or use your existing classes. For example:

```
<?php
# protected/controllers/PageController.php
class PageController extends Controller {
    // Other attributes and methods.
    // Add service!
}
```

Or, since the CMS example already has a `PageController`, I might do this:

```
<?php
# protected/controllers/PageServiceController.php
class PageServiceController extends Controller {
    // Add services!
}
```

To create the service within that class, you create a method. Unlike most of the controller methods used in this book and in Yii, this method:

- Must be prefaced with a specific *docblock*
- Won't use "action" in its name

For the "page" fetching service, it would take an integer as a parameter—the ID of the page to be fetched—and return a `Page` object. This information must be indicated in a docblock before the method. Here's an example, which I'll then explain in a bit more detail:

```
<?php
# protected/controllers/PageServiceController.php
/**
 * @param int the ID of the page to fetch
 * @return Page a Page class instance
 * @soap
 */
public function getPage($id) {
    // Do the work!
}
```

The docblock standard is a strictly defined way of documenting one's code, and is useful in many ways. Even when not working with SOAP, using docblock to formally document one's work is the hallmark of a true professional. But when using the Yii framework, the docblock before the method is used to identify services. Specifically, you should include:

- The parameters (type and purpose)

- What the method returns (type)
- The keyword `@soap`

The available types are:

- `str` or `string`
- `int` or `integer`
- `float` or `double`
- `bool` or `boolean`
- `date`
- `time`
- `datetime`
- `array`
- `object`
- `mixed`

You can also reflect specific object types by naming the object's class, as in the above code (e.g., `Page`).

*{TIP}* To indicate that a service returns an array of an object type, use `ClassName[]`.

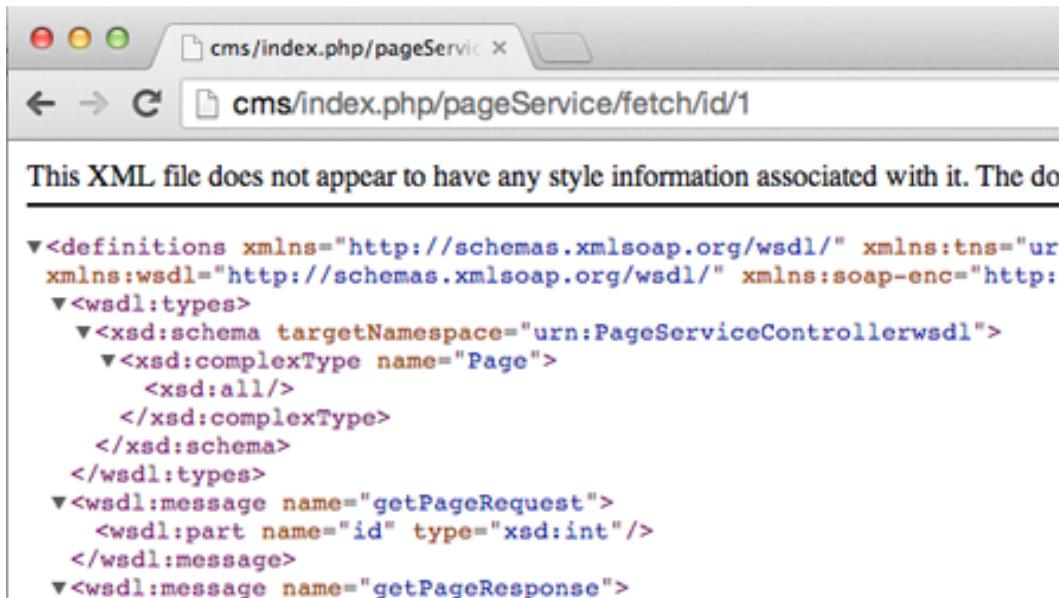
Having defined the method, you must make the controller aware of it as a Web service. That's done in the `actions()` method:

```
<?php
# protected/controllers/PageServiceController.php
class PageServiceController extends Controller {
    public function actions() {
        return array(
            // Other actions.
            'fetch'=>array(
                'class'=>'CWebServiceAction',
            ),
        );
    }
}
```

The `actions()` method returns an array. The indexes of the array should be unique identifiers. The values are the classes associated with that action. Here, I'm naming the service "fetch", which is of type `CWebServiceAction`.

There are two key Yii classes for creating SOAP services: `CWebService` and `CWebServiceAction`. Your services within the controller will always use the `CWebServiceAction` class.

By taking these steps, Yii will automatically create the WSDL specification for the service. A WSDL specification indicates to clients what options are available and how they are to be used. If you were to view `index.php/page/fetch` in your browser, you'd see the generated WSDL, which is in XML format (**Figure 16.1**).



The screenshot shows a web browser window with the URL `cms/index.php/pageService/fetch/id/1`. The page content displays the generated WSDL XML. The XML starts with a `<definitions` tag and defines a schema for a 'Page' complex type, which contains an 'id' integer. It also defines two messages: 'getPageRequest' and 'getPageResponse'.

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:PageServiceControllerwsdl" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/">
  <wsdl:types>
    <xsd:schema targetNamespace="urn:PageServiceControllerwsdl">
      <xsd:complexType name="Page">
        <xsd:all/>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="getPageRequest">
    <wsdl:part name="id" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="getPageResponse">
```

**Figure 16.1:** The WSDL information created by Yii, thanks to SOAP.

## Using SOAP Services

SOAP services are used via a SOAP client. A SOAP client is also part of PHP's [SOAP extension](#). To use it, first create a new `SoapClient` object, passing it the URL of the resource:

```
# protected/controllers/SomeController.php::someAction()
$url = 'http://www.example.com/index.php/page/fetch';
$client = new SoapClient($url);
```

Once the object has been created, you can invoke any of the services as if it's a method of that object, passing along parameters in the process:

```
# protected/controllers/SomeController.php::someAction()
$url = 'http://www.example.com/index.php/page/fetch';
$client = new SoapClient($url);
$page = $client->getPage(1);
```

A simple `var_dump()` of the `$page` variable shows what the PHP script received (**Figure 16.2**).

# Data Received

```
object(stdClass)[20]
```

**Figure 16.2:** PHP received a generic object from the service call.

If you wanted to have multiple tasks as part of a service provider, just define other methods in your class and then call the appropriate one through your `SoapClient` instance.

Note that for this particular example, the assumption is that the service is part of one site and the client is part of another.

## Clearing the WSDL Cache

When you create a SOAP service in Yii, the framework automatically creates the WSDL (see Figure 16.1). This is the specification for how the service is to be used.

When you create a SOAP client that has to connect to that service, it first reads in the WSDL to know what options exist, what parameters the services take, and what they return. Because that's an extra, expensive network call, PHP will cache the WSDL result by default. That's great on a production site, but a real pain when you're developing. If you were to run the previous example once and then add a method (i.e., service), PHP would throw an error when you attempted to call that new method (**Figure 16.3**).

## SoapFault

```
Function ("getPages") is not a valid method for this service
```

**Figure 16.3:** This error suggests that the method does not exist, but it does.

To prevent this from happening when developing a service, tell PHP *not* to cache WSDL results. That's done by changing PHP's configuration using the `ini_set()` function:

```
ini_set('soap.wsdl_cache_enable', 0);
ini_set('soap.wsdl_cache_ttl', 0);
$client = new SoapClient($url);
```

Now the WSDL will be re-examined with every request.

{WARNING} Make sure you remove those two lines before going into production! To see what functions are available to be called in a service, use this code:

```
$functions = $client->__getFunctions();
var_dump ($functions);
```

## Mapping Classes

I previously said that one of the benefits of SOAP is the ability to transmit more meaningful information, such as objects in a specific class type. With the previous bit of code, the `$page` variable in the client will not be a usable object of type `Page` as the client has no knowledge of the `Page` class definition (see Figure 16.2). The solution is to add *class mapping*: additional service documentation.

Yii will do this for you, once you tell it to. To start, indicate the mapping when configuring the action:

```
# protected/controllers/PageController.php
public function actions() {
    return array(
        // Other actions.
        'fetch'=>array(
            'class'=>'CWebServiceAction',
            'classMap' => array(
                'Page' => 'Page'
            )
        ),
    );
}
```

That line indicates that the `Page` object returned by the service should be mapped to the internal `Page` class.

Next, make Yii aware of the `Page` classes' methods and attributes for the purpose of the service. This is done by adding the proper docblock to the class:

```
# protected/models/Page.php
class Page extends CActiveRecord {
    /**
     * @var integer page ID
     * @soap
     */
    public $id;
    /**
     * @var string page title
     * @soap
     */
    public $title;
    // Etc.
}
```

Understand that you'll want to do this for every attribute or method that you want to make known in the SOAP service.

Once this is done, the client can reference `$page->id`, `$page->title`, etc. (**Figure 16.4**).

## Data Received

```
object(stdClass)[20]
public 'id' => int 1
public 'title' => string 'Aliquam malesuada, ligula sit amet.' (length=35)
```

**Figure 16.4:** The returned object now has two properties.

## RESTful Services

SOAP services are great when meaningful data needs to be transmitted or when transmissions should occur over non-HTTP protocols. However, although it may not be clear thanks to the work done by Yii, SOAP requires considerable amount of extra effort, by the programmer, by the server, and by the client. Further, SOAP is not very efficient when it comes to the amount of data that must be transmitted or the number of server calls that must be made by the client.

An alternative to SOAP is REST (Representational State Transfer). REST provides an easy, lightweight way to implement Web services. A REST service is used very much like a Web browser requests a page from a server. Sometimes additional data will be passed as part of the request, other times not. The server's response will normally be plain text, JSON, or XML. Some of the Ajax examples from Chapter 14 could be defined as a type of RESTful service.

Here, let's formally implement the "page" service in a RESTful manner.

## Creating the API

As with SOAP services, RESTful services are defined as part of an API. Unlike with SOAP, the RESTful API is not formally declared, but grouped together. To do so in Yii, it makes sense to create an "API" controller, through which all RESTful requests can be processed:

```
# protected/controllers/ApiController.php
class ApiController.php extends Controller {
}
```

Next, create an "action" method for each service you want to provide:

```
# protected/controllers/ApiController.php
class ApiController.php extends Controller {
    public function actionFetchPage($id) {
        // Do the work.
    }
}
```

Now you've created a RESTful service! This service is available like any other controller action, which you can test by going to this URL in your browser:

`http://www.example.com/index.php/api/fetchPage/id/x`

Of course, as written, that page doesn't actually do anything.

Most commonly, RESTful services output either JSON or XML. Let's look at how both are done.

## Outputting JSON

JSON is perhaps the most commonly used format for data returned by RESTful services. Like REST itself, JSON is lightweight and easy to use. It's also quite easy to create. To start, you should send a header that indicates that JSON content is about to follow:

```
header('Content-Type: application/json');
```

Then, once you have the data ready to be transmitted, run it through the Yii `CJSON::encode()` method, printing the output in the process. Here's the complete example:

```
<?php
# protected/controllers/ApiController.php
class ApiController extends Controller {
    public function actionFetchPage($id) {
        $page = Page::model()->findByPK($id);
        header('Content-Type: application/json');
        echo CJSON::encode($page);
    }
}
```

You can see the result in **Figure 16.5**.



**Figure 16.5:** The representation of a Page object in JSON format.

If the service returns an array of information, that's easily done, too:

```
# protected/controllers/ApiController.php
class ApiController extends Controller {
    public function actionFetchPages() {
        $pages = Page::model()->findAll();
        header('Content-Type: application/json');
        echo CJSON::encode($pages);
    }
}
```

Only one thing is missing in this process: error reporting. I'll get to that after I discuss outputting XML.

## Sending XML

Sometimes you might want your service to return XML instead of JSON. The downside to XML is that it's fairly particular in its syntax and adds extra size to the response because of all the additional markup. (JSON is particular, too, but the `CJSON::encode()` method handles that for you.)

Because of all the additional work required by an XML response, I like to use a different approach in my XML Yii services. To start, I would create an XML layout that can be a container for any XML response. The layout file would send the proper header, indicate the XML version, and create the root XML tag:

```
# protected/layouts/xml.php
<?php header("Content-type: text/xml"); ?>
<?php echo '<?xml version="1.0" encoding="UTF-8"?>' ?>
<response>
    <?php echo $content; ?>
</response>
```

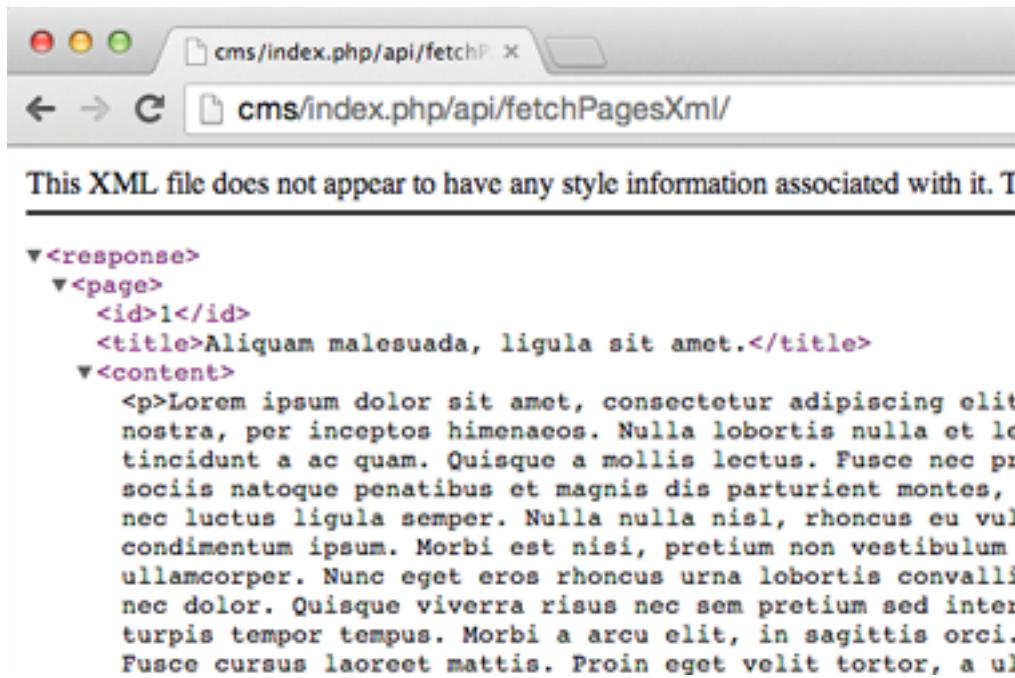
With that layout file, the individual view file is responsible for creating the `$content` value, which is all the other tags and values. Here's what the controller might do:

```
# protected/controllers/ApiController.php
class ApiController extends Controller {
    public function actionFetchPagesXml() {
        $this->layout = 'xml';
        $pages = Page::model()->findAll();
        $this->render('pages_xml', array('pages' => $pages));
    }
}
```

The controller method first changes the layout used, then passed the data to the rendered view file. Here's how that might look:

```
# protected/views/api/pages_xml.php
<?php foreach ($pages as $page) : ?>
    <page>
        <id><?php echo CHtml::encode($page->id); ?></id>
        <title><?php echo CHtml::encode($page->title); ?></title>
        <content><?php echo CHtml::encode($page->content); ?></content>
    </page>
<?php endforeach; ?>
```

That's all there is to it. You can see the resulting XML output in (**Figure 16.6**).

A screenshot of a web browser window. The title bar says "cms/index.php/api/fetchPagesXml". The address bar also shows "cms/index.php/api/fetchPagesXml/". The main content area displays an XML document. At the top, it says "This XML file does not appear to have any style information associated with it. To see the XML code, please change your browser's "Content-Type" setting or choose "View" > "View XML Source" in Internet Explorer." Below this, the XML code is shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <page>
    <id>1</id>
    <title>Aliquam malesuada, ligula sit amet.</title>
    <content>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit
      nostra, per inceptos himenaeos. Nulla lobortis nulla et le
      tincidunt a ac quam. Quisque a mollis lectus. Fusce nec pr
      sociis natoque penatibus et magnis dis parturient montes,
      nec luctus ligula semper. Nulla nulla nisl, rhoncus eu vul
      condimentum ipsum. Morbi est nisi, pretium non vestibulum
      ullamcorper. Nunc eget eros rhoncus urna lobortis convallis
      nec dolor. Quisque viverra risus nec sem pretium sed inter
      turpis tempor tempus. Morbi a arcu elit, in sagittis orci.
      Fusce cursus laoreet mattis. Proin eget velit tortor, a ul
```

Figure 16.6: The XML output from the service.

## Reporting Errors

To make this service more polished, it ought to include an approach for indicating problems. As written, the service will return an individual page record when provided with a corresponding page ID, but what if the provided page ID does not correlate to an existing record? The service could, in that case, return an empty data set, but that approach won't differentiate among:

- The page ID not matching a valid record
- An invalid request being made of the service
- The service generally not working
- And other possibilities

Smarter people than I have already recognized this need and come up with a clever solution. The solution is so smart that it uses a communication device that already exists: since the RESTful request is being made over HTTP, HTTP status codes are perfect for indicating the success of the request.

Some of the most common status codes, and their meaning, are listed in the following table.

Code	Meaning
200	OK

Code	Meaning
201	Created
204	No Content
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
418	I'm a teapot
500	Internal Service Error

These are logically and directly mappable to the service responses, except perhaps for the “I’m a teapot” status, which is actually assigned to 418 (you can look it up).

The next issue is how you indicate the proper response. From the perspective of the Web server, the RESTful services are equivalent to HTML pages. The Web server will automatically return 2xx and 3xx codes for you. To send the status codes yourself, you’ll need to use the PHP `header()` function:

```
<?php header('HTTP/1.1 400 Bad Request'); ?>
```

If you’re using PHP 5.4 or later, this can be simplified by invoking the new `http_response_code()` function. It takes the code as its lone argument:

```
<?php http_response_code(400); ?>
```

Provided with the HTTP status code, `http_response_code()` will output the correct header, including both the code and the corresponding message.

For example, the `actionFetchPage()` method might be defined like so:

```
# protected/controllers/ApiController.php
class ApiController extends Controller {
    public function actionFetchPage($id) {
        $page = Page::model()->findByPK($id);
        if ($page === null) {
            http_response_code(404); // PHP 5.4!
            $data['error'] = 'No such page record exists.';
            echo CJSON::encode($data);
        } else {
            header('Content-Type: application/json');
            echo CJSON::encode($page);
        }
    }
}
```

If you wanted to make this more flexible and modular, you could create a separate `ApiController` method that sets the status code, as explained in [this excellent Yii wiki article](#). Doing so would allow you to, for example, return a 201 Created code when a RESTful service creates a new resource (e.g., a new record). That same wiki article also explains how you can map different request types—GET, POST, PUT, DELETE—to different services. It's a bit more advanced, but very smart.

## Web Service Extensions

If you'd rather not go through all the effort to create Web services yourself, there are many [Yii extensions](#) that can step in. Some of these extensions simplify the process of contacting to Web services on other sites (e.g., Google, Facebook, Twitter, or Amazon). Other extensions assist in creating services as part of your own site. One such extension is [restfullyii](#). This simple extensions creates a complete RESTfull API for basic CRUD functionality.

A more advanced, but still young, extension is [Backvendor](#) by MobiDev. This is a fairly complete tool for generating services, including functional tests, API versioning, automatic documentation, and more.

## Accessing Web Services

The one possible remaining issue is: how do you access Web services? This chapter has already included one approach: using PHP's `SoapClient` class:

```
# protected/controllers/SomeController.php::someAction()
$url = 'http://www.example.com/index.php/page/fetch';
$client = new SoapClient($url);
$page = $client->getPage(1);
```

For RESTful services, you'll often access them using Ajax within the browser (see Chapter 14). To access RESTful services in PHP, you have multiple options. Two good ones are cURL and `file_get_contents()`.

The PHP `file_get_contents()` function takes a URL as its argument and returns the read data as a string:

```
# protected/controllers/SomeController.php::someAction()
$url = 'http://www.example.com/index.php/api/fetchPage/id/1';
$data = file_get_contents($url);
$data_json = CJSON::decode($data);
```

As you can see in that code, if the service returns JSON data, the Yii `CJSON::decode()` method can turn it into usable JSON. If the service returns XML, you can use one of PHP's excellent XML parsers.

## Creating a Console Application

The last subject to discuss in this chapter are *console applications*. Console applications, or console scripts, are intended to be run not in a Web browser (i.e., accessed over HTTP) but within a console or command-prompt environment.

For beginning Web developers, the need for, let alone actually using, console applications can often be unclear. Console applications are great in situations where:

- The resulting output will be minimal or immaterial
- The execution will take excessive time (more than even a complex Web page)
- The execution should be run on a schedule

Common uses of console applications include:

- Generation of resources, such as images, PDFs, and even code
- Maintenance of the file system, database, search indexes, etc.
- Other services, like sending emails

When you have tasks like these that need to be performed in conjunction with a Yii-based Web site, it makes sense to create a Yii-based console application.

### Revisiting the `yiic` Script

You've already seen an example of a Yii-based console command: the `yiic` tool uses Yii to generate files and folders from a command-line interface. One of the files that `yiic` creates is an application-specific version of `yiic`, placed in the **protected** folder by default.

The `yiic` script written to the **protected** directory looks like:

```
1 #!/usr/bin/env php
2 <?php
3 require_once(dirname(__FILE__).'/yiic.php');
```

As you can see, all the executable `yiic` does is use the environment's version of PHP (line 1) to require the `yiic.php` script located in the same directory. Here's what `yiic.php` looks like:

```
1 <?php
2 // change the following paths if necessary
3 $yiic=dirname(__FILE__).'../../../../framework-yii-1.1.13/yiic.php';
4 $config=dirname(__FILE__).'/config/console.php';
5 require_once($yiic);
```

From that code, you can see that the framework's **yiic.php** script is identified (line 4). On line 5, the console-specific configuration file, found within the site's **protected/config** directory, is also identified. Then this script includes the framework's **yiic.php** script from the framework's directory.

You can take a look at the framework's **yiic.php** script to see the code, but I'll explain a synopsis of what it does. The framework's **yiic.php** script creates an instance of a **CConsoleApplication** class. This is the console equivalent to **CWebApplication**, created by the bootstrap **index.php** file (see Chapter 3, "[A Manual for Your Yii Site](#)").

To restate this, the **protected/yiic** script creates an instance of a console application, using **protected/config/console.php** to configure how it runs. So this is similar to what the bootstrap **index.php** script does, except for the type of application being created: web or console. Also, because it's in the **protected\*** directory, **the protected/yiic\*\*** script should not be accessible via a Web browser.

{TIP} The **yiic.bat** file, also found in the **protected** directory, is the Windows equivalent of **yiic**.

## Configuring the Application

The structure of **console.php** is the same as the other two configuration files: **main.php** and **test.php**. But due to the narrower scope of console applications, the console-specific configuration file will never need as much customization.

You should start with what's absolutely necessary: the database connection information (assuming the console application will use the database). You can copy the correct values from **main.php** and paste it into the proper location in **console.php**. Other than that, you probably won't do much console configuration unless you depend upon console scripts a lot.

As always, you can find the full list of **CConsoleApplication** properties in the [Yii class docs](#).

## Creating Commands

Having configured your console, the next step is to create the commands to be executed via the command-line interface. A single console application can have one or more commands to be executed.

Each command is defined as its own class. That class must extend the **CConsoleCommand** class. The class name takes the format **CommandNameCommand**:

```
<?php
class PurgeCommand extends CConsoleCommand {
    // Do the work!
}
```

The class must be defined in a file whose name matches the class name (as always), and be stored in the **protected/commands** directory. Hence, the above would be found in **protected/commands/PurgeCommand.php**.

{TIP} The directory where your commands should be stored is dictated by the console application's **commandPath** property. This is changeable in your console application configuration file.

The class can have (or not have) attributes and methods like any other class. Within the class, you can create multiple subcommands by defining a series of methods. For example, the **DatabaseCommand** class might define methods for optimizing tables, backing up the database, etc. The end result is a structure very much the same as the use of actions in a Web-based controller.

To implement this, create one or more methods with the name of "action" followed by the command name: **actionOptimize()**, **actionBackup()**, etc.

```
<?php
# protected/commands/DatabaseCommand.php
class DatabaseCommand extends CConsoleCommand {
    public function actionOptimize($args) {
        echo 'Optimizing...' . PHP_EOL;
        // Do whatever!
    }
    public function actionBackup($args) {
        echo 'Backing up...' . PHP_EOL;
        // Do whatever!
    }
}
```

As you can see in those examples, each method must be defined to take array as its lone argument. Conventionally, this is named **\$args**. You'll see how this variable is used momentarily.

Within the function you can do whatever needs to be done, including making use of the Yii application's defined models. As with a Web-based scripts, the application instance is available through **Yii::app()**, although here that instance is of type **CConsoleApplication**, not **CWebApplication**.

Keep in mind that the console application method would not render any views, as the command is being executed in the command-line environment. You can print

simple messages, if you want, although if you plan on executing the script through a cron, no one would ever see those messages (but they can be useful for debugging purposes while you’re developing the code).

## Executing Commands

Once you’ve created your command, it’s ready to be executed. The exact syntax you’ll use for doing so will differ from one environment to the next, so I’ll explain some common options. The general syntax is:

```
yiic <command-name> <action-name>
```

You’d need to do this within the **protected** directory, where the `yiic` script can be found. With the example “optimize” command, that syntax would be:

```
yiic database optimize
```

{NOTE} Neither the command name nor the action name needs to be quoted.

Like running the framework’s `yiic` script, whether or not this works for you depends upon your operating system, how PHP was installed, your configuration, and so forth. For some setups, you may need to just preface “`yiic`” with “`./`”:

```
./yiic database optimize
```

Those additional characters say to execute the `yiic` script in the current directory. Other setups will require that you directly invoke PHP and reference the PHP script (**Figure 16.7**).

```
php yiic.php database optimize
```

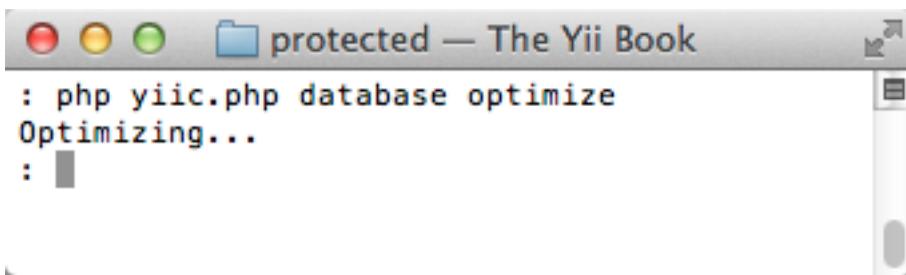
And still other setups will require that you provide the full path to the PHP executable:

```
C:\xampp\php\php.exe yiic.php purge
```

You’ll need to play around with these variations, and pay attention to any error messages you might see, until you find the syntax that works for your situation. For the purposes of the rest of this chapter, I’ll just use `yiic <command-name> <action-name>` to be consistent; change it to suit your proper solution.

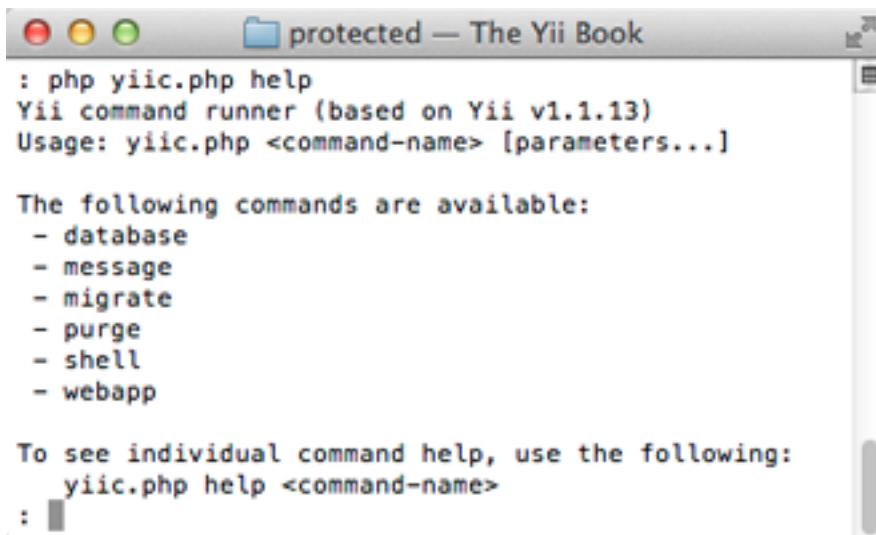
Before moving on, there are two more options you should be familiar with. First, run `yiic help` to list all available commands. This will include both the commands you’ve defined, and those defined by the framework (**Figure 16.8**).

Second, use `yiic help <command-name>` to get more information about using any particular command (**Figure 16.9**).



```
: php yiic.php database optimize
Optimizing...
:
```

Figure 16.7: Running the “optimize” action of the “database” command.

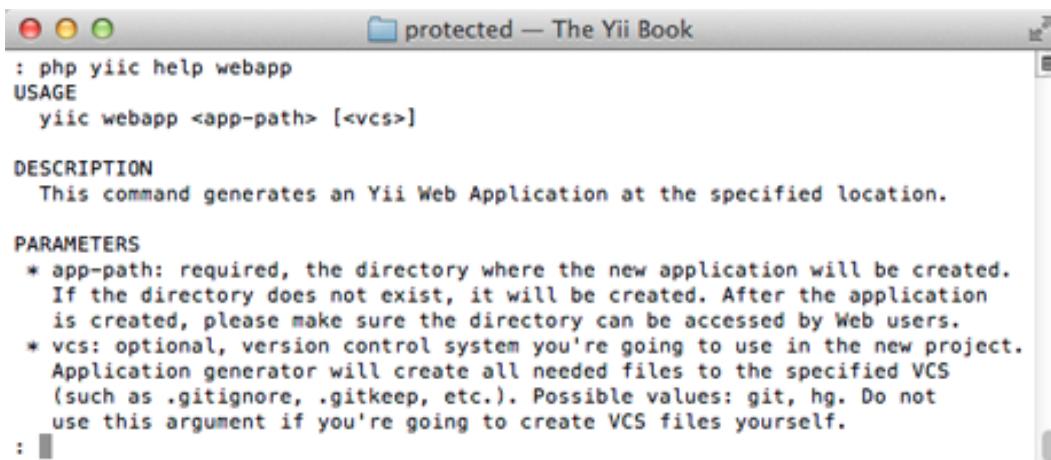


```
: php yiic.php help
Yii command runner (based on Yii v1.1.13)
Usage: yiic.php <command-name> [parameters...]

The following commands are available:
- database
- message
- migrate
- purge
- shell
- webapp

To see individual command help, use the following:
    yiic.php help <command-name>
:
```

Figure 16.8: The list of available commands.



```
: php yiic help webapp
USAGE
    yiic webapp <app-path> [<vcs>]

DESCRIPTION
    This command generates an Yii Web Application at the specified location.

PARAMETERS
    * app-path: required, the directory where the new application will be created.
        If the directory does not exist, it will be created. After the application
        is created, please make sure the directory can be accessed by Web users.
    * vcs: optional, version control system you're going to use in the new project.
        Application generator will create all needed files to the specified VCS
        (such as .gitignore, .gitkeep, etc.). Possible values: git, hg. Do not
        use this argument if you're going to create VCS files yourself.
:
```

Figure 16.9: The help for the “webapp” command.

## Passing Command Arguments

You may commonly write shell scripts that do different things based upon different arguments. For example, maybe the maintenance would be performed on a specific table or list of tables, to be indicated when the command is run.

The PHP class that represents the command also already been written to accept arguments. Any arguments passed when executing the command get assigned to the `$args` variable in the corresponding method. To pass arguments to that method, use this syntax:

```
yiic <command-name> <action-name> <arg1> <arg2>...
```

For example, if the “optimize” command optimizes tables, you could pass the tables to be optimized:

```
yiic database optimize session reset
```

With that command, in the `actionOptimize()` method, `$args[0]` would have a value of “session” and `$args[1]` would have a value of “reset”.

You don’t need to quote any strings unless they contains spaces or other potentially problematic characters. (That being said, there’s no harm in quoting the parameter values, either.)

As mentioned, `$args` will be an indexed array by default. If you’d rather create an associative array, use this syntax instead:

```
yiic <command-name> --arg1=value1 --arg2=value2...
```

As a hypothetical, let’s say that there’s a “purge” command with a “clear” action. That action may can take as arguments: the type of thing to purge, an expiration unit (e.g., hour or day), and an expiration quantity. It might then be called like so:

```
yiic purge clear --type=cache --interval=hour --number=2
```

Now, in the `actionClear()` method, `$args['type']` would have a value of “cache”, `$args['interval']` would have a value of “hour”, and `$args['number']` would have a value of “2”. That would be true for this command as well:

```
yiic purge clear --number=2 --type=cache --interval=hour
```

## Returning Exit Codes

The final thing you should think about when it comes to creating command-line scripts are exit codes. Exit codes aren't commonly used by PHP programmers, as the Web server itself normally handles the proper codes to indicate the success or failure of an operation. But in the command-line world, including in languages such as C and Java, having a code be returned to indicate the success or failure of an operation is the norm.

In the command-line interface, returning a code from an action or command is as simple as having the associated method use a `return` statement. The standard is for an integer to be returned, with 0 being returned to indicate success:

```
<?php
# protected/commands/DatabaseCommand.php
class DatabaseCommand extends CConsoleCommand {
    public function actionOptimize($args) {
        echo 'Optimizing...' . PHP_EOL;
        // Do whatever!
        return 0;
    }
    public function actionBackup($args) {
        echo 'Backing up...' . PHP_EOL;
        // Do whatever!
        return 0;
    }
}
```

This always struck me as a little backwards (after all, 0 equates to false in many situations), but you can think of it as “0 errors occurred”.

At a minimum, returning 1 indicates that an error occurred. You can also use escalating integers (up to 254) to indicate a level of error. I would caution not to go overboard with that, though, as the returned number would not have inherent meaning outside of the command-line script itself.

## Chapter 17

# IMPROVING PERFORMANCE

Using a framework such as Yii provides an exponentially faster development time, but it does so at a cost. That cost is site performance. Tapping into frameworks can be a faster way to create software, and the end result may even be more secure and feature-rich, but in all likelihood, the generated software will perform more poorly than it would have if written from scratch (assuming general competency on the programmer's part with both approaches).

However, there is a counter argument as to why this performance hit is acceptable. First, there are limited ways to speed up your development time (when not using a framework), and it's impossible to get wasted time back. Second, you *can* improve the performance of your framework-based project. Third, a framework-based site might be able to scale better than a non-framework site, as much of the code will have already been abstracted.

In this chapter I'll explain the myriad ways you can improve your Web site's performance. Some approaches will make use of smart features the Yii framework has, while others will mostly be a matter of just making smart decisions while you develop the site.

The chapter will also cover how you test a site's performance, as you can't know whether your performance improvements are making a difference or not without concrete numbers. The chapter concludes with a suggested, cohesive plan for how you should develop and then speed up your Yii applications.

### Testing Performance

Due to the number of possible factors, there are very few absolutes when it comes to improving a site's performance. An approach that has a huge impact in one situation, may have no benefit, or even a cost, in another. Thus it's imperative that

you know how to test a site's performance so that you can make the decisions that best benefit your site running on your server.

When it comes to a Web site, performance can be lumped into two broad categories:

- The server: how quickly the resulting page is outputted
- The browser: how quickly the resulting page is downloaded and rendered

On the server side of things, you'll want to write good code, make the database as efficient as possible, and throw in some caching. On the browser side, you'll want to limit how much data is transmitted, restrict the number of HTTP requests that have to be made, and watch how external resources such as JavaScript files are loaded.

I'll cover these topics in detail throughout the chapter, but just as there are different approaches for server performance and browser performance, there are different testing methodologies as well.

## Benchmarking the Server

A somewhat technical but highly useful way of testing your site's performance is to use [ApacheBench](#). ApacheBench comes with the Apache Web server and is used to benchmark a Web server's performance. The most important information this tool will return is the number of requests per second (RPS) the server can handle.

Requests per second reflects many aspects of a server, from its processor speed and available memory, to how efficient the underlying code is. Simply put, more requests roughly equates to better performance. If your site does not perform well, the server won't be able to handle that many requests for your site, which means:

- Your users will see long delays in loading pages
- Your site won't scale well
- Your site won't be able to handle traffic spikes well

The ApacheBench tool is run from a command-line interface. Assuming you have ApacheBench installed on your computer (which generally means you have Apache installed), you can use it with the command (note that you're testing some other server from your computer):

```
ab -n # -c # http://www.example.com/
```

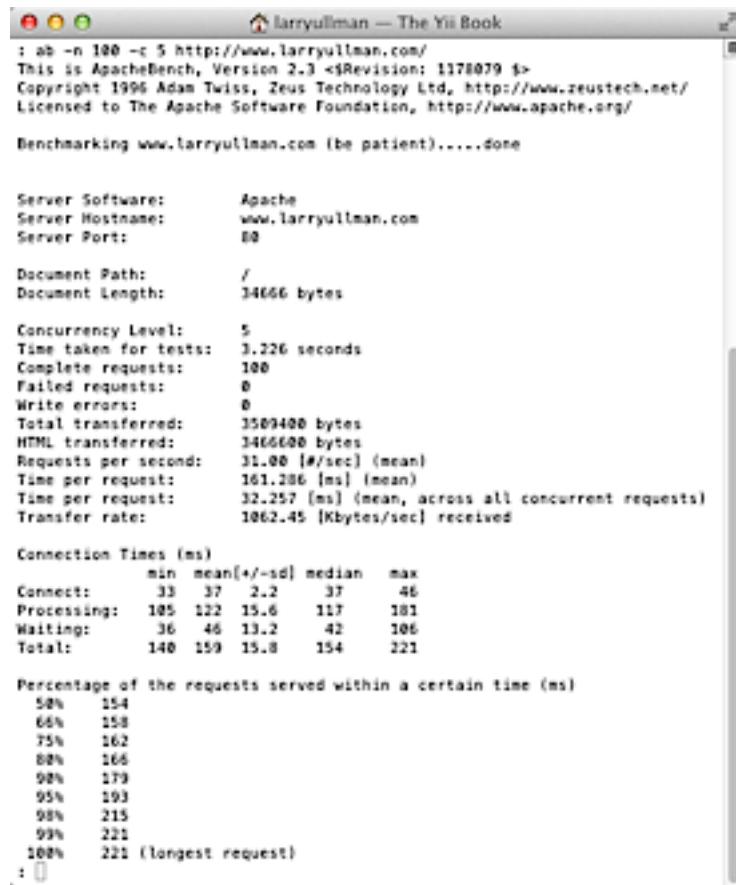
{NOTE} You need to end your URL with a slash (unless it ends with a filename).

The `-n` flag indicates the number of requests to make. The `-c` flag dictates the number of *concurrent* requests to make. This is great because no matter how fast you are with your browser, you can't single-handedly make multiple simultaneous requests of a server.

If you look at the ApacheBench documentation (linked above or available through the `--help` command), you'll see that you can also indicate a time limit, send POST data along with each request, and so forth.

As a basic test, you can see the performance of a site for 100 requests in groups of 5 (**Figure 17.1**):

```
ab -n 100 -c 5 http://www.larryullman.com/
```



**Figure 17.1:** The ApacheBench test results for my Web site.

You'll get a bunch of information back, after a few seconds or minutes. You'll see the number of complete and failed requests, which is nice, but the requests per second (RPS) is the most common benchmark for a site's performance. In other words, this server can handle X number of requests for this URL per second. The more RPS, the better: it's that simple.

With this in mind, and as I'll explain at the end of the chapter, the basic sequence will be this:

1. Run `ab` to benchmark your initial RPS.
2. Make a tweak to improve the performance.
3. Re-benchmark the site to find the new RPS.
4. Repeat.

## Profiling Your Code

There are two ways you can profile your PHP code in a Yii-based site. The first option is the standard approach for profiling any PHP code:

```
<?php
$start = microtime();
// Do whatever.
$end = microtime();
echo sprintf('The code took %f microseconds.',
    $end - $start);
```

With Yii, this can be a bit tricky, however, as you'll need to decide where (in the code) to start and stop the profiling and you'll need to factor in where you display the output in your view files. Once again, Yii has come up with its own solution.

The first step is to identify in your code where to start and stop profiling. The most common and logical code to profile would be your controller methods. To do that, use the `beginProfile()` and `endProfile()` methods, providing both with the same identifier:

```
# protected/controllers/PageController.php
public function actionView($id) {
    Yii::beginProfile('load page');
    $this->render('view',array(
        'model'=>$this->loadModel($id),
    ));
    Yii::endProfile('load page');
}
```

That code will profile the entire method block. And it's perfectly fine for the `endProfile()` to come after calls to `render()`. In fact, that's appropriate.

(Note that this approach does not profile the complete time it takes Yii to render the page, as it doesn't factor in the effort required to find and call the controller method. That's okay, though, as that effort will be generally comparable regardless of the controller method involved.)

With the block identified, you should now tell Yii to display the results. That's done in the configuration file:

```
# protected/config/main.php
array(
    // Other stuff.
    'preload'=>array('log'),
    'components'=>array(
        'log'=>array(
            'class'=>'CLogRouter',
            'routes'=>array(
                array(
                    'class'=>'CFileLogRoute',
                    'levels'=>'error, warning',
                ),
                array(
                    'class'=>'CProfileLogRoute',
                    'report'=>'summary'
                ),
                array(
                    'class'=>'CWebLogRoute',
                ),
            ),
        ),
    ),
    // More other stuff.
);
```

The code enables the `CProfileLogRoute`. By doing this, the profiling results will be added to the log at the bottom of the browser window, before the application log, assuming that `CWebLogRoute` is also enabled (**Figure 17.2**).

Profiling Summary Report (Time: 0.07686s, Memory: 6,050KB)					
Procedure	Count	Total (s)	Avg. (s)	Min. (s)	Max. (s)
load page	1	0.05980	0.05980	0.05980	0.05980
Application Log					
Timestamp Level Category Message					

**Figure 17.2:** The profiling results at the bottom of the browser window.

As you can see in that figure, the identifier is used in the output. You can profile multiple blocks at the same time:

```
# protected/controllers/AnyController.php
public function actionSomething() {
    Yii::beginProfile('Model Load');
    // Do whatever.
    Yii::endProfile('Model Load');
    Yii::beginProfile('Network Call');
    // Do whatever.
    Yii::endProfile('Network Call');
}
```

You can even profile subsections of a block. To do that, just nest one block within another, giving each a unique identifier:

```
# protected/controllers/AnyController.php
public function actionSomething() {
    Yii::beginProfile('Something Method');
    Yii::beginProfile('Model Load');
    // Do whatever.
    Yii::endProfile('Model Load');
    Yii::beginProfile('Network Call');
    // Do whatever.
    Yii::endProfile('Network Call');
    Yii::endProfile('Something Method');
}
```

The goal of profiling code is to find the bits that:

- Are taking the most time
- Can be improved, performance-wise

Sometimes you'll find that an "expensive" section of code from a performance perspective is what you'd expect and perhaps can't be improved, such as a database interaction. Other times you'll find a section of code that isn't expensive, but can easily be optimized, such as swapping a loop for a recursive function call.

As a caveat, keep in mind that profiling your code by definition will make the result less efficient. In other words, the act of setting timers, recording periods of execution, and outputting the additional results will affect the overall performance. That's okay, though, as you'll remove all code profiling before you go live.

{TIP} A great tool for profiling PHP code is [Xdebug](#).

## Testing the Database Application

Another type of profiling you can do is to profile the database interactions. Again, you can do this with Yii and without Yii.

Yii provides a nice option for profiling the SQL command executions (i.e., profile every query being run by the site). To enable this, simply set `enableProfiling` to true in the database configuration:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'db'=>array(
            'connectionString' => 'mysql:host=localhost;dbname=whatever',
            'emulatePrepare' => true,
            'username' => 'root',
            'password' => 'blah',
            'charset' => 'utf8',
            'enableProfiling' => true
        ),
        ),
    // More other stuff.
);
```

Then, so long as the `CProfileLogRoute` and `CWebLogRoute` are also configured, at the bottom of the browser window you'll see how long each query took to execute (**Figure 17.3**).

Profiling Summary Report (Time: 0.05035s, Memory: 6,055KB)					
Procedure	Count	Total (s)	Avg. (s)	Min. (s)	Max. (s)
system.db.CDbCommand.query(SHOW FULL COLUMNS FROM `page`)	1	0.00314	0.00314	0.00314	0.00314
system.db.CDbCommand.query(SELECT * FROM `page` `t` WHERE `t`.`id`='6' LIMIT 1)	1	0.00054	0.00054	0.00054	0.00054
system.db.CDbCommand.query(SHOW CREATE TABLE `page`)	1	0.00038	0.00038	0.00038	0.00038

Application Log	

**Figure 17.3:** The time it took to execute the page's queries is now reflected in the profile results.

If you'd like to look at the macro perspective of database interactions, invoke the `CDbConnection::getStats()` method. It returns an array consisting of the:

- The number of queries executed on the page
- The total amount of time required to execute all of those queries

Here's how you might use that (**Figure 17.4**):

```
# protected/controllers/AnyController.php
public function actionSomething() {
    // Do whatever.
    $stats = Yii::app()->db->getStats();
    // Use $stats[0] and $stats[1]
}
```

Queries run: 3

Total time: 0.004321813583374

**Figure 17.4:** An overview of the page's queries.

Without using Yii, there are a couple of tools you can use to profile your database interactions, although these are a bit more technical and may require administrative control over your database.

The first option is to use your database application's "slow query" log. The slow query log records every SQL command that took longer than a specified amount of time to execute. For MySQL, the slow query log needs to be enabled and configured. For information on MySQL's slow query log, see [this page of the MySQL manual](#).

A more proactive approach is to run all your queries through MySQL, prefacing them with EXPLAIN. The EXPLAIN command tells MySQL to report how it would execute the query, including what indexes would be used, how many rows would be returned, and so forth. EXPLAIN is a good way to confirm that you've got the right indexes in place. You can also use it to experiment with queries to see if there's not a better solution than the one you're currently using.

## Testing the Browser Performance

The last way you can profile your Web page is to see how quickly it loads in the browser. This can be further broken down into two areas:

- How quickly the data is received by the browser from the server
- How quickly the browser renders the whole page

The former will be impacted by a host of factors, including:

- Your server's performance
- The amount of data being transferred
- The applicable networks

- The use of a Content Delivery Network (CDN)
- Browser caching

You can test how quickly a live page loads on your server using any number of tools found online (via a quick Google search) or already in your browser (e.g., Opera Dragonfly or Chrome's Developer Tools).

How quickly the browser renders the page is impacted by:

- The browser in use
- The amount of data being transferred
- The number of HTTP request being made
- The complexity of the Document Object Model (DOM)
- Your use of JavaScript, CSS, and various media

Again, there are lots of resources online for profiling how quickly a page loads in the browser. I've always liked Yahoo!'s [YSlow](#), which provides both analytics and recommendations.

## Changing the Bootstrap

Now that you know how to test your site's performance, you can start learning the different ways you can improve it.

To begin, there are a couple of changes you can easily make that will impact your site's performance. First, you should make edits to the **index.php** bootstrap file. You should do two things there:

- Change the inclusion of the framework file from **yii.php** to **yiilite.php**
- Disable debugging

The default **index.php** as created by the **yiic** script looks like:

```
<?php
// change the following paths if necessary
$yii=dirname(__FILE__).'/../framework/yii.php';
$config=dirname(__FILE__).'/protected/config/main.php';

// remove the following lines when in production mode
defined('YII_DEBUG') or define('YII_DEBUG',true);
// specify how many levels of call stack should be shown in each log message
defined('YII_TRACE_LEVEL') or define('YII_TRACE_LEVEL',3);

require_once($yii);
Yii::createWebApplication($config)->run();
```

You would want to change that to just:

```
<?php
// change the following paths if necessary
$yii=dirname(__FILE__).'/../framework/yiilite.php';
$config=dirname(__FILE__).'/protected/config/main.php';

require_once($yii);
Yii::createWebApplication($config)->run();
```

The **yiilite.php** script, which comes with the framework, is a slightly optimized version of the default **yii.php**. It's particularly beneficial when using APC for caching (discussed next in the chapter).

{WARNING} As with everything related to performance, the actual results of any particular change can vary from server to server, environment to environment. You should always benchmark and profile all changes to see the actual results for yourself.

Second, debugging needs to be disabled for both performance and security reasons.

## Changing the Configuration

Turning to the configuration file, you should employ as few route rules as possible. The application has to evaluate each rule for each request until a match is found, which can be expensive. Towards that end, you could also make sure the rules are ordered such that matches can be found as quickly as possible.

By default, application components in Yii are created on demand. This means an application component may not be created at all if it is not accessed during a user request. As a result, the overall performance may not be degraded even if an application is configured with many components. So you don't need to go around removing components from your configuration file, but be prudent about which ones are preloaded (such as **CLogRouter**), which don't have this same constraint.

## Caching

Perhaps the most potent tool available for improving the performance of your Web site is *caching*. Caching is a way of saving output for future use. For example, browsers will automatically cache Web page resources, such as images, CSS, and JavaScript. By caching those resources, when the visitor views another page that uses those same resources, the browser will retrieve them from storage and not need

to download them again. Hopefully this is a concept with which you are already familiar.

I'll return to the topic of browser caching (and browser performance in general) later in the chapter, but first I want to discuss server-side caching. There are several different server-side components that can be cached:

- Discrete pieces of data (e.g., variables)
- Fragments of pages
- Entire Web pages
- Database schemas
- Session data

I'll explain how to do each of these, but first you need to know how to choose and enable a *caching engine*.

## Choosing and Enabling Caching

In order to make use of caching, one has to first enable it in the application. To do that, you must choose which caching engine to use. The Yii framework supports all the popular PHP caching tools:

- [Alternative PHP Cache \(APC\)](#)
- [Memcached](#)
- [XCache](#)
- [EAccelerator](#)
- [Redis](#)
- Zend Data Cache

{NOTE} Redis is supported as a caching mechanism as of Yii 1.1.14.

Note that these are all tools that must be installed on your server. Some, like APC, may be built into your PHP installation (run a `phpinfo()` script to confirm support for it). Others, such as Memcache, require a separate Memcache installation (and it needs to be running). Rather than discuss the pros and cons of the various caching tools in any detail, my recommendation is to start with what you have. For example, although APC may not be the fastest option, it's the most likely to already be installed.

{TIP} Most of the caching tools store data in memory. The more memory you have available for caching, the better the performance, in general.

There are also Yii classes for using a database or the file system as a caching mechanism. And because every one of these classes extends the **CCache** class, it's easy to use any tool you want for caching (such as technologies to later come out, or those of your own creation).

Once you've selected a caching mechanism to use, you need to enable and configure it in your primary configuration file. Here's how you would enable APC:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'cache'=>array(
            'class'=>'CApcCache',
        ),
    ),
    // More other stuff.
);
```

The specific configuration properties that are available will depend upon the caching mechanism used, and therefore the associated class. The **CMemCache** class, for example, has a **servers** attribute that's used to point to the Memcached server instances:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'cache'=>array(
            'class'=>'CMemCache',
            'servers'=>array(
                array(
                    'host'=>'server1',
                    'port'=>11211,
                    'weight'=>60,
                ),
                array(
                    'host'=>'server2',
                    'port'=>11211,
                    'weight'=>40,
                ),
            ),
        ),
        // More other stuff.
    );
);
```

How you configure any particular caching mechanism will depend upon the mechanism in use, your server, the application, and so forth. This chapter will get you started down this path, but I recommend you really learn and master the caching tool of your choice.

## Data Caching

The most atomic type of caching you can utilize is data caching. This is just the storing of a variable's value in the cache (well, any data). To do this, you first get a reference to Yii's cache:

```
$c = Yii::app()->cache;
```

Regardless of what caching mechanism you're using, the above line will reference Yii's cache.

Next, call the `set()` method to store some data in the cache. Its first argument is a unique identifier and its second is the value being stored:

```
$c = Yii::app()->cache;
$c->set('foo', 'bar');
```

That will store the data in the cache without an expiration (although it could be removed if the caching mechanism itself clears it). To cache the data for a certain period, add a third argument, a time in seconds to persist:

```
$c = Yii::app()->cache;
$c->set('foo', 'bar', 60*60); // One hour
```

If you'd only like to store the data in the cache if it is not already cached, use `add()`. It takes the same three initial arguments as `set()`:

```
$c = Yii::app()->cache;
$c->add('foo', 'bar', 60*60); // One hour
```

To retrieve cached data, use the `get()` method, providing the same unique identifier:

```
$c = Yii::app()->cache;
$data = $c->get('foo');
```

If the data was no longer available, the value `false` will be returned:

```
$c = Yii::app()->cache;
$data = $c->get('foo');
if ($data !== false) {
    // Use $data
} else {
    // Use non-cached version and re-cache.
}
```

If you'd prefer, you can use array syntax to set and get cached data:

```
$c = Yii::app()->cache;
$c['foo'] = 'bar';
echo $c['foo'];
```

You cannot set an expiration when using this approach, however.

If you need to fetch multiple pieces of cached data, you can do that in one step using `mget()`. It takes an array of identifiers to fetch and returns an array of key=>value pairs:

```
$c = Yii::app()->cache;
$c->set('x', 1);
$c->set('y', 2);
$c->set('z', 3);
$data = $c->mget(array('x', 'z'));
echo $data['z']; // 3
```

For some caching mechanisms (e.g., Memcached and APC), using `mget()` can also result in better performance.

To remove data from the cache, invoke `delete()`, providing it with the identifier:

```
$c = Yii::app()->cache;
$c->delete('foo');
```

To fine-tune the caching behavior, you can add *dependencies*. Dependencies are different types of criteria you can associate with cached data that determine when cached data will be used as opposed to uncached data.

To add a dependency to a data cache, provide a fourth argument to `set()` or `add()`. This argument needs to be an object of a type of cache dependency:

- `CFileCacheDependency`, dependent upon the provided file having been changed

- `CDirectoryCacheDependency`, dependent upon the provided directory having been changed (including its subdirectories)
- `CDbCacheDependency`, dependent upon the results of the provided SQL statement changing
- `CGlobalStateCacheDependency`, dependent upon the value of a provided global variable changing

You can also use the `CChainedCacheDependency` and `CExpressionDependency` to make more complex dependency scenarios.

As an example, you might cache an array that represents the files found in a specific directory. You would want this array to be recreated (and re-cached) if that directory changes. To pull that off, add the `CDirectoryCacheDependency` to the caching of the variable:

```
$c = Yii::app()->cache;  
$dir = 'somedir';  
$pics = glob($dir . '/*.png');  
// Keep the array cached for a day  
// Or until the directory changes:  
$c->set('pics', $pics, 60*60*24,  
        new CDirectoryCacheDependency($dir)  
)
```

If the contents of that directory have changed since the array was stored in the cache, then the `get()` method will return false and you'd want to re-cache the data (using the code just demonstrated).

Later in the chapter, I'll also show some examples of the database caching, which can be very helpful in improving the performance of your site.

## Fragment Caching

A broader type of caching than simple data caching is fragment caching. This is used to cache parts of a page. Fragment caching is performed differently than data caching. In fact, the approach is similar to using content decorators as explained in Chapter 6, “[Working with Views](#)”.

Fragment caching is a breeze to implement. Within a view file, wrap the content fragment to be cached within `beginCache()` and `endCache()` calls. The former should be provided with a unique identifier:

```
<?php if ($this->beginCache('newsSidebar')) : ?>  
<div class="span-5 last">  
<div id="sidebar">
```

```
<?php $this->widget('ext.widgets.News'); ?>
<!-- sidebar --></div>
<?php $this->endCache(); ?>
<?php endif; ?>
```

In that hypothetical example, the “news” widget output is being cached. By wrapping the `beginCache()` call in a conditional, the fragment will automatically be retrieved from the cache if it exists, or generated and cached if it does not.

The `beginCache()` method takes an optional second parameter used to configure the caching behavior. This should be an array. The possible values are any public, writable attributes of the `COOutputCache` class. The most important of these is `duration`, used to set the caching duration in seconds.

```
<?php if ($this->beginCache('newsSidebar',
    array('duration' => 60*60))) : ?>
<div class="span-5 last">
<div id="sidebar">
<?php $this->widget('ext.widgets.News'); ?>
<!-- sidebar --></div>
<?php $this->endCache(); ?>
<?php endif; ?>
```

As with data caching, you can add dependencies to fragment caching, although the syntax differs. Add a “dependency” index item to the customization array, passing key values to configure that dependency. This example makes use of the global variable dependency:

```
<?php if ($this->beginCache('promo',
    array(
        'duration' => 60*60,
        'dependency' => array(
            'class' => 'CGlobalStateCacheDependency',
            'stateName' => 'promocode'
        )
    )))
: ?>
<div class="span-5 last">
<div id="sidebar">
<?php $this->widget('ext.widgets.Promotion'); ?>
<!-- sidebar --></div>
<?php $this->endCache(); ?>
<?php endif; ?>
```

With that code, the fragment cache will be used until the duration passes or the global state variable “`promocode`” changes. You can assign a value to it using:

```
Yii::app()->setGlobalState('promocode', 'value');
```

You can also impact the caching by setting the `requestTypes` property. For example, you can limit the caching to just GET or POST requests. And you can use the `varyBy*` methods to change when caching applies in other ways:

- When certain parameters are present in the URL
- Based upon the route
- By session (i.e., each unique session would have its own fragment caching)
- Using a custom expression of your choosing

All of these possibilities, and the corresponding methods, are detailed in the Yii manual.

## Page Caching

Continuing to move along, a more expansive type of caching than fragment caching is page caching. This is, as you would imagine, the caching of an entire rendered page. To cache an entire page, you need to add the `COutputCache` class as a filter to the controller:

```
# protected/controllers/SomeController.php
public function filters() {
    return array(
        'accessControl',
        array(
            'COutputCache',
            'duration'=>100,
            'varyByParam'=>array('id'),
        ),
    );
}
```

Again, you configure how the caching behaves by assigning values to the key `COutputCache` properties as in the above. In that example code, the caching will vary based upon the `id` parameter, which is logical.

*{TIP}* To confirm that page caching is working, check out your profile summary report to see if queries are still being run (assuming that page normally runs queries).

As with any filter, you'll likely want to use the plus operator to limit the filter to only certain actions. You'd likely want to cache a “view” or “index” action, but not “add” or “update”:

```
# protected/controllers/SomeController.php
public function filters() {
    return array(
        'accessControl',
        array(
            'COutputCache + view, index',
            'duration'=>100,
            'varyByParam'=>array('id'),
        ),
    );
}
```

The above performs server-side caching. The Yii framework will only re-execute the applicable controller actions once the cache has expired. An alternative is to make use of browser-side caching. This is simply a matter of indicating caching recommendations to the user's browser, giving it the information it needs to do what it'd do anyway.

Browser caching is controlled (or to be more precise, informed) by a couple of things. The two most important are:

- The Last Modified header
- Entity Tags (aka, ETags)

The Last Modified header is a simple indicator of when the content last changed. The browser can use this to know whether the cached version should be displayed (if it exists) or a version should be pulled from the server.

ETags are server-provided unique identifiers for a resource. By associating an ETag with a page, an update to the page results in a new ETag. The browser can then see if the ETag for the version in cache matches the ETag for the resource being requested. If so, the cached version will be used. Otherwise, the updated version will be pulled from the server and the ETag will be updated.

You can manually manage both the Last Modified header and the ETags for any page, but once again Yii provides a nice tool for doing this automatically. The `CHttpCacheFilter` class can be assigned as a filter to your controller and will send the Last Modified and ETag headers for you.

```
# protected/controllers/SomeController.php
public function filters() {
    return array(
        'accessControl',
        array(
            'CHttpCacheFilter + view',
        ),
    );
}
```

```
        'lastModified'=>$timestamp,
    ),
);
}
```

The `lastModified` value can be a Unix timestamp or a human-readable date. To dynamically determine this value, you might query your database (depending upon the underlying table structure):

```
# protected/controllers/SomeController.php::filters()
return array(
    'accessControl',
    array(
        'CHttpCacheFilter + view',
        'lastModified'=> Yii::app()->db->createCommand("SELECT
            MAX(`date_updated`) FROM tableName")->queryScalar(),
    ),
);
```

To also set an ETag, provide a value to the `etagSeed` property. This value can be any string or array that will properly identify the state of the resource. Again, the last updated date would make sense:

```
# protected/controllers/SomeController.php::filters()
return array(
    'accessControl',
    array(
        'CHttpCacheFilter + view',
        'lastModified'=> Yii::app()->db->createCommand("SELECT
            MAX(`date_updated`) FROM tableName")->queryScalar(),
        'etagSeed'=> Yii::app()->db->createCommand("SELECT
            MAX(`date_updated`) FROM tableName")->queryScalar(),
    ),
);
```

## Preventing Some Caching

When you're using page caching (and less often, fragment caching), you'll sometimes have parts of the page that should not be cached. For example, maybe your whole home page should be cached, except for:

- The snippet that greets a logged-in user by name
- The widget that displays the Twitter feed

- Any other element that's highly variable and/or time sensitive

To prevent Yii from caching some content, you can flag the content as dynamic. Doing so requires changing your views a bit. To start, among some content that is otherwise being cached, insert some dynamic content using the `renderDynamic()` method:

```
<!-- HTML page being cached. -->
<div class="span-5 last">
<div id="sidebar">
<?php $this->renderDynamic('getNews'); ?>
<!-- sidebar --></div>
<!-- Rest of the HTML page being cached. -->
```

The `renderDynamic()` method takes a single argument: a callback to a function that will create and return the dynamic content. The easiest way to provide this value is as a method in the current controller. Thus, using the above code, the current controller should have a method named `getNews` that returns the dynamic content:

```
# protected/controllers/AnyController.php
public function getNews() {
    // return dynamic content
}
```

## Session Caching

Another type of caching you can add to your site to improve its performance is session caching. Session caching isn't quite the same kind of caching as data, fragment, or page, but rather an easy way to move the storing of session data from an inefficient methodology to a more efficient one.

By default, PHP stores session data in plain text files in a common, world-writable directory on the server. The file system interactions required—all of those reads and writes—are slow. Moreover, on a shared server, session data can be compromised due to its publicly-available status in a common directory. A solution to both problems is to use session caching in Yii. With this enabled, instead of storing the data in the file system, it'll be stored in the caching mechanism. This can have a nice performance benefit.

To enable session caching, configure the “session” component in your primary configuration file:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'session'=>array(
            'class'=>'CCacheHttpSession',
            'cacheID' => 'sessionCache',
        ),
        'sessionCache' => array(
            'class'=>'CRedisCache',
            'hostname'=>'localhost',
            'port'=>6379,
            'database'=>0,
        ),
    ),
    // More other stuff.
);
```

First, the “session” component is configured to use the `CCacheHttpSession` class, instead of the default `CHttpSession` class. This configuration is also assigned an identifier of “`sessionCache`”. In the next bit of code, this identifier is used to tell Yii to use Redis caching for that specific cache type.

If you have an even larger Yii application that runs on multiple servers, you can use Memcached for your sessions caching. Assuming that Memcached is running on one common server, by storing all your session data there, you also end up resolving the issue of supporting user sessions across multiple Web servers.

## Improving Database Performance

As a rule of thumb, file system interactions tend to be the biggest bottlenecks in any site’s performance. And a database, of course, is essentially a managed file system. Finding ways to improve the performance of your database interactions will go a long way towards improving the overall performance of your site.

Improving the performance of your database interactions comes down to:

- Preventing the execution of any query that you can avoid
- Improving the execution of any queries that you must run

The first thing you’ll want to do is using profiling and logging to see what queries are being run on a page and how long each is taking to execute. That will give you a baseline to begin with (see Figures 17.3 and 17.4).

Next, make sure that you're only running the queries you absolutely must. It's unlikely that you'll have unnecessary queries, but sometimes you slipped in your logic and added a process that wasn't necessary.

*{TIP}* Caching pages or fragments will also reduce the number of queries being executed.

The third step is to examine how you might make a query more efficient. For non-Active Record queries, this can be a matter of changing what columns are selected, how many rows are returned, how JOINs are performed, and so forth.

For Active Record queries, it is possible to limit what columns are selected, but doing so undermines much of the point of using AR in the first place. You may want to consider switching from AR to Query Builder or Direct Access Objects (DAO) for straight-up SQL. You'll see the biggest benefit to dropping AR when you have complex, demanding queries. The pro's and con's of AR vs. Query Builder vs. DAO were explained in Chapter 8, “[Working with Databases](#)”.

Those are some general recommendations for improving your database performance. Let's look at some database design thoughts, and then move to specific Yii features that will help here.

*{TIP}* If you have administrative control over your MySQL installation, tweaking its configuration can also greatly improve performance.

## Better Database Design

A lot of performance comes down to just employing best practices:

- Don't create variables you don't need.
- Watch how many files you include.
- And so forth.

This is especially true with your database. Proper database design will go a long way towards improving your site's performance. Hopefully you've made the right decisions well before this point, but as a reminder, here are some best practices.

First, try to use numbers whenever possible. Databases (and, well, pretty much all computer programs) work with numbers faster than they do strings. So when you have a choice, go with a number (e.g., a zip code).

Second, make good use of indexes. To start, you should index columns that are:

- The primary key
- Frequently used in WHERE clauses

- Frequently used in ORDER BY clauses
- Frequently used as the basis for a JOIN

You should *not* index columns that:

- Allow NULL values
- Have a very limited range of values (such as Y/N or 1/0)

Again, watch the slow query log and use EXPLAIN to identify and rectify problematic queries.

Third, higher-end and more active applications tend to move away from a strictly normalized structure. Normalization is great for data integrity but bad for performance (JOINS are especially costly). A performance solution is to start breaking normalization and creating tables that will perform better.

For example, you might have a `user` table that stores everything about a user. Each time a user logs in, or any time any query references that table, all extraneous information stored in the table—the user’s address, gender, whether the user likes watermelon or not, whatever—impacts the performance of that query, even when that information is not referenced by the query itself. The fix in such cases is to put the bare minimum required information in the primary `user` table (e.g., username, email address, password, user ID, and registration date), and move everything else to another table. Again, multiple queries will be required to retrieve all the information, but multiple smaller queries can execute faster than single larger queries. You’ll also find that all queries will run better if large text and binary columns are moved to their own separate tables.

If you do this right, the first table could have SELECT queries performed using any column (e.g., the email address or the username), but the secondary tables would always use SELECT queries off of foreign keys: the FK related to the PK from the original table.

Of course, the downside to this is that it’ll wreak havoc with your AR models.

Similarly, I like to use views to represent complex, relational data sets. Again, you won’t be able to use AR with views, but you can use DAO effectively.

## Schema Caching

Active Record is a lovely design but isn’t so kind when it comes to performance. In order for the framework to do anything with a particular model, it must first run two queries:

- SHOW FULL COLUMNS FROM tablename
- SHOW CREATE TABLE tablename

Again, to be clear, any use of, say, an `employee` table requires these two commands be run first. If you have the related `Employee` and `Department` models, a JOIN across them requires four queries before any data is retrieved at all!

This is the downside of Active Record: it does a lot of the work for you, but at a cost of performance. You can quickly and easily improve your Yii application's performance by limiting how often these basic queries are run. To do so, you'll need to cache the database schema (i.e., tell Yii to remember what columns exist in the tables).

Caching the database schema is done in the `main.php` configuration file, in the section for configuring the "db" component:

```
'db'=>array(
    'connectionString' => 'mysql:dbname=test',
    'schemaCachingDuration' => 604800, // One week
    'emulatePrepare' => true,
    'username' => 'user',
    'password' => 'pass',
    'charset' => 'utf8',
),
```

The `schemaCachingDuration` line needs to be added there. Assign to it a logical value, such as an hour or a week (the value is in seconds). That line will enable caching of the database schema but only if Yii has a caching component registered.

To confirm this is working, look at your Web profiling log for any page that uses a model (**Figure 17.5**). The two queries mentioned earlier should not be run (after you've reloaded the page to cache the results once, that is). This is very important: if you follow these steps and don't see the `SHOW` commands disappear from the Web profiling log, the caching isn't working.

Profiling Summary Report (Time: 0.04713s, Memory: 5,942KB)					
Procedure	Count	Total (s)	Avg. (s)	Min. (s)	Max. (s)
system.db.CDbCommand.query(SELECT * FROM `page` `t` WHERE `t`.`id`='5' LIMIT 1)	1	0.00068	0.00068	0.00068	0.00068
Application Log					

**Figure 17.5:** Thanks to schema caching, two queries are no longer necessary.

## Query Caching

After you've had Yii cache the database schema, you may want to consider having Yii cache the results of specific database queries. This concept is much the same as the data caching explained already.

The first thing you'll need to do is ensure that caching is enabled. Then, call the `cache()` method when executing any query. Provide to the method a length of time, in seconds, to cache the results:

```
$q = 'SELECT * FROM table_name';
$cmd = Yii::app()->db->cache(60*60)->createCommand($q);
$rows = $cmd->queryAll();
```

With that additional `cache(60*60)` code, Yii will look for cached results prior to actually running the query. If there are cached results, those will be returned instead. If no cached results exist for that SQL command, Yii will actually execute the query to get the results, and then cache those.

That example uses DAO, but you can cache AR queries, too. Again call `cache()`, this time just after `model()`. This query fetches a page, along with the page's author, and all the comments posted on that page, along with the authors of those comments (this is from Chapter 8):

```
$page = Page::model()->cache(3600)
    ->with('pageUser', 'pageComments',
        'pageComments.commentUser')->findPk($id);
```

There are two more things you ought to know about query caching. First, you can use dependencies with query caching, just as you can with data caching. For example, if a query retrieves the latest pages in a CMS example, you may want to cache those results for a certain amount of time or until a new page is added (whichever comes first):

```
$dep = new CDbCacheDependency('SELECT MAX(date_published)
    FROM pages');
$page = Page::model()->cache(3600, $dep)->findAll();
```

The second thing to know is that, by default, the cache only applies to the query being executed. This is logical and appropriate, but when you're using Active Record, one query might lead to others behind the scenes.

In the above example, the `findAll()` method results will be cached. But if lazy or eager loading were to be used (e.g, to find the author of each page), those queries would *not* be cached. You can add a third argument to `cache()` to indicate the number of subsequent queries to also cache:

```
$dep = new CDbCacheDependency('SELECT MAX(date_published)
    FROM pages');
$page = Page::model()->cache(3600, $dep, 1)->findAll();
```

## Flexible Relational Queries

Executing queries is demanding on your application, and executing queries simultaneously across multiple tables in a related database is even more so. In some situations, running one query that fetches all of the related data from multiple tables is preferred. In other situations, it would actually be more efficient to run multiple, separate queries, each from a single table.

One option you have is to use Yii's logging to see what queries are being run, and then to use database profiling techniques to see whether those queries would be more efficient separated out or together. Or you could just let Yii make this decision for you.

When using Active Record with relational queries, you have the option of setting the `CDbCriteria` class's `together` property. When this value is set to true, you force Yii to always run a single query to select all related records. Setting this value to false allows Yii to use separate queries to fetch related records.

You can set this in your `relations()` method of the model:

```
# protected/models/User.php
public function relations() {
    return array(
        'comments' => array(self::HAS_MANY, 'Comment',
            'user_id', 'together'=>false),
        'files' => array(self::HAS_MANY, 'File', 'user_id'),
        'pages' => array(self::HAS_MANY, 'Page', 'user_id'),
    );
}
```

You can also set this value when executing the query:

```
$page = Page::model()->with(
    array(
        'pageUser' => array(
            'select'=>'username',
            'together'=>true
        )
    )
)->findPk($id);
```

And you can set this when explicitly using `CDbCriteria`. But understand that "together" only applies to Active Record when there is a `HAS_MANY` or `MANY_MANY` relationship between two or more models.

## Using Multiple Caching Mechanisms

Since you can implement caching in so many different ways, a common question is whether you can use different types of caching mechanisms for different purposes. For example, you might want to use Memcached for your sessions, data, schema, and query caching, but APC or file caching for your fragment and page caches. This is definitely possible, although it's not obvious from any of the Yii documentation.

The trick here is the cache ID. By default, Yii uses a cache ID of “cache” for all caching mechanisms. This means that anything that uses a cache—data, queries, pages, etc.—has, by default, a cache ID of “cache”. This cache is configured like so:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'cache'=>array(
            'class'=>'CApcCache',
        ),
    ),
    // More other stuff.
);
```

If you wanted to use a different caching mechanism for anything, simply assign it a different caching ID. For example, let's store the database schema in Memcache, but have APC be the default cache. Here's how that configuration file would look:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'db'=>array(
            'connectionString' => 'mysql:dbname=test',
            'schemaCachingDuration' => 604800,
            'schemaCacheID' => 'schemaCache',
            'emulatePrepare' => true,
            'username' => 'user',
            'password' => 'pass',
            'charset' => 'utf8',
        ),
        'cache'=>array(
            'class'=>'CApcCache',
        ),
        'schemaCache'=>array(
            'class'=>'CMemCache',
        ),
    ),
);
```

```
        ),  
    ),  
    // More other stuff.  
);
```

If you want to use a different caching mechanism for the query cache, you'd assign a unique identifier to the `queryCacheID` property of the "db" component. Most of the other caching options just have a `cacheID` property to which you'd assign the identifier.

## Browser Improvements

The focus thus far in the chapter has been on improving the performance on the server-side of things. But the server is only one side of the equation, all sites should optimize the browser experience as well. Doing so when using Yii involves largely the same techniques as you'd use on any site, although once again there are a couple of Yii-specific tools you can utilize. Naturally I want to focus on the Yii-specifics here, but I'll mention the basics quickly.

### Fundamental Browser Improvements

With respect to the browser, the general goal is to reduce the amount of data being transmitted to the browser and the number of HTTP requests required. The former is accomplished by:

- Optimizing your images
- Reducing the amount of HTML
- Minifying your HTML, JavaScript, and CSS

For lots of concrete recommendations, check out the excellent resource "[How to lose weight \(in the browser\)](#)". That resource also explains the best places to put your JavaScript and CSS references within the HTML document (it matters).

The second task—reducing the number of requests required—can be accomplished by:

- Using CSS sprites
- Combining multiple CSS files into one
- Combining multiple JavaScript files into one

With these last two concepts, Yii can help.

## Minimizing Requests with Yii

Minimizing the number of HTTP requests that a browser has to make—how many resources it must download—can have a significant impact on how quickly the page is rendered. You can minimize the number of requests by using CSS sprites to combine lots of your images. On a more advanced level, you can also minimize the number of *initial* requests by having some resources be downloaded after the page has been rendered. But for the JavaScript and CSS resources that are needed by the page from the outset, the only solution is to use but a single file for each type (i.e., one JavaScript file and one CSS file).

As an example, **Figure 17.6** shows the number of requests required to view a page using the standard Yii template. There are 8 requests for that page alone, without any media whatsoever!

Network			
Name Path	Method	Status Text	Type
5 /yii-test-db/index.php/page	GET	200 OK	text/html
screen.css /yii-test-db/css	GET	200 OK	text/css
print.css /yii-test-db/css	GET	200 OK	text/css
main.css /yii-test-db/css	GET	200 OK	text/css
form.css /yii-test-db/css	GET	200 OK	text/css
styles.css /yii-test-db/assets/e93435ed/detailvi	GET	200 OK	text/css
jquery.js /yii-test-db/assets/36ff9c79	GET	200 OK	application/javascript
jquery.yii.js /yii-test-db/assets/36ff9c79	GET	200 OK	application/javascript

**Figure 17.6:** Viewing the number of required requests using the browser tools.

Other pages might have extra JavaScript files to be included.

To start minimizing these requests, you might take your base JavaScript files—

- **menus.js**
- **forms.js**
- **llamas.js**

—and combine them into one called **all.js**. (You would not normally combine your custom JavaScript files into your jQuery or other libraries.)

And you'd do the same with your CSS files, combining these into **all.css**:

- **screen.css**
- **print.css**
- **main.css**
- **forms.css**
- **styles.css**

Now you'll do this in your layout:

```
<link rel="stylesheet" type="text/css"
      href="php echo Yii::app()-&gt;request-&gt;baseUrl; ?&gt;/css/
            all.css" /&gt;
&lt;script src="<?php echo Yii::app()-&gt;request-&gt;baseUrl;
            ?&gt;/js/all.js"&gt;&lt;/script&gt;</pre
```

By taking these steps, you've reduce six or eight or X HTTP requests down to two. That's great! But...

Unfortunately, you're using a widget in your sidebar that requires its own CSS file, **widget.css**. And some pages use an extension that requires its own **ext.js** and **ext.css** files. Now you're back up to 5 requests again! Fortunately, there's a relatively easy solution.

The first thing you'll need to do is copy the contents of those other resources—**widget.css**, **ext.css**, and **ext.js**—into your **all.css** and **all.js** files.

Second, tell Yii that when requests are made for **widget.css**, **ext.css**, and **ext.js**, the **all.css** and **all.js** files should be provided instead. This is accomplished by configuring Yii's “clientScript” component. Provide to this component's **scriptMap** property the names of the files and what they should be mapped to:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'clientScript'=>array(
            'scriptMap'=>array(
                'ext.js'=>'/js/all.js',
                'ext.css'=>'/css/all.css',
                'widget.css'=>'/css/all.css'
            )
        ),
    ),
)
```

```
),
// More other stuff.
);
```

Now, whenever a page makes a request for `ext.js`, Yii will convert that request into one for `all.js` instead. And since that file will have already been included by your header, no additional request will be required (**Figure 17.7**).

Name Path	Method	Status Text	Type
 5 /yii-test-db/index.php/page	GET	200 OK	text/html
 all.css /yii-test-db/css	GET	304 Not Modified	text/css
 jquery.js /yii-test-db/assets/36ff9c79	GET	304 Not Modified	application/javascript
 jquery.yii.js /yii-test-db/assets/36ff9c79	GET	304 Not Modified	application/javascript

**Figure 17.7:** All of the CSS requests have been combined into one.

Understand that the “clientScript” component does not:

- Create the HTML tags required to include any resource
- Automatically combine the files into one for you
- Need the full path to the file to be mapped, just the file’s name

The “clientScript” component also doesn’t affect resources that are manually included in a page, such as through a LINK or SCRIPT tag.

## Using Hosted Libraries

Another way you can improve performance when it comes to how quickly the browser renders your page is to switch from using your own versions of common libraries to using a hosted one. For example, if you include the jQuery library from Google instead of your own site, the browser will likely load that library faster because:

- Google has faster servers than you
- The Google jQuery library is on a CDN, and the library will be pulled from a closer location to the user

- The user’s browser may have already cached that version of jQuery on Google, so it won’t need to be downloaded again

To use Google’s version of jQuery (or anything other than the version included with the framework), start by disabling these files in your “clientScript” component:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'clientScript'=>array(
            'scriptMap'=>array(
                'jquery.js'=>false,
                'jquery.min.js'=>false,
            )
        ),
    ),
    // More other stuff.
);
```

Those lines disable the default behavior of having Yii automatically include the local version of jQuery. (While you’re at it, if you’re not using the default Yii template, you ought to set **core.css**, **styles.css**, **pager.css**, and **default.css** to false as well.)

Next, you need to tell Yii what version of jQuery to use. You *could* set the direct URL to the Google hosted version in your configuration file (by assigning that value in place of false). But Yii provides the **CGoogleApi** class for this purpose. Just place it in the HEAD of your layout file:

```
<?php echo CHtml::script(
    CGoogleApi::load('jquery','1.8.3') . "\n" .
); ?>
```

If you’d rather use the latest version of jQuery hosted by jQuery, you could do this:

```
# protected/config/main.php
array(
    // Other stuff.
    'components'=>array(
        'clientScript'=>array(
            'scriptMap'=>array(
                'jquery.js'=>false,
                'jquery.min.js'=>'http://code.jquery.com/
                    jquery-latest.min.js',
            )
        ),
    ),
);
```

```
        ),  
    ),  
    // More other stuff.  
);
```

## Creating a Plan

Now that you've been presented with a whole host of possible performance improvements, how should you proceed? One recommendation I would make before getting into specifics is that you avoid *premature optimization*. The goal while developing a site should be to get it working while also employing best practices. In fact, many best practices have performance benefits anyway. Once the functionality is in place, and the site is almost ready to go live, then you can implement some of these other performance approaches, such as caching.

Here's a step-by-step approach:

1. Develop the site using best practices, completing all necessary functionality and appearance.
2. Test the site extensively, revise and tweak as needed, and get approval from the client (if applicable).
3. Compile all your JavaScript code(including those used by extensions) into a single, minified file.
4. Compile all your CSS code (including those used by extensions) into a single, minified file.
5. Update your layouts to use the new files.
6. Update your configuration to map calls to other JavaScript and CSS files to the one master file.
7. Profile the site, in detail, on its destination server, or an extremely comparable development server.
8. Identify the caching mechanisms available.
9. Enable and configure the caching mechanisms, if applicable.
10. Change the bootstrap file, switching to **yiilite.php** and disabling debugging.
11. Re-profile to ensure that **yiilite.php** is the right file to use.
12. Enable logging and profile logging in your configuration file.
13. Identify, profile, and optimize your database queries. Repeat as needed.
14. Re-profile the site, adjusting as necessary.
15. Profile any specific blocks of code that you might think are problematic. Refactor, re-profile, and repeat as necessary. Use caching where appropriate.
16. Remove your profile blocks and any other profile-specific code (such as that in your configuration file).
17. Test, test, test!

## 18. Deploy!

Note that you'll profile your site in many ways, depending upon what, exactly, you're examining. Sometimes you'll use ApacheBench, other times a browser-based tool, and still other times Yii's built-in profiling tools.

## Chapter 18

# ADVANCED DATABASE ISSUES

Although the book has covered and explained lots of database-related issues when it comes to the Yii framework, there are still some remaining. In this chapter, I'll discuss a smattering of topics that are either more advanced than most, or just more esoteric. Because these are more advanced and unusual conditions, I won't be able to explain all of them in absolute, concrete detail. But I will always provide as much information as possible in terms of the problem and solution.

If there's something you still don't understand related to databases, let me know and I'll see about addressing it in a future revision.

### Database Migration

Whether it's a matter of bug fixes, or adding features, sites are rarely absolutely done, and there's always code that could be updated. Adding code changes can be done with relative ease in Yii, particularly if you're also making use of version control, unit testing, and other techniques that help ensure smooth transitions.

Some site changes, however, require changes to the underlying database, too. Normally these are *additive* changes:

- Adding one or more tables
- Adding one or more columns to existing tables
- Adding one or more indexes to existing tables
- Adding records to existing tables (this is less common)

Obviously you can make these changes merely by executing the proper SQL commands. But that approach has a couple of downsides. First, it's a manual step, disassociated from the application, meaning that you'd have to remember to do this

when going back and forth from development to production servers. Second, with straight SQL commands, there are no inherent ways to revoke changes, or to only implement only part of the proposed changes.

The solution to these problems is to use *database migration*. Database migration is a system for managing database changes: defining changes, applying them, revoking them, and viewing them.

## Setting Up Migration

In Yii, database migration is handled through the command-line `yiic` script. You'll want to use the application-specific `yiic` script found in your **protected** directory. Before continuing, make sure you know how to execute this script:

```
cd /path/to/site/protected  
yiic <command>
```

(Depending upon your operating system, setup, and so forth, you may need to use a variation on that specific command.)

Second, you should create a **migrations** folder, if one does not exist already. That folder also needs to be writable by the Web server:

```
cd /path/to/site/protected  
mkdir migrations  
chmod 0777 migrations
```

Third, make sure that your **config/console.php** file is configured for your database. Note that this is specifically the *console* configuration file, as that's what the command-line `yiic` will use.

## Creating Migrations

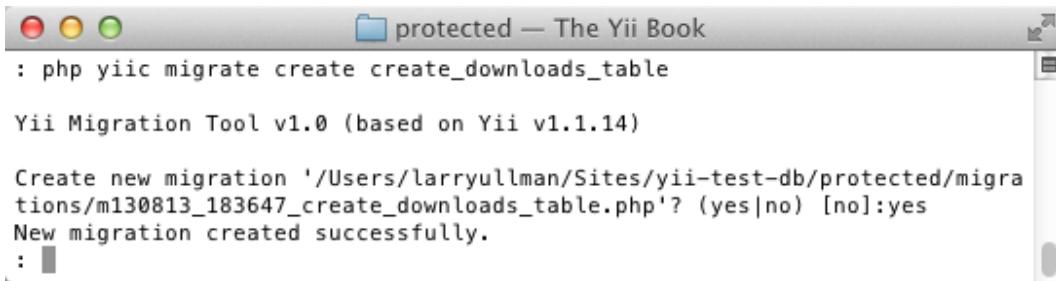
New migrations—database changes—are created using the syntax:

```
yiic migrate create <name>
```

The name should be clearly meaningful, but can only contain letters, numbers, and the underscore (for reasons to be explained shortly).

As an example, let's say that you've later decided to add a `downloads` table to the CMS project to track the number of times a file has been downloaded (**Figure 18.1**):

```
yiic migrate create create_downloads_table
```



```
: php yiic migrate create create_downloads_table
Yii Migration Tool v1.0 (based on Yii v1.1.14)

Create new migration '/Users/larryullman/Sites/yii-test-db/protected/migrations/m130813_183647_create_downloads_table.php'? (yes|no) [no]:yes
New migration created successfully.
:
```

**Figure 18.1:** Creating a new migration.

Then answer “yes” at the prompt. You should see that the new migration was created successfully.

This command will create a new file within the **migrations** directory. The file will be named **mTIMESTAMP\_NAME**, with the timestamp in the format *yyymmdd\_hhmmss*, and the name coming from the value provided to the **create** command. In my example, this file might be **m130813\_183647\_create\_downloads\_table.php**.

The file defines a class that has the same name as the file itself (in keeping with Yii’s conventions).

The class will already have two methods: **up()** and **down()**. These methods define the actions to be taken when invoking the migration and revoking it, respectively. Two other methods will be present, but commented out:

```
<?php
class m130813_183647_create_downloads_table
    extends CDbMigration {
    public function up() {
    }
    public function down() {
        echo "m130813_152544_create_downloads_table does
not support migration down.\n";
        return false;
    }
/*
// Use safeUp/safeDown to do migration with transaction
public function safeUp() {
}
public function safeDown() {
}
*/}
```

Let’s look at what these methods are and how they’re used.

## Migration Actions

The `up()` method is where the migration changes are executed: creating tables, modifying them, possibly adding records, and so forth. This is not done using the traditional, Yii approaches—Active Record, Query Builder, or Database Access Objects, but instead using the `CDbMigration` class.

If you look at a generated migration class file, you'll see that it extends `CDbMigration`. This class defines a couple dozen methods for interacting with the database. For example, if the migration needs to add an index, you'd use the `createIndex()` method (**Figure 18.2**):

```
#protected/migrations/something.php
public function up() {
    $this->createIndex('login', 'user', 'email,pass');
}
```

createIndex() method		
<pre>public void createIndex(string \$name, string \$table, string \$column, boolean \$unique=false)</pre>		
<b>\$name</b>	string	the name of the index. The name will be properly quoted by the method.
<b>\$table</b>	string	the table that the new index will be created for. The table name will be properly quoted by the method.
<b>\$column</b>	string	the column(s) that should be included in the index. If there are multiple columns, please separate them by commas. The column names will be properly quoted by the method.
<b>\$unique</b>	boolean	whether to add UNIQUE constraint on the created index.

**Source Code:** [framework/db/CDbMigration.php#373](#) ([show](#))

Builds and executes a SQL statement for creating a new index.

**Figure 18.2:** The method definition for creating indexes.

That's all there is to it! When the migration is run, the `up()` method will be executed, and that line will create an index named “login” on the `email` and `pass` columns of the `user` table.

The corresponding `down()` method would look like this:

```
#protected/migrations/something.php
public function down() {
    $this->dropIndex('login', 'user');
}
```

To add a column, use the `addColumn()` method:

```
$this->addColumn('someTable', 'someCol', 'string');
```

Yii actually accepts conversational strings for the column's type:

String	Meaning
pk	INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY
string	VARCHAR(255)
text	TEXT
integer	INT(11)
boolean	TINYINT(1)
float	FLOAT
decimal	DECIMAL
datetime	DATETIME
timestamp	TIMESTAMP
time	TIME
date	DATE
binary	BLOB

With that in mind, the above `addColumn()` example actually creates a `VARCHAR(255)`. (Note that this assumes you're using MySQL; with other database applications, the results may differ.)

If you want to add additional characteristics, you can:

```
$this->addColumn('someTable', 'someNum',
    'int UNSIGNED NOT NULL');
$this->addColumn('someTable', 'someTS',
    'timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP');
```

The corresponding `down()` method would contain:

```
$this->deleteColumn('someTable', 'someCol');
```

The `createTable()` method is used to make a new table. Its first argument is the table name; its second is an array of column names and tables; and, its third is for any additional SQL, such as setting the storage engine or adding indexes:

```
$this->createTable('downloads', array(
    'id' => 'pk',
    'file_id' => 'int UNSIGNED NOT NULL',
    'count' => 'int UNSIGNED NOT NULL',
    'last_downloaded' => 'timestamp DEFAULT CURRENT_TIMESTAMP'
));
```

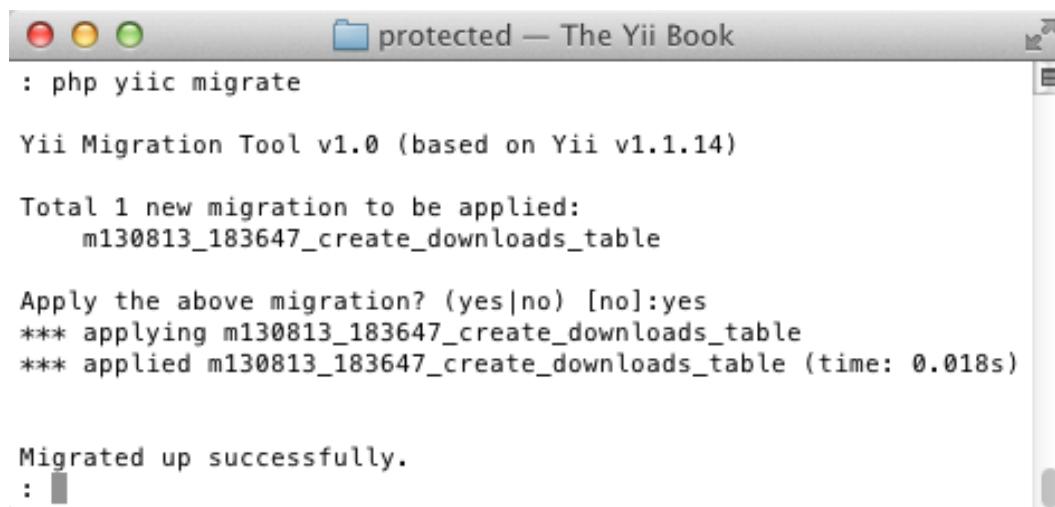
If you wanted, you could then add a foreign key constraint (using the `addForeignKey()` method). There are also methods for renaming columns, inserting records, updating records, deleting records, truncating tables, and more. See the documentation for the `CDbMigration` class for details.

As an added measure, you can perform transactional migrations. For example, you would only want to add foreign key constraints or populate a table if a sequence of commands could successfully be completed. To perform transactional migrations, use the `safeUp()` and `safeDown()` methods of the class instead of `up()` and `down()`. Not all database applications support transactions, though, and some database actions cannot be revoked anyway. But see the [Yii documentation](#) if you're curious about this feature.

## Implementing Migrations

Once you've defined a migration, you implement it—execute its `up()` method—using the `yiic migrate` command (**Figure 18.3**):

```
yiic migrate
```



The screenshot shows a terminal window titled "protected — The Yii Book". The command `: php yiic migrate` is entered. The output shows the Yii Migration Tool v1.0 (based on Yii v1.1.14) identifying 1 new migration to be applied: `m130813_183647_create_downloads_table`. It then prompts "Apply the above migration? (yes|no) [no]:yes" and shows the migration being applied with timestamps. Finally, it displays "Migrated up successfully.".

```
: php yiic migrate

Yii Migration Tool v1.0 (based on Yii v1.1.14)

Total 1 new migration to be applied:
    m130813_183647_create_downloads_table

Apply the above migration? (yes|no) [no]:yes
*** applying m130813_183647_create_downloads_table
*** applied m130813_183647_create_downloads_table (time: 0.018s)

Migrated up successfully.
:
```

**Figure 18.3:** Applying migrations.

That command will show you a list of new migrations to apply (it creates, and uses, a `tbl_migration` table in the database to track this information). Just type “yes” and press Return/Enter to do so.

The migrations will be executed in timestamp order, from oldest to youngest.

If you have a lot of migrations cued up and only want to implement a subset of them, you can limit what migrations are invoked in a couple of ways.

One option is to use the `up` command, followed by a number, to indicate how many of the available migrations (from oldest to newest) should be applied:

```
yiic migrate up 1
```

Or you could tell Yii to migrate up until (and including) a specific timestamp:

```
yiic migrate to 130813_152544
```

To view the list of new migrations available, execute:

```
yiic migrate new
```

To view the list of migrations that have already been executed, use:

```
yiic migrate history
```

Both the `history` and `new` commands take numeric limit arguments restricting how many implemented or upcoming migrations are listed.

If you use migrations a lot, you may want to look at the Yii Guides instructions for [customizing the migration command](#).

## Reverting Migrations

If it was a mistake to implement a migration, you *may* be able to revert it. The actual revocation would be defined in the migration's `down()` method. That method could drop a table, column, or index that was created by the corresponding `up()` method.

To execute the `down()` method, use the `down` command:

```
yiic migrate down
```

That command revokes the previous migration. If you pass a numeric argument after `down`, that many previous migrations will be revoked. The undoing of migrations is in reverse chronological order: from most recent (i.e., newest) to the oldest.

Understand that not all migrations can be undone, although that's really up to you. By definition, if the associated migration class's `down()` method returns false, as in the default generated code, the migration is considered to be un-revokable. A common example would be a migration that has subtractive results: removes data by deleting records, removes columns, or drops tables. Although you can recreate columns and tables, there's no way to repopulate removed data.

If something didn't quite go right with a migration, you can edit the migration class file and then redo the migration:

```
yiic migrate redo 1
```

That line first revokes the most recent migration and then re-implements it.

## Calling Stored Procedures

A topic not previously covered but worth a couple of pages are *stored procedures*. There are huge benefits to using stored procedures, including improved security and performance. On the other hand, not everyone is comfortable with using stored procedures, and many cheaper hosting companies won't let you use them anyway. But if you are comfortable with and capable of using stored procedures, rest assured that you can use them in your Yii-based site, too. In fact, it's easier to use stored procedures in Yii than you might think.

To call a stored procedure in MySQL from a regular PHP script, you'd execute a query along the lines of `CALL procedure_name(arg1, arg2)`. This assumes that the stored procedure `procedure_name` has already been created in the database, of course.

Therefore, to call a stored procedure from Yii, you'd execute that same query. To execute queries directly, you use Direct Access Objects:

```
$q = 'CALL procedure_name(arg1, arg2)';
$cmd = Yii::app()->db->createCommand($q);
$cmd->execute();
```

Naturally, you'll want to toss in bound parameters to pass values to the call:

```
$q = 'CALL procedure_name(:arg1, :arg2)';
$cmd = Yii::app()->db->createCommand($q);
$cmd->bindParam(':arg1', $some_var, PDO::PARAM_STR);
$cmd->bindParam(':arg2', $other_var, PDO::PARAM_INT);
$cmd->execute();
```

Those are examples of *inbound parameters*. If you're using *outbound parameters* in your stored procedure, there's an extra step required. In this next example, the stored procedure will (theoretically) store some information in the `@val` SQL variable. The PHP script then just needs to select this variable in order to access the value:

```
$q = 'CALL procedure_name(:arg1, :arg2, @val)';
$cmd = Yii::app()->db->createCommand($q);
$cmd->bindParam(':arg1', $some_var, PDO::PARAM_STR);
```

```
$cmd->bindParam(':arg2', $other_var, PDO::PARAM_INT);
$cmd->execute();
$result = Yii::app()->db->createCommand('SELECT @val')
    ->queryScalar();
// Use $result
```

And that's all there is to that!

Again, stored procedures are just a somewhat unique type of query that you'd execute using DAO. Other than that, most of this syntax and behavior comes from PHP's PDO, so see those documentation pages if you have any questions. And also see the MySQL documentation if you're unfamiliar with stored procedures in general.

## Using Complex Keys

For most database applications, related tables have a simple relationship between one primary key column in Table A and one foreign key column in Table B. Once you've identified the relationship in your models, Yii can easily pull together related data as needed.

Sometimes you'll have more complex relationships, normally because the primary or foreign keys are comprised of more than one column. These are called either *composite* or *compound* keys, depending upon the nature of the underlying keys. Yii is capable of supporting these situations, you just need to tell Yii more about the relationships.

In Chapter 5, “[Working with Models](#),” I explained how to identify standard relationships:

```
# protected/models/Comment.php
public function relations() {
    return array(
        'page' => array(self::BELONGS_TO, 'Page', 'page_id'),
        'user' => array(self::BELONGS_TO, 'User', 'user_id'),
    );
}
```

In that chapter, I also explained how to use intermediary tables:

```
# protected/models/Page.php::relations()
return array(
    'files' => array(self::MANY_MANY, 'File',
        'page_has_file(page_id, file_id)'),
)
```

That code applies in situations where you have a many-to-many relationship between Table A and Table B (such as `page` and `file` in the above). The intermediary Table C (e.g., `page_has_file`) has a compound primary key but Table A and Table B have single column foreign keys.

In short, although there's a bit of complexity in that database design, it's relatively not that complex.

Let's now look at how you handle situations where you either have complex foreign keys or complex primary keys (but aren't using an intermediary table).

{TIP} Some people distinguish between compound and composite keys, others don't. In theory, a compound key contains two or more columns, all of which relate to other tables. A composite key contains two or more columns, but at least one column is not related to another table.

## Complex Primary Keys

As an example of a composite primary key, let's say you have an application that tracks when a site's users have accessed site content (e.g., pages), and perhaps even what rating the user gave that page:

The `reading` table has compound primary key: the combination of the user ID and the page ID:

```
CREATE TABLE reading (
    user_id INT(10) UNSIGNED NOT NULL,
    page_id INT(10) UNSIGNED NOT NULL,
    PRIMARY KEY (user_id, page_id)
);
```

(Note that this design assumes that only a visited/not visited state is being recorded, not the number of times a user has visited a specific page.)

The `Reading` model needs to implement the `primaryKey()` method to reflect its compound primary key:

```
# protected/models/Reading.php
public function primaryKey() {
    return array('user_id', 'page_id');
}
```

This overrides the default `primaryKey()` method, which returns `id` as the primary key column. But this is all you need to do to identify a complex primary key in a model.

The relations in all the models would still be defined in the standard ways, as a single column in the `user` and `page` tables relates to a single column in the `reading` table.

## Complex Foreign Keys

Complex foreign keys are trickier than complex primary keys. Complex foreign keys occur in situations wherein more than one column in Table B refers to more than one column in Table A.

For example, let's say you have a database about cars. There's the `make` table, which stores "Audi", "Ford", "Honda", and so forth:

```
CREATE TABLE carmake (
    id INT(10) UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    make VARCHAR(30) NOT NULL,
    UNIQUE (make)
);
INSERT INTO carmake (make) VALUES ('Audi'), ('Ford'), ('Honda');
```

Then there's the `carmodel` table:

```
CREATE TABLE carmodel (
    make_id INT(10) UNSIGNED NOT NULL,
    model VARCHAR(30) NOT NULL,
    PRIMARY KEY (make_id, model)
);
INSERT INTO carmodel VALUES (1, 'TTS Roadster'), (2, 'Fusion'),
(3, 'Fit'), (2, 'Focus'), (3, 'Civic');
```

This table relates to `carmake` through the `make_id`. The `model` values would be like "TTS Roadster", "Fusion", "Fit", etc. The `carmodel` table's primary key would be a composite of the `make_id` and the `model`.

*{NOTE}* You would also add a foreign constraint to the `model` table, naturally.

Personally, I don't end up with complex foreign keys often, as I prefer to create surrogate primary keys, which always leaves me with a simple integer as a unique reference point. In other words, in the above design, I would create an `id` field in `model` that would act as a unique identifier.

But if you do prefer the design I'm laying out here, then the next table, for representing individual cars or years of release would have a complex foreign key to `carmodel`:

```
CREATE TABLE car (
    make_id INT(10) UNSIGNED NOT NULL,
    model VARCHAR(30) NOT NULL,
```

```
release_year YEAR,  
PRIMARY KEY (make_id, model, release_year)  
)
```

In fact, not only is the combination of `make_id` and `model` a foreign key reference to `carmodel`, but this table would have a composite primary key comprising `make_id`, `model`, and `release_year`.

Although you can work with complex keys in Yii, the Gii CRUD generator does not handle complex keys itself (**Figure 18.4**); you'll need to set these relations manually in your models.

## Crud Generator

This generator generates a controller and views that implement CRUD operations for the specified data model.

*Fields with \* are required. Click on the highlighted fields to edit them.*

**Model Class \***

Car

Table 'car' has a composite primarykey which is not supported by crud generator.

**Figure 18.4:** Gii cannot scaffold tables with complex keys.

To acknowledge this complex relationship, you'll need to use all applicable columns in the relations definition:

```
# protected/models/Carmodel::relations()  
return array(  
    'make' => array(self::BELONGS_TO, 'Carmake', 'make_id',  
    'cars' => array(self::HAS_MANY, 'Car', 'make_id, model')  
)
```

And:

```
# protected/models/Car::relations()  
return array(  
    'model' => array(self::BELONGS_TO, 'Carmodel', 'make_id, model')  
)
```

That's all you need to do, although be certain to list the columns in the same order in which they appear in the tables.

## Validating Complex Keys

An added complication appears when you go to add validation with complex keys. For example, you should not be able to add a record to the `car` table unless the `make_id` and `model` combination exists in `carmodel`. Or, in the hypothetical `reading` table, the combination of the `user_id` and `page_id` would have to be unique.

In some situations, you can customize the “unique” and “exists” validators to suit your needs. Otherwise, you might be better off using one of the available complex key validation extensions you can find online.

## Using Complex Relationships

Next, let’s move from complex keys to complex relationships. There are two more advanced relationship topics to cover. The first is using “through” relationships. The second addresses how one handles inheritance.

### Defining Through Relationships

Sometimes you’ll have two models (and database tables) that have an indirect, but meaningful relationship to each other. For example, in the CMS example, users post comments and pages have comments, but users and pages are not directly related. What if you wanted to easily find what pages a user has commented on?

In such situations, one option would be to go through the existing relationships. For example, you could do a `find()` on `User`, and toss in a `with('comments')` and then perform lazy loading from the comments results to get the pages. That will work, but it’s a lot of extra work and overhead.

The alternative is to set up a “through” relationship. Through relationships create a relationship between two not-directly-related models, through a third model. Here’s how that would be defined for the `User` example:

```
# protected/models/User.php::relations()
return array(
    'pages' => array(self::HAS_MANY, 'Page',
        array('page_id' => 'id'),
        'through'=>'comments'
    );
}
```

In the third argument (the array), the index value is the column in the “through” relationship that equates to this model’s primary key. In other words, `comments.page_id` connects to `page.id`.

With that defined in `User`, you can now perform the query using:

```
User::model()->with('pages')->findByPK();
```

You can use “through” whenever a HAS\_MANY or HAS\_ONE relationship exists between the two models being connected. Also keep in mind that this is only meaningful when using Active Record. You can also always get the same results using Query Builder or Direct Access Objects.

## Addressing Inheritance

The examples in the book to this point could all be described as flat, or single-layer databases. That’s to say that all of the tables are related to each other (if at all) on the same level. No tables reflect an inheritance relationship.

A common enough question I see is: how do you handle database inheritance in Yii? You’ve seen class inheritance multiple times over by now, but how do you translate database inheritance into models? Most relational databases don’t support inheritance, but your models do. The models, though, are also already inherited from Active Record. So how do you proceed?

As an example, let’s say the application is dealing with pets in some regard. From a code perspective, you’d want an `Pet` model that would have the generic properties and methods that apply to all animal types: `name`, `age`, `eat()`, `sleep()`, etc. Then you’d have individual animal classes that extend `Pet`: `Dog`, `Cat`, etc.

This logical approach results in what’s called an [object relational impedance mismatch](#). What this means is that that models exist for which there is no underlying database table. In particular, the `Pet` model doesn’t correlate to a `pet` table (because you wouldn’t store records of a generic pet type).

The best solution here is to implement *single table inheritance*: put all the children in a single table, but use a column to differentiate among them. This is an appropriate solution when the parent table or class would not have any records or be instantiated on its own, as in the pets example.

In that example, create a `pets` table:

```
CREATE TABLE pet (
    id INT(10) UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    type ENUM('dog', 'cat'),
    /* Other columns */
);
```

Then model it:

```
class Pet extends CActiveRecord {
}
```

The other models will extend `Pet`:

```
class Dog extends Pet{  
}
```

This can give you the basic inheritance outlined in the models. But it gets a bit trickier since Active Record uses the table and class names extensively.

By default Active Record assumes the table name is the same as the class name. This means that the code `Dog::findAll()` will attempt to use the `dog` table, which does not exist. The first thing you need to do then is to tell Yii to always use the `pet` table:

```
class Pet extends CActiveRecord {  
    public function tableName() {  
        return 'pet';  
    }  
}
```

Because `Dog`, `Cat`, and the others extend `Pet`, they'll all have this same method. (If your `Dog`, `Cat`, and other models have their own `tableName()` methods, you'll want to remove those.)

Next, you need to also tell Yii what class is being used for `find()` and the other methods that use model instances. This is done by overriding the `instantiate()` method in `Pet`. The method should now return a different class name based upon the `type` attribute of the provided model:

```
class Pet extends CActiveRecord {  
    protected function instantiate($attributes){  
        switch($attributes['type']){  
            case 'dog':  
                $class='Dog';  
                break;  
            case 'cat':  
                $class='Cat';  
                break;  
            default:  
                $class=get_class($this);  
                break;  
        }  
        $model=new $class(null);  
        return $model;  
    }  
}
```

Now, whenever you use the code `Dog::find()` or `Cat::find()` or whatever, the proper, specific model will be used, not the generic `Pet`.

Finally, you'll want to create default scopes in all the children (`Dog`, `Cat`, etc.) so that all `SELECT` queries only apply to the correct type:

```
class Dog extends Pet{
    public function defaultScope() {
        return array('condition' => 't.type="dog"');
    }
}

class Cat extends Pet{
    public function defaultScope() {
        return array('condition' => 't.type="cat"');
    }
}
```

(The table alias `t` is used in the condition to avoid ambiguity issues if you were to reference the same `pet` table multiple times in a single query.)

That's all you should need to do to get single table inheritance working on your site. This is a relatively clean and easy solution.

Unfortunately, clean and easy is not always possible. In particular, in situations with multiple levels of inheritance, a single table design won't work. It is technically possible to implement multiple table inheritance in Yii, although it's not easy considering how Active Record works. The fact is that I could spend ten pages on the subject, still not create code that's applicable to all situations, and that wouldn't work for all possible Active Record uses.

If you do have a hard requirement for multiple table inheritance, I'd recommend you read [this blog post](#), which provides a nice overview of the issue and possible solutions. The post also has links to two of the best Yii forum threads on the subject.

## Chapter 19

# EXTENDING YII

The Yii framework is fairly complete in its own right, and through the extensions made available by others, you might be able to go a long time before you hit a development wall. But should that day come, Yii is easily extendible by anyone with decent Object-Oriented Programming skills and comfort with the framework. Extensions in Yii are simply that: the creation of additional functionality not inherently built into the framework.

The term “extension” can be used in a couple of ways. Most strictly, an extension in Yii extends the core framework functionality but are *intended to be shared with third parties*. In other words, instead of writing a chunk of code that will help you out on one or two projects, a Yii extension is also a way to share the extremely useful code that you’ve come up with.

Although that’s the normative use of the term “extension” in Yii, in this chapter I am also going to talk about what could be loosely described as “custom classes”. In fact, the application template created by the `yiic` command already defines a couple of these, such as the `Controller` and `UserIdentity` classes (found within **protected/components**).

In this chapter, I’ll explain everything you need to know to create, use, and even deploy (for the use of others) your own extensions or custom classes.

## Fundamental Concepts

Before getting into the particulars of creating an extension, let’s look at the fundamental concepts when it comes to the subject:

- Guidelines
- Structure
- Publishing assets
- Types of extensions

## Guidelines

The purpose of an extension is not just to add needed functionality (including changing existing functionality), but to do so in a manner that's highly and easily reusable. Moreover, a complete extension is easily reusable not just by you, but by other Yii developers as well. Towards that end, there are many guidelines in place for writing extensions. Note that these are not absolute rules, just good and logical recommendations:

- Give your extension a unique but meaningful name
- If your name could possibly conflict with another extension name, use a unique but meaningful prefix (as a tip, use the same prefix for every extension you create)
- Write the extension to be self-contained, managing its own external dependencies (i.e., the user should not have to download additional packages to use your extension, as a general rule)
- All extension files should go within an extension directory that uses the same name as the extension itself (but in all lowercase letters)
- Extension classes should be named with a prefix to avoid collusion issues with other classes (again, your own unique prefix would be good here)
- Include good, reliable installation and configuration instructions
- Provide good documentation for usage, FAQ, and so forth
- Use meaningful version numbers and provide a CHANGELOG file when you update the extension
- Include a license
- Do your best to maintain the extension

*{WARNING}* Don't use "C" for an extension class prefix as that's reserved for Yii's core classes.

*{NOTE}* The Yii community is wonderfully generous. Even if you worry that your work might be amateurish, or too limited in scope, I would highly recommend you consider sharing your extensions with others.

And, of course, your extension should be written under the best practices, with respect to the programming in general and how code is written in Yii.

*{TIP}* For help in picking the license that's most appropriate for you and the extension itself, see an online resource such as [Choose a License](#).

## Structure

Per the extension guidelines, all extension files should go within an extension directory that uses the same name as the extension itself, but in lowercase letters.

So if you're creating the *ArgyleBargle* extension, all of its files would go in the **argylebargle** directory.

Within the extension directory, you'd place your main class in the root folder. In this hypothetical example, that might be the file **ArgyleBargle.php**.

If **ArgyleBargle** is a fairly common term (e.g., you're making a jQuery or database-related extension), you'd probably want to prefix the extension directory and class to distinguish it. I might use **luargylebargle** and **LUArgyleBargle.php**.

Some extensions require only a single class file, meaning that the extension directory contains only that. Other extension types have their pieces appropriately divided into multiple subdirectories:

- **extDir/assets**
- **extDir/models**
- **extDir/views**

This is often the case when creating modules, which have an application-like structure, or widgets, which have view files.

Regardless of the complexity of the extension, you should also be in the habit of including:

- A README file
- A LICENSE file
- A CHANGELOG file

These files should be in plain text or Markdown format.

## Publishing Assets

More complex extensions, particularly widgets and modules, often make use of resources that need to be in a public directory: CSS, JavaScript, or media. But extensions generally go in non-public directories, such as **protected/extensions**. And the browser should be prevented from loading any resources from the extension's directory, as browsers shouldn't have access to **protected** at all. This is an important security feature, that can be taken further by moving the entire **protected** directory out of the web root.

Because an extension might require publicly accessible resources, the Yii framework provides the **CAssetManager** class. This tool will handle the copying of resources from a non-public directory to the public **assets** directory.

{NOTE} By default the resources will be copied to the **webroot/assets** directory, but this can be changed, if needed.

Assets are copied by invoking the `CAssetManager` class's `publish()` method. The first argument to this method is the path to the file(s) to be copied. Your primary extension class will normally invoke this method in a constructor or initialization method:

```
public function init() {
    $source = dirname(__FILE__) . '/assets/filename.css';
    Yii::app()->assetManager->publish($source);
```

By default, all published assets will go within an `assets` subdirectory, with a simple hashed name like `8e98dfc4`. This name is based upon the basename of the file(s) being copied. If you set the second argument of the `publish()` method to true, a common assets subdirectory can be used by multiple extensions.

```
public function init() {
    $source = dirname(__FILE__) . '/assets/filename.css';
    Yii::app()->assetManager->publish($source, true);
```

*{WARNING}* The `assets` directly must be writeable by the web server, but that is the default expectation.

If the asset being published is a single file, as in the above, it will be placed within the assets subdirectory (e.g., `assets/8e98dfc4/filename.css`). If the asset being published is a folder of files, then the files will be moved to an assets subdirectory:

```
public function init() {
    $source = dirname(__FILE__) . '/assets/css';
    Yii::app()->assetManager->publish($source);
```

Assuming the extension's `css` directory had both `style.css` and `print.css`, those two files would end up as `assets/8e98dfc4/style.css` and `assets/8e98dfc4/print.css` (not `assets/8e98dfc4/css/style.css`). If you have a nested directory structure that you would like to copy intact (**Figure 19.1**), you'd use this code:

```
public function init() {
    $source = dirname(__FILE__) . '/assets';
    Yii::app()->assetManager->publish($source);
```

When the assets are published, the structure will be maintained (**Figure 19.2**).

The third argument to `publish()` is an indicator of what level of copying is performed. By default, every subdirectory and file is copied. This equates to an argument value of -1. If you use the value 0, only the immediate files in the named

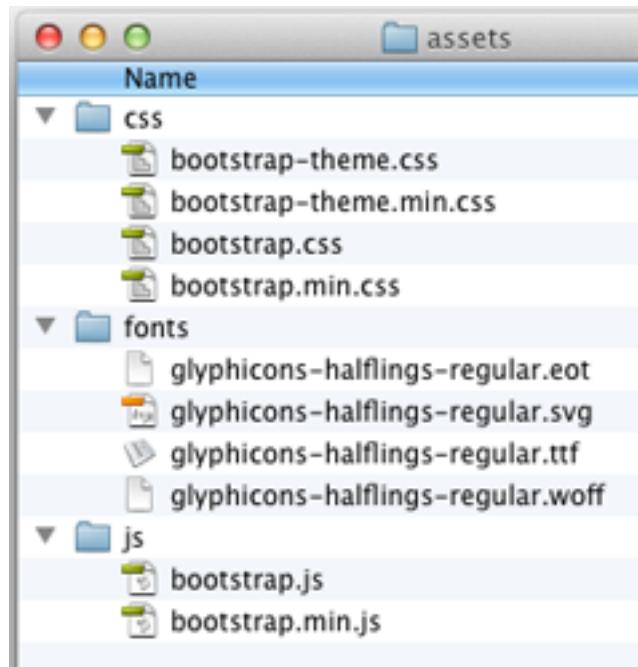


Figure 19.1: An extension's collected assets.

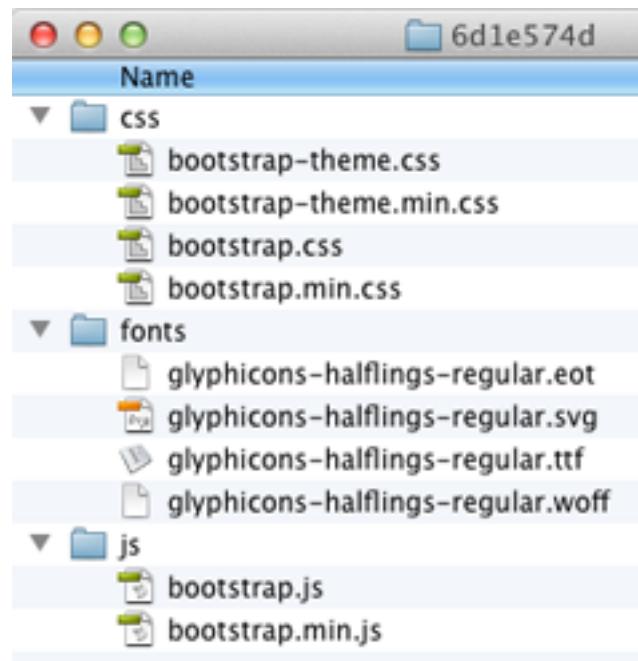


Figure 19.2: The published assets.

directory will be copied. If you use an N value, where N is any integer, that number of files and subdirectories will be copied.

The ability to specify how recursive of a copy to perform allows you to copy, for example, the CSS directory without copying the LESS directory you've also included with the extension. Or, as another example, the JavaScript file could be copied without copying the unit tests directory or un-minified JavaScript files.

When publishing a subdirectory with limited recursion, you'll likely need to invoke `publish()` more than once:

```
public function init() {
    $base = dirname(__FILE__) . '/assets/';
    $am = Yii::app()->assetManager;
    $am->publish($base . 'css', false, 0);
    $am->publish($base . 'js', false, 0);
```

The fourth and final argument to `publish()` is a Boolean indicating whether existing files should be overridden. When you're developing a new extension, and changing the files frequently, you'd likely want to set this to true:

```
public function init() {
    $source = dirname(__FILE__) . '/assets';
    Yii::app()->assetManager
        ->publish($source, false, -1, true);
```

However, this setting adversely affects performance, and would not be desired on a live, distributed extension.

With the default values, if the named asset is a file and it already has been published, its modification time will be checked before re-publishing. If the asset is a directory and the fourth argument is false, the asset manager will only check for the existence of the directory (no recursive check will be performed). If the fourth argument is true, every file and directory will be checked.

*{WARNING}* Never delete a file from an assets subdirectory, as the assets manager doesn't check the detailed contents of a directory when deciding whether or not to republish assets.

When you publish assets, you'll need to know where they ended up in order to create proper references to them after the fact. The `publish()` method will return the new path to the asset. This value can be stored in an internal variable for later use:

```
class MyExt {
    private $_assetsUrl;
    public function init() {
        $source = dirname(__FILE__) .
            '/assets/filename.css';
        $this->_assetsUrl = Yii::app()
            ->assetManager->publish($source);
    }
}
```

Now, other methods within the class can reference `$this->_assetsUrl`. Understand that even though the asset manager won't republish assets if no changes have been made (unless you pass a fourth argument of true), the `publish()` function will still return the used path for the extension's assets.

The final thing to note is that you'll need to tell the application to include your CSS or JavaScript file. This is done using the `registerCssFile()` and `registerScriptFile()` methods. You'd call either after copying the assets:

```
public function init() {

    // Set the sources:
    $source = dirname(__FILE__) . '/assets/';
    $css = $source . 'css/bootstrap.css';
    $js = $source . 'js/bootstrap.js';

    // Publish them:
    $am = Yii::app()->assetManager;
    $cssUrl = $am->publish($css);
    $jsUrl = $am->publish($js);

    // Register them:
    $cs=Yii::app()->clientScript;
    $cs->registerCssFile($cssUrl);
    $cs->registerScriptFile($jsUrl);

}
```

## Extension Types

The particulars of any extension will be dependent upon the *type* of extension being created. Fundamentally, it's a question of what role the extension will play in a Yii application, and therefore, from what class the extension's primary class will be derived (if any).

Save for modules, I will next walk through the basic types of extensions, and outline how they're written, before delving into some examples of specific types. Due to the more complex nature of modules, I'll discuss and demonstrate creating a module later in the chapter.

## Application Components

An application component extension serves the same purpose as the application components built into the Yii framework, such as the URL manager, the asset manager, the caching component, the session component, and so forth. This is where you implement specific functionality that a website needs that's not particular to any model. Application components are particularly good for making code or data available to every aspect of an application; controllers, models, and views can all readily access application components.

Application components must implement the **IApplicationComponent** interface or extend the **CApplicationComponent** class:

```
class MyAppComponent extends CApplicationComponent {  
}
```

Within the class, you can define any attribute or method you want:

```
class MyAppComponent extends CApplicationComponent {  
    public $var = false;  
    private $_thing = 42;  
    public function getThing()  
    {  
        return $this->_thing  
    }  
}
```

Once defined, you need to register the component with the application. This is done in the primary configuration file:

```
# protected/config/main.php  
// Other stuff.  
'components' => array(  
    'myappcomponent' => array(  
        'class' => 'ext.myappcomponent.MyAppComponent',  
    ),  
)
```

*{TIP}* The path reference `ext` equates to `protected/extensions` (or wherever your extensions directory is).

That code tells the application that the `myappcomponent` component is an instance of the `MyAppComponent` class. (And the code assumes that the `MyAppComponent.php` file is within the `protected/extensions/myappcomponent` directory.)

With the component registered, you can access an instance of that class using `Yii::app()->myappcomponent`:

```
Yii::app()->myappcomponent->var = true;  
Yii::app()->myappcomponent->getThing();
```

Application components are often configurable, affecting how they behave. For example, the user component can be told to allow for auto login (i.e., “remember me” functionality):

```
# protected/config/main.php  
// Other stuff.  
'user'=>array(  
    // enable cookie-based authentication  
    'allowAutoLogin'=>true,  
,  
// More stuff.
```

The configuration names and values just correspond to public properties of the underlying class (or “setter” methods, if you’ve gone that route). To make your application configurable, just add public variables such as `$var` in the trivial example above. Then assign a value to that variable in your configuration:

```
# protected/config/main.php  
// Other stuff.  
'components' => array(  
    'myappcomponent' => array(  
        'class' => 'ext.myappcomponent.MyAppComponent',  
        'var' => true,  
,  
,
```

Do be certain to give your public attributes default values, however, so that a failure to configure the value does not result in errors.

A final consideration for application components is whether there’s any initial setup or other work to be performed before the component can be used. For example, does a database connection need to be made? If there is something to be done, define an `init()` method that performs any initialization work. Your component’s `init()` method should also call `parent::init()` to make sure the inherited initialization work is also performed.

## Widgets

Widgets are a specific, and common, type of extension used to add complex functionality to view files without embedding a lot of code and logic. Yii comes with a number of widgets built-in, as described in Chapter 12, “[Working with Widgets](#).“ The built-in widgets:

- Present loads of information in grid format
- Represent jQuery UI components
- Present individual records in a nice format
- And more

A widget must extend the `CWidget` class (or an existing child class of `CWidget`). For example, to create your own variation on the Yii `CCaptcha` widget, you could just extend that class and override its methods, or change its default behaviors, to suit your needs.

When creating a brand-new widget, you extend the `CWidget` class and must define at least the `init()` and `run()` methods. The `init()` method will be called when `$this->beginWidget()` is used in a view file to create the widget instance. The `run()` method is called when the view file invokes `$this->endWidget()`:

```
# protected/extensions/mywidget/MyWidget
class MyWidget extends CWidget {
    public function init() {
        // Do whatever to start.
    }
    public function run() {
        // Do whatever to end.
    }
}
```

```
# protected/views/controllerID/view.php
<?php $this->beginWidget('ext.mywidget.MyWidget'); ?>
<?php $this->endWidget(); ?>
```

Some widgets, such as `CActiveForm` expect you to create data between the `beginWidget()` and `endWidget()` method calls (in the case of `CActiveForm`, the data created are form elements). The data placed between those method calls is captured by the widget by starting output buffering within the `init()` method and ending it in `run()`:

```
# protected/extensions/mywidget/MyWidget
class MyWidget extends CWidget {
```

```
public function init() {
    // Do whatever to start.
    ob_start();
}
public function run() {
    // Do whatever to end.
    $data = ob_get_clean();
    // Use $data.
}
}

# protected/views/controllerID/view.php
<?php $this->beginWidget('ext.mywidget.MyWidget'); ?>
<p>This data is being captured.</p>
<?php $this->endWidget(); ?>
```

Note that the output buffering captures the data, it does not output it. You'll need to echo the `$data` variable to do that.

If your widget doesn't need to capture any input, it can be created in a view file using just `widget()`:

```
# protected/views/controllerID/view.php
<?php $this->widget('ext.mywidget.MyWidget'); ?>
```

In that case, the `init()` and `run()` methods are still invoked in that order.

{NOTE} You always need `init()` and `run()` methods in your widget class.

You can configure how a widget functions when it's created (in the view file) by passing additional arguments to the `beginWidget()` or `widget()` method:

```
# protected/views/controllerID/view.php
<?php $this->widget('ext.mywidget.MyWidget',
    array('name' => 'value')); ?>
```

The names of the array elements must match the public properties defined in the widget class:

```
# protected/extensions/mywidget/MyWidget
class MyWidget extends CWidget {
    public $var1;
```

```
public $var2;
public function init() {
    // Do whatever to start.
}
public function run() {
    // Do whatever to end.
}
}

# protected/views/controllerID/view.php
<?php $this->widget('ext.mywidget.MyWidget',
array('var1' = 42, 'var2' = false); ?>
```

In that example, the public `$var1` property in the class will be assigned the value 42, `$var2` will be assigned false.

If a property value must be assigned to use the widget, you would throw an exception in the `init()` method:

```
# protected/extensions/mywidget/MyWidget.php
class MyWidget extends CWidget {
    public $var1;
    public $var2;
    public function init() {
        // Do whatever to start.
        if ($this->$var1 === null) {
            throw new CException('You must provide a `'$var1`'
                'value.');
        }
    }
    public function run() {
        // Do whatever to end.
    }
}
```

You can also reference the public attributes in the `run()` method, of course, including performing validation there, if you'd like.

Most widgets output HTML. For anything but the very simplest output, you should create your own view file (or files) for the widget. The view files would go within the `views` subfolder of your extension's directory, and be given the name `viewfile.php`, as is the case with any controller's view file. You'd then use the `render()` method within the widget's `run()` method, as you would within a controller.

To put this all together, as a stupid, but comprehensible, example of this, say your `MyWidget` outputs some text within a DIV or a paragraph (which is not a good use of a widget). The view files would be:

```
# protected/extensions/mywidget/views/div.php
<div><?php echo CHtml::encode($output); ?></div>
```

```
# protected/extensions/mywidget/views/p.php
<p><?php echo CHtml::encode($output); ?></p>
```

Use of the widget (in a view file) would be:

```
# protected/views/controllerID/view.php
<?php $this->beginWidget('ext.mywidget.MyWidget',
    array('tag' => 'p')); ?>
This is the output.
<?php $this->endWidget(); ?>
```

Or:

```
# protected/views/controllerID/view.php
<?php $this->beginWidget('ext.mywidget.MyWidget',
    array('tag' => 'div')); ?>
This is the output.
<?php $this->endWidget(); ?>
```

And the widget itself would be defined as:

```
# protected/extensions/mywidget/MyWidget.php
class MyWidget extends CWidget {
    public $tag;
    public function init() {
        ob_start();
    }
    public function run() {
        $output = ob_get_clean();
        if ($this->tag == 'p') {
            $this->render($this->tag,
                array('output' => $output));
        } else {
            $this->render('div',
                array('output' => $output));
        }
    }
}
```

## Controllers

The default Yii template, used when creating a new application, already creates a new base controller class, aptly named **Controller**:

```
# protected/components/Controller.php
<?php
class Controller extends CController {
    public $layout='//layouts/column1';
    public $menu=array();
    public $breadcrumbs=array();
}
```

Although very useful, and a reasonable step to take with any of your own applications, the **Controller** class is not a good candidate as a *distributable* extension. In Yii, if you create a controller intended as an extension, it should actually extend the **CExtController** class, not **CController**:

```
# protected/extensions/mycontroller/MyController.php
class MyController extends CExtController {
}
```

The reason for this difference is that the **CController** class will look within the **protected/views/ControllerID** directory by default when it comes to rendering views. Presumably, with a controller extension, the views will be distributed along with the class file. That puts the view files in, for example, **ext/mycontroller/views**, not **protected/views/ControllerID**. But if you base your controller extension on **CExtController**, calls to **render()** will pull from the **views** directory found with the extension class file.

Other than that, a controller extension is written and used like any other.

## Actions

Actions are controller methods invoked as part of a request. These are methods defined within the controller, whose names start with *action*. For example, the **actionIndex()** method is normally the default method called for every controller (i.e., when no other action is specified).

It's standard for controller actions to be defined within the controller itself, but they can also be defined independently, thereby creating an *action extension*. As with any extension, the benefit of an action extension is reusability: if you have an action whose logic could be used by multiple controllers without change, you could define that as an extension and then tell each controller about it.

Actions must extend the **CAction** class, or an existing child class of **CAction**. Actions must define a **run()** method, which does the bulk of the work:

```
# protected/extensions/myaction/MyAction.php
class MyAction extends CAction {
    public function run() {
        // Do the work.
    }
}
```

To use the action in any controller, you just need to tell the controller about it. This next bit of code tells the `MyController` class to use the `MyAction` class for the `myaction` request:

```
# protected/controllers/MyController.php
class MyController extends CController {
    public function actions() {
        return array(
            'myaction' => 'ext.myaction.MyAction'
        );
    }
    // Other controller code.
}
```

If the action needs parameters—values passed to the action when it's executed, you can do that by creating arguments in the `run()` method:

```
# protected/extensions/myaction/MyAction.php
class MyAction extends CAction {
    public function run($thing) {
        // Do the work.
    }
}
```

When the `myaction` action is executed (associated with a controller), the value of `$_GET['thing']` will now be passed to the `run()` method.

## Filters

Like actions, filters are also used by controllers. Filters perform a task before and/or after an action is executed. For example, the “access control” filter is used to restrict access to the execution of controller actions.

To create a filter extension, you'll need to extend the `CFilter` class (or extend a child of that class). The class should have a `preFilter()` method, which will be executed before the controller action. The class should also have a `postFilter()` method, which will be executed after the controller action. The `preFilter()` method needs

to return a Boolean indicating whether or not the controller action should be allowed to execute. The `postFilter()` method does not need to return anything. Both functions should take one argument, a filter chain.

The filter chain argument allows the filter to be one of many applied to an action. For example, the `actionDelete()` method of a controller may have both the “access control” and the “postOnly” filters applied to it.

Because a filter may be one in a sequence to be executed, you ought to call the `preFilter()` and `postFilter()` methods of the parent class in your filter extension. You can use the returned value as the value to be returned by your methods. Here, then, is the shell of a filter extension:

```
# protected/extensions/myfilter/MyFilter.php
class MyFilter extends CFilter {
    public function preFilter($fc) {
        // Do whatever.
        return parent::preFilter($fc);
    }
    public function postFilter($fc) {
        // Do whatever.
        return parent::postFilter($fc);
    }
}
```

The `filters()` method of the controller is used to apply the filter:

```
# protected/controllers/MyController.php
class MyController extends CController {
    public function filters() {
        return array(
            'accessControl',
            'postOnly + delete',
            'ext.myfilter.MyFilter + index'
        );
    }
    // Other controller code.
}
```

## Behaviors

Behavior extensions provide the ability to add additional functionality to common MVC pieces through an external source. Put another way, behaviors allow you to emulate multiple inheritance by making methods not directly inherited by a class available for use within that class.

For example, a behavior might be defined that adds the ability to smoothly handle checkboxes in any model. As another example, the `TimestampBehavior` will automatically set a model's `create_time` attribute to the current moment when the model is first created. It will also do the same for the `update_time` attribute when the model instance is updated.

The behavior extension must implement the `IBehavior` interface. This is normally accomplished simply by extending the `CBehavior` class. Behaviors intended for specific contexts, such as models in general or Active Record in particular, can extend the more specific `CModelBehavior` or `CActiveRecordBehavior` classes instead:

```
# protected/extensions/mybehavior/MyBehavior.php
class MyBehavior extends CActiveRecordBehavior {
}
```

To attach a behavior to a model, use the model's `behaviors()` method. This method returns an array of behaviors to attach. For each item, provide a behavior name. The name's value is an array. The most important element in this subarray will be the class associated with the added behavior:

```
# protected/models/SomeModel.php
public function behaviors() {
    return array(
        'behaviorName' => array(
            'class' => 'ext.mybehavior.MyBehavior'
        )
    );
}
```

The result is that the model now has access to the methods defined in the behavior as if the methods were defined in the model itself. The main difference being that these methods—the behavior—can be added to *any* model.

As you'll see in a later example, behaviors tied to models often use `beforeSave()` and other event-driven methods to perform steps in conjunction with the life of a model. This is how the `TimestampBehavior` works.

## Validators

Validators are used by models to validate the model's properties. There are dozens of validators built into Yii, but you may need your own custom validator, too. Chapter 5, “[Working with Models](#),” showed how to define a validator within a model. To make a more reusable validator, you could build it as an extension instead.

A validator extension needs to extend `CValidator`. The class must define the `validateAttribute()` method. This method takes two arguments: a model instance and the attribute to validate. The method doesn't need to return anything, but instead adds an error to the method upon failure to validate:

```
# protected/extensions/myvalidator/MyValidator.php
class MyValidator extends CValidator {
    public function validateAttribute($model, $attr) {
        if /* $model->$attr is NOT valid */ {
            $model->addError($attribute, 'Error message');
        }
    }
}
```

And that's all there is to it! To have your model use this validator, add it to the list of rules:

```
public function rules() {
    Yii::import('ext.myvalidator.MyValidator');
    return array(
        // Other rules.
        array('attrName', 'ext.myvalidator.MyValidator'),
    );
}
```

To make your validator configurable, define a public attribute for every configuration option:

```
# protected/extensions/myvalidator/MyValidator.php
class MyValidator extends CValidator {
    public $var1;
    public function validateAttribute($model, $attr) {
        if /* $model->$attr is NOT valid */ {
            $model->addError($attribute, 'Error message');
        }
    }
}
```

Then, when you apply the rule to your model, provide an array of name-value pairs, where each name matches a public attribute:

```
public function rules() {
    return array(
        // Other rules.
    )
}
```

```
        array('attrName', 'ext.myvalidator.MyValidator',
              array('var1' => 'value'))),
    );
}
```

## Console Commands

Console commands are like controller actions meant to be run from a command-line interface. Similarly, you can create a console command as an extension that's distributable and usable in any application.

To create a console command extension, you need to extend the `CConsoleCommand` class. The class's `run()` method will do the actual work. This method can be written to accept one parameter:

```
class MyCommand extends CConsoleCommand {
    public function run($args) {
        // Do whatever.
    }
}
```

The `$args` array will store whatever values were passed as command-line parameters when the command is invoked:

```
yiic mycommand --param1=value
```

To make your command a bit more professional, you can also implement the `getHelp()` command. This method will be invoked when the user types `command-name --help`. The function should return a string, which will be printed:

```
class MyCommand extends CConsoleCommand {
    public function run($args) {
        // Do whatever.
    }
    public function getHelp() {
        return "Usage: mycommand --param"
    }
}
```

To make the command extension available to your application, you need to add it to the `commandMap` property of the console application. This is done in the `console.php` configuration file:

```
# protected/config/console.php
return array(
    'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',
    'name'=>'My Console Application',
    'commandMap' => array(
        'mycommand' => array(
            'class' => 'ext.mycommand.MyCommand'
        )
    ),
// Other stuff.
```

If you want to be able to configure the console command, add public properties to the class and assign values to those properties in the configuration file.

## Other Extension Types

Finally, you may have an extension or custom class that does not fit neatly into any of the listed categories. Remember that in Yii you can extend almost anything you need to extend!

As one example, you might like all of your ActiveRecord models to have certain attributes or methods. To do that, you'd create your own model class based upon **CActiveRecord**:

```
# protected/components/My ActiveRecord.php
class My ActiveRecord extends CActive Record {
    // Do whatever.
}
```

Then you can have all your ActiveRecord models extend **My ActiveRecord** instead of **CActive Record**. The only caveat is that you need to tell Yii about the existence of your class, first.

Yii can (obviously) recognize any class defined within the framework itself. Any other class your application uses must be imported into the framework in order to be usable. There are two ways of doing so.

First, to import a class on a global scale—so that every application element has access to it, use the configuration file:

```
# protected/config/main.php
// Other stuff.
'import'=>array(
    'application.models.*',
    'application.components.*',
```

```
),
// More other stuff.
```

That code, from the standard configuration, automatically imports every class found within the **protected/models** and **protected/components** directories.

A global import is practical when a class might be used in multiple places. When that's not the case, import the specific class (or classes) when needed using:

```
Yii::import('path/to/classfile');
```

## More Examples

With the fundamental concepts of exceptions explained, and a couple of simple examples in place, it's time to look at some more specific and real-world examples. Traditionally, as a writer, I always come up with my own ideas for every example in a book. However, in this case, I'm going to use a combination of new examples, as well as analysis of existing extensions (with due credit given, of course). I'm using existing extensions for a couple of reasons:

1. Lots of great extensions have already been created, and spending time coming up with second-rate examples just for the sake of being new leaves a lot to be desired.
2. This approach shows how you can learn to write your own extensions by looking at existing extensions, now that you know what to look for.

If, when all is said and done with this book, you'd like another example of a specific extension type, let me know and I'll see what I can do.

## Creating Behaviors

Earlier I mentioned the timestamp behavior as a useful concept. This behavior, part of the Zii library and written by Jonah Turnquist, can automatically set the value of model attributes to the current timestamp. The extension is mostly defined like so (some content has been removed for brevity and clarity):

```
<?php
class CTimestampBehavior extends CActiveRecordBehavior {
    public $createAttribute = 'create_time';
    public $updateAttribute = 'update_time';
    public $setUpdateOnCreate = false;
    public $timestampExpression;
```

```
public function beforeSave($event) {
    if ($this->getOwner()->getIsNewRecord() &&
        ($this->createAttribute !== null)) {
        $this->getOwner()->{$this->createAttribute} =
            $this->getTimestampByAttribute($this->createAttribute);
    }
    if ((!$this->getOwner()->getIsNewRecord()
        || $this->setUpdateOnCreate)
        && ($this->updateAttribute !== null)) {
        $this->getOwner()->{$this->updateAttribute} =
            $this->getTimestampByAttribute($this->updateAttribute);
    }
}
```

As you can see, the class extends `CActiveRecordBehavior`, as the behavior is meant to be applied to models that extend `CActiveRecord`. The class has five public attributes, making these configurable options when using the behavior:

```
public function behaviors() {
    return array(
        'CTimestampBehavior' => array(
            'class' => 'zii.behaviors.CTimestampBehavior',
            'createAttribute' => 'create_time_attribute',
            'updateAttribute' => 'update_time_attribute',
        )
    );
}
```

Being a behavior, the most important code is defined within the `beforeSave()` method. Within that method, the model instance that triggered the behavior is available through `$this->getOwner()`. The code is a bit complicated from there (for maximum flexibility), but it simply updates the appropriate attribute of that model, varying whether this is a new record or not, and whether the “`setUpdateOnCreate`” option is true or not.

As another example, in Chapter 9, “[Working with Forms](#),” I discussed how to work with model attributes associated with checkboxes. In particular, you might have an attribute whose value stored in the database is Y/N, which will need to be properly represented by a form checkbox. The catch being when a checkbox is not checked in the form, the resulting value will be 0 (by default). To address this problem, you could make a behavior extension. I’ll intermingle the code and the explanation:

```
<?php

class LUCheckboxBehavior extends CActiveRecordBehavior {
    public $trueValue = 'Y';
    public $falseValue = 'N';
    public $attr = null;
```

The class extends `CActiveRecordBehavior`, of course. Within the class are three public properties, making it configurable. Two properties set the “true” and “false” values as they’re stored in the database. The defaults are Y and N. The third property is the attribute to which the behavior should be applied. The behavior would be used like so:

```
public function behaviors() {
    return array(
        'LUCheckoutBehavior' => array(
            'class' => 'ext.luchCheckoutBehavior.LUCheckoutBehavior',
            'trueValue' => 'Yes',
            'falseValue' => 'No',
            'attr' => 'receiveEmails'
        )
    );
}
```

Next, the behavior has a `beforeSave()` method. Its role is to convert the form data from the default 1/0 to those expected by the database:

```
public function beforeSave($event) {
    $model = $this->getOwner();

    if (($this->attr === null) ||
        !in_array($this->attr, get_object_vars($model)) ) {
        throw new CException('You must indicate a valid model
                            attribute!');
    }

    if ($model->{$this->attr} == 1) {
        $model->{$this->attr} = $this->trueValue;
    } else {
        $model->{$this->attr} = $this->falseValue;
    }
}
```

First, the method gets a reference to the model instance. Next, the method verifies that an attribute value was provided and that it exists within the model instance.

If not, an exception is thrown. Finally, the method converts the values from 1/0 to whatever the user wants.

The behavior also needs to perform a conversion when the record is retrieved (so that the form works properly). That code goes in an `afterFind()` method, and is similar to that in the other method:

```
public function afterFind($event) {
    $model = $this->getOwner();
    if (($this->attr === null) ||
        !in_array($this->attr, get_object_vars($model)) ) {
        throw new CException('You must indicate a valid model
            attribute!');
    }
    if ($model->{$this->attr} == $this->trueValue) {
        $model->{$this->attr} = 1;
    } else {
        $model->{$this->attr} = 0;
    }
}
```

The only difference in this method is that it checks if the existing values match the “true” and “false” values, and assigns 1 or 0 accordingly.

And there you have a functional behavior extension that’s easy to use and quite helpful. Once applied to a model, you’ll never have to think about converting between database and checkbox form values again!

*{NOTE}* If you’re copying this code, you also need a closing bracket to complete the class.

## Creating Widgets

For the next example, let’s create a widget, one of the most common extension types around. From a development perspective, creating a widget requires a decent amount of knowledge and effort, although not so much as a full-blown module. Widgets are used to provide more complex logic, normally including HTML, as a component that’s separate from your own view files.

Instead of trying to come up with a perfect example, that doesn’t yet exist, I’ve decided upon a somewhat decent example that I can use to help explain the general process of creating a widget. The specific example is a widget-based implementation of the [Stripe Checkout](#) form.

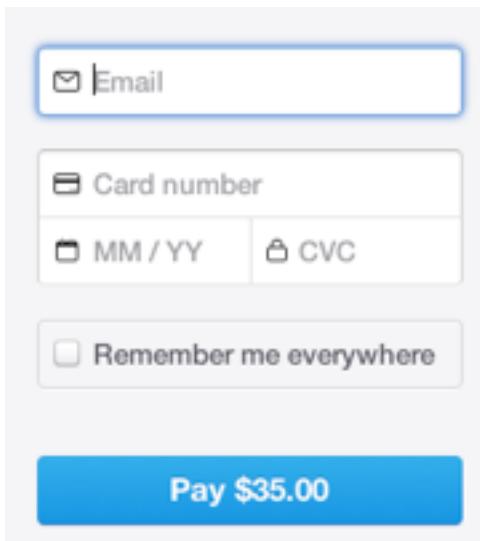
In case you’re not familiar with [Stripe](#) or how its system works, you use a form on your Web site for taking the customer’s payment information (i.e., credit card

details). This data is sent to Stripe via Ajax, so that it never touches your server. Not having the credit card information on your server relieves almost all of the PCI compliance burden.

{NOTE} Stripe is not currently live in every country, but you can test Stripe with any email address, or without creating an account at all.

Stripe takes the customer's credit card information, stores it on its system, and returns a representative token to your site. The form on your site is then submitted to your server (so your server receives the token and any other form data, but not the payment details). The form handling script on your server then actually processes the charge, using the supplied token to represent the card to be used.

Checkout is Stripe's easy and convenient modal form for taking a customer's payment information and securely sending it to Stripe. You don't need to create the form itself or write any JavaScript. Although you can do those things manually (and commonly will), Checkout is a no-brainer implementation (**Figure 19.3**).



**Figure 19.3:** Part of the Checkout modal window

There are two caveats to this widget example. First, as of the summer of 2013, I work for Stripe (as a Support Engineer). But I work at Stripe because I'm such a huge fan of their product, not the other way around.

The second caveat is that this widget only does half of the task: the part that takes the customer's payment information, passes it to Stripe, receives a token in return, and submits it to your server. To complete the implementation of Stripe Checkout, you would need to:

- Download and install the [Stripe PHP library](#)

- Process a charge in your form handling code

That being said, what this widget does do it does just fine, and I think it explains the process of creating widgets well enough. Further, this widget example will make the fuller Stripe example (as a module) a bit easier to understand when that time comes later in the chapter.

## Looking at the End Result

To create this widget, I begin by identifying the desired end result. In the case of Stripe Checkout, the end result will be something like:

```
<form action="/charge" method="POST">
    <script
        src="https://checkout.stripe.com/checkout.js" class="stripe-button"
        data-key="pk_test_APr32Tly9WH6K9XfZpJeEKCH"
        data-image="/square-image.png"
        data-name="Demo Site"
        data-description="2 widgets ($20.00)"
        data-amount="2000">
    </script>
</form>
```

That's from the Stripe documentation. The Checkout JavaScript file is hosted on Stripe's servers, so the widget doesn't need to worry about that.

The questions I must now ask are:

- What information must the widget user provide?
- What information should be optionally configurable?

In the above, the form's "action" attribute is mandatory, although it would make sense to default this to the current page. In terms of Stripe, the "key" value (which identifies your account to Stripe) is required. The other "data-\*" items are optional, along with a couple more mentioned in the documentation.

The combination of the required and optional information will be the widget's properties, configurable on a widget-by-widget basis.

## Defining the Class

Next, it's time to create the widget class. It must extend `CWidget`:

```
<?php
class LUStripeCheckout extends CWidget {
    public $action;
    public $key;
    public $amount;
    public $description;
    public $name;
    public $currency = 'usd';
    public $panelLabel;
    public $billingAddress = false;
    public $shippingAddress = false;
    public $email;
    public $label = 'Pay with Card';
    public $image;
```

As you can see, there are a slew of public properties. A couple of them have default values, primarily the default values used by Stripe Checkout already.

In a real-world (i.e., non-book) class, there would be plenty of documentation included, starting with phpDoc syntax for the class itself and the properties. I would also include an example of the widget's usage in the documentation.

Next, the widget must have two methods: `init()` and `run()`. Here's the first of those:

```
public function init() {
    if ($this->key == null) {
        throw new CException('You must provide your public
            Stripe API key.');
    }
    if (($this->amount == null) ||
        !is_integer($this->amount)) {
        throw new CException('You must provide an amount as
            an integer in cents.');
    }
    if ($this->action == null) {
        $this->action = Yii::app()->request->url;
    }
    if (!Yii::app()->getRequest()->isSecureConnection) {
        throw new CException('This form must be loaded over
            HTTPS.');
    }
}
```

In this widget, the role of the `init()` method is to perform the necessary validation.

To start, the method confirms that a key was provided. If not, an exception is thrown (**Figure 19.4**).

### CException

You must provide your public Stripe API key.

/Users/larryullman/Sites/yii-test-ch19/protected/extensions/lustripecheckout/LUStripeCheckout.php(25)

**Figure 19.4:** A thrown exception.

Next, the amount is not required, but ought to be used, so it's validated. (To be clear, the amount is only required on the server-side.) More importantly, Stripe requires that the amount be provided as an integer in cents, so it makes sense (ha!) to enforce that here (**Figure 19.5**).

### CException

You must provide an amount as an integer in cents.

/Users/larryullman/Sites/yii-test-ch19/protected/extensions/lustripecheckout/LUStripeCheckout.php(29)

**Figure 19.5:** Another exception.

The action value is required, but instead of throwing an exception, if one is not provided, the current page will be used.

Finally, the Checkout form must be loaded over HTTPS in order to be PCI compliant. The method makes a last check for that.

The other method, `run()` has nothing to do other than to render the view file:

```
public function run() {
    $this->render('form');
}
} // End of widget class.
```

Now it's time to define that view file.

### Defining the View File

The view file, named `form.php`, should be placed within the `views` directory of the widget extension's directory. It might look like this:

```
<form action="<?php echo CHtml::encode($this->action); ?>" method="POST">
    <script
        src="https://checkout.stripe.com/checkout.js"
        class="stripe-button"
        data-key="<?php echo CHtml::encode($this->key); ?>
<?php if (!empty($this->image)) echo ' data-image=' . .
    CHtml::encode($this->image) . '''; ?>
<?php if (!empty($this->name)) echo ' data-name=' . .
    CHtml::encode($this->name) . '''; ?>
<?php if (!empty($this->description)) echo ' data-
    description=' . CHtml::encode($this->description) .
    '''; ?>
<?php if (!empty($this->amount)) echo ' data-amount=' .
    CHtml::encode($this->amount) . '''; ?>
<?php if (!empty($this->currency)) echo ' data-currency=' .
    CHtml::encode($this->currency) . '''; ?>
<?php if (!empty($this->panelLabel)) echo ' data-panel-
    label=' . CHtml::encode($this->panelLabel) . '''; ?>
<?php if (!empty($this->billingAddress)) echo ' data    billing-address=' .
    CHtml::encode($this->billingAddress) .
    '''; ?>
<?php if (!empty($this->shippingAddress)) echo ' data-
    shipping-address=' .
    CHtml::encode($this->shippingAddress) . '''; ?>
<?php if (!empty($this->email)) echo ' data-email=' .
    CHtml::encode($this->email) . '''; ?>
<?php if (!empty($this->label)) echo ' data-label=' .
    CHtml::encode($this->label) . '''; ?>
    ></script>
</form>
```

The two required pieces of information—the action and the key—are printed automatically. Everything else is printed only if its value is not empty (**Figures 19.6** and **19.7**). For security, everything value is run through `CHtml::encode()`.

```
<form action="/yii-test-ch19/index.php/site/test" method="POST">
    <script
        src="https://checkout.stripe.com/checkout.js" class="stripe-button"
        data-key="pk_test_APr32Tly9WH6K9XfZpJeEKCH"
        data-amount="3500" data-currency="usd" data-label="Pay with Card" ></script>
</form></div><!-- content -->
```

**Figure 19.6:** The bare minimum source.

```
<form action="/yii-test-ch19/index.php/site/test" method="POST">
    <script>
        src="https://checkout.stripe.com/checkout.js" class="stripe-button"
        data-key="pk_test_APr32Tly9WH6K9XfZpJeEKCH"
        data-amount="2299"
        data-currency="usd"
        data-billing-address="1"
        data-shipping-address="1"
        data-label="Pay with Card"
    </script>
</form></div><!-- content -->
```

Figure 19.7: Different source based upon the provided values.

## Using the Widget

Finally, there's the use of the widget. Assuming you've provided good documentation, using the widget should be very easy. This would go in one of the site's view files:

```
<?php $this->widget('ext.lustripecheckout.LUStripeCheckout',
    array(
        'shippingAddress' => true,
        'billingAddress' => true,
        'key' => 'pk_test_APr32Tly9WH6K9XfZpJeEKCH',
        'amount' => 2299
)); ?>
```

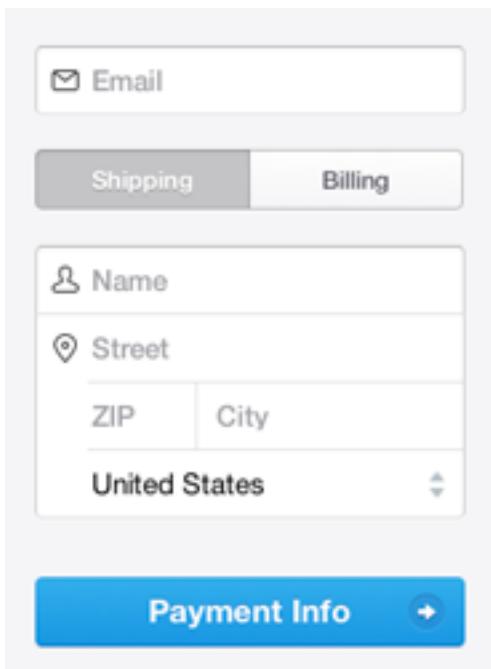
Any other public property can be set in the configuration. Failure to meet the minimum requirements results in an exception being thrown. **Figure 19.8** shows the result of the above code (after clicking the “Pay with Card” button that's also generated).

If you'd like to see what the page receives upon successful checkout, fill out the form using sample data, then dump out the value of `$_POST`.

## Working with Modules

In a way, modules are the easiest extension type to understand. A module is an entire Web application, used as a component of a larger application. A module will have the same core pieces as any application—models, views, and controllers, but be particular to one aspect of a site. For example, your site may have:

- A main area
- Support forums
- A shop



**Figure 19.8:** A custom Checkout integration.

- A blog
- An administrative area

In this example, the main area of your site would be the primary Yii application. The other three areas could each be their own module. The result is a fairly complex and sophisticated site that's still easy to use. Moreover, because you'll have created modules for three of these areas, those modules could be turned into reusable extensions for other projects you do.

The only thing you can't do with a module is deploy it on its own. This is really the primary distinction between a module and an application: modules are always subservient to an application.

To best explain how to create and use modules, let's first walk through the basics and then create a specific example as a module.

## Module Basics

One of the nice things about creating modules in Yii is that the Gii tool will create the shell of the module for you (just as it does the entire application).

## Creating a Module

The first thing you must do, before heading to Gii, is make sure you have a **protected/modules** directory that's writable by the Web server. This is where Gii will attempt to create the module.

Once you've created that,

1. Enable Gii, if it's not already.
2. Access Gii for your site in your browser.
3. Click Module Generator.
4. On the resulting page, enter the name of the module to be generated, and click Preview.
5. On the resulting page, confirm the files being created, and click generate.

This is all pretty straightforward. The most common complication will be the lack of an existing **modules** directory, which can result in odd errors.

Give serious thought to your module name prior to following those steps. Modules need unique names, which will be their identifier. To avoid bugs, it's best to name the module in all lowercase letters, and do not use any numbers, punctuation, or other symbols.

Gii will create a new directory, **protected/modules/modulename**. Within it, you'll find:

- **controllers**
- **controllers/DefaultController.php**
- **views**
- **controllers/default**
- **controllers/default/index.php**
- **ModulenameModule.php**

This last item will be the main module class, which will extend **CWebModule**. In many ways, this class is a corollary to the application's **CApplication** class.

## Using a Module

Once you've defined a module, or added an existing module extension to your site, you just need to tell your site about the module in order to use it. This is done in the "modules" section of the primary configuration file. In fact, you've already seen an example of this with Gii:

```
# protected/config/main.php
'modules'=>array(
    'gii'=>array(
        'class'=>'system.gii.GiiModule',
        'password'=>'password',
        'ipFilters'=>array('127.0.0.1','::1'),
    ),
),
```

To use a different module, just add its name to the configuration file:

```
# protected/config/main.php
'modules'=>array(
    'modulename',
    'gii'=>array(
        'class'=>'system.gii.GiiModule',
        'password'=>'password',
        'ipFilters'=>array('127.0.0.1','::1'),
    ),
),
```

The value used needs to match the ID value of the module itself (which also corresponds to the name of the module's directory).

With the module enabled via the configuration file, it's now usable by your site. You can test this by heading to <http://www.example.com/index.php/modulename> in your browser.

As with the primary application, if no controller or action IDs are provided, the module will run the default action of the default controller. With a module, that's the "index" action of `DefaultController`. As generated, this action merely renders the `modulename/views/default/index.php` view file (**Figure 19.9**).



**Figure 19.9:** A default module layout.

```
# protected/modules/modulename/controllers/DefaultController.php
<?php
class DefaultController extends Controller {
    public function actionIndex() {
        $this->render('index');
    }
}
```

As you can see, this controller extends the `Controller` class (which, in turn, extends `CController`), so it can be used and behaves like any other controller in your application. You can now also create other controllers, other actions, and other view files. If your module needs models, you can create and use those, too. You'll see more of this in a forthcoming example.

{TIP} You can nest one modules within another by adding the submodule to the configuration of the parent module, and placing the submodule in the `modules` directory of the parent module.

## Routing Modules

When using modules, an additional dimension is added to your URLs. The full syntax of a URL in Yii is:

<Domain>/index.php/<Module>/<Controller>/<Action>/<Params>

If no action ID is provided, the default action of the requested controller is used. If no controller ID is provided, the default controller of the application or module is used. If no module ID is provided, the default application is used.

When it comes to creating URLs with modules, you'll still use the `createUrl()` or `createAbsoluteUrl()` methods of the `CController` class. As is the case when not using modules, if you specify the action and nothing else, the route will assume the same controller, and, in the case of a module, module:

```
# protected/modulename/controllers/DefaultController.php
public function actionDummy() {
    // modulename/default/index:
    $url = $this->createUrl('index');
```

If you specify the controller, the URL will be to that controller and action, but still within the same module:

```
# protected/modulename/controllers/DefaultController.php
public function actionDummy() {
    // modulename/other/index:
    $url = $this->createUrl('other/index');
```

If you specify the module name, you can create a URL to a different module within the application:

```
# protected/modulename/controllers/DefaultController.php
public function actionDummy() {
    // testmodule/other/index:
    $url = $this->createUrl('/test/other/index');
```

Note that in this case, you need to start with a slash, to indicate the base of the URL. The same goes if you want to create a URL to the primary application from within a module:

```
# protected/modulename/controllers/DefaultController.php
public function actionDummy() {
    // /other/index:
    $url = $this->createUrl('/other/index');
```

## Configuring the Module

Modules, like any application component or the application itself, can be pre-configured in the primary configuration file. To do that, assign a value to any public property of the main module class:

```
<?php
class TestModule extends CWebModule {
    public $var;

# protected/config/main.php
'modules'=>array(
    'test' = array(
        'var' => 'value',
    ),
),
```

{TIP} If you'd rather, you can create your own configuration file for the module, and then have the main configuration file include it.

Because the base class extends `CWebModule`, you can configure any public property of that class, too:

```
# protected/config/main.php
'modules'=>array(
    'test' = array(
        'defaultController' => 'something',
        'layout' => 'home'
    ),
),
```

Within the module or the application, you can also access any public property through the module itself. First, you'd get access to the module, which returns a class instance. Then you could access its properties:

```
$m = Yii::app()->getModule('moduleID');
echo $m->var;
```

Within a module itself, you can always refer to the current module using `Yii::app()->controller->module`:

```
$m = Yii::app()->controller->module;
echo $m->var;
```

Further, Yii will create a root alias for each module. Having created the “test” module, in path names in your application, “test” will equate to `protected/modules/test`.

## An Example Module

To wrap up this discussion of modules, I'm going to write a more complete version of the Stripe extension begun earlier. This was originally an incomplete widget example, that would be better developed as a module.

Stripe doesn't offer shopping carts or inventory management, it simply provides an easy and secure way to process a credit card payment. As explained earlier, it's a two-step process:

- In the client, take the customer's payment information, pass it to Stripe, and receive a token in return
- On the server, use the token to process the charge

Stripe's Checkout functionality handles all of step one, as already demonstrated. Now, let's extend (ha!) that functionality so that the charge is actually processed.

Because the processing of the payment is separate from the other aspects of the e-commerce site, the only dynamic piece of information this module would need is the amount to be charged.

To make this module more complex, configurable, and interesting, I'm tossing in a few more features:

- A custom payment form, in lieu of Stripe Checkout
- Recording of the charges in a database

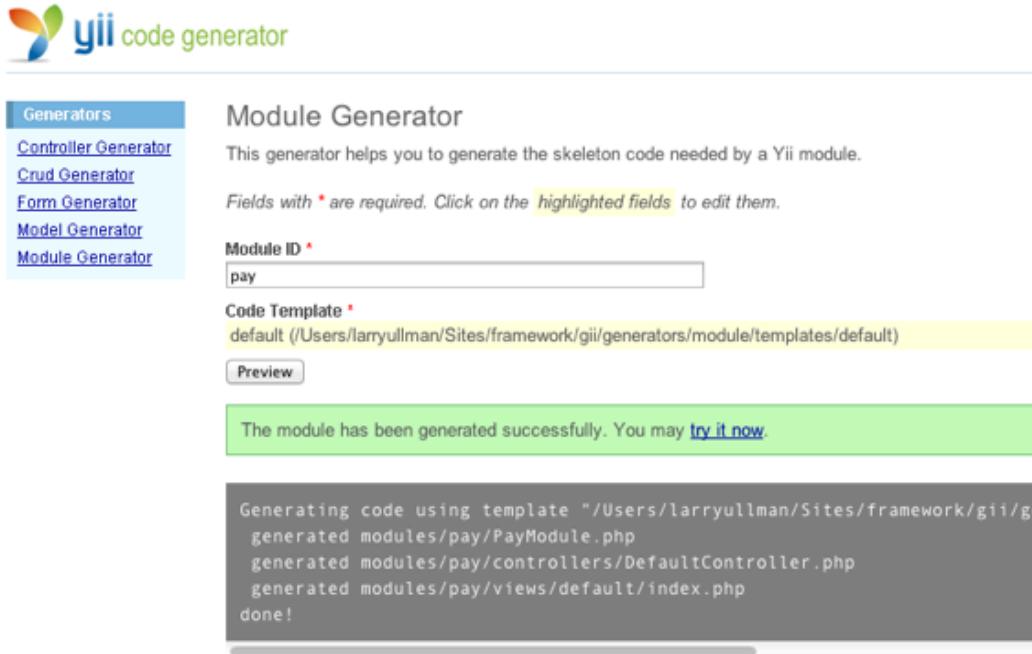
With all this in mind, let's create and use this module.

{TIP} You'll probably find this easier to follow if you also download the code from [<http://yii.larryullman.com>].

## Generating the Shell

The first thing you'll need to do is enable Gii, and follow the steps outlined earlier to create the shell of the module. This will require that you have a writable **protected/modules** directory, if you don't already.

For this module, I'm going to call it "pay" (**Figure 19.10**).



**Figure 19.10:** Creating a new module.

The result is the **protected/modules/pay** directory. Within it, you'll find:

- **controllers**
- **controllers/DefaultController.php**
- **views**
- **controllers/default**
- **controllers/default/index.php**
- **PayModule.php**

Now it's time to edit, and add to, those files and directories.

### **Adding the Stripe Library**

Performing a charge requires the Stripe PHP library, so the extension needs to have that.

Commonly, third-party libraries go within a **vendors** directory, so start by creating **protected/modules/pay/vendors**. Within **vendors**, create a **stripe** folder.

Next, head to <https://stripe.com/docs/libraries> and download the PHP library.

Expand the downloaded file. The result will be a folder named something like *stripe-php-1.10.1*. Within that, you'll find:

- **CHANGELOG**
- **composer.json**
- **lib**
- **LICENSE**
- **README.rdoc**
- **test**
- **VERSION**

Copy the **lib** folder to **protected/modules/pay/vendors/stripe**.

There! Now the extension has its required external libraries.

### **Updating the Module Class**

Next, I'm going to think about how I might want the module to be configured when added to a site. With this particular module, there are two pieces of information that must be provided: the user's public and private Stripe keys. These can be provided during the configuration so long as they're public attributes of the main module class:

```
class PayModule extends CWebModule {  
    public $public_key;  
    public $private_key;
```

Because these values are required (no default value would work), the class's `init()` method should throw exceptions if they're not provided (**Figure 19.11**):

## CException

Your public Stripe key is required.

/Users/larryullman/Sites/yii-test-ch19/protected/modules/pay/PayModule.php(12)

Figure 19.11: Stripe keys are required.

```
public function init() {  
    if ($this->public_key == null) {  
        throw new CException('Your public Stripe key is  
required.');//  
    }  
    if ($this->private_key == null) {  
        throw new CException('Your private Stripe key is  
required.');//  
    }  
}
```

## Creating the Database Table

The Stripe extension module will need one model class defined. The class will be used to create the form, add validation, and store the data in the database. For that reason, a model based upon Active Record makes sense. Gii can create the model, but the underlying database table needs to exist first. I imagine the database table would be defined like so:

```
CREATE TABLE payment (  
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    charge_id VARCHAR(60) NOT NULL,  
    email VARCHAR(80) NOT NULL,  
    amount INT UNSIGNED NOT NULL,  
    date_added TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
    UNIQUE (charge_id),  
    INDEX (email)
```

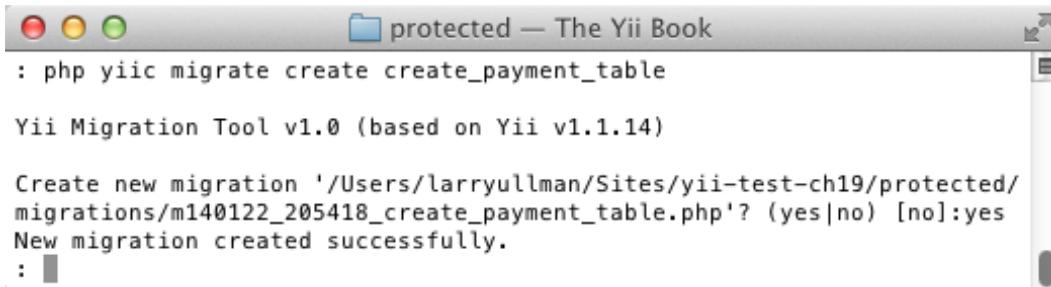
```
) ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

{NOTE} You may notice that the table is not storing any credit card details. When using Stripe, absolutely no credit card information will touch your server.

You *could* define that SQL command in a text file that you distribute with the extension. But since reusability is the goal, and to do things the Yii way, let's create a *migration* for it.

The first thing you'll want to do is setup your application to support migrations, if you have not already. See Chapter 18, “[Advanced Database Issues](#)”, for those explicit instructions.

Next, from the command-line, within the **protected** directory, execute this command (**Figure 19.12**):



```
protected — The Yii Book
: php yiic migrate create create_payment_table
Yii Migration Tool v1.0 (based on Yii v1.1.14)

Create new migration '/Users/larryullman/Sites/yii-test-ch19/protected/
migrations/m140122_205418_create_payment_table.php'? (yes|no) [no]:yes
New migration created successfully.
:
```

Figure 19.12: Creating a migration.

```
yiic migrate create create_payment_table
```

{NOTE} You may need to change the command used to execute `yiic` to suit your environment.

This will create a file named something like `m140122_205418_create_payment_table.php`, and stored in the **protected/migrations** directory.

Since this migration isn't particular to the application—it's part of the module, create a **migrations** directory within your module folder, and then move this new file there. Then open the file for editing.

The migration class has two methods: `up()`, for implementing the migration, and `down()`, for undoing its impact. In this case, the `up()` method should create the `payment` table and `down` should delete it:

```
public function up() {
    $this->createTable('payment', array(
        'id' => 'pk',
        'charge_id' => 'string NOT NULL',
        'email' => 'string NOT NULL',
        'amount' => 'integer UNSIGNED NOT NULL',
        'date_added' => 'timestamp NOT NULL DEFAULT
            CURRENT_TIMESTAMP'
    ));
    $this->createIndex('charge_id', 'payment', 'charge_id', true);
    $this->createIndex('email', 'payment', 'email');
    echo "The 'payment' table has been created.\n";
}
public function down() {
    $this->dropTable('payment');
    echo "The 'payment' table has been dropped.\n";
    return false;
}
```

With that file defined, you now just need to execute the migration. Normal application migrations are executed from the command-line (from within the **protected** directory) using:

```
yiic migrate
```

By default, this command looks for migrations to be executed found within the **protected/migrations** folder. To change that, provide the path to the module's **migrations** directory (**Figure 19.13**):

```
yiic migrate --migrationPath=application.modules.pay.migrations
```

And now the database table should be created! Next, you can generate and edit the corresponding model. More importantly, you've just made your extension that much easier for others to use!

### Creating the Model

For the model, start by having Gii generate it based upon the database table definition:

1. Log into Gii.
2. For the table name, use “payment”.
3. For the model class, use “Payment” (the default).

```

protected — The Yii Book
: php yiic migrate --migrationPath=application.modules.pay.migrations
Yii Migration Tool v1.0 (based on Yii v1.1.14)

Total 1 new migration to be applied:
m140122_205418_create_payment_table

Apply the above migration? (yes|no) [no]:yes
*** applying m140122_205418_create_payment_table
  > create table payment ... done (time: 0.111s)
  > create unique index charge_id on payment (charge_id) ... done (time: 0.225s)
  > create index email on payment (email) ... done (time: 0.106s)
The 'payment' table has been created.
*** applied m140122_205418_create_payment_table (time: 0.445s)

Migrated up successfully.
:

```

**Figure 19.13:** Implementing the migration.

4. Click Preview.
5. Assuming everything is okay, click Generate (**Figure 19.14**).

The resulting code will end up in **protected/models/Payment.php**. Create a **models** directory for your extension (in the extension's folder), and move this file there. Open the file to edit it.

First, you'll want to add one public property to the model:

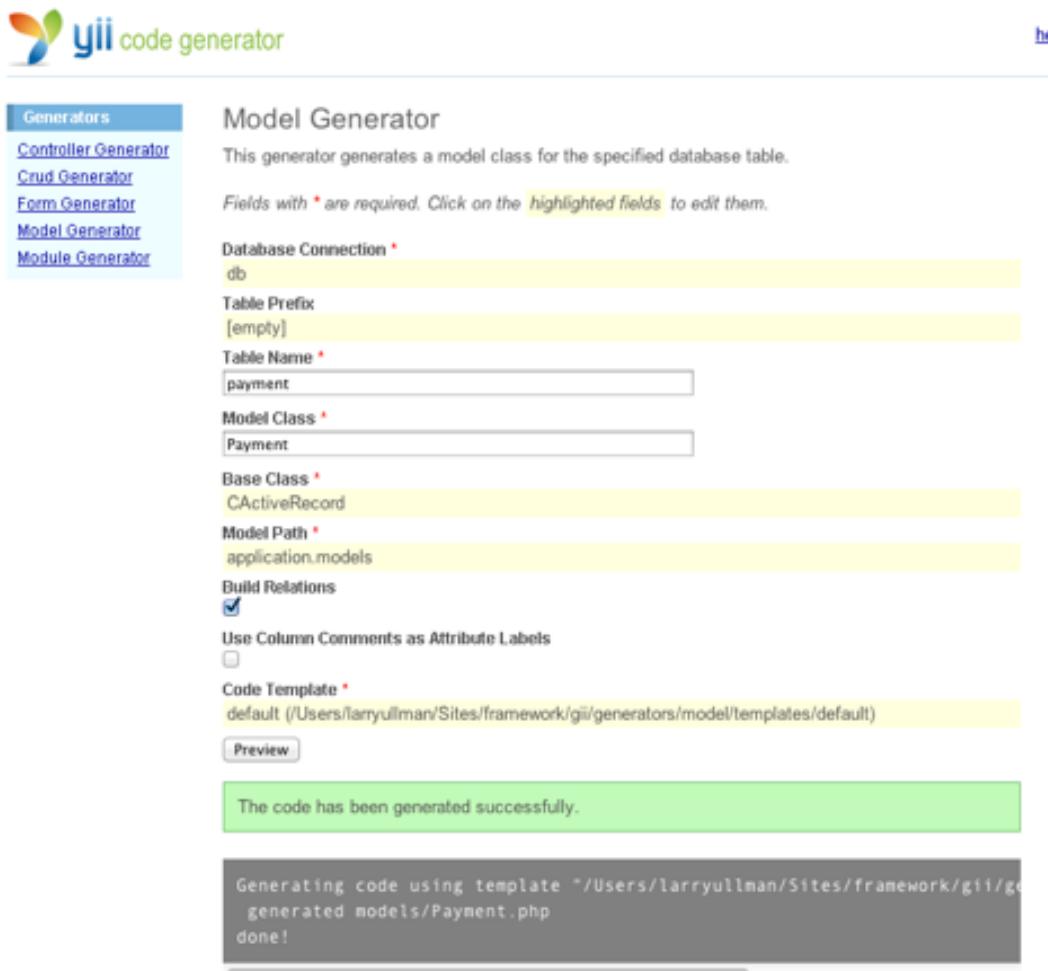
```
class Payment extends CActiveRecord {
    public $token;
```

The token is the only additional piece of information that is required through the HTML form, but won't be stored in the database. For this reason, it becomes a public property.

None of the credit card information—number, CVC, or expiration date—will touch the server, so there's little point in representing those in the model.

Now you'll want to tweak the validation rules slightly:

```
public function rules() {
    return array(
        array('charge_id, email, amount, token',
              'required'),
        array('charge_id, email', 'length', 'max'=>255),
        array('email', 'email'),
        array('charge_id', 'unique'),
        array('amount', 'numerical', 'integerOnly'=>true,
```

**Figure 19.14:** Creating a new model.

```
'min'=>50),
array('charge_id', 'email', 'amount', 'date_added',
      'safe', 'on'=>'search'),
);
}
```

Hopefully there's nothing too surprising here, but see Chapter 5 for more on validation rules, if need be. The token is marked as required here. The amount must be an integer at least 50 or larger, which corresponds to the minimum amount that can be processed through Stripe.

You might want to update the labels, too:

```
public function attributeLabels()
{
    return array(
        'id' => 'ID',
        'charge_id' => 'Charge ID',
        'email' => 'Email',
        'amount' => 'Amount',
        'date_added' => 'Date Charged',
    );
}
```

{TIP} You may also want to set custom error messages.

And, it's not a big deal, but I would remove `id` from being a search criteria. I only put that property in the model because I prefer my database tables to be numerically indexed, rather than using the `charge_id`.

## Editing the Controller

With the model created and edited, you can turn to the controller (and then the view files). By default, modules use the “index” action of the `DefaultController`. Although I might be inclined to use more meaningful terminology, changing these defaults just for that purpose is moot.

The workflow for the extension is pretty simple:

1. The payment form should be shown.
2. Upon successful completion of the payment form, the charge attempt needs to be made of Stripe.
3. If the charge attempt succeeds, the user should be shown a “thanks” page.

All of this can be done in two actions and two view files, although one of those view files will require some heavy JavaScript. First, though, the display of the form should only be loaded over HTTPS, so a filter can be added for that:

```
# pay/controllers/DefaultController.php
public function filters() {
    return array(
        'httpsOnly + index',
    );
}
public function filterHttpsOnly($fc) {
    if (Yii::app()->getRequest()->getIsSecureConnection()) {
        $fc->run();
    } else {
        throw new CHttpException(400, 'This page needs to be
            accessed securely.');
    }
}
```

The first chunk of code applies the “httpsOnly” filter to the “index” action. Next, the “httpsOnly” filter is defined as part of the controller. For an even better integration, you could create “httpsOnly” as a filter extension, include it with the project, and then use it here.

{TIP} If you’re testing this and don’t have SSL setup, remove the use of the filter.

The rest of the controller has two actions: “index” and “thanks”. The “index” action does the bulk of the work, and I’ll build it up in two sections of this chapter. All of the following code (until I say otherwise), goes, in order, within the “index” action, starting by creating a model instance:

```
$model=new Payment;
```

Most of this action’s code is similar to that you’d find in any controller’s “create” action. Note that you can create an instance of a `Payment` model here because the primary module class imports classes from the module’s `models` directory:

```
# protected/modules/pay/PayModule.php
class PayModule extends CWebModule {
    public function init() {
        $this->setImport(array(
            'pay.models.*',
            'pay.components.*',
        ));
    }
}
```

```
    ));  
}
```

Returning to the controller, next, you'll need to validate that an amount was provided:

```
$model->amount = 12575;
```

After creating the model instance, it should access the amount to be charged. On a real, live site, this amount would most likely be determined dynamically, and would be provided to the extension in some manner:

- Saved in the session
- Calculated by another controller
- Retrieved from the database

For this demonstration, I'm just hardcoding a value there. Remember that it has to be an integer, representing the amount in cents!

Next, the action checks if the form has been submitted. If so, the safe attributes are assigned. Then the `validate()` method is called to confirm that all the data matches the validation rules. Note that `save()` isn't called here yet, as the model is only saved after successfully making the charge.

```
if(isset($_POST['Payment'])) {  
    $model->attributes=$_POST['Payment'];  
    // Temporary:  
    $model->charge_id = 'temp';  
    if($model->validate()) {
```

In the model, `charge_id` is required, but this value will only be provided once the charge is processed with Stripe. To circumvent this issue, but still pass validation, the `charge_id` is assigned a temporary value first.

Next, it's time to process the charge:

```
// Charge via Stripe!  
if /* charged */ {  
    $model->save();  
    $this->redirect(array(  
        'thanks',  
        'amount' => $model->amount  
    ));  
}
```

If the payment went through, then the model is saved, and the user is redirected to the “thanks” page, passing along the amount in the URL. (This code goes within the `if ($model->validate())` conditional.)

Finally, the action renders the “form” page, passing along the model instance and the public Stripe key:

```
    } // Not validated.  
} // Not posted.  
// Show the form:  
$this->render('form',array(  
    'model'=>$model,  
    'key' => Yii::app()->controller->module->public_key  
));
```

As with any create-like action, the code will only get to this point if the user was not redirected to the “thanks” page. That would happen in three scenarios:

- The form is loaded for the first time
- The form is not completely filled out and errors need to be shown
- The payment could not be made with Stripe

The code above handles all of this except for the payment attempt, to be added later.

Finally, you’ll need the “thanks” action. It receives the amount charged in the URL and passes it to the view file:

```
public function actionThanks($amount) {  
    $this->render('thanks',array(  
        'amount'=>$amount,  
    ));  
}
```

That’s the bulk of the controller (lacking only the most important piece!). Fleshying out this example completely, I would be inclined to add an admin feature to this extension, allowing the administrator to view the list of payments. Neither “update” nor “delete” actions would ever be warranted.

*{TIP}* Stripe supports the creation of customers and recurring billing via subscriptions. Ability to do both would be a great addition to this extension.

## Creating the View Files

With the bulk of the controller in place, let's create the two view files. The "thanks" page is easy, something like:

```
# protected/modules/pay/views/default/thanks.php
<?php
/* @var $this DefaultController */

$this->breadcrumbs=array(
    $this->module->id,
);
?>
<h1>Thank you for your order!</h1>

<p>You have been charged $<?php echo number_format($amount/100, 2); ?>. </p>
```

The "form" page is much more complicated. It must display an HTML form and apply some JavaScript. And because the functionality of Stripe depends upon the JavaScript, it's a bit more complicated than most uses.

To create the form, I started with the standard syntax for forms in Yii, like those created by Gii:

```
<h1>Pay</h1>
<div class="form">
<?php $form=$this->beginWidget('CActiveForm', array(
    'id'=>'pay-form',
    'enableAjaxValidation'=>false,
)); ?>
<p class="note">Fields with <span class="required">*</span> are required.</p>
<?php echo $form->errorSummary($model); ?>
<div class="row">
    <?php echo $form->labelEx($model, 'email'); ?>
    <?php echo $form->textField($model, 'email',
        array('size'=>60, 'maxlength'=>60)); ?>
    <?php echo $form->error($model, 'email'); ?>
</div>
```

Ajax validation should definitely be disabled, as the form's data should not be sent to the server. And, most of the form's data—specifically, the credit card information—isn't represented by the model anyway. Speaking of which...

```
<div class="row">
    <label for="cc-num" class="required">Credit Card Number
    <span class="required">*</span></label>
    <input size="20" maxlength="20" id="cc-num" type="text"
        autocomplete="off">
</div>
<div class="row">
    <label class="required">Credit Card Expiration (MM/YYYY)
    <span class="required">*</span></label>
    <input size="2" maxlength="2" id="cc-exp-month"
        type="text" autocomplete="off"> / <input size="4"
        maxlength="4" id="cc-exp-year" type="text"
        autocomplete="off">
</div>
<div class="row">
    <label class="required">Credit Card CVC
    <span class="required">*</span></label>
    <input size="4" maxlength="4" id="cc-cvc" type="text"
        autocomplete="off">
</div>
```

For the credit card elements—the number, the expiration month and year, and the CVC—these are not tied to the model, and so they’re created using straight HTML, not even `CHtml`. I made this choice for a few reasons:

1. Because they aren’t tied to models, no server-side validation will be used, and no element-specific error messages will automatically apply.
2. It’s absolutely imperative that the values entered here aren’t submitted to the server (i.e., they cannot have `name` attributes).
3. The JavaScript will be easier if the ID values of the elements are known and constant.

Next, the form is completed:

```
<div class="row buttons">
    <?php echo CHtml::submitButton('Pay $' .
        number_format($model->amount/100, 2),
        array('id'=>'submit-btn')); ?>
</div>
<?php $this->endWidget(); ?>
</div><!-- form -->
```

For extra flair, the submit button shows the amount to be charged, which can be found in the model. Because it’s an integer, it needs to be divided by 100 (**Figure 19.15**).

**Pay**

Fields with \* are required.

Email \*

Credit Card Number \*

Credit Card Expiration (MM/YYYY) \*

 / 

Credit Card CVC \*

**Pay \$125.75**

**Figure 19.15:** The payment form.

Now it's a matter of adding the JavaScript. First, the page needs to include the **Stripe.js** file:

```
<?php echo CHtml::scriptFile('https://js.stripe.com/v2/'); ?>
```

{TIP} You can find more detailed explanations of much of this code in my [Stripe blog series](#).

Next, the page needs to do a bunch of stuff:

- Set the Stripe public key
- Validate the payment information
- Send the payment information to Stripe
- Handle the Stripe response
- Report any errors

Arguably, most of this could go in an external JavaScript library that is then published to the **assets** folder. Instead, I'm just adding it to the page. All of the JavaScript goes within this:

```
<?php  
Yii::app()->clientScript->registerScript('stripe', "  
// All JavaScript here.  
");  
?>
```

Because all of the JavaScript is placed between double quotes, the JavaScript code has to use only single quotes, or escaped double quotes. I'll explain that code in pieces (again, the following all goes within the second `registerScript()` argument).

First, the Stripe public key is set:

```
Stripe.setPublishableKey('$key');
```

This is provided to the view file from the controller, and the main module class throws an exception if it doesn't exist.

The next bit of code handles the form submission. Within that code, the JavaScript must:

- Disable the submit button
- Get the four form values (for the credit card details)
- Validate the form values
- Pass the details to Stripe

Here's how the submit event handler is created, using jQuery:

```
$('#pay-form').submit(function(){
    // Prevent the form from submitting:
    return false;
}); // Submit function.
```

Now the other steps are performed within that bit (just before `return false`:

```
// Flag variable:
var error = false;

// disable the submit button to prevent repeated clicks:
$('#submit-btn').attr('disabled', 'disabled');

// Get the values:
var ccNum = $('#cc-num').val(), cvcNum = $('#cc-cvc').val(),
expMonth = $('#cc-exp-month').val(),
expYear = $('#cc-exp-year').val();

// Validate the number:
if (!Stripe.card.validateCardNumber(ccNum)) {
    error = true;
    reportError('The credit card number appears to be invalid.');
}
```

```
// Validate the CVC:  
if (!Stripe.card.validateCVC(cvcNum)) {  
    error = true;  
    reportError('The CVC number appears to be invalid.');//  
  
// Validate the expiration:  
if (!Stripe.card.validateExpiry(expMonth, expYear)) {  
    error = true;  
    reportError('The expiration date appears to be invalid.');//  
  
// Validate other form elements, if needed!  
  
// Check for errors:  
if (!error) {  
  
    // Get the Stripe token:  
    Stripe.card.createToken({  
        number: ccNum,  
        cvc: cvcNum,  
        exp_month: expMonth,  
        exp_year: expYear  
    }, stripeResponseHandler);  
  
}
```

There are comments inline that explain each step, and most of the information can also be found in Stripe's documentation. The validation steps make repeated reference to a `reportError()` function. For now, that's defined as so:

```
function reportError(msg) {  
  
    // Show the error in the form:  
    alert(msg);  
  
    // re-enable the submit button:  
    $('#submit-btn').prop('disabled', false);  
    return false;  
}
```

Again, this all goes within the `registerScript()` line. Obviously there are many ways you could improve the `reportError()` function; I'm mostly focusing on easy-to-understand and functional-enough here.

Finally, when the Stripe request returns a response, it's handled by the `stripeResponseHandler()` function:

```
function stripeResponseHandler(status, response) {  
  
    // Check for an error:  
    if (response.error) {  
  
        reportError(response.error.message);  
  
    } else { // No errors, submit the form:  
  
        var f = $('#pay-form');  
  
        // Token contains id, last4, and card type:  
        var token = response['id'];  
  
        // Insert the token into the form so it gets submitted to the server  
        f.append('<input type=\"hidden\" name=\"Payment[token]\"  
value=\"' + token + '\">');  
  
        // Submit the form:  
        f.get(0).submit();  
  
    }  
}  
} // End of stripeResponseHandler() function.
```

This code checks the response from Stripe for an error. If one exists, it's sent to the `reportError()` function. Otherwise, the token is grabbed from the response and then stored in a hidden input. Finally, the form is submitted.

Whew! Complicated JavaScript, but it works. Now on to the PHP code that actually processes the payment with Stripe.

## Making the Payment

Just to reiterate what the process is, Stripe requires both client-side code (the form and the JavaScript) and server-side code. The client-side code securely sends the customer's payment information to Stripe so that it doesn't touch your server. But it's the server-side code that performs the actual charge. That will be done in the controller action that handles the form's submission. All of the following will go in `actionIndex()`, after `if($model->validate()) {`.

All of the new code should go within a `try...catch` block in order to handle the

errors that could occur. But first, the code must import the Stripe library. Next, within the `try`, you then start by setting the secret key:

```
Yii::import('pay.vendors.stripe.lib.Stripe');
try {
    Stripe::setApiKey(Yii::app()->controller
        ->module->private_key);
```

Next, you can attempt the charge at Stripe, passing along:

- The amount
- The currency
- The token
- A description

```
$charge = Stripe_Charge::create(array(
    "amount" => $model->amount,
    "currency" => "usd",
    "card" => $model->token,
    "description" => $model->email
));
);
```

Next, the code should check the response, confirming that it was paid. If so, then the `charge_id` (set in Stripe) can be assigned, the model can be saved, and the user redirected:

```
// Check that it was paid:
if ($charge->paid == true) {
    $model->charge_id = $charge->id;
    $model->save();
    $this->redirect(array('thanks', 'amount' => $model->amount));
}
```

If an exception occurred, it must be caught. Most exceptions will be of type `Stripe_CardError`:

```
} catch (Stripe_CardError $e) {
    $e_json = $e->getJsonBody();
    $err = $e_json['error'];
    $model->addError('Credit Card', $err['message']);
```

# Pay

Fields with \* are required.

Please fix the following input errors:

- Your card was declined.

Email \*

fail@example.net

**Figure 19.16:** How the Stripe error message appears on the page.

That's how you parse the error message out of the Stripe response. The message will be sufficiently useful. To have it display on the page, I'm assigning it to a non-existent model property. The view file will still display this (**Figure 19.16**).

Catch other Stripe exception types:

```
} catch (Stripe_ApiConnectionError $e) {
    // Network problem, perhaps try again.
} catch (Stripe_InvalidRequestError $e) {
    // You screwed up in your programming. Shouldn't happen!
} catch (Stripe_ApiError $e) {
    // Stripe's servers are down!
} catch (Stripe_CardError $e) {
    // Something else that's not the customer's fault.
}
```

You would want to handle these in other ways, but you can use the same construct already laid out to have the error be shown.

And that's it! Should the charge succeed, the model will be saved in the database and the user redirected (**Figure 19.17**).

## Thank you for your order!

You have been charged \$125.75.

**Figure 19.17:** The result of a successful payment.

Should it fail, the error message will be shown on the page, giving the user the chance to try again.

And that completes the module. It is ready to be used. Once you polish it up, provide adequate documentation, and so forth...

To test the Stripe payment integration (after configuring the module), see [Stripe's documentation](#).

## Configuring the Module

To use this module, it just needs to be provided with the right Stripe keys:

```
'modules'=>array(
    'stripetest' => array(
        'public_key' => 'pk_test_APr32Tly9WH6K9XfZpJeEKCH',
        'private_key' => 'sk_test_Ca0U4KkAPr32TZpJeEKCH'
    ),
)
```

Everything else is handled by the module, save for the amount issue, to be discussed in more detail shortly. Other configuration ideas include allowing for the extension user to

- Change the currency
- Customize the payment description
- Opt-in to creating customer objects in Stripe as well as payments

## Improving the Module

There are many ways this module could be written differently or changed. I've already mentioned a couple, but the most important step in making this a distributable extension would be to provide good documentation as to how you'd use it. This is particularly important as the extension's user might want to customize the layout of the form, and you'd want to set the right rules for doing so.

Towards that end, one improvement would be to change the form creation from a static view file to a use of form builder. This would further separate the requirements from the display, making it much easier for the extension user to change the look of the form.

The only requirement that's not well handled by the extension is how the amount gets to it. Truly, an exception ought to be thrown when no amount value exists, as it is required. But the amount could come from many places, so a sophisticated solution would be to have this as a configurable option. The extension user could indicate the source and attribute or index where the amount can be found: the X attribute of the Y model; the X index of the session; etc. Then the module could retrieve it and throw an exception on error.

I welcome anyone to improve upon this extension in any way, and even to distribute it!

## Deploying Extensions

If you've created an extension that you think is worth sharing (yay!), you can offer it up on your own Web site, but it'd be better placed among the other [Yii extensions](#).

In order to share an extension, you'll need to be a registered [Yii forum](#) member. You'll also want to create a new forum thread within the extensions forum where you can answer questions and get feedback. User feedback is a great way to improve the quality of your work (believe me, I know!), so pay attention to your forum thread, duly consider the feedback you get, and continue to maintain your extension.

And thanks for sharing! Not to be overly sentimental about it, but the strength of the community is one of the things makes Yii so great.

## Chapter 20

# WORKING WITH THIRD-PARTY LIBRARIES

The Yii framework in itself is a powerful tool, but it's made even more powerful by its easy support for third-party libraries. The Yii creators never intended Yii to be all things to all people. Instead, they designed Yii to readily work with external tools that might be better suited for individual tasks.

Before approaching this chapter, I informally asked as to what third-party libraries you'd like to see me integrate and demonstrate. The most interest was in integrating other frameworks, most specifically [Symfony](#) and [Zend](#). I'm also going to use this chapter to discuss how you can implement a search engine using [Elasticsearch](#) and its PHP library. And I'll walk through a usage of [Swift Mailer](#) to send out email.

The focus of the book is on the Yii framework, of course, but using third-party libraries with Yii is relatively straightforward. In fact, Chapter 19, "Extending Yii," already demonstrated one example of this with the [Stripe](#) PHP library. But to add some weight, and to show more real-world code, this chapter will spend more time discussing the third-party libraries and tools in more detail, most particularly Elasticsearch.

### Installation

To be as obvious as possible when it comes to using third-party libraries with Yii, I'll start by clearly stating that you must always download and install the third-party libraries to your server yourself. That is not something Yii will do for you.

It's recommend that you put all third-party libraries within the **protected/vendor** directory, with each vendor having its own subdirectory. If you're using multiple third-party libraries, this might mean you'd end up with a structure like:

- **protected/vendor/zend**

- `protected/vendor/stripe`
- `protected/vendor/elasticsearch`

This organization is not required by Yii, but simply makes the most sense. (You'll also sometimes see it as a `vendors` directory, plural, although I try to stick to the singular `vendor`.)

In just a few pages, I'll discuss the installation of libraries using [Composer](#). Again, although this is a *more automated* way of installing third-party libraries, it's not completely automatic.

{NOTE} ] Yii 2 will make extensive use of Composer, which means that management of third-party libraries will also be made easier and more automatic.

## Accessing Library Classes

Once you've installed the requisite third-party library, how easy or hard it is to use the third-party library's code is entirely dependent upon how complex the library is, and the status of its autoloader. To appreciate what steps you'll need to take, let's look at how Yii works and what could go wrong.

Yii relies upon *autoloading* to grab required class files on demand. Take the following bit of code:

```
$model = new User;
```

When Yii executes that code, it knows that it needs to load the `User.php` file that will define the `User` class. (Presumably, that's `protected/models/User.php`.) Yii is aware of the existence of that file thanks to this line in the application's configuration:

```
# protected/config/main.php
// autoloading model and component classes
'import'=>array(
    'application.models.*',
    'application.components.*',
),
```

When Yii cannot find a class definition, it'll throw an exception ([Figure 20.1](#)).

This is true of PHP in general, of course.

When it comes to using third-party libraries, the most common problem will be an inability for Yii to find the corresponding class file for the object type you're trying

## PHP warning

```
include(Users.php): failed to open stream: No such file or directory
```

```
/Users/larryullman/Sites/framework/YiiBase.php(427)
```

**Figure 20.1:** *The misspelling of the class name results in an exception.*

to create. This is particularly true when a library has a very complex class hierarchy or just uses multiple classes. But the right Yii code will resolve these issues.

Over the next several pages I'll introduce and explain many different ways to make third-party library classes available to your Yii application. If it's still not clear, subsequent specific examples should hammer the points home.

The most important lesson is that when you see an exception like that in Figure 20.1, the reason is almost always one of the following:

- You misspelled the class name
- Yii cannot find the file where that class is defined

To debug such problems, start by confirming the class name. If that doesn't clearly resolve the issue, you'll need to fix where Yii looks for files.

## Requiring Class Files

One way to include a library class definition in your Yii application is the standard PHP approach: using `require()`.

```
require('vendor/VendorName/ClassName.php');
```

Having included that library, you can now create objects of the `ClassName` type:

```
$obj = new ClassName;
```

One downside to this approach is that it automatically includes the file at the invocation of `require()`, even if the class definition isn't used until much later, or not at all.

A bigger problem with this approach is that it won't work with especially complicated class structures. For example, if `ClassName` is just one of many classes used

by the library, and objects of other class types will be created on the fly, this code won't work.

Third, and most trivially, using `require()` is not in keeping with the general Yii approach. A better solution is to import the class definition into the application, as Yii already does with the fundamental application models and controllers.

## Importing Classes

Yii uses the `spl_autoload()` function of the Standard PHP Library (SPL) to handle its autoloading necessities. PHP uses this function to find and include class definitions that are now needed, but have yet to be included. With it, so long as Yii knows where to find your class definitions, autoloading will work without problem.

As already mentioned, the “import” section of the primary configuration file is where you can identify the locations of class files. If you'll be using a third-party library anywhere in the application, you can add the library's directory to the list of folders in the configuration file:

```
# protected/config/main.php
// autoloading model and component classes
'import'=>array(
    'application.models.*',
    'application.components.*',
    'application.vendor.VendorName.*'
),
```

If instead you'll only be using the third-party library in specific controllers or controller actions, it'd be better to use the `import()` method within the controller or controller action:

```
Yii::import('application.vendor.VendorName.*');
```

Note that in both cases you're *not* importing a specific file, as in a `require()` call, but rather importing an entire directory. When you provide a directory to the `import()` method or the “import” configuration, this is equivalent to adding that directory to PHP's include path.

As with any other import in Yii, the framework does not automatically read in the class files. Instead, Yii will only do so when an instance of a corresponding object is created (i.e., class files are loaded on demand).

Assuming the `VendorName` directory has a root-level class file, such as `Thing.php`, you can now create objects of that type like so:

```
$thing = new Thing;
```

However, the import does not automatically include any subdirectories, so the potential problems of having multiple classes, possibly in subdirectories, still exists. For example, if the `Thing` class makes use of `VendorName/Subdir/Charlie.php`, which defines the `Charlie` class, Yii will again throw an exception.

As an aside, this wasn't a problem with the inclusion of the Stripe library in Chapter 19:

```
Yii::import('pay.vendor.stripe.lib.Stripe');
Stripe::setApiKey(Yii::app()->controller->module->private_key);
$charge = Stripe_Charge::create(array(
    "amount" => $model->amount,
    "currency" => "usd",
    "card" => $model->token,
    "description" => $model->email
));
);
```

In this particular case, use of the `Stripe_Charge` class did not result in an exception because the `Stripe.php` file does a direct inclusion of every other Stripe class file.

In that same vein, if the `VendorName` directory has classes below the root level that you want to use, you can require those files directly:

```
require('Subdir/Charlie.php');
$charlie = new Charlie;
```

(You'll note that the code does not reference the `VendorName` directory. This still works because `VendorName` was previously added to the include path.)

Again, though, this approach still has the issues with needing to manually require every class file, or import every subdirectory, and not really being in keeping with the preferred Yii approach.

The better solution is to use an autoloader that's knowledgeable about the libraries structure, classes, and subdirectories.

## Autoloading Classes

The solution to the problem of libraries with multiple and nested classes is to use the library's own autoloader mechanism. Any library with a complex structure should have one. The only remaining trick is getting Yii to use it. Often this is as simple as included the autoloader file that comes with the library:

```
$al = dirname(__FILE__) .  
    '/protected/vendor/vendorname/autoload.php';  
require_once($al);
```

In many cases, that will suffice. Once again, if the library will be used by the entire application, you can put this code in the bootstrap file (**index.php**) to register the autoloader for the whole site. You'd want to place this after the application has been configured but before the `run()` method call. This does require breaking the original one line in the index file into two separate ones:

```
// Create the application instance:  
$app = Yii::createWebApplication($config);  
  
// Include the vendor's autoloader:  
$al = dirname(__FILE__) .  
    '/protected/vendor/vendorname/autoload.php';  
require_once($al);  
  
// Run the application:  
$app->run();
```

On the other hand, if the library will be used sparingly, you'll want to add the inclusion of the autoloader to the specific controller actions.

A third option is to register the third-party autoloader with Yii so that Yii will attempt to use it automatically. This is done through the `registerAutoLoader()` method of the `YiiBase` class.

```
Yii::app()->registerAutoloader(array(  
    'Vendor_Autoloader_ClassName',  
    'vendorAutoloaderMethodName'  
));
```

The method's lone argument needs to point to the classname and method name of the autoloader (when passing an array). That class must already be within Yii's include path, which means that class file (or the directory it's in) must already have been imported.

{NOTE} You can also create your own autoloaders and register those with the Yii application.

Sometimes an autoloader won't play nicely with Yii's autoloader and you'll still get "class not found" error messages. This can happen when Yii only uses its own autoloader, never turning to the third-party library's one. In those cases, you'll need

to unregister the Yii autoloader, include the library's autoloader, and then reregister the Yii autoloader. Here's how that looks:

```
Yii::import('application.vendor.vendorname.*');
spl_autoload_unregister(array('YiiBase', 'autoload'));
$al = dirname(__FILE__) .
    '/protected/vendor/vendorname/autoload.php';
require_once($al);
spl_autoload_register(array('YiiBase', 'autoload'));
```

The reason this approach works is that the application will now first use the third-party library's autoloader, then Yii's. This means Yii will no longer complain about not finding a class definition.

## Mapping Classes

Yet even one more way to resolve the issue of being unable to find classes is to tell Yii about your classes via class mapping. This is simply a matter of creating aliases so that when you attempt to create an object of type X, Yii will know to use the definition found in **path/to/X.php**.

Class mapping is accomplished by assigning an array to the `$classMap` property of the `YiiBase` class:

```
Yii::$classMap = array(
    'Stripe' => 'protected/vendor/stripe/lib/Stripe.php',
    'Another' => 'protected/vendor/another/lib/Another.php'
);
```

I think of class mapping as a last resort, as proper use of autoloaders should suffice, but I wanted to at least mention that this option exists.

## Using Namespaces

Thanks to support for namespaces in PHP 5.3, many PHP frameworks have turned to a namespace structure. This is true for Zend Framework 2, Symfony 2, and will be the case with Yii 2 as well.

The PHP Specification Request level 0 (PSR-0) defines the rules for which PSR-0 compliant libraries should autoload classes. If you want to use a third-party library that abides by PSR-0, then you can use namespace references to the library's classes in your Yii application (assuming you're using PHP 5.3 or greater).

First, you'll want to create an alias to the library's root folder:

```
Yii::setPathOfAlias('VendorName',  
    Yii::getPathOfAlias('application.vendors.vendorname'));
```

Now you can use the namespace syntax to create objects, starting with *VendorName* as the root namespace. Again assuming that there is a **VendorName/Subdir/Charlie.php** file, you can now do this:

```
$charlie = new VendorName\Subdir\Charlie();
```

Of course this only works with a properly setup autoloader.

## Working with Composer

Many PHP frameworks and libraries these days are installed using [Composer](#). In fact, Yii 2 will use Composer as well. Composer is a *dependency manager* for PHP. This simply means that you create a file that defines the requirements for a project, you run a command in the terminal, and Composer will make sure those requirements are met, downloading and installing the necessary packages. Composer is not hard to learn and use, and you really can't be a modern PHP programmer without using it. If you haven't picked Composer up yet, there's no time like the present!

To install Composer:

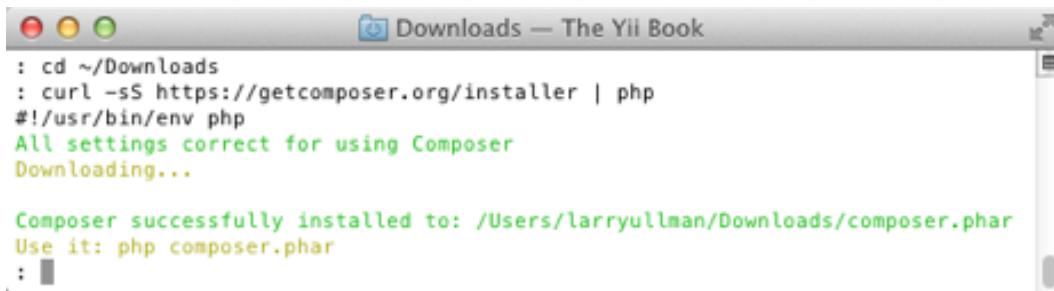
1. Access your computer from a command-line interface.
2. Move to a logical destination directory for Composer

```
cd /path/to/dir
```

You can install Composer anywhere, and you wouldn't want to install it within any specific project's directory. On Windows, I might install Composer within my home directory or in the XAMPP folder, if I've installed that. On Mac OS X, I might install Composer in my **Sites** folder.

3. Execute the following command (**Figure 20.2**):

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



```
Downloads — The Yii Book
: cd ~/Downloads
: curl -sS https://getcomposer.org/installer | php
#!/usr/bin/env php
All settings correct for using Composer
Downloading...
Composer successfully installed to: /Users/larryullman/Downloads/composer.phar
Use it: php composer.phar
:
```

Figure 20.2: Installing Composer.

That line will use cURL to download the Composer installer, and then use your local version of PHP to run the installer. If you get an error message about not being able to find or recognize PHP, you'll need to change the end of that command to include a full path to your PHP executable (such as C:\xampp\php\php.exe).

Those steps take care of the installation of Composer, assuming the process worked. You should find the file **composer.phar** in the folder you used (in Step 2). That script will do the work of installing dependencies.

{NOTE} You need to install Composer only once on each computer, not once for each site.

With Composer installed, you then identify the dependencies for a project. That's accomplished by creating a file of JSON (JavaScript Object Notation) data named **composer.json**. You'd place this file within one of your project's directories. In the case of a Yii-based site, I'd use **protected**, for reasons to be explained shortly.

{NOTE} Composer installs libraries for a specific project only, it does not perform a global installation.

At the very minimum, the **composer.json** file would identify the requirements for the project. This goes within a "require" section, using the syntax **package:version**:

```
{
    "require": {
        "library": "x.y.z"
    }
}
```

The package name includes the vendor name and the project name, with the two often being the same. You can find a list of available packages at [Packagist](#). This

is the main repository for Composer packages, although some frameworks use their own.

To add repositories for Composer to use, change the JSON syntax to first list the additional repositories, such as the one for the Zend Framework:

```
{  
    "repositories": [  
        {  
            "type": "composer",  
            "url": "https://packages.zendframework.com/"  
        }  
    ],  
    "require":  
        {  
            "zendframework/zend-mail": "2.0.*"  
        }  
}
```

The above adds the Zend Framework's repository to the list of repositories for Composer use, allowing Composer to install libraries from it, too.

As for the version, if you use a specific version, such as 1.2.3, only that version would ever be installed. If you use an asterisk for any number, that's a wildcard: 1.2.\* will install the latest version within the 1.2 family. If you want the latest version regardless of the numbers, you'd use ... Composer only installs stable versions, by default.

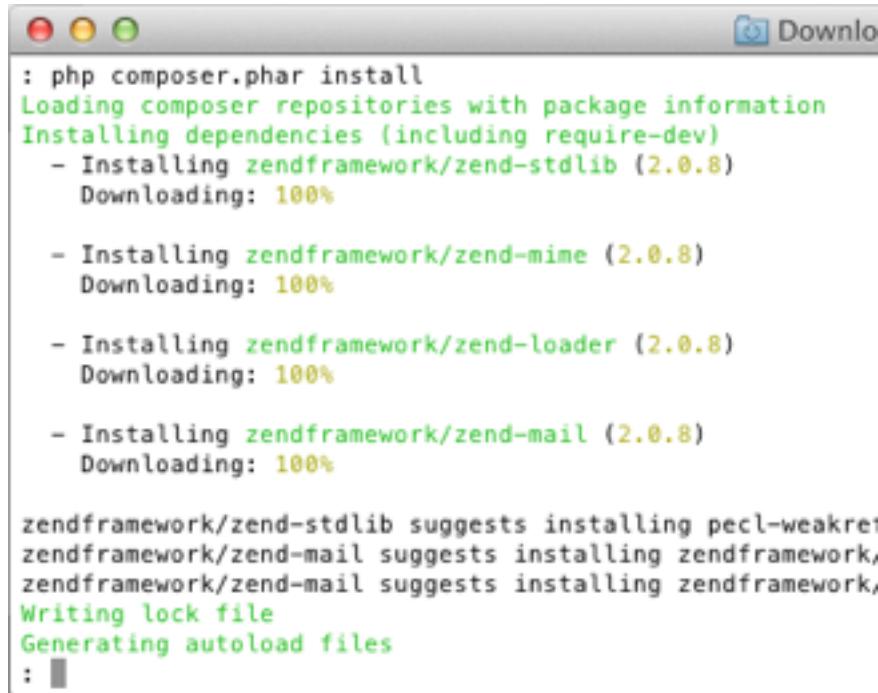
With Composer installed and the dependencies for the project identified, the final step is to have Composer install the dependencies. For this, you again turn to the command line:

1. Access your computer from a command-line interface.
2. Move to the directory where you placed the **composer.json** file.
3. Execute the following command (**Figure 20.3**):

```
php /path/to/composer.phar install
```

This line uses the **composer.phar** script to install the necessary dependencies. Depending on your setup and your operating system, you'll most likely need to explicitly set the path to the PHP executable and/or the path to **composer.phar**.

*{TIP}* Ideally, you should add your PHP executable to your environment's path. Look online to find instructions for your operating system.



```
: php composer.phar install
Loading composer repositories with package information
Installing dependencies (including require-dev)
- Installing zendframework/zend-stdlib (2.0.8)
  Downloading: 100%

- Installing zendframework/zend-mime (2.0.8)
  Downloading: 100%

- Installing zendframework/zend-loader (2.0.8)
  Downloading: 100%

- Installing zendframework/zend-mail (2.0.8)
  Downloading: 100%

zendframework/zend-stdlib suggests installing pecl-weakref
zendframework/zend-mail suggests installing zendframework,
zendframework/zend-mail suggests installing zendframework,
Writing lock file
Generating autoload files
:
```

Figure 20.3: Using Composer.

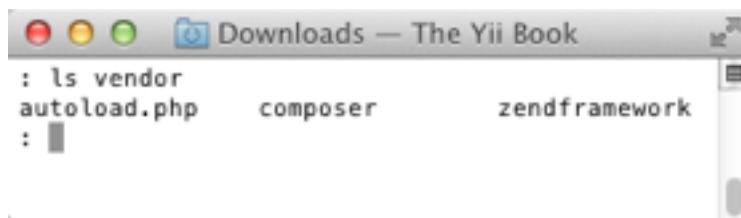


Figure 20.4: Composor installed these files.

4. Look within the **vendor** directory to find the installed stuff (**Figure 20.4**).

The above steps also create a **composer.lock** file in the same directory as **composer.json**. This file acts as a record of what was installed when.

After changing the requirements in **composer.json**, or when packages are updated, you'll want to have Composer update your installation. This is done via the Composer's **update** command:

```
php /path/to/composer.phar update
```

To update Composer itself, execute this command:

```
php /path/to/composer.phar self-update
```

Composer itself will warn you if you haven't updated it in more than a month.

## Using Symfony

[Symfony](#) is a popular PHP framework that's been around since 2005 and is currently in version 2. Supported by [SensioLabs](#), Symfony has numerous useful components, pretty good documentation, and is easily used as a third-party tool in a Yii-based application. In my informal poll for what libraries I should demonstrate in this chapter, Symfony was frequently mentioned.

For the Symfony example, I'm going to use its [DomCrawler](#) component. It provides an easy and powerful way to traverse HTML and XML documents. The specific example could be the basis of a crawler you create to index Web pages (perhaps in conjunction with the Elasticsearch example later in this same chapter).

I'm not going to discuss Symfony in detail. For more information on this framework, see the Symfony site or search the Web. The focus here is on using Symfony within Yii.

### Installing the Components

To install the necessary Symfony component, add the following to your **composer.json** file:

```
{  
    "require": {  
        "symfony/css-selector": "*",  
        "symfony/dom-crawler": "*"  
    }  
}
```

The DomCrawler component is the key one, but its extended functionality uses the Symfony CSSSelector component.

Once you've configured Composer, run its `install` command, as already explained. The result will be a `protected/vendor/symfony` folder, with appropriate subfolders.

## Autoloading the Components

Since Symfony was installed via Composer, it's best to tell Yii about the Composer autoloader by replacing the last line in the `index.php` file with:

```
// Use the Composer autoloader:  
$app = Yii::createWebApplication($config);  
$composerAutoloader = dirname(__FILE__) . '/protected/vendor/autoload.php';  
require_once($composerAutoloader);  
$app->run();
```

This code was explained earlier in the chapter.

## Using Symfony

To use Symfony, create the appropriate controller and action. For simple testing purposes, I normally just throw these attempts into a new action in the "Site" controller:

```
public function actionSymfony() {  
    // Do stuff here!  
}
```

For this example, I want to read in a page of HTML, find every link, and pass those links to the view file to be displayed. Alternatively, you might:

- Read in the page
- Index the site's content for your search engine
- Pull all of the links out of the content
- Repeat these steps for each link on that same site

So here's what the complete function would look like, with inline comments:

```
public function actionSymfony() {
    // Read in the HTML:
    $html = file_get_contents('http://localhost/');

    // Find all the links on the page:
    $crawler = new Crawler($html);
    // Filter by "a" tag:
    $found = $crawler->filter('a');

    // Store the links in an array:
    $links = array();
    foreach ($found as $link) {
        // $link->nodeValue will be the link text
        $links[$link->nodeValue] = $link->getAttribute('href');
    }

    // Pass the links to the view file:
    $this->render('symfony',array('links'=>$links));
}
```

Even without knowing Symfony, the commented code above should be easy enough to follow. There's one catch, however. The `Crawler` class is namespaced in Symfony, meaning that the Composer autoloader won't be able to find it (**Figure 20.5**).

## PHP warning

```
include(Crawler.php): failed to open stream: No such file or directory
```

```
/Users/larryullman/Sites/framework/YiiBase.php(427)
```

**Figure 20.5:** *Yii can't find the Crawler definition.*

The solution is to tell Yii about the location of `Crawler` within the namespace. That's done via PHP's `use` command:

```
use Symfony\Component\DomCrawler\Crawler;
```

The next catch, though, is that you can't place this code within a function, as PHP won't let you use `use` within a block. That line must go outside of the class definition:

```
<?php  
use Symfony\Component\DomCrawler\Crawler;  
class SiteController extends Controller
```

The final step is to create the view file that will display this links:

```
<h2>Links Found By Symfony</h2>  
  
<div>  
<?php foreach ($links as $name => $url) {  
    echo '<p><strong>' . $name . '</strong>: ' . $url . '</p>';  
}  
<?  
</div>
```

**Figure 20.6** shows the output after reading in the default Yii template home page.

## Links Found By Symfony

**Home:** /yii-test-ch20/index.php/site/index

**About:** /yii-test-ch20/index.php/site/page?view=about

**Contact:** /yii-test-ch20/index.php/site/contact

**Login:** /yii-test-ch20/index.php/site/login

**documentation:** <http://www.yiiframework.com/doc/>

**forum:** <http://www.yiiframework.com/forum/>

**Yii Framework:** <http://www.yiiframework.com/>

**Figure 20.6:** The names and URLs of the found links.

## Using Swift Mailer

**Swift Mailer** is a library that provides an object-oriented interface for reliably sending out email using PHP. As you may know, sending plain-text email in PHP is blazingly simple, assuming the server is properly configured. If it's not, then...it'll be much harder. Sending HTML email, on the other hand, pretty much always requires the use of a third-party library in order to be done reliably.

Swift Mailer has many great features:

- Ability to send email via sendmail, postfix, SMTP, or other mechanisms
- Support for authentication
- Prevents header injection attacks to reduce the chance of spam being sent
- Support for attachments and inline images

There are Yii extensions for Swift Mailer, such as [wkd-swiftmailer](#), but I'll just tap into Swift Mailer directly here.

## Installing Swift Mailer

The first thing you'll need to do is install Swift Mailer in your application.

1. Create a **protected/vendor/swiftmailer** directory.
2. Download the Swift Mailer library from <http://swiftmailer.org/>.
3. Expand the downloaded file (the download will be named something like **Swift-5.1.0.tar.gz**).
4. Copy the **Swift-x.y.z/lib** folder (created by Step 3) to **protected/vendor/swiftmailer**.

And now you're ready to use the Swift Mailer.

## Fixing the Autoloading

Again, for testing purposes, I'll just create a new controller action for sending an email via Swift Mailer:

```
public function actionEmail() {  
}
```

Swift Mailer has its own autoloader, but it's not one that's easily used in conjunction with the Yii autoloader. So the solution, already explained, is to unregister Yii's autoloader, include the Swift Mailer autoloader, and then re-register Yii's autoloader. Here's that code:

```
Yii::import('application.vendor.swiftmailer.lib.*');  
spl_autoload_unregister(array('YiiBase', 'autoload'));  
require_once 'swift_required.php';  
spl_autoload_register(array('YiiBase', 'autoload'));
```

That goes within the `actionEmail()` method.

Now Swift Mailer can be used to send out an email. As with Symfony, I'm not going to go into too much detail, but will provide enough to get you started. For more information, see the Swift Mailer site or search online.

## Using Swift Mailer

First, to be clear, all of the following code would also go in the `actionEmail()` method, after the adjustments to the autoloaders that were just made.

As for using Swift Mailer, you'll want to start by creating a new instance of the `Swift_Message` class. This represents the email itself. You would then configure it by setting the subject, from address, to address, body, etc. Alternatively, you can do all that in one step by chaining method calls:

```
// Create the message:  
$message = Swift_Message::newInstance()  
// Give the message a subject:  
->setSubject('Testing Swift Mailer')  
// Set the From address using an array:  
->setFrom(array('larry@larryullman.com' => 'Larry Ullman'))  
// Set the To addresses using an array  
->setTo(array('testing@example.com'))  
// Give the message a plain text body:  
->setBody('Here is the message itself')  
// Optionally add an HTML version of the body:  
->addPart('<p>Here is the message itself</p>', 'text/html');
```

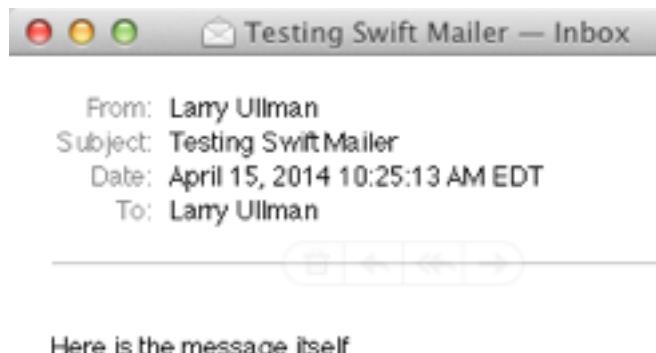
Having defined the message, the next step is to send it out. This is done via a “transport”. A transport can be an SMTP configuration, PHP’s `mail()` function, or your own custom transport. For example, to send the email via your SMTP server, with authentication, you’d use this code:

```
// Connect to smtp.example.com using port 587:  
$tr = Swift_SmtpTransport::newInstance(  
    'smtp.example.com', 587)  
// Set the username:  
->setUsername('larry@larryullman.com')  
// Set the password:  
->setPassword('1234password');
```

Finally, the message is sent through the transport using the `Swift_Mailer` class:

```
$mailer = Swift_Mailer::newInstance($tr);  
$result = $mailer->send($message);
```

And that should do it (**Figure 20.7**).



**Figure 20.7:** The received email (it's stunning).

## Using Elasticsearch

For the last example of this chapter, I want to use the third-party Elasticsearch library. [Elasticsearch](#), in case you're not familiar with it, is, well, the latest and greatest search engine around.

Elasticsearch is a real-time, distributed search engine and more. Elasticsearch supports full-text searching, structured searching, and analytics. Elasticsearch, like Solr, is built on top of Apache Lucene, a Java-based full-text search engine. Lucene, though, is not quite so easy to use (I've played with it and Solr extensively and...it wasn't fun). Elasticsearch uses a RESTful API and JSON for its interactions, making common activities—indexing content, searching content, etc.—as simple as performing any other API call.

Elasticsearch is already in use by Wikipedia, StackOverflow, GitHub, and many other extremely popular and active sites. Still, Elasticsearch is remarkably easy to begin using. I'll provide a basic introduction to Elasticsearch here, and demonstrate how you'd use it with Yii to create a search engine for your site.

{TIP} Yii 2 will have some Elasticsearch support built-in!

### Installing Elasticsearch

To start, you need to install Elasticsearch, which is quite simple.

1. Head to <http://www.elasticsearch.org/>.
2. Click DOWNLOAD.
3. Download the ZIP version (or whichever you prefer).
4. Expand the downloaded file.

The download will be a file named something like **elasticsearch-1.1.0.zip**. Expand this file to get a folder named something like **elasticsearch-1.1.0**. That's it! You've now installed Elasticsearch!

{NOTE} Elasticsearch does require that you have the Java Development Kit installed on your machine.

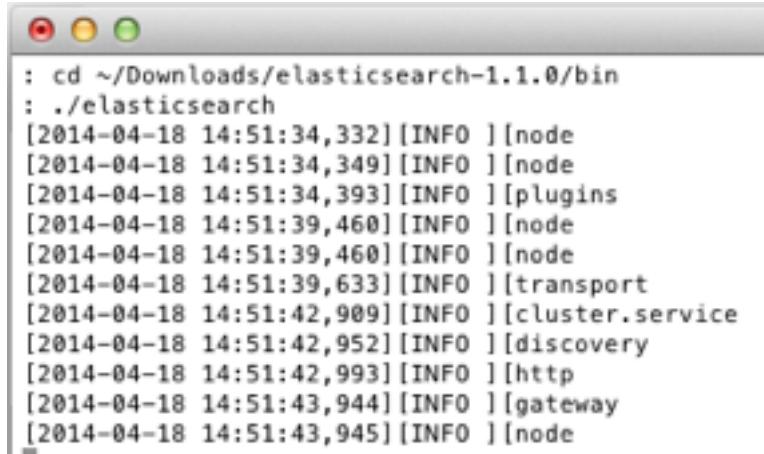
## Running Elasticsearch

Like the MySQL database, Elasticsearch must be running in order to be used. It's started via a command-line interface.

1. Access your computer via a command-line interface.
2. Navigate to the Elasticsearch's **bin** directory.

```
cd /path/to/elasticsearch-1.1.0/bin
```

3. Execute this command: `./elasticsearch` (**Figure 20.8**).



A screenshot of a Mac OS X terminal window. The title bar says "Terminal". The window contains the following text:

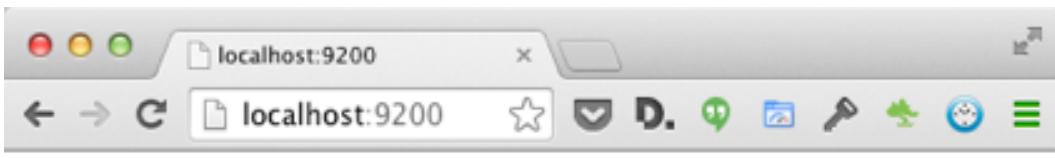
```
: cd ~/Downloads/elasticsearch-1.1.0/bin
: ./elasticsearch
[2014-04-18 14:51:34,332] [INFO ] [node
[2014-04-18 14:51:34,349] [INFO ] [node
[2014-04-18 14:51:34,393] [INFO ] [plugins
[2014-04-18 14:51:39,460] [INFO ] [node
[2014-04-18 14:51:39,460] [INFO ] [node
[2014-04-18 14:51:39,633] [INFO ] [transport
[2014-04-18 14:51:42,909] [INFO ] [cluster.service
[2014-04-18 14:51:42,952] [INFO ] [discovery
[2014-04-18 14:51:42,993] [INFO ] [http
[2014-04-18 14:51:43,944] [INFO ] [gateway
[2014-04-18 14:51:43,945] [INFO ] [node
```

**Figure 20.8:** Starting Elasticsearch.

And that should be it! You should see that Elasticsearch is running. You can confirm this by loading <http://localhost:9200> in your Web browser (**Figure 20.9**).

When it's time to stop Elasticsearch, head back to the same terminal or DOS prompt window where you started it, and press Control+C.

{NOTE} I'm not going into the more advanced uses of Elasticsearch, such as creating clusters of nodes.



```
{  
    "status" : 200,  
    "name" : "Fixx",  
    "version" : {  
        "number" : "1.1.0",  
        "build_hash" : "2181e113dea80b4a9e31e58e9686658a2d46e363",  
        "build_timestamp" : "2014-03-25T15:59:51Z",  
        "build_snapshot" : false,  
        "lucene_version" : "4.7"  
    "tagline" : "You Know, for Search"  
}
```

Figure 20.9: Elasticsearch is now running.

## How Elasticsearch Works

As I already mentioned, Elasticsearch uses a RESTful API. This means that accessing different URLs has different effects. For example, searches will be performed through **http://localhost:9200/\_search**. But before you can search, you must have an index of data to search through.

Indexes are created by sending a PUT request to: **http://localhost:9200/index/type/id** (with those three values being replaced by actual ones). Elasticsearch stores its data in “indexes”, which are the equivalent to databases. Within a single index (or database), you can have multiple “types”, which are equivalent to tables. Each record of data stored in a type can be associated with an ID value. In Elasticsearch parlance, a record that is indexed is a “document”, containing both metadata and a body.

With all of this in mind, this means that you can create an index for a page of content in a CMS example by performing a PUT request to **http://localhost:9200/cms/page/1**. I’ll return to the syntax of that request later.

Note that unlike a database, you don’t have to take any steps to create an index or type in advance. The simple act of making the PUT request creates everything necessary.

Existing indexes are searched using a POST request to—

**http://localhost:9200/index/type/\_search**

—with both the index and type being optional. This means that a request to **http://localhost:9200/cms/\_search** will perform a search within the “cms” in-

dex and a request to `http://localhost:9200/cms/page/_search` will perform a search within the “page” type of that index.

{NOTE} Indexes allows you to easily use one Elasticsearch instance with multiple data sets.

When it comes time to perform a search, Elasticsearch has its own query domain specific language (DSL). This is roughly equivalent to SQL. At the most basic level, you can pass a “q” value to perform a search:

`http://localhost:9200/cms/page/_search?q=yii`

Basic queries use the Lucene query parser syntax. By default, all indexed fields will be searched. As you’ll soon see, you can indicate fields when indexing content, such as author, date, and content for a page in a CMS site.

{TIP} You can simultaneously search multiple indexes or types by separating the names by commas in the URL.

## Interacting with Elasticsearch

For the rest of the chapter, most of the interactions with Elasticsearch will be done through the Elasticsearch PHP library. However, I think it helps to first communicate with Elasticsearch directly in order to understand the basics of the engine. This will be much the same as learning how to first execute SQL queries straight against the database before doing so from a PHP script.

Communications are done through the URL, `http://localhost:9200`. You can access this through your browser or using cUrl in the command-line. When it comes to interacting with Elasticsearch, I highly recommend the excellent Sense extension for the Chrome browser, available in the Chrome store (it’s free).

After installing Sense, you’ll see a pane on the left for editing and a pane on the right for showing Elasticsearch’s response. Sense supports autocompletion (using Enter/Tab), code indentation, code folding, and a history of executed commands. Click the green triangle (pointing right, like a Play button) to execute the edited command.

To start, let’s create an index, type, and ID to store a bit of data for later searching. To do that, you’ll need to make a PUT request. You can execute the following using Sense:

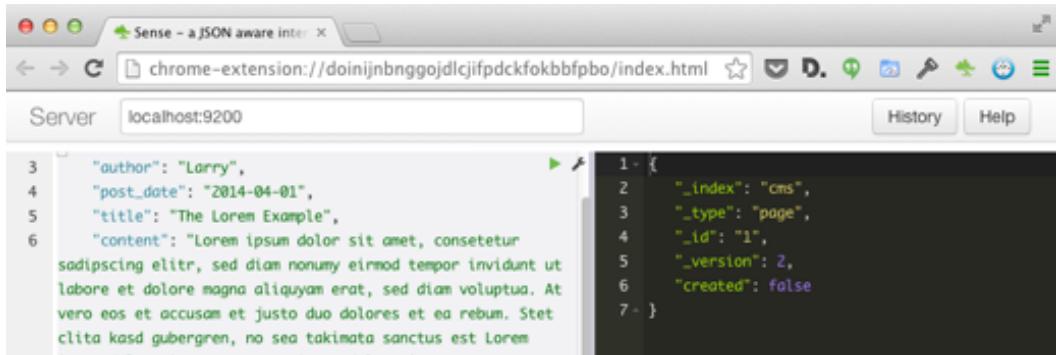
```
PUT /cms/page/1
{
    "author": "Larry",
    "post_date": "2014-04-01",
```

```

    "title": "The Lorem Example",
    "content": "Lorem ipsum dolor..."
}

```

After clicking the green arrow, you should see a positive response on the right (**Figure 20.10**).



The screenshot shows the Sense interface with a JSON editor on the left and a results panel on the right. The JSON editor contains the following document:

```

3   "author": "Larry",
4   "post_date": "2014-04-01",
5   "title": "The Lorem Example",
6   "content": "Lorem ipsum dolor sit amet, consetetur
sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
labore et dolore magna aliquyam erat, sed diam voluptua. At
vero eos et accusam et justo duo dolores et ea rebum. Stet
clita kasd gubergren, no sea takimata sanctus est Lorem
"

```

The results panel shows the indexed document with its ID, type, and version:

```

1 - {
2   "_index": "cms",
3   "_type": "page",
4   "_id": "1",
5   "_version": 2,
6   "created": false
7 - }

```

**Figure 20.10:** A document has been indexed.

{TIP} If you make another PUT request to an existing index/type/ID combination, the result will be an update of that stored content.

To search through that indexed document, use this syntax:

```

GET /cms/page/_search
{
  "query": {
    "match": {
      "author": "Larry"
    }
  }
}

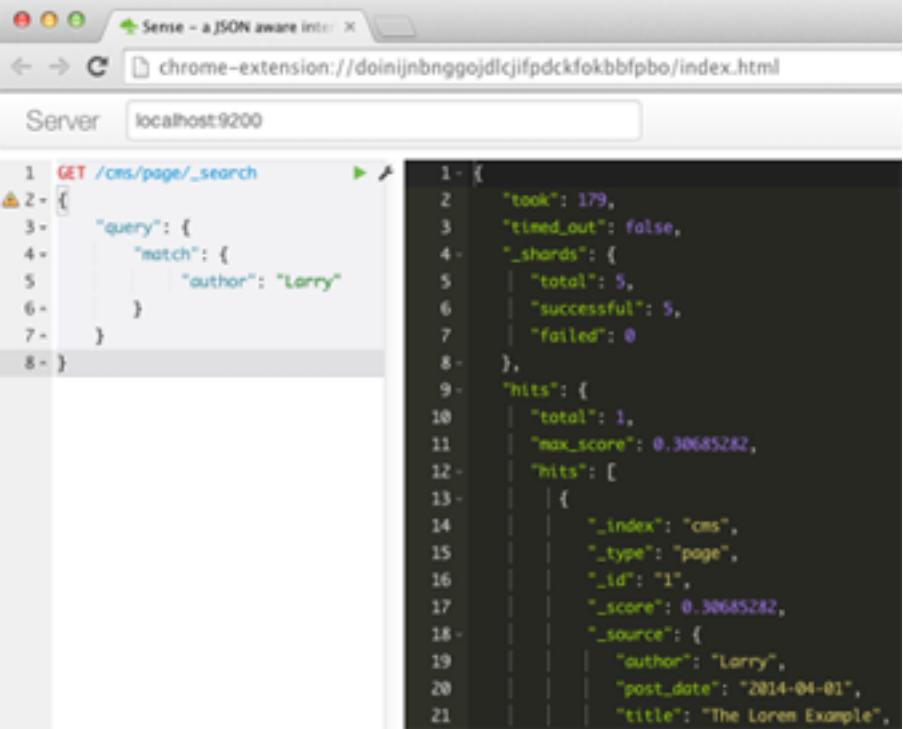
```

That query looks for an author match, which should return one record (**Figure 20.11**).

This is the most basic search one can do. The query results begin with the general information: how long it took, if the request timed out, how many shards were searched, etc. Then you'll see how many hits were found and a maximum relevancy score. This will be followed by the records that count as hits.

This is equivalent to accessing this URL (directly in your browser or via cURL, not using Sense):

[http://localhost:9200/cms/page/\\_search?q=author:Larry](http://localhost:9200/cms/page/_search?q=author:Larry)



The screenshot shows the Sense interface, a JSON-aware interface for Elasticsearch. The URL in the browser is chrome-extension://doinijnbnggojdjcifpdckfokbbfpbo/index.html. The search query is:

```

1 GET /cms/page/_search
2 {
3   "query": {
4     "match": {
5       "author": "Larry"
6     }
7   }
8 }

```

The response is:

```

1 {
2   "took": 179,
3   "timed_out": false,
4   "_shards": {
5     "total": 5,
6     "successful": 5,
7     "failed": 0
8   },
9   "hits": {
10    "total": 1,
11    "max_score": 0.30685282,
12    "hits": [
13      {
14        "_index": "cms",
15        "_type": "page",
16        "_id": "1",
17        "_score": 0.30685282,
18        "_source": {
19          "author": "Larry",
20          "post_date": "2014-04-01",
21          "title": "The Lorem Example",
22        }
23      }
24    ]
25  }
26 }

```

**Figure 20.11:** A basic search, with one hit.

If you wanted to search through every field, you'd use “\_all” instead of the field name.

Again, “match” provides a very simple search, but it does work as a full-text search on multiple words:

```

GET /cms/page/_search
{
  "query": {
    "match": {
      "content": "sed diam"
    }
  }
}

```

That search would return any record that contains “sed” or “diam”, with records that contain both being ranked higher. To turn this into a phrase match—only return records with “sed diam” exactly, you'd use “match\_phrase” instead of “match”.

As a cool feature, Elasticsearch has a built-in highlighter, wrapping found words or phrases in HTML EM tags. To use that, change the JSON request to:

```
GET /cms/page/_search
{
  "query": {
    "match": {
      "content": "sed diam"
    }
  },
  "highlight": {
    "fields": {
      "content": {}
    }
  }
}
```

The query results will be the same, but extra information will be returned in a “highlight” response field. It will contain the portion of the text that contains the highlight (**Figure 20.12**).

```
"highlight": {
  "content": [
    "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, <em>sed</em> <em>diam</em>
    " labore et dolore magna aliquyam erat, <em>sed</em> <em>diam</em> voluptua. At vero
```

**Figure 20.12:** Search results with highlighted terms.

Next, if you change the search term to “Lore”, and click the green arrow, you’ll notice that no hits are returned, despite the first word in the content being “Lorem”. This is due to the default Elasticsearch indexing, which I will address in more detail shortly.

If you want, you can run any of the above searches on the entire index (all of “cms”) by using the URL [http://localhost:9200/cms/\\_search](http://localhost:9200/cms/_search) instead.

Results can also be changed by providing:

- `size`, the number of results to return
- `from`, an offset
- `fields`, the specific fields to return in the hit results
- `sort`, an alternative sort order

{TIP} By default, Elasticsearch returns the top 10 hits for any search query.

The real power comes from the query DSL. You can use configurations to indicate what must be present, what must not be, ranges of numbers or dates, even geolocation! You can also tap into filters to apply limitations to the query results. And

analytics can be used to aggregate data. Elasticsearch is an extremely powerful search engine!

Before moving back to Yii, I'll recommend that you read Joel Abrahamsson's [Elasticsearch 101](#) tutorial. It's easy to digest and does a great job of explaining query DSL in particular.

## Configuring the PHP Client

Interacting with Elasticsearch from PHP makes use of the Elasticsearch PHP library. When you use it, you'll start by creating an instance of the Elasticsearch client:

```
$client = new Elasticsearch\Client();
```

Subsequent tasks—indexing content, performing searches, etc.—will be performed through that object.

To configure the interactions, pass an array of parameters to the creation of the client:

```
$params = array();
// Configure $params.
$client = new Elasticsearch\Client($params);
```

You can see a list of configuration options in the [Elasticsearch PHP library documentation](#).

*{NOTE}* In the code that follows, the client won't be configured, but I wanted to include this bit of information just to be thorough.

## Indexing Content with PHP

To index content using PHP, you'll invoke the `index()` method of the client object, passing along the details. Using the previous example content, creating the same index in PHP would look like so:

```
$client = new Elasticsearch\Client();
$data = array();
$data['author'] = 'Larry';
$data['post_date'] = '2014-04-01';
$data['title'] = 'The Lorem Example';
$data['content'] = 'Lorem ipsum dolor...';
$client->index($data);
```

If you wanted to index that content in Yii, you'd either read through an HTML page (perhaps using the Symfony component) or pull the data from a database. I'll demonstrate the latter later in the chapter.

## Better Indexing

When you store data in Elasticsearch, the engine indexes the content. With text, this means *tokening* it: breaking it into chunks and filtering them. By default, Elasticsearch will break words using spaces and punctuation, filter them as all lowercase—both “Lorem” and “lorem” will be represented as “lorem”, and then filter out any stopwords (such as “the” in English). The indexing engine relies upon an *analyzer* to perform these steps.

When you do a search, another analyzer is applied. It may or may not take the same steps as the index analyzer, depending upon the configuration.

In order to be able to search through stored data making matches on parts of a word, you’ll need to configure how the indexing analyzer works. Unlike in the previous example, where an index was created by indexing content, you can also create and configure an index as a first, separate step. I’ll go through an example, but you may want to read the Elasticsearch documentation for all the gritty details, or [this tutorial](#) for a quick overview.

To allow for partial word matches in the example I’m using, the index analyzer needs to be told to tokenize the words not just on spaces and punctuation, but also on a range of lengths. Reasonably, the range could go from 3 characters to 10. The result would be that the word “Lorem” would be indexed as all of the following:

- lor
- lore
- lorem
- ore
- orem
- rem

(All in lowercase because of the filter.)

In a few pages, I’ll explain the exact code needed to make this change.

Another way you can improve the quality of the indexing and searching is by applying *mapping*. By default, Elasticsearch is schema-less, meaning you don’t need to tell it what kind of data you’re storing. Elasticsearch will dynamically determine the proper data type and index it according to some standard rules.

That’s often fine, and certainly okay for basic examples and practicing, but there are advantages to defining your schema by mapping the data you’ll be indexing to certain types:

- string
- date
- long

- double
- byte
- short
- integer
- float
- boolean
- object
- ip (as in IP address)

Along with those types, there are a couple of geolocation-related types.

At this point, I could demonstrate how to perform better indexing and mapping using just Elasticsearch, or using PHP, but I think it's time to turn to interacting with Elasticsearch from within your Yii application.

## Using Elasticsearch with Yii

To begin interacting with Elasticsearch from the Yii application, you'll need to first install the Elasticsearch PHP library. This is done with Composer, meaning you just need to add this to your **protected/composer.json** file:

```
{  
    "require": {  
        "elasticsearch/elasticsearch": "~1.0"  
    }  
}
```

Then perform an install or update with Composer. The result will be a new **elasticsearch** folder within **vendor**.

To use the Elasticsearch PHP library, it's best just to use the Composer autoloader. You can either include it in the bootstrap file or within a specific controller. If you're following this entire chapter sequentially, this will have already been done in the bootstrap file:

```
require_once($yii);  
  
// Use the Composer autoloader:  
$app = Yii::createWebApplication($config);  
  
$composerAutoloader=dirname(__FILE__).'/protected/vendor/autoload.php';  
require_once($composerAutoloader);  
  
$app->run();
```

For the rest of the chapter, I'm going to assume you'd create a "SearchController", with these methods:

- `actionCreate()`, for creating the index
- `actionIndex()`, for indexing content
- `actionSearch()`, for search the index

The last method will be the default, and you'll probably want to add access control to the other two in a real-life site, limiting the ability to update or create indexes to just an administrator. Better yet, those two methods could be converted into command-line versions (see Chapter 16, "[Leaving the Browser](#)").

For the specific example, I'll use something similar to CMS, this time indexing some book content. The fields will be: title, author, and content. You can download the SQL required to create the database table and records from the [book download page](#).

## Creating the Index

As just mentioned, the `actionCreate()` method of the "Search" controller will create the index. This is going to be a bit complicated, and took some trial-and-error on my part to figure out, but I'll explain the code as best as I can. The following assumes that you've already installed the Elasticsearch PHP library (using Composer) and included the Composer autoloader in your Yii application. All of the following code would then go within the `actionCreate()` method.

To start, create an array of parameters. One parameter can identify the name of the index being created:

```
$params = array();
$params['index'] = 'books';
```

The next goal is to set the index to allow for partial matches. This is done by configuring the "analysis" body setting, which means you'll be assigning an array to `$params['body']['settings']['analysis']`. The term "body" is used there, because it applies to the body of the documents being indexed.

This main array should have two subarrays: "filter" and "analyzer". The filter should identify the filter that parses out strings of 3 to 10 characters in length. The analyzer is then told to use this filter. Here's that code:

```
$params['body']['settings']['analysis'] = array(
    'filter' => array(
        'my_filter' => array(
            'type' => 'ngram',
```

```
        'min_gram' => 3,
        'max_gram' => 10
    )
),
'analyzer' => array(
    'my_analyzer' => array(
        'type' => 'custom',
        'tokenizer' => 'standard',
        'filter' => array('lowercase', 'my_filter')
    )
)
);

```

“ngrams” is the type of filter that can be used to break a string into substrings. It’s provided minimum and maximum values, and the whole configuration is assigned the name “my\_filter”.

The analyzer is of type “custom”, and given the name “my\_analyzer”. This analyzer uses the standard tokenizer, and applies two filters. The first is the normal lowercase filter; the second is the custom filter.

Next, the mappings can be established. Specifically, the mappings for a “book” type. First the configuration will say that the source should be included. Then each property of a “book” is identified by name and type. For the title, author, and content, the custom analyzer is assigned.

```
$params['body']['mappings']['book'] = array(
    '_source' => array(
        'enabled' => true
    ),
    'properties' => array(
        'id' => array(
            'type' => 'integer',
        ),
        'title' => array(
            'type' => 'string',
            'analyzer' => 'my_analyzer'
        ),
        'author' => array(
            'type' => 'string',
            'analyzer' => 'my_analyzer'
        ),
        'content' => array(
            'type' => 'string',
            'analyzer' => 'my_analyzer'
        )
)
```

```
)  
);
```

Hopefully there's nothing too confusing there, but you can read more online about creating analyzers if you're curious.

With all of the parameters defined, you can create an Elasticsearch client and invoke the `create()` method of the `indices()` method:

```
$client = new Elasticsearch\Client();  
$client->indices()->create($params);
```

And that should do it! If a problem occurs, an exception will be thrown. Obviously you can render a view file that says something (although a method like this would only be invoked once and could be invoked from the command-line).

If, while playing around with this, you find you made a mistake in creating the index, you can delete it using this code:

```
$client->indices()->delete(array('index' => 'books'));
```

## Indexing Content with Yii

With the index created and configured, you can start throwing documents into it. As already mentioned, the content could be derived by crawling a Web site or come direct from the database. In this case, I'm going to do the latter. There's no need for ActiveRecord here, you can just run a query on the database, fetch the results, and index them.

This code would go in the `actionIndex()` method:

```
$q = 'SELECT * FROM books';  
$cmd = Yii::app()->db->createCommand($q);  
$result = $cmd->query();  
foreach ($result as $row) {  
    // Index $row.  
}
```

To index the content, you'd again create an array and the Elasticsearch client:

```
$params = array();  
$params['index'] = 'books';  
$params['type'] = 'book';  
$client = new Elasticsearch\Client();
```

This would be before the `foreach` loop. You can see that the first two parameters identify the index name and type name. These will be the same for each indexed item.

Next, you need to assign an array to `$params['body']`. The array should be the names and values of the document being indexed. In this case, that's:

- title
- author
- content

So within the `foreach` loop, you'd have code like this:

```
$params['body'] = array(
    'author'=>$row['author'],
    'title'=>$row['title'],
    'content'=> $row['content']
);
```

Note that you're not referencing the book ID value here, because that value isn't being indexed. The ID value will instead be the value of the index itself (assigned next).

That code needs to be followed by the ID, assigned to `$params['id']`:

```
$params['id'] = $row['id'];
```

And now you can index that content:

```
$client->index($params);
```

Here's the mostly complete method body then:

```
$params = array();
$params['index'] = 'books';
$params['type'] = 'book';
$client = new Elasticsearch\Client();
$q = 'SELECT * FROM books';
$cmd = Yii::app()->db->createCommand($q);
$result = $cmd->query();
foreach ($result as $row) {
    $params['body'] = array(
        'author'=>$row['author'],
        'title'=>$row['title'],
    );
}
```

```
    'content'=> $row['content']
);
$params['id'] = $row['id'];
$client->index($params);
}
```

Again, you'll want to create a view file to show the results, or use this from the command-line.

To confirm this worked, you could now perform a GET request (in the browser) of <http://localhost:9200/books/book/1> (**Figure 20.13**).

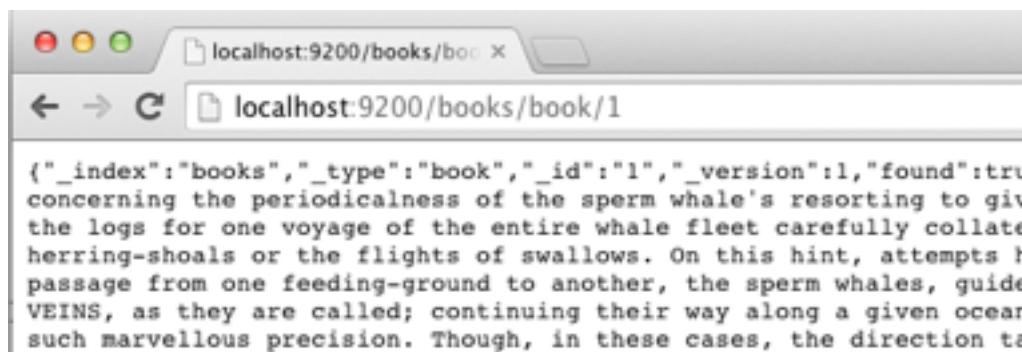


Figure 20.13: The first indexed book.

## Performing a Search with Yii

Last, and most importantly, it's time to search the index. My assumption is that you'd create a static HTML form with a text input named "terms". This form would use the GET method, and point to the `actionSearch()` method of the "Search" controller. That would be set as the default method of the controller:

```
class SearchController extends Controller {
    public $defaultAction = 'search';
```

Thanks to that line, just the URL `/search?terms=blah` will work (as opposed to having to use `/search/search?terms=blah`).

The search form would be one of the very rare forms in Yii that wouldn't need an underlying model.

The `actionSearch()` method should perform a search if terms were provided. Unlike running a search on a database, you don't need to worry about SQL injection attacks here. The search results should be passed as an array to the view file for display.

You can perform a basic search by invoking the `search()` method of the Elasticsearch client, passing an array of parameters to it. For example:

```
$params = array();
$params['index'] = 'books';
$params['type'] = 'book';
$params['body']['query']['match']['_all'] = $terms;
$client = new Elasticsearch\Client();
$results = $client->search($params);
```

The results of executing that code will be the same as running a match query within the browser, as shown earlier. This means that you can loop through the results and find every returned hit.

To be more specific, `$results` will be an array, corresponding to the main array shown in the results in Sense. This means that you'll have these array elements:

- `took`
- `timed_out`
- `_shards`
- `hits`

The `$results['hits']` element will itself have these elements:

- `total`
- `max_score`
- `hits`

So in your PHP code, `$results['hits']['total']` is the number of hits found by Elasticsearch.

If you're ever in doubt as to what's returned by a search, take the same debugging steps you would with MySQL. With MySQL, you would print out the query being run and execute it using another interface. With Elasticsearch, you'd run the same search using the browser or cURL to see the results.

`$results['hits']['hits']` is going to be an array with one element for each hit. That subarray will have these elements:

- `_index`
- `_type`
- `_id`
- `_score`
- `_source`

Within `_source`, you'll find the document's properties. In this case that's author, title, and content.

To reiterate all this, `$results['hits']['total']` is the number of hits found by Elasticsearch. And `$results['hits']['hits'][0]` is the first found hit (the highest ranking one, by default). And `$results['hits']['hits'][0]['_source']['title']` would be the title value for the first found hit. Whew!

Again, just look at the results in Sense if you're ever lost with all of this.

So now, after running a simple search, you have search results ranked by relevancy. And you can display the particulars—author, title, and content—with ease. Further, since the index ID values correlate to the database ID values in this example, it would be easy to link the search results to the page where they could be found (assuming you fleshed out this example).

The only problematic aspect of this minimal approach is that it wouldn't be easy to display the specific content that matched the search term. The example data I used had a few paragraphs for each book. If I indexed an entire book, it'd be ridiculous to show the entire content in the search results.

As previously mentioned, the “highlight” feature of Elasticsearch does a wonderful job in this area, so it'd be great to use that. Unfortunately, this feature is not easily accessible using the client library. That is until you figure out the that the “body” parameter can take a JSON string, just the same as you'd use in the browser via Sense.

## Passing JSON to Search

A very basic search can be accomplished by assigning the search terms to `$params['body']['query']['match']['_all']`. To perform more customized searches, the easiest way is to pass a string of JSON data as `$params['body']`. This is the equivalent to the previous example:

```
$params = array();
$params['index'] = 'books';
$params['type'] = 'book';
$params['body'] = '{
    "query": {
        "match": {
            "_all": "' . $terms . '"
        }
    }
}';
$client = new Elasticsearch\Client();
$results = $client->search($params);
```

The benefit of going with the JSON route is that you can easily customize the search to replicate what you're seeing in the browser using Sense. For example, to add "highlight", you'd do this:

```
{  
    "query": {  
        "match": {  
            "_all": " ". $terms . "  
        }  
    },  
    "highlight": {  
        "fields": {  
            "content": {}  
        }  
    }  
}
```

As in Figure 20.12, this will now return both the hits and the highlighted section of the hits. The highlighted snippets will be found in:

```
$results['hits']['hits'][0]['highlight']['field_name']
```

(In this case, the "field\_name" would be "content", as that's where the search is finding the terms.)

My goal is to show these results, with the highlights, in a display as that shown in **Figure 20.14**.

## Search Results

2 Record(s) Found Searching for "years"

### The Scarlet Plague

... tasted in sixty years and shall never taste again. I sometimes think the most wonderful...

### Moby Dick

... technical phrase—the Season-on-the-Line. For there and then, for several consecutive years, Moby...

**Figure 20.14:** Search results with highlighted terms in the browser.

To accomplish that, I use the above code to customize `$params['body']`.

Next, I create the Elasticsearch client and perform the search:

```
$client = new Elasticsearch\Client();  
$results = $client->search($params);
```

Now I can fetch the number of results:

```
$total = $results['hits']['total'];
```

Finally, I loop through the hits, assigning the data I want into a new array:

```
$hits = array();
foreach ($results['hits']['hits'] as $hit) {
    $id = $hit['_id'];
    $hits[$id]['title'] = $hit['_source']['title'];
    $hits[$id]['highlight'] = $hit['highlight']['content'][0];
}
```

(Still with me so far? If not, make liberal use of `print_r()` or breakpoints to see what you have at each stage of the code.)

To pass all this to the view file, the method concludes with:

```
$this->render('search', array(
    'total' => $total, 'hits' => $hits, 'terms' => $terms));
```

And the view file does this:

```
<?php echo '<h2>' . $total . ' Record(s) Found Searching for "' .
    $terms . '"</h2>';
foreach ($hits as $hit) {
    echo '<div><h3>' . $hit['title'] . '</h3>';
    echo '<p>...' . $hit['highlight'] . '...</p>';
    echo '</div>';
}
```

But that's not all...

## Changing the Highlight

There are two things I don't like about the current highlighting. One is that the fragment returned is rather short. This can be changed by customizing the fragment size within highlight:

```
"highlight": {
    "fields": {
        "content": {
            "fragment_size": 300
        }
    }
}
```

The default is 100 characters; now 300 will be returned (**Figure 20.15**).

## Search Results

2 Record(s) Found Searching for "years"

The Scarlet Plague

... got the food for many men. The other men did other things. As you say, I talked. I talked all the time, and for this food was given me—much food, fine food, beautiful food, food that I have not tasted in sixty years and shall never taste again. I sometimes think the most wonderful...

Moby Dick

... attained, when all possibilities would become probabilities, and, as Ahab fondly thought, every possibility the next thing to a certainty. That particular set time and place were conjoined in the one technical phrase—the Season-on-the-Line. For there and then, for several consecutive years, Moby...

**Figure 20.15:** More text is returned in the highlighted results.

Second, right now Elasticsearch will also find results in the title and author fields, but those won't be highlighted (because only "content" is configured to be). Let's add highlighting to these fields, but this time returning the entire length of the field's value instead just a fragment:

```
"highlight": {
    "fields": {
        "content": {
            "fragment_size": 300
        },
        "title": {
            "number_of_fragments": 0
        },
        "author": {
            "number_of_fragments": 0
        }
    }
}
```

By setting the "number\_of\_fragments" to 0, Elasticsearch won't return a fragment; instead it returns the whole value.

Now Elasticsearch will highlight hits found within the other fields, too (**Figure 20.16**).

This does mean that the PHP code needs to be updated, as it can no longer assume that there will only be one highlight, or that it'll be under "content". Here's that updated code, that goes within the `foreach`:

```
foreach ($hit['highlight'] as $field => $h) {
    $hits[$id]['highlight'][$field] = $h[0];
}
```

The loop goes through each highlight, finding the field and the highlight text. These are then assigned to a `$hits[$id]['highlight']` array.

```
    "highlight": {
        "content": [
            | " of this has proved true. In general, the same remark, only within a
            | among the matured, aged sperm whales. So that though <em>Moby</em> Dick had
            | led the Seychelle ground in the Indian ocean, or Volcano",
            | " attained, when all possibilities would become probabilities, and, a
            | to a certainty. That particular set time and place were conjoined in the one
            | there and then, for several consecutive years, <em>Moby</em>",
            | "; if by chance the White Whale, spending his vacation in seas far re
            | up his wrinkled brow off the Persian Gulf, or in the Bengal Bay, or China S
            | at Monsoons, Pampas, Nor'-Westers, Harmattans, Trades; any wind but the Leva
            | he devious zig-zag world-circle of the Pequod's circumnavigating wake.</p>"
        ],
        "title": [
            | "<em>Moby</em> Dick"
        ]
    }
```

**Figure 20.16:** The title field is now highlighted.

To be clear, this is a replacement of:

```
$hits[$id]['highlight'] = $hit['highlight']['content'][0];
```

Now the view has to be updated to acknowledge an array of highlights:

```
foreach ($hit['highlight'] as $field => $h) {
    echo '<p><strong>' . ucfirst($field) . '</strong>: ' . $h . '</p>';
}
```

You can see the results in (**Figure 20.17**).

## Partial Word Searching (Again)

The final problem with the results so far is that they still aren't showing partial matches. If you were to search using "mob", "Moby Dick" wouldn't be returned. The fix is to change the syntax of the "query" section of the JSON:

```
"query": {
    "multi_match": {
        "query": '"' . $terms . '"',
        "fields": [
            "title^1",
```

## 1 Record(s) Found Searching for "moby"

### Moby Dick

**Content:** of this has proved true. In general, the same remark, only wi  
sperm whales. So that though *Moby Dick* had in a former year been si  
Volcano

**Title:** *Moby Dick*

**Figure 20.17:** Search results with multiple highlighted fields.

```
        "author^1",
        "content"
    ],
    "minimum_should_match": "70%"
}
},
```

This is quite a switch, so I'll explain it in detail. First of all, I've switched from just "match" to "multi\_match", which matches against multiple fields. The search "query" value is still assigned the incoming terms.

Next, I specify the fields to match against. When doing it this way, you can add weight via the caret. Now title and author are both weighted more than content.

Finally, I add the "minimum\_should\_match" field. This is necessary when doing tokenization as, for example, "herman" would be broken down into almost a dozen fragments. Thus, "herman" will match:

- farther
- her
- here
- hermits
- man
- many
- other
- where
- woman

And many others. As the search terms get longer, more and more irrelevant results will be returned. This "minimum\_should\_match" value says that at least X percent of the search terms need to be found in the results. With this example, if I set

that value to 65%, then matches would have to contain one of the following (at a minimum):

- herm
- erma
- rman

In this case, that would still return “Herman Melville” but rule out all the other extraneous results (**Figure 20.18**).

## 1 Record(s) Found Searching for "herman"

### Moby Dick

**Content:** of this has proved true. In general, the same remark, only with sperm whales. So that though Moby Dick had in a former year been seen Volcano

**Author:** Herman Melville

**Figure 20.18:** Properly restricted results.

And partial word matches work, too (**Figure 20.19**).

## 1 Record(s) Found Searching for "mob"

### Moby Dick

**Content:** of this has proved true. In general, the same remark, only with sperm whales. So that though Moby Dick had in a former year been seen Volcano

**Title:** Moby Dick

**Figure 20.19:** Found the white whale.

## Chapter 21

# TESTING YOUR APPLICATIONS

Testing your code is simple in theory, but complex in reality, requires a decent amount of work, and adds a lot of code. Those are the negatives. What you get in return is a more reliable and less buggy application, both now and as you update the code in the future.

Testing was adopted relatively late in the PHP community, which is unfortunate, but at least it is being used more and more. Naturally, Yii supports testing, too.

There are two kinds of testing you can implement in Yii. The first is *unit testing*. The premise behind unit testing is that you write tests that verify that your code does exactly what it should on the smaller-picture, behind-the-scenes level. These tests should inspect every little component of the application— every class and method, asking the question: is the result of executing this code always what it should be?

The second type of testing is *functional*. Functional testing takes a big-picture approach to the application, verifying that it *behaves* as it should.

Loosely speaking, you can think of unit testing as focusing on the code itself, and functional testing as focusing on the user interface. Or you could say that unit tests focus primarily on the models and functional tests look at the views and controllers.

As previously mentioned, testing starts off simple, but real-world and thorough testing is an involved process. But by applying tests to your application, your code and site should be more predictable. Further, subsequent code changes or alterations won't be able to create new bugs in existing code, a common problem as projects are expanded and modified.

It can take some time to become completely fluent in testing, especially when it comes to effectively testing complex structures and processes. But in this chapter, I'll introduce the basic concepts of the two testing types, and explain how to begin adding testing to your Yii projects. I'll also include some tips and tricks, and recommend some resources for learning more about testing in general.

{NOTE} How you test your application will change significantly between Yii 1 and Yii 2, as Yii 2 swaps out the testing framework directly used.

## Test Directories

After creating a new Yii application using the `yiic webapp` command, you'll find a **tests** directory within the **protected** folder. By default, that folder will contain:

- **fixtures**, stores database fixture files
- **functional**, stores functional tests
- **report**, stores test coverage reports
- **unit**, stores unit tests
- **bootstrap.php**, the script executed to run tests
- **phpunit.xml**, the PHPUnit configuration file
- **WebTestCase.php**, the base class for Web-based functional tests

You'll write your unit tests in code stored in the **unit** directory, functional tests in the **functional** directory, and define any fixtures required—to be covered later in the chapter—in **fixtures**. If you generate coverage reports (also covered later), those will be written into **report** (much like Yii itself uses the **runtime** directory to write data on the fly).

## Using PHPUnit

The most popular tool for unit testing in PHP is [PHPUnit](#). Over the next several pages, you'll download, install, and use PHPUnit to implement basic tests of your code.

Understand that the focus in this section will be on using PHPUnit with a Yii-based site. I cannot provide full coverage of PHPUnit itself. For that, you'll want to see the [PHPUnit documentation](#).

The current version of PHPUnit at the time of this writing is 4.1. It requires PHP 5.3.3 or greater. Yii requires that you use at least PHPUnit 3.5 or greater.

### Installation

PHPUnit used to be installed via [PEAR](#), but support for PEAR is being phased out. PHPUnit needs to be installed as an executable on your computer, so you can install it as either a:

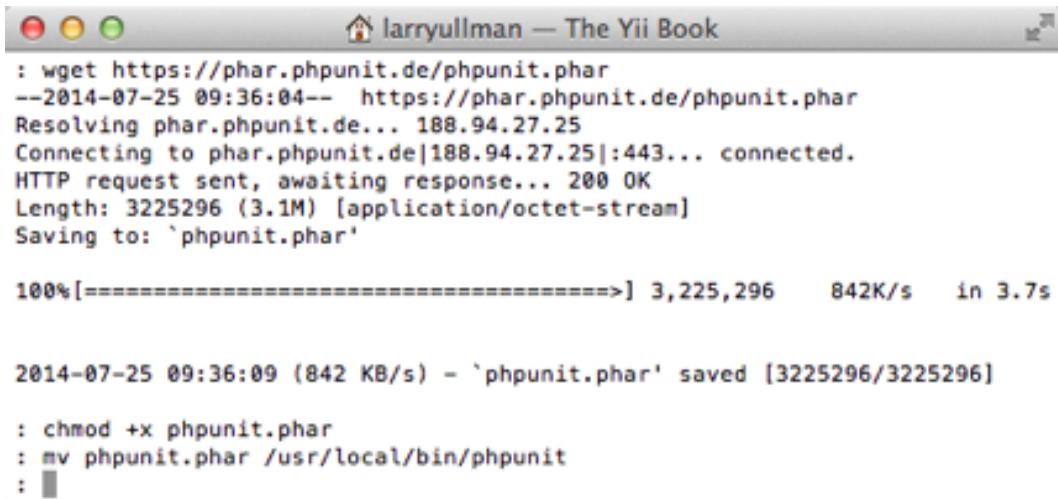
- PHP Archive (PHAR)

- Composer package

To be completely frank, getting PHPUnit installed successfully so that it ran without error via Yii was effortful, only eventually resolved through many Internet searches. Towards that end, I'm going to start with the theoretical instructions, then explain what worked for me.

To install PHPUnit as a PHAR, run the following commands in your terminal (**Figure 21.1**):

```
wget https://phar.phpunit.de/phpunit.phar
chmod +x phpunit.phar
mv phpunit.phar /usr/local/bin/phpunit
```



The screenshot shows a terminal window titled "larryullman — The Yii Book". It displays the command-line process for downloading and installing PHPUnit. The output includes the wget command, file download details (HTTP request, length, save path), completion message, and the chmod and mv commands used to make the file executable and move it to the /usr/local/bin directory.

```
: wget https://phar.phpunit.de/phpunit.phar
--2014-07-25 09:36:04--  https://phar.phpunit.de/phpunit.phar
Resolving phar.phpunit.de... 188.94.27.25
Connecting to phar.phpunit.de|188.94.27.25|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3225296 (3.1M) [application/octet-stream]
Saving to: 'phpunit.phar'

100%[=====] 3,225,296     842K/s   in 3.7s

2014-07-25 09:36:09 (842 KB/s) - 'phpunit.phar' saved [3225296/3225296]

: chmod +x phpunit.phar
: mv phpunit.phar /usr/local/bin/phpunit
:
```

**Figure 21.1:** *Installing PHPUnit as a PHAR.*

If you're using Windows, you'll need to change the last line to move the **phpunit.phar** file into an appropriate directory. On Windows, you'll also want to either add the executable to your path, or provide a full path to it when executing subsequent commands.

If you're already using Composer on a project, see Chapter 20, “[Working With Third-Party Libraries](#),” you may want to just install PHPUnit as a Composer package. To do so, you'd add this code to your **composer.json** file:

```
{
  "require-dev": {
    "phpunit/phpunit": "4.1.*",
    "phpunit/phpunit-selenium": ">=1.2"
  }
}
```

With those requirements defined (or added to your existing requirements), you'd run either `install` or `update` to have Composer install the dependencies (again, see Chapter 20).

That being said, the instructions that worked best for me was to use this `composer.json` configuration:

```
{  
    "require-dev": {  
        "phpunit/phpunit": "3.7.*",  
        "phpunit/phpunit-selenium": ">=1.2",  
        "phpunit/dbunit": ">=1.2",  
        "phpunit/phpunit-story": "*"  
    },  
    "autoload": {  
        "psr-0": {"": "src"}  
    },  
    "config": {  
        "bin-dir": "bin/"  
    }  
}
```

This comes from [a useful blog post](#), and is what got me where I needed to be. Newer versions of PHPUnit just don't seem to work as well, and installing the PHAR didn't get me all the extensions I needed. But running this installation did work, putting a usable PHPUnit executable into the `protected/bin` directory.

## Creating Tests

After you've installed the framework, the next step is to begin defining tests. Let's look at the basic concept, and then expand on that into real-world examples. Once again, I'll point out that knowing how to use PHPUnit itself is necessary in order to fully comprehend everything involved.

To create a unit test, you define a class:

- Whose name ends with the string “Test”
- That extends either `CTestCase` or `CDbTestCase`
- Saved in a file named `ClassName.php`, in `protected/tests/unit`

For example, if you wanted to test the `LoginForm` class from the default Yii example, you'd create the `protected/tests/unit/LoginFormTest.php` file:

```
<?php
class LoginFormTest extends CTestCase {
}
```

Each individual test is defined as a method of the test class. These methods are given the name “testMethodName”, where “MethodName” comes from the method in the associated class that’s being tested:

```
<?php
class LoginFormTest extends CTestCase {
    public function testAuthenticate() {
    }
    public function testLogin() {
}
}
```

{TIP} The testing class constitutes a “suite” of test cases.

Within the test method, you’ll run one assertion for every possible scenario you’ll want to test. The assertions come from PHPUnit, and are defined in [Appendix A of the PHPUnit manual](#). These assertion methods are run on the `$this` object, as `$this` refers to an instance of the test class within the testing method. Since that test class extends from `CTestCase` or `CDbTestCase`, it has all the PHPUnit assertions.

For a completely stupid example, just to get the process started, here’s a dummy test class:

```
<?php
class DummyTest extends CTestCase {
    public function testTrue() {
        $var = true;
        $this->assertTrue($var);
    }
}
```

The class has one test that confirms that a variable has a true value. (Later in the chapter, you’ll create real tests.)

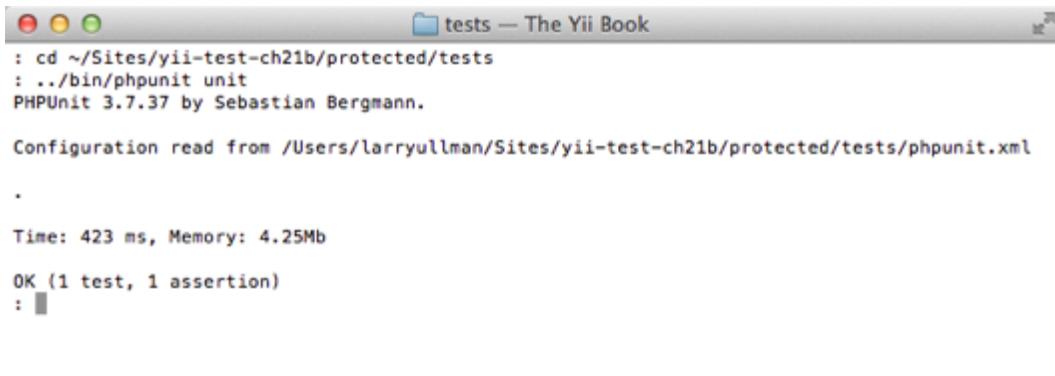
## Running Tests

Once you’ve defined one or more tests, you can run them. This is done via the command-line and the `phpunit` command. The particulars of how you do this will depend upon your PHPUnit installation and configuration. If PHPUnit is a globally accessible executable, you would do the following:

```
cd /path/to/protected/tests  
phpunit <thing to test>
```

For this example, you could use either `phpunit unit/DummyTest.php` to run just that test suite, or `phpunit unit`, which would run every test suite within that folder.

That's how you would normally run PHPUnit, but using the Composer configuration that worked for me, I ran tests using `../bin/phpunit unit`, from within the `protected/tests` directory (**Figure 21.2**).



The screenshot shows a terminal window titled "tests — The Yii Book". The command entered was `cd ~/Sites/yii-test-ch21b/protected/tests` followed by `../bin/phpunit unit`. The output indicates that PHPUnit 3.7.37 by Sebastian Bergmann was used. Configuration was read from `/Users/larryullman/Sites/yii-test-ch21b/protected/tests/phpunit.xml`. The test results show 1 test and 1 assertion were OK, and the total time taken was 423 ms. Memory usage was 4.25Mb. The final status line shows a single vertical bar, indicating all tests passed successfully.

**Figure 21.2:** The test results.

There are additional execution options when running PHPUnit, mentioned in the manual (or via `phpunit --help`). Execution can also be configured by editing the `phpunit.xml` file, which I'll return to.

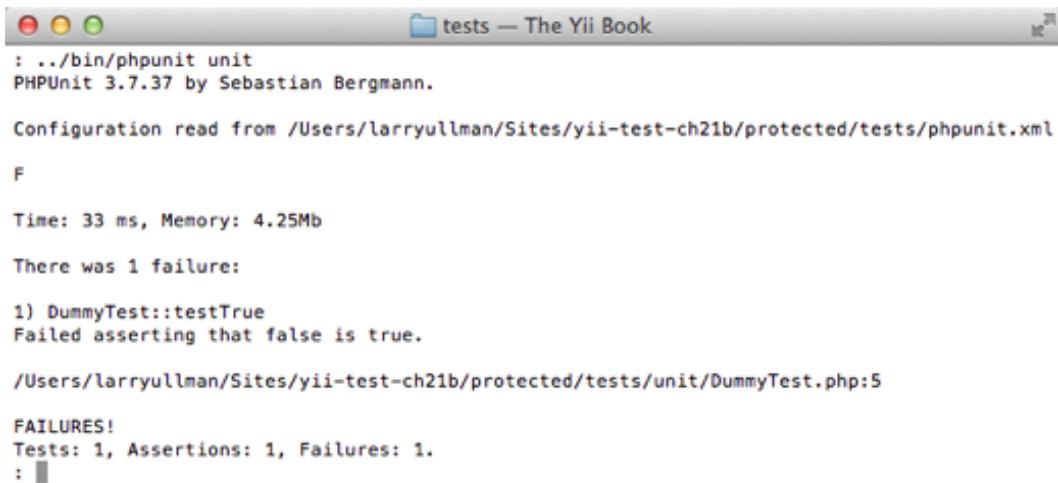
In the PHPUnit output, a single period will be created for every successful test. An "F" is shown when an assertion fails (**Figure 21.3**).

And you'll see other letters when errors occur, a test is skipped, and so forth (see the PHPUnit manual). The total results are shown last.

That's the basic premise, but, again, this didn't work out of the box for me (specifically, a failure results in a warning about being unable to include "PHP\_Invoker"). The final fix is needed in the `protected/tests/bootstrap.php` file.

To understand where that bootstrap file comes in, look at the PHPUnit configuration, set in `phpunit.xml`:

```
<phpunit bootstrap="bootstrap.php"  
    colors="false"  
    convertErrorsToExceptions="true"  
    convertNoticesToExceptions="true"  
    convertWarningsToExceptions="true"  
    stopOnFailure="false">
```



```
: ./bin/phpunit unit
PHPUnit 3.7.37 by Sebastian Bergmann.

Configuration read from /Users/larryullman/Sites/yii-test-ch21b/protected/tests/phpunit.xml

F

Time: 33 ms, Memory: 4.25Mb

There was 1 failure:

1) DummyTest::testTrue
Failed asserting that false is true.

/Users/larryullman/Sites/yii-test-ch21b/protected/tests/unit/DummyTest.php:5

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
:
```

Figure 21.3: A failed test.

```
<selenium>
    <browser name="Internet Explorer" browser="*iexplore" />
    <browser name="Firefox" browser="*firefox" />
</selenium>
</phpunit>
```

The first line says that the **bootstrap.php** file is the bootstrap file for these tests. In other words, PHPUnit should run everything through it. That file looks like so:

```
<?php
// change the following paths if necessary
$yiit=dirname(__FILE__).'/../../../../framework/yiit.php';
$config=dirname(__FILE__).'/../config/test.php';
require_once($yiit);
require_once(dirname(__FILE__).'/WebTestCase.php');
Yii::createWebApplication($config);
```

This is a Yii bootstrap file, very similar to **index.php**. In order to get rid of the last errors when executing PHPUnit with Yii, add this line to the end of that file:

```
YiiBase::$enableIncludePath=false;
```

And now the testing should work.

## Setting Up and Tearing Down

Real-world testing often requires that certain things happen before a test can be run. For example, perhaps an object needs to be created or a connection to a database

established. When test requires certain things to exist or have happened before being run, you can use a “setup” method to do the preparation, rather than doing the work within each individual test method.

To create a setup method, define a method in your class named `setUp()`. This method will automatically be called once before each test is run. Continuing with the silly example, the `$var` variable had been created within `testTrue()`. If multiple tests might use that variable, it could be assigned a value within `setUp()`:

```
<?php
class DummyTest extends CTestCase {
    public $var;
    public function setUp() {
        $this->var = false;
    }
    public function testTrue() {
        $this->assertTrue($this->var);
    }
}
```

This, of course, is still trivial, but the point should be sufficiently clear.

Often, just to be safe, you’ll want to call the parent class’s `setUp()` method first thing within yours:

```
<?php
class DummyTest extends CTestCase {
    public $var;
    public function setUp() {
        parent::setUp();
        $this->var = false;
    }
}
```

Conversely, the `tearDown()` method is executed after each test is run. You won’t need a `tearDown()` method as frequently as you will `setUp()`, but if you tied up resources in the `setUp()` method, such as open a file or network connection, you could free up those resources (e.g., close the file or connection) in `tearDown()`.

{NOTE} The `tearDown()` method will always be called after a test is run, whether or not the test succeeded.

Understand that the `setUp()` and `tearDown()` methods are called *before and after each test case* (i.e., each method is run). PHPUnit also has the `setUpBeforeClass()` and `tearDownAfterClass()` methods. These will each only be called once, regardless of how many test methods you have. The `setUpBeforeClass()` method will be executed before the running of the first test case, and `tearDownAfterClass()` will be executed after the running of the last test case.

## Creating Fixtures

The `setUp()` and `tearDown()` methods just explained help define, and clear up, a state of being for when tests are run. For example, an object of a certain type may need to exist. This state, in unit testing, is called a *fixture*.

The previous example isn't much of a fixture: a simple variable was populated. Real-world testing requires more complex fixtures, and fixtures will sometimes be used by more than one test suite (i.e., class). This most often occurs in cases where a database is involved.

Yii has created a special class for testing with databases: `CDbFixtureManager`.

When tests are run, this tool will:

- Before any tests are run, reset all tables involved to a known state
- Before a specific test is run, reset any involved tables to a known state
- Provide access to the table's data during the test's execution

To use `CDbFixtureManager`, you'll create one or more "fixture files" in **protected/tests/fixtures**. Each file should return an array of representative data for a table, and each file would use the same name as the table for which it's a fixture.

For example, the `Comment` class in the on-going CMS example has the following properties (corresponding to the same-named columns in the `comment` table):

- `id`
- `user_id`
- `page_id`
- `comment`
- `date_entered`

A fixture file for that class would look like so:

```
<?php
# protected/tests/fixtures/comment.php
return array(
    'comment1' => array(
        'user_id' => 23,
        'page_id' => 14,
        'comment' => 'This is the comment.',
        'date_entered' => '2014-07-20 12:09:23'
    ),
    'comment2' => array(
        'user_id' => 3,
        'page_id' => 8,
```

```
    'comment' => 'This is another comment.',  
    'date_entered' => '2014-08-01 19:43:08'  
) ,  
);
```

Each subarray represents one row of data, and each is given an alias as a reference point. Each subarray's element is indexed using the table's column names. If a database table has certain automatic behavior, such as using an auto-incremented integer for the primary key, you don't have to provide that value.

Understand that having defined this file, `CDbFixtureManager` will run a `TRUNCATE TABLE comment` command when first accessed, and then insert those new rows of data into the table. So you'll want to run your tests on a non-live database, for both performance and safety purposes.

*{TIP}* You can prevent `CDbFixtureManager` from automatically manipulating the database using another PHP script, as explained in the Yii guide.

## Configuring the Database

Before using the fixture, you should configure the database connection in Yii such that you're using a test database, not a production one. (Because, again, `CDbFixtureManager` will wipe out the table data.)

The `bootstrap.php` file within the `tests` directory includes the `protected/config/test.php` file. It looks like so, by default:

```
<?php  
return CMap::mergeArray(  
    require(dirname(__FILE__).'/main.php'),  
    array(  
        'components'=>array(  
            'fixture'=>array(  
                'class'=>'system.test.CDbFixtureManager',  
            ),  
            /* uncomment the following to provide test database connection  
            'db'=>array(  
                'connectionString'=>'DSN for test database',  
            ),  
            */  
        ),  
    )  
);
```

You can see that it first includes the primary configuration file. Then it adds the `CDbFixtureManager` fixture tool. Then it allows you to override the database configuration. Change the DSN value there for to use your test database.

## Using Fixtures

To use fixtures in a test suite, define a `$fixtures` public array in the test class. The array's indexes are the fixture names (a generic name), and the values are the corresponding model names:

```
<?php
class CommentTest extends CDbTestCase {
    public $fixtures = array(
        'comment' => 'Comment'
);
```

Do notice that this class extends `CDbTestCase`, not `CTestCase`.

Another example of using fixtures might be:

```
<?php
class PageTest extends CDbTestCase {
    public $fixtures = array(
        'page' => 'Page',
        'comment' => 'Comment'
);
```

{WARNING} Be certain to configure the fixtures to use your test database before running any fixture-based tests.

With the class and fixture defined, the database tables will automatically be populated using that data when you run the tests. You can reference a fixture in a test method using `$this->fixtureName`. Further, you can reference specific rows of data, pulled from the database, using the file's indexes, such as `$this->fixtureName['index']`.

{TIP} Remember that Yii will create database records for you using the fixture data.

```
<?php
class CommentTest extends CDbTestCase {
    public $fixtures = array(
        'comment' => 'Comment'
```

```
);

public function testComment() {

    // Fetch the first record:
    $fetch = Comment::model()->findByPk(1);

    // Confirm it's a Comment object:
    $this->assertInstanceOf('Comment', $fetch);

    // Confirm the stored value:
    $this->assertEquals($this->comment['comment1']['comment'],
        $fetch->comment);

    // Create and save a new comment:
    $comment = new Comment($this->comment['comment1']);
    $comment->user_id = 99;
    $comment->page_id = 88;
    $comment->comment = 'Test';
    $this->assertTrue($comment->save(false));

}
```

First, the test method fetches the comment just created. It can do so using the primary key of 1, as the table was truncated before the previous insertion. After that, the test confirms that `$fetch` is of type `Comment`, which means that the retrieval worked (otherwise it'd be null). And then the test confirms that the stored value equals the provided value.

After that, the test method creates a new comment. Then the method asserts that calling the `save()` method on the object returns true. In other words, given this data, a comment can be saved to the database.

That's some of the basic ideas. You could also test:

- Record updates
- Record deletions
- Data validation routines
- Related models

## Good Testing

In this chapter, I'm essentially covering the fundamentals of testing in Yii, walking through the mechanics more than the theory. That's because the theory can take months to master, and just an example or two or three can't really ingrain testing into you. That being said, I'll end this section with some general tips.

When it comes to designing unit tests for your code, your goals should be making your tests:

- Thorough
- As atomic as possible (some would argue that you should only use one assertion per test, rather than completely test an entire class method at a time)
- Easy to read, write, and execute
- Never superfluous (i.e., don't use any unnecessary assertions)

Your unit tests should never be used to validate user input or handle problems that could occur unexpectedly on a live site. That's what exception handling is for, after all. The point of unit tests is to confirm valid results when valid data is used and appropriate results when invalid data is used. In other words: is the code doing what it should for all possible cases? Test what absolutely should happen and what absolutely shouldn't.

*{TIP}* PHPUnit supports testing the exceptions that should occur for invalid code. See the [PHPUnit manual](#) for more.

Finally, to further your studies, learn as much as you can about PHPUnit. For additional expert advice on testing your PHP applications, check out [Grumpy Learning](#), and pretty much everything that Chris Hartjes does.

*{TIP}* Testing can be taken further to the concept of Test-Driven Development (TDD). With TDD, you define your tests first, and then write code that pass the tests.

## Using Selenium

The second testing component to be explained in this chapter is *functional* testing: looking at how the site operates in the browser. Put another way, functional testing helps verify that the site will work for end users as it should.

Functional testing is accomplished using [Selenium](#). Selenium is a browser automation tool that executes commands in Web browsers and records the results. Yii requires Selenium Remote Control 1.0 or greater (Selenium Remote Control is a Java-based server, since renamed as just Selenium Server).

Selenium can test your site in a number of browsers, the specifics depending upon the OS and selenium version in use. For your Yii application, you'll run Selenium tests through PHPUnit.

## Installation

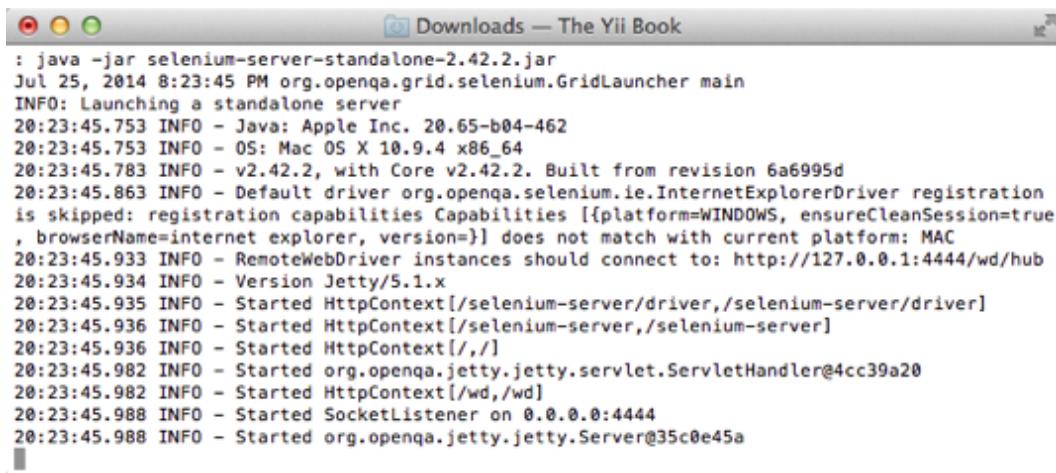
Most of what you'll need to perform testing through Selenium has already been installed along with PHPUnit. In fact, the definition of Selenium tests and the execution of them will also go through PHPUnit. But the last remaining steps are to download the Selenium Server (formerly called the Selenium Remote Control) and start it up.

1. Download the latest version of Selenium Server from <http://docs.seleniumhq.org/download/>.

It's a Java file, so it doesn't matter what operating system you're running. Also, you need to have Java installed.

2. Access your computer via the command line.
3. Move to the directory to where you've downloaded the Selenium Server (`cd path/to/directory`).
4. Start Selenium Server using `java -jar selenium-server-standalone-2.42.2.jar` (**Figure 21.4**).

You'll need to change the specific version to match what you downloaded.



A screenshot of a terminal window titled "Downloads — The Yii Book". The window contains the command "java -jar selenium-server-standalone-2.42.2.jar" followed by its output. The output shows the server launching, detecting the platform as Mac OS X 10.9.4 x86\_64, and starting various components like Jetty and HttpContext.

```
: java -jar selenium-server-standalone-2.42.2.jar
Jul 25, 2014 8:23:45 PM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a standalone server
20:23:45.753 INFO - Java: Apple Inc. 20.65-b04-462
20:23:45.753 INFO - OS: Mac OS X 10.9.4 x86_64
20:23:45.783 INFO - v2.42.2, with Core v2.42.2. Built from revision 6a6995d
20:23:45.863 INFO - Default driver org.openqa.selenium.ie.InternetExplorerDriver registration is skipped: registration capabilities Capabilities [{platform=WINDOWS, ensureCleanSession=true, browserName=internet explorer, version=}]} does not match with current platform: MAC
20:23:45.933 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:4444/wd/hub
20:23:45.934 INFO - Version Jetty/5.1.x
20:23:45.935 INFO - Started HttpContext[/selenium-server(driver),/selenium-server(driver)]
20:23:45.936 INFO - Started HttpContext[/selenium-server,/selenium-server]
20:23:45.936 INFO - Started HttpContext[/,/]
20:23:45.982 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@4cc39a20
20:23:45.982 INFO - Started HttpContext[/wd,/wd]
20:23:45.988 INFO - Started SocketListener on 0.0.0.0:4444
20:23:45.988 INFO - Started org.openqa.jetty.Server@35c0e45a
```

**Figure 21.4:** Selenium Server is now running.

When it comes time to stop Selenium Server, you'll want to use Control + C.

## Creating Tests

To perform functional tests using Selenium, you need an understanding of Selenium itself, of course. For that, I'd recommend you check out the [Selenium documentation](#). But the Yii-specific parts will be quite similar to what you do with the unit tests, as the Selenium tests are run through PHPUnit.

First, you define a class:

- Whose name ends with the string “Test”
- That extends `CWebTestCase`
- Saved in a file named `ClassName.php`, in `protected/tests/functional`

Note that the Selenium test class extends `CWebTestCase`, not `CTestCase` or `CDbTestCase`. For the name, it'll often make sense to use the name of the *controller* being tested. For example, the `yiic` command will create a sample test suite for you, named `SiteTest`, found in the `protected/tests/functional/SiteTest.php` file.

{TIP} For a great introduction to functional testing, checkout the `SiteTest` class generated for you.

Again, within the class, each individual test is defined as a method. These methods are given the name “testSomething”, where “Something” comes from the specific functionality being tested. Note that because functional tests aren't normally tied to specific class methods, the name will likely reflect either the view page being tested, or specific capabilities (e.g., login and logout).

```
<?php
class PageTest extends CWebTestCase {
    public function testViewPage() {
    }
    public function testEditPage() {
    }
}
```

Within the test method, you'll again run one assertion for every possible scenario you'll want to test. The assertions come from both [PHPUnit](#) and [Selenium](#). Again, these assertion methods are run on the `$this` object, as `$this` refers to an instance of the test class within the testing method.

With the assertions, you'll also use Selenium actions. For example, you'll want to load a specific page, maybe click on a link, enter values in a form, and so forth. Understand that the Selenium Server will open a Web browser and perform these

steps for every test. In fact, you'll see that in action when you run your first Selenium test suite.

You can see some example tests in the `protected/tests/functional/SiteTest.php` file:

```
public function testIndex() {
    $this->open('');
    $this->assertTextPresent('Welcome');
}

public function testContact() {
    $this->open('?r=site/contact');
    $this->assertTextPresent('Contact Us');
    $this->assertElementPresent('name=ContactForm[name]');

    $this->type('name=ContactForm[name]', 'tester');
    $this->type('name=ContactForm[email]', 'tester@example.com');
    $this->type('name=ContactForm[subject]', 'test subject');
    $this->click("//input[@value='Submit']");
    $this->waitForTextPresent('Body cannot be blank.');
}
```

The first test method opens the home page (I'll discuss the base URL shortly) and then confirms that the text "Welcome" is found there. The second test method opens the contact page, and the confirms that both some text and a specific form element (whose `name` value is `ContactForm[name]`) exists.

Next, the test will then populate three form elements and submit the form. Finally, the test waits for the text "Body cannot be blank." to appear. This is the expected error message for submitting a contact form with no body value provided. Pretty cool, eh?

{NOTE} Make sure the URLs you use within your tests reflect the proper syntax you're using (e.g., whether or not URL rewriting is enabled).

Other useful action methods include:

- `type()`
- `click()`
- `clickAndWait()`
- `waitForTextPresent()`

With these, and the other commands and assertions, you can have Selenium literally click a checkbox that triggers some JavaScript and then confirm the result. This means Selenium brings to your site cross-browser client-side testing.

## Configuring Selenium

Before you can run any tests, you'll need to configure Selenium. Specifically, you need to tell it:

- The root URL to use
- What browsers to test in

You can do this in a few places:

- **phpunit.xml**
- **WebTestCase.php**
- The test classes

For example, **phpunit.xml** has a section for configuring Selenium:

```
<selenium>
    <browser name="Internet Explorer" browser="*iexplore" />
    <browser name="Firefox" browser="*firefox" />
</selenium>
```

You can edit that file to choose what browsers to test in, if you'd like. For example, if I'm doing this on my Mac, I need to comment out the Internet Explorer option.

The **WebTestCase.php** file has a place already wherein you can define the base URL:

```
define('TEST_BASE_URL','http://localhost/index-test.php');
```

This URL should point to the root of your Web site. For a Yii site, that's a bootstrap file. During testing, it should be **index-test.php**, which includes the test configuration file.

You can also configure Selenium in the test classes themselves, to be explained next.

## Creating Better Tests

Since Selenium tests are run through PHPUnit, many of the same concepts and structures supported in PHPUnit are usable in your Selenium tests, including:

- `setUp()`
- `tearDown()`
- Fixtures

Fixtures here would be used to provide data for HTML forms, for example. The syntax for using fixtures in Selenium tests is exactly the same as using fixtures in unit tests.

In my experience, I found the only way to avoid errors when running Selenium was to configure the root URL in the `setUp()` method:

```
public function setUp() {
    $this->setBrowserUrl('http://localhost/index-test.php');
}
```

You may also want to use `setUp()` to login or otherwise create browser state. Understand that each test is a separate browser interaction, so state will not be maintained (by the browser, and therefore the Selenium tests) from one test to the next.

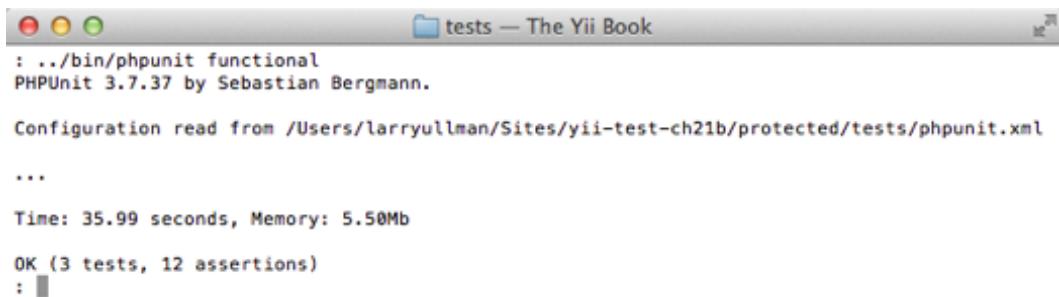
## Running Tests

Once you've defined your Selenium tests, you execute them the same way you execute other tests, using PHPUnit. This time you'll want to point PHPUnit to the **functional** directory:

```
cd /path/to/protected/tests
phpunit functional
```

With the Composer installation I used, this command ran the Selenium tests (**Figure 21.5**):

```
`./bin/phpunit functional`
```



```
: ./bin/phpunit functional
PHPUnit 3.7.37 by Sebastian Bergmann.

Configuration read from /Users/larryullman/Sites/yii-test-ch21b/protected/tests/phpunit.xml

...
Time: 35.99 seconds, Memory: 5.50Mb
OK (3 tests, 12 assertions)
:
```

**Figure 21.5:** Running the functional tests.

While running the tests, you'll see your browsers open and close. Because of this, functional tests will run much slower than unit tests.

{TIP} For an awesome testing experience, you can tell Selenium to take screenshots upon failure. See the PHPUnit manual's section on Selenium for details.

## Creating Tests More Easily

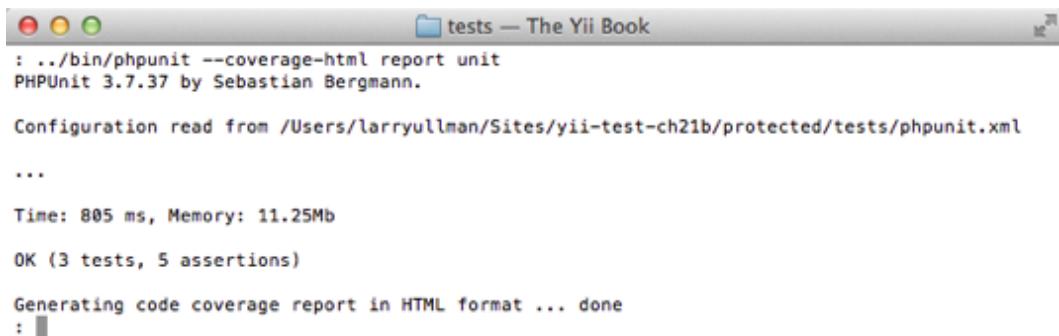
If you really get into Selenium, you may find that writing out the tests by hand quickly becomes tedious, not least of all because you have to look up all the commands. An alternative is to use the [Selenium IDE](#). This tool runs through the Firefox browser and records your tests for you.

If you also use the [Selenium IDE: PHP Formatters](#) add-on, you can export the tests you've created in the IDE as a source file for your Yii project.

## Checking Coverage

After you've put so much time and effort into writing tests, you can become complacent, thinking you'll never have another bug. That's only true if your tests cover every bit of code. But how can you be sure of that? Well, PHPUnit has the ability to generate coverage reports that detail how much of your code is tested.

To create a report, your PHP installation will need to support [Xdebug](#) (which you ought to be using anyway). If you meet that lone requirement, execute `phpunit --coverage-html report unit` (**Figure 21.6**).



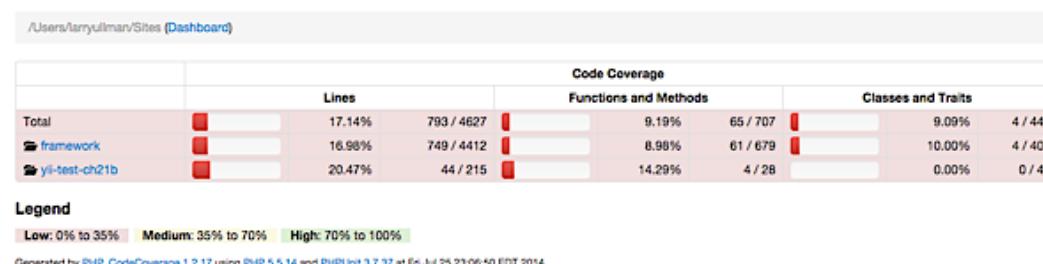
```
tests — The Yii Book
: ./bin/phpunit --coverage-html report unit
PHPUnit 3.7.37 by Sebastian Bergmann.

Configuration read from /Users/larryullman/Sites/yii-test-ch21b/protected/tests/phpunit.xml

...
Time: 805 ms, Memory: 11.25Mb
OK (3 tests, 5 assertions)

Generating code coverage report in HTML format ... done
:
```

PHPUnit will write a coverage report to the `protected/tests/report` directory. If there are meaningful lines of code in your application that are not tested, PHPUnit will catch them (**Figure 21.7**).



You'll notice in the figure that the coverage report looks at not only your site, but the framework, too (since it's part of your site).

{TIP} Many organizations use automated testing, not allowing code be moved to production until it has complete test coverage, and passes all tests.

## Chapter 22

# CREATING A CMS

In my [other books](#), example chapters have been popular additions. Instead of trying to cover new material, the primary goal in example chapters is to present previously explained content within a more complete context. In such chapters, the rubber meets the road, so to speak. Accordingly, I've decided to include two sample chapters in this book as well.

In this sample chapter, I'll walk through the creation of a Content Management System (CMS). A CMS has been used as a hypothetical example throughout most of the book, so putting it all together in one place makes sense. In truth, this is more of a blog than a full CMS, but there's still plenty to learn and see. For alternative examples, check out:

- The Yii blog example (found in the **demos** folder of the framework download itself)
- Vince G's [CMS application](#)

I'll undoubtedly repeat this next sentiment frequently, but do understand that this is not a complete example. The goal with the code and the chapter is to focus on the heart of the application, and how one might go about developing one. There are myriad ways this code and example could be improved, made more secure, made to perform better, and so forth. But it is a reasonable, working example that should be educational.

You can find the complete code in a [GitHub repository](#). You can also use the commit history to roughly track the development of this application. The rest of the chapter will approximately map to the same development path as that commit history. Many of the individual steps are simply applications of ideas, code, and processes explained earlier in the book.

## Project Goals

Every project begins not with a computer, but with pen and paper. (Well, okay, you could also use a note-taking application on your computer.) You need to identify the project's goals, needs, target audience, and so on. The basis of this chapter's example is a CMS or a blog (more of a blog, really). Unlike the variations on this example that have littered the book thus far, in this chapter, the blog features:

- Ability for anyone to comment on a page (not just registered users)
- Three user roles: author, editor, and admin
- Password management using the most secure method: PHP's `password_hash()` function
- Comments shown on the blog page
- Styling accomplished using Twitter Bootstrap 3
- HTML WYSIWYG editor for creating and editing posts
- More SEO-friendly URLs
- A custom widget to browse for posts by month
- Blog searching via ElasticSearch

As you can see, there's a lot to this application, even though it's not 100% finished ([Figure 22.1](#)).

# My Web Application

This is the site description.

## "Wuthering Heights" by Emily Bronte

By Another Author | Oct. 26, 2014

1801.—I have just returned from a visit to my landlord—the solitary neighbour that I shall be troubled with. This is certainly a beautiful country! In all England, I do not believe that I could have fixed on a situation so completely removed from the stir of society.

### About

*Etiam porta sem malesuada magna mollis euismod. Cras mattis consectetur purus sit amet fermentum. Aenean lacinia bibendum nulla sed consectetur.*

### Archives

[October 2014](#)  
[September 2014](#)

## Alice in Wonderland

By larry | Oct. 20, 2014

### Elsewhere

[GitHub](#)  
[Twitter](#)  
[Facebook](#)

**Figure 22.1:** The site's home page.

At the end of the chapter, you'll see some of the notes and thoughts I have for how the example could be completed or expanded.

## Creating the Database

The SQL commands for creating the database are as follows, with a few comments following each.

```
CREATE TABLE `page` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(10) unsigned NOT NULL,
  `live` tinyint(1) unsigned NOT NULL DEFAULT '0',
  `title` varchar(100) NOT NULL,
  `content` longtext,
  `date_updated` datetime DEFAULT NULL,
  `date_published` date DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_page_user_idx` (`user_id`),
  CONSTRAINT `fk_page_user` FOREIGN KEY (`user_id`)
    REFERENCES `user` (`id`) ON DELETE CASCADE
    ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The `page` table is the most important one, representing a page of content (aka, a blog post). It references the `user` table, which reflects the page's author. The `live` column allows pages to be created in draft mode. There are also two dates: one for when the page was last updated and another for when it was, or will be, published.

```
CREATE TABLE `comment` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `page_id` int(10) unsigned NOT NULL,
  `username` varchar(45) NOT NULL,
  `user_email` varchar(60) NOT NULL,
  `comment` mediumtext NOT NULL,
  `date_entered` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  KEY `fk_comment_page_idx` (`page_id`),
  KEY `date_entered` (`date_entered`),
  CONSTRAINT `fk_comment_page` FOREIGN KEY (`page_id`)
    REFERENCES `page` (`id`) ON DELETE CASCADE
    ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The `comment` table links to the `page` table. Each comment has a username, a user's email address, the comment itself, and the date the comment was entered.

```
CREATE TABLE `user` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `username` varchar(45) NOT NULL,
    `email` varchar(60) NOT NULL,
    `pass` varchar(255) DEFAULT NULL,
    `type` enum('author','editor','admin') NOT NULL,
    `date_entered` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    UNIQUE KEY `username_UNIQUE` (`username`),
    UNIQUE KEY `email_UNIQUE` (`email`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The `user` table represents the site's registered users: administrators, editors, and authors. Readers would be non-registered users in my version of this CMS.

## Getting Started

With the database created, it's time to get started creating the application. Logically, this begins by using the `yiic` script to create a new Web application:

```
yiic web app ../path/to/yiobook-cms-ch22
```

For my own personal development, at this point I also:

- Create a new virtual host so that I can access the site using something like `http://ch22`
- Initialize Git
- Open up the permissions on the **models**, **views**, and **controllers** directories (required on my Mac when using Gii in steps to follow)
- Register the new site with [CodeKit](#), which will refresh the browser when I make changes to site files, among many other features (it's Mac-only, but I adore it)

Next, edit the `main.php` configuration file. The most important steps at first are to:

- Enable Gii
- Configure the database
- Enable Web log routing
- Create the models using Gii
- Create CRUD functionality using Gii

By this point, the above should be obvious and easy to do, so I won't go into more detail on those steps.

## Editing the Models

At this point in the process, with very little actual work done on my part, the heart of the application is in place. From here on out, it's mostly about editing (which is one thing I love about Yii). First, let's edit the model classes, specifically focusing on the rules and labels.

For the `User` class, the final rules are:

```
// Required fields when registering:  
array('username, email, pass', 'required', 'on'=>'insert'),  
// Encrypt the password on insert:  
array('pass', 'encryptPassword', 'on'=>'insert'),  
// Username must be unique and less than 45 characters:  
array('email, username', 'unique'),  
array('username', 'length', 'max'=>45),  
// Email address must also be unique (see above), an email address,  
// and less than 60 characters:  
array('email', 'email'),  
array('email', 'length', 'max'=>60),  
// Set the type to "author" by default:  
array('type', 'default', 'value'=>'author'),  
// Type must also be one of three values:  
array('type', 'in', 'range'=>array('author', 'editor', 'admin')),  
// Set the date_entered to NOW():  
array('date_entered', 'default', 'value' =>  
    new CDbExpression('NOW()'), 'on'=>'insert'),  
// The following rule is used by search().  
array('id,username, email, type, date_entered', 'safe', 'on'=>'search'),
```

Most of the rules come from the database constraints. There are specific rules that apply to registration (insertion of a new record), as the username and date wouldn't be required when logging in. The user type has three defined possible values—"author", "editor", and "admin", with "author" being the default. The `date_entered` value is set upon registration. The password field is run through the `encryptPassword()` function (well, rule), upon registration. I'll return to that shortly.

The rules for the `Page` model are:

```
// Only the title is required from the user:  
array('title', 'required'),  
// User must exist in the related table:  
array('user_id', 'exist', 'attributeName'=>'id', 'className'=>'User',  
    'message'=>'The specified author does not exist.'),
```

```
// Live needs to be Boolean; default 0:  
array('live', 'boolean'),  
array('live', 'default', 'value'=>0),  
// Title has a max length and strip tags:  
array('title', 'length', 'max'=>100),  
array('title', 'filter', 'filter'=>'strip_tags'),  
// Filter the content to allow for NULL values:  
array('content', 'default', 'value'=>NULL),  
// Set the date_updated to NOW() every time:  
array('date_updated', 'default', 'value'=>new CDbExpression('NOW()')),  
// date_published must be in a format that MySQL likes:  
array('date_published', 'date', 'format'=>'yyyy-MM-dd'),  
// The following rule is used by search().  
// @todo Please remove those attributes that should not be searched.  
array('user_id', live, title, content, date_updated, date_published',  
    'safe', 'on'=>'search'),
```

Only the page title is required. The user ID value must exist in the `user` table. By default, the `live` property is 0 (not live). The title is run through the `strip_tags()` function to remove any HTML. The `date_updated` value is set to now whenever the model is updated, and the `date_published` value needs to be in a specific format.

Later in the `Page` definition, for the `user_id` label, the value is set to “Author”.

Finally, the rules in the `User` class:

```
// Required attributes (by the user):  
array('username', user_email, comment', 'required'),  
// Must be in related tables:  
array('page_id', 'exist', 'attributeName'=>'id', 'className'=>'Page',  
    'message'=>'The specified page does not exist.'),  
// Strip tags from the comments:  
array('comment', 'filter', 'filter'=>'strip_tags'),  
// Set the date_entered to NOW():  
array('date_entered', 'default', 'value' =>  
    new CDbExpression('NOW()'), 'on'=>'insert'),  
// Username limited to 45:  
array('username', 'length', 'max'=>45),  
// Email limited to 60 and must be an email address:  
array('user_email', 'length', 'max'=>60),  
array('user_email', 'email'),  
// The following rule is used by search().  
// @todo Please remove those attributes that should not be searched.  
array('page_id', username, user_email, comment, date_entered',  
    'safe', 'on'=>'search'),
```

Remember that in this version of the application, the commenter is not a registered user. The username, email, and comment values are required. The `page_id`, which wouldn't be set by the user, must match an existing page in the system. The comments are stripped of any PHP, HTML, or JavaScript using `strip_tags()`. The `date_entered` is set to now.

And that's it for editing the rules.

## Creating Login Functionality

The site requires registered users to create and edit pages of content (i.e., blog posts), so the next step is to implement registration, login, and logout functionality. For best security, the application uses the relatively new `password_hash()` function to encrypt the password. Most of these edits were explained in Chapter 11, “[User Authentication and Authorization](#)”

Logging in will be done with the email address and password. The `protected/views/site/login.php` page is edited towards that end (**Figure 22.2**).

# Login

(Only site writers that have previously been asked to register can log in.)

Fields with \* are required.

**Email Address \***

**Password \***

**Figure 22.2:** The login form.

```
<div class="form-group">
    <?php echo $form->labelEx($model,'email'); ?>
    <?php echo $form->textField($model,'email',
        array('class'=>'form-control')); ?>
    <?php echo $form->error($model,'email'); ?>
</div>
<div class="form-group">
```

```
<?php echo $form->labelEx($model,'pass'); ?>
<?php echo $form->passwordField($model,'pass',
array('class'=>'form-control'))); ?>
<?php echo $form->error($model,'pass'); ?>
</div>
```

(Note that I use “pass”, not “password”, which I prefer because the `User` table uses the former. Also, the above HTML, and the corresponding image, already show the application of the Twitter Bootstrap formatting.)

For security purposes, disable auto-login in the configuration file:

```
'user'=>array(
    'allowAutoLogin'=>false,
),
```

The login form is associated with the `LoginForm` model. The model needs to be updated to use the email address instead of the username. This is really just a matter of replacing uses of “username” with “email”.

The `UserIdentity` class does the rest of the login functionality. The updated `authenticate()` method looks like so:

```
$user = User::model()->findByAttributes(
    array('email'=>$this->username));
if ($user === null) {
    // No user found!
    $this->errorCode=self::ERROR_USERNAME_INVALID;
} elseif (!password_verify($this->password, $user->pass)) {
    // Invalid password!
    $this->errorCode=self::ERROR_PASSWORD_INVALID;
} else { // Okay!
    $this->errorCode=self::ERROR_NONE;
    // Store the type in the session:
    $this->setState('type', $user->type);
    // Store the user ID:
    $this->_id = $user->id;
    // Store the username in the session:
    $this->username = $user->username;
}
return !$this->errorCode;
```

As explained in Chapter 11, `UserIdentity` extends `CUserIdentity`, so it uses “username” everywhere. When making these edits, just remember to treat “email” as “username”.

The method first finds the `User` record by email address. An error is thrown if it doesn't exist. Then the method uses `password_verify()` to compare the provided password (upon logging in) with the stored password. If it's not a match, another error is thrown.

If there were no errors, the user's type, ID, and username (actual username) are stored.

The last step for this to work is to create the `encryptPassword()` method within `User` that properly encrypts the password upon registration. The class's rules already require its usage. Here's its definition:

```
public function encryptPassword($attr, $params) {
    $this->pass = password_hash($this->pass, PASSWORD_DEFAULT);
}
```

Remember that this goes in the `User` class.

To complete the registration, the `UserController` class's access rules need to be changed to allow anyone to create a user. And the `protected/views/user/_form.php` (i.e., the registration form) is edited such that it doesn't provide elements for the type and date entered fields.

## Creating Pages

After establishing the registration, login, and (implicitly) logout functionality, the next step is to implement the heart of the application's functionality: the ability to create and display pages of content (e.g., blog posts). While developing the project, I did this in two steps:

- Implement core capability
- Add in WYSIWYG support

*{TIP}* When developing sites, I normally find it most helpful to populate the database with sample records first, and then implement record creation via PHP.

Towards the first goal, start by updating `protected/views/page/_form.php`:

- Remove the “user ID” element (whose value will come from the user storage)
- Convert the “live” element to a checkbox
- Remove the “date updated” element (because it will be populated by the rules)

# Create Page

Fields with \* are required.

**Title \***

**Content**

**Date Published**

**Live**

**Create**

**Figure 22.3:** The add page form, with the WYSIWYG editor.

At this point there's a simple, but usable, HTML form. Later on, the WYSIWYG editor and the jQuery UI date picker (for the publish date) will be added (**Figure 22.3**).

Finally, the `Page` model needs to pull in the user ID in order to represent the author of the page. That's accomplished by adding this method to `protected/models/-Page.php`:

```
protected function beforeValidate() {
    if(empty($this->user_id)) {
        $this->user_id = Yii::app()->user->id;
    }
    return parent::beforeValidate();
}
```

For new pages, where the `user_id` property would be empty, the current user's ID is assigned to it. For exist pages, perhaps being updated by an editor or administrator, the `user_id` property is not touched. Thus the tie between a page and its original author is maintained.

To make pages display more nicely, add a method to `Page` that outputs the date in a formatted way:

```
public function formattedDate() {
    if ($this->date_published) {
        return DateTime::createFromFormat('Y-m-d',
            $this->date_published)->format('M. d, Y');
    }
}
```

The page view file can then be updated to (**Figure 22.4**):

```
<p class="blog-post-meta">By
<?php echo $data->user->username; ?> |
<?php echo CHtml::encode($data->formattedDate()); ?></p>
```

## Making Comments

Now that pages exist to be viewed, it's time to support the ability to add and view comments. This is a bit more complicated than some of the other examples in the book because the comment form and display of existing comments needs to happen within the context of the page view files. To pull this off, I stole liberally from the Yii blog example that comes with the framework.

To start, the comment form needs to be pulled into the page view file (**Figure 22.5**):

# Alice in Wonderland

By larry | 2014-10-20

All the time they were playing the Queen never left off quarreling about "Off with her head!" or "Off with his head!" Those whom she sentenced were always executed immediately.

**Figure 22.4:** The start of a page's display.

```
<h3>Leave a Comment</h3>

<?php if(Yii::app()->user->hasFlash('commentSubmitted')): ?>
    <p class="text-success">
        <?php echo Yii::app()->user->setFlash('commentSubmitted'); ?>
    </p>
<?php else: ?>
    <?php $this->renderPartial('/comment/_form',array(
        'model'=>$comment,
    )); ?>
<?php endif; ?>
```

The `renderPartial()` method pulls in the comment form from the `views/comment` directory. The `$comment` instance is passed along, so it'll need to be created in the controller. Just before the `renderPartial()` code, a flash message is used to report upon a successful comment post (**Figure 22.6**).

For the comment form itself, three updates are required from the Gii-generated code:

- The action is set to an empty string
- Ajax validation is enabled
- The submit button's label is changed to “Post Comment”

The most complicated part of handling comments within a page view takes place in the “Page” controller. Within the `actionView()` method, a `Comment` object needs to be created. This object could be empty upon first viewing a page or come from the comment form data upon a comment submission. The approach I stole from the Yii blog example is to create a new method within the “Page” controller for handling both scenarios. The “view” action is then updated, too:

**Leave a Comment**

Fields with \* are required.

**Name \***

---

**Email \***

---

**Comment \***

---

**Figure 22.5:** The comment form, with Bootstrap styling.

**Leave a Comment**

**Thank you for your comment.**

**Figure 22.6:** Local version control diagram.

```
public function actionView($id) {
    $page = $this->loadModel($id);
    $comment = $this->newComment($page);
    $this->render('view',array(
        'model'=>$page,
        'comment'=>$comment
    ));
}
```

The key line there is `$comment = $this->newComment($page);`. The `newComment()` method, also defined within the “Page” controller looks a lot like the “create” action would in the “Comment” controller:

```
protected function newComment($page) {
    $comment=new Comment;
    $comment->page_id = $page->id;
    if(isset($_POST['ajax']) && $_POST['ajax']=='comment-form') {
        echo CActiveForm::validate($comment);
        Yii::app()->end();
    }
    if(isset($_POST['Comment'])) {
        $comment->attributes=$_POST['Comment'];
        if($comment->save()) {
            Yii::app()->user->setFlash('commentSubmitted',
                'Thank you for your comment.');
            $this->refresh();
        }
    }
    return $comment;
}
```

As you can see, the method starts by creating a new comment and then assigning over the current page’s ID. Next, the method performs Ajax validation, if an Ajax request was made (this is fairly standard Yii code at this point). Finally, if the form was submitted, the remaining properties are populated, the comment is saved in the database, the flash message is set, and the page is refreshed.

And that takes care of handling comment submissions on the page view.

Finally, the page ought to display the comments, too. First, again stealing from the Yii blog example, create a relationship within the `Page` class that counts the number of comments on a page:

```
public function relations() {
    return array(
```

```
    'comments' => array(self::HAS_MANY, 'Comment', 'page_id'),
    'user' => array(self::BELONGS_TO, 'User', 'user_id'),
    'commentCount' => array(self::STAT, 'Comment', 'page_id'),
);
}
```

The count relationship is created by defining a relation that performs a STAT query on the `Comment` model using the current `page_id`. See Chapter 8, “[Working with Databases](#),” for more on this type of relation.

With that defined (with the page being able to see how many comments it has), the page’s view file is updated:

```
<?php if ($model->commentCount >= 1): ?>
<h3><?php echo $model->commentCount > 1 ?
$model->commentCount . ' Comments' : 'One Comment'; ?></h3>
<?php $this->widget('zii.widgets.CListView', array(
    'dataProvider' => new CActiveDataProvider('Comment',
        array(
            'criteria' => array(
                'condition' => 'page_id=' . $model->id,
                'order' => 'date_entered DESC'
            ),
            'pagination' => array(
                'pageSize'=>20
            )
        )
    ),
    'itemView'=>'comment/_view',
    'template'=>'{items}{pager}'
)); ?>
<?php else: ?>
<h3>Be the first to comment on this page!</h3>
<?php endif; ?>
```

The `CListView` widget is used to display the comments, using the `protected/views/comment/_view` file as the item display. It’s basically the same as what Gii generated, although you’d remove the comment ID and page ID references ([Figure 22.7](#)).

The data for the `CListView` widget is a `CActiveDataProvider` instance, using the “Comment” model, with the condition that the comment’s `page_id` value matches the current page’s ID.

Thanks to the built-in Yii widget, displaying a paginated list of comments is a breeze.

## 2 Comments

---

**Name:** Jane Doe

**Email:** jane@example.net

**Comment:** What a wonderful thing.

**Date Entered:** Oct. 17, 2014

---

**Name:** testing

**Email:** blah@blah.com

**Comment:** Lorem ipsum dolor sit amet, c  
labore et dolore magna aliquyam erat, sed

**Figure 22.7:** The display of comments.

## Converting to Twitter Bootstrap

For this application, I choose to use Twitter Bootstrap for the layout and formatting. I'm pretty comfortable with Twitter Bootstrap, but getting everything "just so" took some effort. I started by downloading the [Twitter Bootstrap blog example](#), which is the design I was hoping to replicate. And, of course, you have to install the Bootstrap framework, its CSS, and its JavaScript files in the right places within your application for even that template to work.

Next, throw the HTML from the Twitter Bootstrap example into the `protected/views/layouts/main.php` file. All the references to external resources get prefaced with `<?php echo Yii::app()->request->baseUrl; ?>`.

The default Yii skeleton has two available layouts: a single column and a double column (i.e., one with a sidebar). I decided to keep this format for the blog, having both a full-page and a with-sidebar view (see Figure 22.1 and **Figure 22.8**).

To manage this design approach, the `protected/views/layouts/column1.php` and `protected/views/layouts/column2.php` files need to be updated accordingly. The single column layout to be the default, so all the controllers (except for "Site") get set to use it:

```
public $layout='//layouts/column1';
```

[Home](#) [Create Page](#) [List Pages](#) [List Comments](#) [My Info](#) [Logout \(larry\)](#)

# My Web Application

This is the site description.

# This is the First Test

By larry | 2014-09-30

*Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.* At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo

**Figure 22.8:** *The one-column layout.*

For the home page, preview versions of the blog posts are shown (see Figure 22.1). To accomplish that, create a `getSnippet()` method within the `Page` class definition. This method should return the first couple of sentences or so:

```
public function getSnippet() {
    $sentencesToDisplay = 3;
    $nakedBody = preg_replace('/\s+/', ' ', strip_tags($this->content));
    $sentences = preg_split('/(\\.|\\?|\\!)(\\s)/', $nakedBody);
    $stopAt = 0;
    foreach ($sentences as $i => $sentence) {
        $stopAt += strlen($sentence);
        if ($i >= $sentencesToDisplay - 1)
            break;
    }
    $stopAt += ($sentencesToDisplay * 2);
    return trim(substr($nakedBody, 0, $stopAt));
}
```

I took that code from a [Stack Overflow](#) post.

The home page uses a `CLListView` widget to display the most recent 3 live posts:

```
<?php $this->widget('zii.widgets.CListView', array(
    'dataProvider' => new CActiveDataProvider('Page'),
    array(
        'criteria' => array(
            'condition' => 'live=1',
            'order' => 'date published DESC'
        )
    )
))
```

```
        ),
        'pagination' => array(
            'pageSize'=>3
        )
    ),
    'itemView'=>'/page/_view',
    'template'=>'{items}{pager}'
)); ?>
```

The “itemView” is `protected/views/page/_view.php`, which looks like so:

```
<div class="blog-post">
    <h3 class="blog-post-title">
        <?php echo CHtml::link(CHtml::encode($data->title),
array('/page/view', 'id'=>$data->id,
'title'=>$data->title)); ?></h3>
    <p class="blog-post-meta">By
        <?php echo $data->user->username; ?> |
        <?php echo CHtml::encode($data->formattedDate()); ?></p>
    <p><?php echo $data->getSnippet(); ?></p>
</div><!-- /.blog-post -->
```

With this in place, mostly it’s a matter of tweaking the formatting on forms to use Twitter Bootstrap. Here’s a snippet from the comment form, with the proper Twitter Bootstrap classes in use:

```
<div class="form-group">
    <?php echo $form->labelEx($model, 'username'); ?>
    <?php echo $form->textField($model, 'username',
array('maxlength'=>45,'class'=>'form-control')); ?>
    <?php echo $form->error($model, 'username'); ?>
</div>
```

## Adding WYSIWYG

The blog needs a WYSIWYG editor, of course, so the next step is to install it. In Chapter 13, “[Using Extensions](#),” I used CKEditor, via the `editMe` extension. In this example, I’ve opted to use [Imperavi’s Redactor](#) tool, which gets a lot of love and attention these days. Although Redactor is a commercial product, Yii has purchased an OEM license for it, and it can be installed through the [Imperavi Redactor widget](#).

The trick with getting this widget to work is to manipulate when and how the JavaScript files are loaded when coupled with the Twitter Bootstrap includes. I’ll explain those specific steps.

{TIP} Conflicts with the loading of the JavaScript files is a frequent cause of a widget not working.

First, you have to download the extension and toss it in the **ext/imperavi-redactor-widget** directory.

Next the page's **\_form.php** file should be updated to use the widget for the content text area:

```
<div class="form-group">
    <?php echo $form->labelEx($model, 'content'); ?>
    <?php echo $form->textArea($model, 'content',
        array('rows'=>6, 'class'=>'form-control')); ?>
    <?php echo $form->error($model, 'content'); ?>
    <?php
Yii::import('ext.imperavi-redactor-widget.ImperaviRedactorWidget');
$this->widget('ImperaviRedactorWidget', array(
    'selector' => '#Page_content',
    // Some options, see http://imperavi.com/redactor/docs/
    'options' => array(
        ),
));
?>
</div>
```

There are many ways of applying the extension. I've chosen to leave the textarea alone, and apply Redactor to it separately. With this approach, I simply need to tell Redactor the ID value of the element to Redactor-ify. Obviously you'd want to configure some options there; see the Redactor docs for those possibilities.

This, unfortunately, won't get Redactor to work because of issues with the various jQuery libraries now used by the site thanks to Bootstrap. To resolve that, modify the template and change how the application includes jQuery.

First, configure the "clientScript" component to identify jQuery:

```
# protected/config/main.php
'clientScript' => array(
    'scriptMap' => array(
        'jquery.js'=>
            'https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js',
    ),
),
```

Next, this line (from the Twitter Bootstrap template) gets removed from **protected/views/layouts/main.php**:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/  
        jquery.min.js"></script>
```

That was in the footer and loads the minified version of jQuery. This is theoretically great, except that Redactor loads an unminified version of jQuery, and the two versions collide.

Next, still within the layout file, the jQuery library is registered and told to be loaded in the page head:

```
<?php Yii::app()->clientScript->registerCoreScript('jquery',  
    CClientScript::POS_HEAD); ?>
```

And now the WYSIWYG editor works (**Figure 22.9**)!



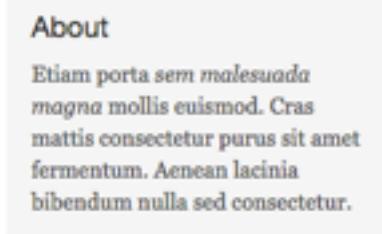
**Figure 22.9:** The WYSIWYG editor for pages.

# Creating a Posts by Month Widget

The Twitter Bootstrap template, and many blogs in general, has a sidebar for navigating posts by month (**Figure 22.10**). To do this in Yii, a custom widget makes the most sense. The widget will list the months that have posts, making each month a clickable link, and the resulting page will be like the home page—a preview of each post, but only show posts for the provided month. For more on creating widgets, see Chapter 19, “[Extending Yii](#)”.

Working backwards, first create a new view file, which will list the posts for a given month:

```
# protected/views/page/archives.php
<?php
/* @var $this PageController */
/* @var $dataProvider CActiveDataProvider */
/* @var $month Integer */
/* @var $year Integer */
$months = array('January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December');
```



## Archives

[October 2014](#)  
[September 2014](#)

**Figure 22.10:** The posts by month widget.

```
?>
<h1>Posts from <?php echo $months[$month] . ' ' . $year; ?></h1>
<?php $this->widget('zii.widgets.CListView', array(
    'dataProvider'=>$dataProvider,
    'itemView'=>'_view',
)); ?>
```

The view file expects to receive the blogs to list as a data provider object. Then it creates a header, showing the month and year. The blog posts themselves are rendered using a `CListView` widget, as on the home page.

Continuing backwards, this view file is rendered by the “archives” action of the “Page” controller:

```
public function actionArchives($year, $month) {

    // Dynamically generate criteria:
    $criteria=new CDbCriteria();
    $criteria->addCondition('live=1');

    // Make sure the year is greater than the first year of the blog:
    if(filter_var($year, FILTER_VALIDATE_INT,
        array('min_range'=>2014))) {
        $y = new CDbExpression('YEAR(date_published)');
        $criteria->addCondition("$y=$year");
    }

    // Validate the month:
    if(filter_var($month, FILTER_VALIDATE_INT,
        array('min_range'=>1, 'max_range'=>12))) {
```

```
$m = new CDbExpression('MONTH(date_published)');
$criteria->addCondition("$m=$month");
}

// Get the data:
$dataProvider=new CActiveDataProvider('Page',
    array(
        'criteria' => $criteria
    )
);

// Render the page:
$this->render('archives',array(
    'dataProvider'=>$dataProvider,
    'month' => $month,
    'year' => $year
));
}
```

The first criteria says to only fetch live posts. Then, the date and the year are added as criteria, provided they meet minimum requirements. The criteria are passed to `CActiveDataProvider`, along with an indicator of the class, to create the data provider object. The data provider object is passed to the view, along with the month and year.

To get this page to work, the “Page” controller needs open permissions on the “archives” action:

```
public function accessRules() {
    return array(
        array('allow',
            'actions'=>array('index','view','archives'),
            'users'=>array('*'),
        ),
    // Et cetera
}
```

And the URL routing needs to identify “archives” as a valid route:

```
'urlManager'=>array(
    'urlFormat'=>'path',
    'rules'=>array(
        '<controller:\w+>/<id:\d+>/<title:.*?>'=>'<controller>/view',
        '<controller:\w+>/<action:\w+>/<id:\d+>'=>'<controller>/<action>',
        '<controller:\w+>/<action:\w+>'=>'<controller>/<action>',
        'page/archives/<year:\d+>/<month:\d+>'=>'page/index',
)
```

```
 ),  
)
```

With all this in place, the last steps are to render and create the widget. The widget is rendered within the “column2” layout file:

```
<?php $this->widget('MonthsPosted'); ?>
```

That code tells Yii to render the “MonthsPosted” widget. The widget itself is defined in **protected/components/MonthsPosted.php**. The widget should extend **CPortlet**, and it needs a **renderContent()** method (again, see Chapter 19 for more detailed explanation). The **renderContent()** method needs to identify the months that have posts, and pass that list to a view file that creates the list of links. Here’s **MonthsPosted.php**:

```
<?php  
Yii::import('zii.widgets.CPortlet');  
class MonthsPosted extends CPortlet {  
    protected function renderContent() {  
        $q = "SELECT DISTINCT(CONCAT(MONTHNAME(date_published), ' ',  
            YEAR(date_published))) AS l, MONTH(date_published) AS m,  
            YEAR(date_published) AS y FROM page  
        WHERE live=1 ORDER BY y DESC, m DESC";  
        $cmd = Yii::app()->db->createCommand($q);  
        $result = $cmd->query();  
        $this->render('monthsPosted', array('data' => $result));  
    }  
}
```

For optimum efficiency, the widget uses Database Access Object (DAO) to run a static SQL command. For each month that has posts, the command selects:

- All the distinct month name plus year combinations (e.g., “January 2015”, “December 2014”, etc.)
- The month number
- The year

This array of data is based to the “monthsPosted” view:

```
/* protected/components/views/monthsPosted.php */  
<div class="sidebar-module">  
<h4>Archives</h4>  
<ol class="list-unstyled">
```

```
<?php foreach ($data as $item) {  
    echo '<li>' . CHtml::link($item['l'],  
        array('/page/archives',  
            'year'=>$item['y'],  
            'month'=>$item['m'])) . '</li>';  
}  
?>  
</ol>  
</div>
```

And that takes care of the widget!

## Applying ElasticSearch

The final big thing to do in this example is implement a search engine, as all blogs do have. For that, let's return to ElasticSearch, explained in detail in Chapter 20, “Working With Third-Party Libraries”. In fact, the steps taken to implement ElasticSearch within this blog are almost exactly those explained in that chapter.

First, ElasticSearch itself has to be installed on the server, and started, using the instructions from Chapter 20. Then, the ElasticSearch PHP library has to be pulled into the Yii application. That's done by creating **protected/composer.json**:

```
{  
    "require": {  
        "elasticsearch/elasticsearch": "~1.0"  
    }  
}
```

And then running `composer install` from the command-line within the **protected** directory.

To make the library available to the framework, the **index.php** bootstrap file has to be updated, specifically changing this line:

```
Yii::createWebApplication($config)->run();
```

To:

```
$app = Yii::createWebApplication($config);  
$composer = dirname(__FILE__).'/protected/vendor/autoload.php';  
require_once($composer);  
$app->run();
```

Next, create a “Search” controller, or just copy the one over from Chapter 20. It needs to create the ElasticSearch file (the index of content), index the site’s content, and handle search results. Here’s how the controller begins:

```
<?php
class SearchController extends Controller {
    public $defaultAction = 'search';
    private $_index = 'pages';
    private $_type = 'page';
```

For convenience in this version, two private variables are declared: identifying the search index name and index type.

The access rules are open for the “search” action, but restricted to administrators for the others:

```
public function filters() {
    return array(
        'accessControl',
    );
}
public function accessRules() {
    return array(
        array('allow',
            'actions'=>array('search'),
            'users'=>array('*'),
        ),
        array('allow',
            'actions'=>array('create', 'index'),
            'users'=>array('@'),
            'expression'=>'isset(Yii::app()->user->type) &&
                (Yii::app()->user->type == "admin")'
        ),
        array('deny', // deny all users
            'users'=>array('*'),
        ),
    );
}
```

The “create” action comes from Chapter 20, creating the index. It only needs to be run once. (See the code on GitHub for details.)

The “index” action indexes the site content. It’s also similar to that in Chapter 20:

```
public function actionIndex() {
    $params = array();
```

```

$params['index'] = $this->_index;
$params['type'] = $this->_type;
$client = new Elasticsearch\Client();
$q = 'SELECT page.id, page.title, page.content, user.username
FROM page LEFT JOIN user ON page.user_id = user.id';
$cmd = Yii::app()->db->createCommand($q);
$result = $cmd->query();
foreach ($result as $row) {
    $params['body'] = array(
        'author'=>$row['username'],
        'title'=>$row['title'],
        'content'=> $row['content']
    );
    $params['id'] = $row['id'];
    $client->index($params);
}
$this->render('index');
} // End of actionIndex() method.

```

The content is indexed, along with the post's author and title. Everything is indexed under the page ID.

After executing the “create” and “index” actions once each (as an administrator), it's now usable as a search engine via the “search” action. That method looks for the search terms in the URL and executes a search query. It begins like so:

```

public function actionSearch() {
    if (isset($_GET['terms'])) {
        $terms = $_GET['terms'];
    } else {
        $this->render('search', array('terms' => null));
        Yii::app()->end();
    }
}

```

First, the action checks if search terms were provided, which would come in through the URL. If not, the search view file can immediately be shown and the action terminated.

Next, the query is configured using JSON, per the instructions in Chapter 20:

```

$params = array();
$params['index'] = $this->_index;
$params['type'] = $this->_type;
$params['body'] = '{
    "query": {

```

```
"multi_match": {
    "query": "' . $terms . '",
    "fields": [
        "title^1",
        "author^1",
        "content"
    ],
    "minimum_should_match": "75%"
},
},
"highlight": {
    "fields": {
        "content": {
            "fragment_size": 300
        },
        "title": {
            "number_of_fragments": 0
        },
        "author": {
            "number_of_fragments": 0
        }
    }
}
}';
```

Again, see Chapter 20 for an explanation of this.

Next, the search query is executed:

```
// Create the client:
$client = new Elasticsearch\Client();
// Perform the search
$results = $client->search($params);
// Get the total:
$total = $results['hits']['total'];
```

And the hits are fetched into an array:

```
$hits = array();
foreach ($results['hits']['hits'] as $hit) {
    $id = $hit['_id'];
    $hits[$id]['title'] = $hit['_source']['title'];
    foreach ($hit['highlight'] as $field => $h) {
        $hits[$id]['highlight'][$field] = $h[0];
```

```
    }
}
```

All of this is explained in Chapter 20.

Finally, the results are passed to the view file and the method is completed:

```
    $this->render('search', array('total' => $total,
        'hits' => $hits, 'terms' => $terms));
    } // End of the actionSearch() method.
}
```

The view file starts by showing the search form:

```
<?php
/*
@var $this SearchController
@var $total Integer number of hits
@var $terms String search terms
@var $hits Array hits
*/
$this->pageTitle=Yii::app()->name;
$this->renderPartial('_form', array('terms' => $terms));
?>
```

The `_form.php` file is:

```
<form class="form-inline" role="form" method="get">
    <div class="form-group">
        <input type="text" class="form-control" id="terms" name="terms"
            placeholder="Search terms"?php if (!empty($terms))
            echo ' value="' . CHtml::encode($terms) . '"';
        ?>
    </div>
    <button type="submit" class="btn btn-default">Search</button>
</form>
<hr>
```

Returning back to the “search” view file, if it receives search terms, it starts by displaying those and the number of results:

```
<?php if (!empty($terms)): ?>
<h1>Search Results</h1>
<?php echo '<h2>' . $total . ' Record(s) Found Searching for "' .
    $terms . '"</h2>';
```

Next, the page iterates through the hits, showing them:

```
foreach ($hits as $id => $hit) {
    echo '<div><h3>' . CHtml::link($hit['title'], array('page/view',
        'id'=>$id, 'title'=>$hit['title'])) . '</h3>';
    foreach ($hit['highlight'] as $field => $h) {
        echo '<p><strong>' . ucfirst($field) . '</strong>: '
            . $h . '</p>';
    }
    echo '</div>';
}
```

And, finally, the script is completed:

```
?>
<?php endif; ?>
```

And that takes care of the search functionality (**Figure 22.11**)!

alice

Search

## Search Results

1 Record(s) Found Searching for "alice"

[Alice in Wonderland](#)

**Content:** of half an hour or so there were no arches left, and all the players, except the King, the Queen, and Alice, were in custody and under sentence of execution.

"HERE!" cried Alice, quite forgetting in the flurry of the moment how large she had grown in the last few minutes, and she jumped up in such

**Title:** Alice in Wonderland

**Figure 22.11:** Search results and the search form.

## Tidying Up

The steps to this point focus on the big, core issues, but there are lots of little things to change. Let's quickly look at a few.

## Checking Permissions

When developing an application like this, you'll find that the generated access rules often conflict with what you're trying to do. You'll create new actions that, by default, are denied, and you'll leave open access to actions that won't actually be used. In the course of developing and testing, you'll quickly catch the former problems, but the latter won't be obvious. For that reason, you'll want to go through your access rules and confirm that you've adjusted all of them accordingly. If an action shouldn't be executable at all—for example, the site shouldn't allow for the deletion of users, then remove the action method itself.

In this particular example, there's one more permission to be tweaked, this one being more contextual. Any editor or administrator should be able to edit any page, but the author of the page should be the only “author” type that can edit it.

Restricting updates to logged in users is easily done in the access rules:

```
array(
    'allow',
    'actions'=>array('create', 'update', 'admin'),
    'users'=>array('@'),
),
```

But to enforce the restriction that only the page's author can edit the page requires logic within the “update” action itself:

```
public function actionUpdate($id) {
    $model=$this->loadModel($id);
    if ((Yii::app()->user->type == 'author') &&
        (Yii::app()->user->id != $model->user_id)) {
        throw new CHttpException(403, 'You do not have
            permission to edit this page.');
}
```

If the user type is “author”, but the current user's ID doesn't match the page's author ID, an exception is thrown.

## Displaying Dates Nicely

When displaying comments and pages (i.e., posts), it's preferable to show the respective dates in a more legible format than the default (the MySQL output). To accomplish this, create a `formattedDate()` method in both the `Comment` and `Page` classes:

```
public function formattedDate() {
    return DateTime::createFromFormat('Y-m-d H:i:s',
        $this->date_entered)->format('M. d, Y');
    parent::afterFind();
}
```

I'm pretty sure I stole this code from a [Stack Overflow answer](#).

Now the view files can call `$model->formattedDate()` when the formatted date is preferred:

```
# protected/views/comment/_view.php
<b><?php echo CHtml::encode($data->getAttributeLabel('date_entered'));
?>:</b>
<?php echo CHtml::encode($data->formattedDate()); ?>
<br />
```

## Better Admin Pages

The default `CGridView` used on the admin pages is a pretty good way for administering records. You won't want to keep using it for all projects, I wouldn't think, but when you do, you will certainly want to make a view edits. For this example, I made slight changes to the comments, users, and pages admin `CGridView` widgets.

On the comments admin page, the page's title would be more useful than the ID (**Figure 22.12**):

```
array(
    'header'=>'Page',
    'name' => 'page_id',
    'value'=>'$data->page->title'
),
```

On the users admin page, the user type should be filterable by the allowed values (**Figure 22.13**):

```
array(
    'header'=>'Type',
    'value' => 'ucfirst($data->type)',
    'filter' => CHtml::dropDownList('User[type]', $model->type,
        array('author' => 'Author', 'editor' => 'Editor',
        'admin' => 'Admin'), array('empty' => '(Select)'))
),
```

Displaying 1-3 of 3 results.						
ID	Page	Name	Email	Comment	Date Entered	
1	This is the First Test	testing	blah@blah.com	<p> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit </p>	2014-10-17 10:31:40	

Figure 22.12: The admin page for comments.

Displaying 1-6 of 6 results.						
ID	Username	Email	Type	User Since		
			(Select)			
1	larry	larry@larryullman.com	Author	2014-10-10 20:02:07		
2	Editor	editor@example.net	Editor	2014-10-14 23:08:54		
3	Admin	admin@example.net	Admin	2014-10-14 23:09:17		

Figure 22.13: The users admin page.

This code was explained in detail in Chapter 12, “Working with Widgets”.

On the pages admin page, three changes are needed (**Figure 22.14**):

- The author’s name should be shown, not the author’s ID
- The live column should say “Live” and “Draft”, not 1/0
- A preview of the page should be shown for the content, not the entire page

Displaying 1-6 of 6 results.								
ID	Author	Live?	Title	Preview	Date Updated	Date Published		
1	larry	Live	This is the First Test	Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.	2014-10-14 23:05:55	2014-09-30		
		(S ↓)						

**Figure 22.14:** The pages admin page.

Here’s how those changes are made, using what was previously explained in Chapter 12:

```
'columns'=>array(
    'id',
    array(
        'header' => 'Author',
        'name' => 'user_id',
        'value' => '$data->user->username'
    ),
    array(
        'header' => 'Live?',
        'filter' => CHtml::dropDownList('Page[live]', $model->live,
            array('0' => 'Draft', '1' => 'Live'),
            array('empty' => '(Select)'), 
            'value' => '($data->live == 1) ? "Live" : "Draft"'
        ),
        'title',
        array(
            'header' => 'Preview',
            'value' => '$data->getSnippet()'
        ),
    )
),
```

## Nicer URLs

To improve the URLs slightly, the blog title is being added to them. Doing so requires two steps:

- Adding the title to the link on the home page
- Telling Yii to handle the routes properly

To add the title to the link, the `protected/views/page/_view.php` file is updated to add that as a third parameter when making the link:

```
<h3 class="blog-post-title"><?php echo CHtml::link(
    CHtml::encode($data->title),
    array('/page/view', 'id'=>$data->id,
        'title'=>$data->title)); ?></h3>
```

With the title added to the URL, the routing needs to be updated to recognize it:

```
'urlManager'=>array(
    'urlFormat'=>'path',
    'rules'=>array(
        '<controller:\w+>/<id:\d+>/<title:.*?>'=>'<controller>/view',
        '<controller:\w+>/<action:\w+>/<id:\d+>'=>'<controller>/<action>',
        '<controller:\w+>/<action:\w+>'=>'<controller>/<action>',
        'page/archives/<year:\d+>/<month:\d+>'=>'page/index',
    ),
),
```

As you can see, the title can include any number of any character.

## Things to Possibly Add

Over the course of this chapter, I walked through the main functionality and issues for creating a fuller CMS example. Of course, there are numerous ways the application could be implemented differently, and even with all the code and changes made so far, the application is still not production-ready. Here, then, are some ways I thought of in which the application could be further improved:

- Create a page synopsis option to be used for SEO purposes and page introductions
- Support an approval mechanism for comments
- Support tags

- Remove the “delete” action for pages and users
- Restrict access to the user registration page so only known people (i.e., those to become authors or editors) can register
- Show recent comments in a sidebar
- Add a file upload mechanism for the HTML editor
- Ajax-ify more functionality
- Cache, cache, cache!
- Insist on blog titles in the URLs everywhere
- Add links to edit the content to the public view of a blog page for logged in users
- Complete the user functionality (e.g., recover password, change password, ability to change a user’s type)
- Update the ElasticSearch index automatically when a new page is created or an existing page is updated

And, if I had more time, I’d fix the styling in the footer pagination links and how errors are shown in the forms. Also, see the two example applications mentioned at the beginning of this chapter for more ideas.

## Chapter 23

# MAKING AN E-COMMERCE SITE

As with the previous chapter, the primary goal in this chapter is to present previously explained content within a more complete context. Here, I'll walk through the creation of an e-commerce site.

As before, the goal with the code and the chapter is to focus on the heart of the application, and how one might go about developing an e-commerce site. There are myriad ways this code and example could be improved, made more secure, made to perform better, and so forth. But it is a reasonable, working example, that should be educational.

You can find the complete code in a [GitHub repository](#). You can also use the commit history to roughly track the development of this application. The rest of the chapter will approximately map to the same development path as that commit history.

*{NOTE}* I changed much of the application's foundation while developing this project, so the commit history is especially messy.

## Project Goals

There are as many variations on an e-commerce site as there are snowflakes in the sky (okay, not really, but it is a diverse subject matter). To me, the primary mitigating factor when developing an e-commerce site is distinguishing between selling physical goods and selling digital goods. With the former, you have to worry about inventory, shipping and handling, accepting shipping addresses, and much, much more. Not to mention, an actual person has to handle the receiving of new inventory and shipping out orders.

Conversely, at most, a purveyor of digital goods only needs to worry about licensing or Digital Rights Management (DRM). For this reason, a virtual e-commerce site practically runs itself once developed.

Considering the constraints of time and space, in this chapter I'll create an e-commerce site that sells digital goods. Specifically, it'll sell books in PDF format.

The features I've chosen to implement include:

- List of books to purchase on the main page
- Detail book item view
- Shopping cart for visitors
- Purchases made through a separate module
- Use of a helper class
- PDF downloads

This site is going to use the default Yii layout. But you'll need to copy the **images** directory (in the webroot), from the GitHub repo, as that contains the book images. And the **protected/books** directory (on GitHub) has the PDFs of the books themselves (these aren't actual PDFs of my books, they're dummy files).

As with the previous chapter, there's a lot to this application, even though it's not 100% finished. At the end of the chapter, you'll see some of the notes and thoughts I had for how the example could be completed or expanded.

## Creating the Database

You can find the SQL statements required to create the database in the application's **protected/data/ch23.sql** file, but I'll quickly run through them here as well. The only table not referenced below is the one created by the "pay" module extension. I'll talk about that more when I get to that point in the development sequence.

The name of the database is "yiobook\_ch23".

The main product table is **book**:

```
CREATE TABLE IF NOT EXISTS `yiobook_ch23`.`book` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(255) NOT NULL,
  `price` INT UNSIGNED NOT NULL,
  `description` TEXT NULL,
  `author` VARCHAR(60) NOT NULL,
  `filename` VARCHAR(60) NOT NULL,
  `date_published` DATE NOT NULL,
  PRIMARY KEY (`id`)
)
ENGINE = InnoDB;
```

Everything should be pretty obvious there. Note that all prices in the database will be stored as integers. The **filename** field refers to the file name of the PDF that

the customer would actually purchase (these are found in the `protected/books` directory).

In a more fleshed-out example, I'd be inclined to create a separate `author` table, and link `author` to `book` through an intermediary table.

You'll want to populate the `book` table using these insert commands (use the SQL file, found in the code on GitHub, if you'd rather):

```
INSERT INTO `book` VALUES
(1,'The Yii Book',2000,'Lorem ipsum...', 'Larry Ullman',
 'yiibook.pdf', '2024-12-31'),
(2,'Effortless E-commerce with PHP and MySQL (2nd Edition)',4499,
 'Lorem ipsum...', 'Larry Ullman', 'effortless-ecom-2nd.pdf', '2013-11-30'),
(3,'PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide (4th
 Edition)',40,'Lorem ipsum...', 'Larry Ullman', 'php-mysql-4th.pdf', '2012-08-16'),
(4,'PHP for the Web: Visual QuickStart Guide (4th Edition)',2499,
 'Lorem ipsum...', 'Larry Ullman', 'php-4th.pdf', '2012-01-10'),
(5,'Modern JavaScript: Develop and Design',4499,'Lorem ipsum...',
 'Larry Ullman', 'modern-javascript.pdf', '2013-02-08'),
(6,'Advanced PHP and Object-Oriented Programming: Visual QuickPro
 Guide (3rd Edition)',4499,'Lorem ipsum...', 'Larry Ullman',
 'adv-php-3rd.pdf', '2012-03-19');
```

The `cart` table stores metadata about the shopping cart. To support the ability to have a shopping cart without being registered, a session ID, stored in a cookie, represents the customer.

```
CREATE TABLE IF NOT EXISTS `yiibook_ch23`.`cart` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `customer_session_id` CHAR(32) NOT NULL,
  `date_modified` TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `customer_session_id_UNIQUE` (`customer_session_id` ASC)
) ENGINE = InnoDB;
```

The `cart_content` table stores the particular items in a given shopping cart. That is the book ID and the quantity:

```
CREATE TABLE IF NOT EXISTS `yiibook_ch23`.`cart_content` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `cart_id` INT UNSIGNED NOT NULL,
  `book_id` INT UNSIGNED NOT NULL,
  `quantity` TINYINT UNSIGNED NOT NULL DEFAULT 1,
  PRIMARY KEY (`id`),
```

```
INDEX `fk_cart_idx` (`cart_id` ASC),
CONSTRAINT `fk_cart_content`
    FOREIGN KEY (`cart_id`)
    REFERENCES `yiibook_ch23`.`cart` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
INDEX `fk_cart_books_idx` (`book_id` ASC),
CONSTRAINT `fk_cart_books`
    FOREIGN KEY (`book_id`)
    REFERENCES `yiibook_ch23`.`book` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

The `customer` table represents people that have made a purchase. It stores an email address, the password, and whether or not the customer wants to receive email updates. That functionality isn't defined in the application yet, but could easily be added.

```
CREATE TABLE IF NOT EXISTS `yiibook_ch23`.`customer` (
`id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
`email` VARCHAR(80) NOT NULL,
`pass` VARCHAR(255) NULL,
`get_emails` TINYINT(1) UNSIGNED NOT NULL DEFAULT 0,
`date_entered` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
UNIQUE INDEX `email_UNIQUE` (`email` ASC))
ENGINE = InnoDB;
```

The `order` table stores the metadata about an individual order. This includes the:

- Customer ID
- Payment ID
- Order total
- Date

The specifics of an order (i.e., what was purchased) will be stored separately.

```
CREATE TABLE IF NOT EXISTS `yiibook_ch23`.`order` (
`id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
`customer_id` INT UNSIGNED NOT NULL,
`payment_id` payment_id INT UNSIGNED NOT NULL,
`total` INT UNSIGNED NOT NULL,
```

```
`date_entered` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
PRIMARY KEY (`id`),  
INDEX `fk_orders_users_idx` (`customer_id` ASC),  
INDEX `fk_orders_payments_idx` (`payment_id` ASC),  
CONSTRAINT `fk_orders_payments`  
    FOREIGN KEY (`payment_id`)  
        REFERENCES `yiibook_ch23`.`payment` (`id`)  
        ON DELETE RESTRICT  
        ON UPDATE RESTRICT)  
CONSTRAINT `fk_orders_users`  
    FOREIGN KEY (`customer_id`)  
        REFERENCES `yiibook_ch23`.`customer` (`id`)  
        ON DELETE RESTRICT  
        ON UPDATE RESTRICT)  
ENGINE = InnoDB;
```

The `order_content` table stores the particular items included in an order: the book, the quantity of that book, and the price paid per copy.

```
CREATE TABLE IF NOT EXISTS `yiibook_ch23`.`order_content` (  
`id` INT UNSIGNED NOT NULL AUTO_INCREMENT,  
`order_id` INT UNSIGNED NOT NULL,  
`book_id` INT UNSIGNED NOT NULL,  
`quantity` TINYINT UNSIGNED NOT NULL DEFAULT 1,  
`price_per` INT UNSIGNED NOT NULL,  
PRIMARY KEY (`id`),  
INDEX `fk_order_content_order1_idx` (`order_id` ASC),  
INDEX `fk_order_content_book1_idx` (`book_id` ASC),  
CONSTRAINT `fk_order_content_order1`  
    FOREIGN KEY (`order_id`)  
        REFERENCES `yiibook_ch23`.`order` (`id`)  
        ON DELETE NO ACTION  
        ON UPDATE NO ACTION,  
CONSTRAINT `fk_order_content_book1`  
    FOREIGN KEY (`book_id`)  
        REFERENCES `yiibook_ch23`.`book` (`id`)  
        ON DELETE NO ACTION  
        ON UPDATE NO ACTION)  
ENGINE = InnoDB;
```

The SQL file in the application code (found on GitHub) includes two other tables, not used in this chapter. I'll explain what those are for at the end of the material.

## Getting Started

(These next instructions are almost verbatim from that in the previous chapter, because, well, this is what you'd do next.)

With the database created, it's time to get started creating the application. Logically, this begins by using the `yiic` script to create a new Web application:

```
yiic web app ../path/to/yiobook-ecom-ch23
```

For my own personal development, at this point I also:

- Create a new virtual host so that I can access the site using something like `http://ch23`
- Initialize Git
- Open up the permissions on the **models**, **views**, and **controllers** directories (required on my Mac when using Gii in steps to follow)
- Register the new site with [CodeKit](#), which will refresh the browser when I make changes to site files, among many other features (it's Mac-only, but I adore it)

Next, edit the **main.php** configuration file. The most important steps at first are to:

- Enable Gii
- Configure the database
- Enable Web log routing
- Create the models using Gii
- Create CRUD functionality using Gii

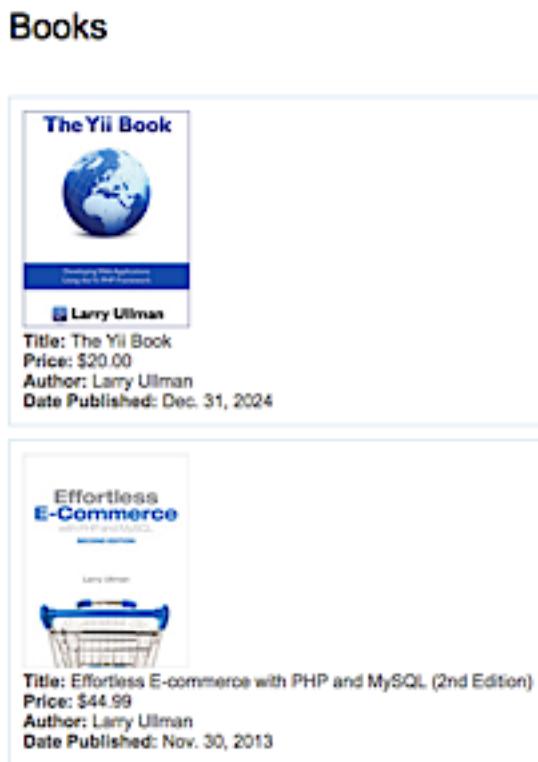
By this point, the above should be obvious and easy to do, so I won't go into more detail on those steps.

## Editing the Models

Having implemented all the models and CRUD functionality, it's editing time! Surprisingly, in this case, I didn't make any major changes to the models at this point. Later in the development, new methods will be added. But based upon the database table definitions, and particularly the foreign key constraints that turn into model relations, no major model edits are needed yet.

## Creating the Home Page

The home page for the site is going to show a list of books that are available for sale (**Figure 23.1**).



**Figure 23.1:** Books listed on the home page.

This is done through the “book” controller, so the site should be configured to use it for the default:

```
# protected/config/main.php
'defaultController' => 'book',
```

With that in place, the “index” action of the “book” controller becomes the default (because “index” is the default action of any controller). That method creates a data provider of Book objects and passes it to the view file:

```
public function actionIndex() {
    $dataProvider=new CActiveDataProvider('Book');
    $this->render('index',array(
        'dataProvider'=>$dataProvider,
    ));
}
```

The “index” view file uses the `CListView` widget to render its content:

```
<?php $this->widget('zii.widgets.CListView', array(
    'dataProvider'=>$dataProvider,
    'itemView'=>'_view',
)); ?>
```

With this in mind, the home page is configured by editing `protected/views/-book/_view.php`:

```
<?php
/* @var $this BookController */
/* @var $data Book */
?>


<?php echo CHtml::link('', array('view', 'id'=>$data->id)); ?>
    <br />
    <b><?php echo CHtml::encode($data->getAttributeLabel('title'));</b>
    <?php echo CHtml::encode($data->title); ?>
    <br />
    <b><?php echo CHtml::encode($data->getAttributeLabel('price'));</b>
    <?php echo Utilities::formatAmount($data->price); ?>
    <br />
    <b><?php echo CHtml::encode($data->getAttributeLabel('author'));</b>
    <?php echo CHtml::encode($data->author); ?>
    <br />
    <b><?php echo CHtml::encode($data->getAttributeLabel('date_published'));</b>
    <?php echo Utilities::formatDate($data->date_published, 'Y-m-d'); ?>
    <br />
</div>


```

Mostly this is the default template, with a couple of changes. First, the book’s image is shown, and hyperlinked to the book’s individual view page. For ease of implementation, each book’s image is the book’s ID plus the “.jpg” extension, found in the `images` directory.

Two pieces of information are run through utilities: the price and the date published. Because the prices are in integers in the dashboard, they’ll need to be formatted to cents in multiple places on the site. Many dates—a book’s published date, an order

date, etc.—will need to be formatted, too. To avoid having to replicate code in multiple places, I created a `Utilities` class in this example. I'll get to it shortly. But, here, two methods in that class are called to format the price and date, accordingly.

By clicking on the image, the user is taken to the book's details page. Obviously you could do much more with the aesthetics and the user interface here, but the functionality is sufficient for the purposes of this example. (And thanks to the use of the widget, making bold changes to the display is easy.)

## Viewing a Book

Individual books are shown on the book's "view" page, which is also where a book can be added to the shopping cart. I removed the default use of `CDetailView` from that page, and instead mostly replicated what was shown on the home page, with the additional description of the book and a link to add the book to the cart (**Figure 23.2**).

### The Yii Book

The screenshot shows a book detail page for "The Yii Book" by Larry Ullman. The page includes the book cover, title, author, price, a detailed description, and the date published. At the bottom, there is a blue "Add to Cart" button.

```
<div class="view">
    <?php echo ''; ?>
    <br />
    <b><?php echo CHtml::encode($model->getAttributeLabel('title'));</b><?php echo CHtml::encode($model->title); ?>
    <br />
    <b><?php echo CHtml::encode($model->getAttributeLabel('price'));</b><?php echo Utilities::formatAmount($model->price); ?>
```

```
<br />
<b><?php echo CHtml::encode($model->getAttributeLabel('description')) ; ?></b><?php echo CHtml::encode($model->description); ?>
<br />
<b><?php echo CHtml::encode($model->getAttributeLabel('author')) ; ?></b><?php echo CHtml::encode($model->author); ?>
<br />
<b><?php echo CHtml::encode($model->getAttributeLabel('date_published')) ; ?></b><?php echo Utilities::formatDate($model->date_published, 'Y-m-d'); ?>
<br />
</div>
<?php echo CHtml::link('Add to Cart', array ('/cart/add', 'id' => $model->id)); ?>
```

As you can see, the link at the bottom goes to the “cart” controller’s “add” action, passing along the book ID. And, as with the home page, this could be made to look a lot nicer (especially if you, unlike me, actually have design skills), which would be easy to do.

## Writing a Utilities Class

As already stated, some functionality will be needed time and again in the application:

- Conversion of amounts from integers to decimals (i.e., from cents to dollars and cents)
- Proper formatting of dates
- Cart retrieval

Rather than replicating the same code in multiple models, it makes sense to create a helper class instead. This will just be a PHP class that does not extend any Yii classes. It will have only static methods:

```
<?php
class Utilities {
    public static function formatAmount($amount) {
        return '$' . number_format($amount/100, 2);
    }
    public static function formatDate($date, $format = 'Y-m-d H:i:s') {
        return DateTime::createFromFormat($format,
            $date)->format('M. j, Y');
    }
}
```

The `formatAmount()` method takes an amount and returns it divided by 100, rounded to 2 decimals, and prefaced with a dollar sign. By creating this method, it wouldn't be too hard to support multiple currencies, passing a currency flag as the second argument, and changing the formatting and currency symbol accordingly.

The `formatDate()` method takes a date and a format as its two arguments. The format is the incoming date's format, which can vary based upon the source. The method then returns a formatted date. Again, because this has been defined in a helper class, it'd be easy to make locale-aware, if you so chose.

The `Utilities` class file—`Utilities.php`—needs to be in the `protected/components` folder to make the application aware of its existence.

The third method in this class—`getCart()`—will be explained separately next, as it's the most complicated helper method.

## Creating Carts

Successful e-commerce, to me, is about providing a product the customer wants to purchase and then getting out of the way of her buying it. From a user experience and development perspective, anything you can do to facilitate completing a purchase is a good thing; any obstacle you can remove is also a benefit. Towards that end, I like to create support for shopping carts before registration and login.

To do that, a cart session identifier is stored in a cookie and in the database. That's the heart of the functionality. Then, when a user accesses the site, some code must look for the presence of the cookie. If one exists, the current cart session is used and extended. If no cookie exists, a new cart session is created. Because many of the site's controllers will need access to the cart, the ability to get a reference to the cart is being put into the `Utilities` class. The comments inline explain what the code is doing:

```
# protected/components/Utilities.php
public static function getCart() {

    // Get or create the cart session ID:
    if (isset(Yii::app()->request->cookies['SESSION'])) {
        $sess = Yii::app()->request->cookies['SESSION'];
    } else {
        $sess = bin2hex(openssl_random_pseudo_bytes(16));
    }

    // Send the cookie, with an expiration of 30 days:
    Yii::app()->request->cookies['SESSION'] = new CHtmlCookie('SESSION',
        $sess, array('expire' => time()+(60*60*24*30)));
}
```

```
// Load the Cart model:  
$cart=Cart::model()->find('customer_session_id=:sess',  
array(':sess' => $sess));  
if($cart==null) {  
    $cart = new Cart();  
    $cart->customer_session_id = $sess;  
    $cart->save();  
}  
  
// Return the Cart instance:  
return $cart;  
}
```

This code can be used like so:

```
$cart = Utilities::getCart();
```

Because the method only relies upon the cookie, it can be called without being passed any parameters. Either an existing cart or a new, empty, cart is returned (as a `Cart` object in either case).

With the ability to create or recreate a cart in place, the next step is to implement “add to cart” functionality. The “cart” controller’s “add” action accepts a product ID as a parameter (specifically, the ID of the product being added to the cart). The method first gets a reference to the cart:

```
public function actionAdd($id) {  
    $cart = Utilities::getCart();
```

Next, the method needs to see if the item is already in the shopping cart:

```
$item=CartContent::model()->find('cart_id=:cart AND book_id=:book',  
array(':cart' => $cart->id, ':book' => $id));
```

The item is already in the cart if the book ID matches the passed ID and the cart ID matches the user’s cart ID.

Next, if the item is in the cart, the quantity is updated. Thanks to this code, if the customer adds an item to the cart once, and then adds it again, the customer will have 2 of that item in the cart.

If the item is not in the cart, it needs to be added. This is done through the `CartContent` model instance:

```
if($item!==null) {
    $item->quantity = $item->quantity + 1;
} else {
    $item = new CartContent();
    $item->cart_id = $cart->id;
    $item->book_id = $id;
    $item->quantity = 1;
}
```

Finally, the new item needs to be saved, and the “view cart” view file rendered:

```
$item->save();
$this->render('view',array(
    'model'=>$cart
));
}
```

The “view cart” view file is intended to be the only cart view file used (i.e., there’s no “update”, “create”, or “index” view file). This view file will be shown when a new product has been added to the cart or when the customer clicks a “view cart” link. It’d also be used upon updating the cart.

The default “view” file uses a `CGridView` widget, so I’ve updated that to display a cart relatively nicely. Admittedly, this will be a pretty heavy customization of the widget, so I’ll explain those steps in detail.

But first, a method needs to be defined on the `Cart` class that will return the cart contents. Specifically, this will be a `CActiveDataProvider` instance of `CartContent` objects for the given cart:

```
# protected/models/Cart.php
public function getContents() {
    $criteria=new CDbCriteria;
    $criteria->compare('cart_id',$this->id, true);
    return new CActiveDataProvider('CartContent', array(
        'criteria'=>$criteria,
    ));
}
```

Again, the method returns a `CActiveDataProvider`. The criterion is the `CartContent` items whose `cart_id` property matches the current cart’s ID.

It will also help if the `Cart` class has a method for returning the order total. This can efficiently be calculated using a direct SQL command:

```
public function getTotal() {
    $id = $this->id;
    $cmd = Yii::app()->db->createCommand('SELECT
        SUM(quantity * book.price) FROM cart_content
        JOIN book ON book_id=book.id
        WHERE cart_id=:id');
    $cmd->bindParam(':id', $id, PDO::PARAM_INT);
    return $cmd->queryScalar();
}
```

Both of these new `Cart` class methods will be used by the view file. Again, the view file uses a `CGridView` widget to display the cart contents:

```
# protected/views/cart/view.php
<?php $this->widget('zii.widgets.grid.CGridView', array(
    'id'=>'cart-grid',
    'dataProvider'=>$model->getContents(),
    'columns'=>array(
        'book.title',
        array(
            'name' => 'Price',
            'value' => 'Utilities::formatAmount($data->book->price)'
        ),
        'quantity',
    )
),
```

The `dataProvider` for the widget is `Cart::getContents()`, the method just explained.

For the columns, the widget first displays the book's title, price, and the quantity of that item within the cart. To properly format the book's price, it's run through the `formatAmount()` utility method (**Figure 23.3**).

Title	Price	Quantity
The Yii Book	\$20.00	1
Effortless E-commerce with PHP and MySQL (2nd Edition)	\$44.99	1
PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide (4th Edition)	\$0.40	1

In the last column, the widget is going to display buttons. By default, the buttons are to view, edit, and delete the item. For the time being, just the view and delete options will be implemented. In the default widget, both buttons would link to the “Cart” controller. However, the “view” should be linked to “Book” and the “delete” needs to go to a custom action for removing items from the cart:

```
array(
    'class'=>'CButtonColumn',
```

```
'header'=>'Action',
'template'=>'{view}{delete}',
'buttons'=>array(
    'view'=>array(
        'label' => 'View',
        'url' => 'Yii::app()->createUrl("book/view",
            array("id"=>$data->book->id))'
    ),
    'delete'=>array(
        'label' => 'Delete',
        'url' => 'Yii::app()->createUrl("cart/deleteItem",
            array("id"=>$data->id))'
    ),
),
),
```

As you can see, “view” is linked to “book/view”, passing along the item’s book ID. The “delete” item is linked to “cart/deleteItem”, passing along the item’s ID.  
(The widget is then completed with ),)); ?>.)

Here is the “deleteItem” definition, from the “Cart” controller:

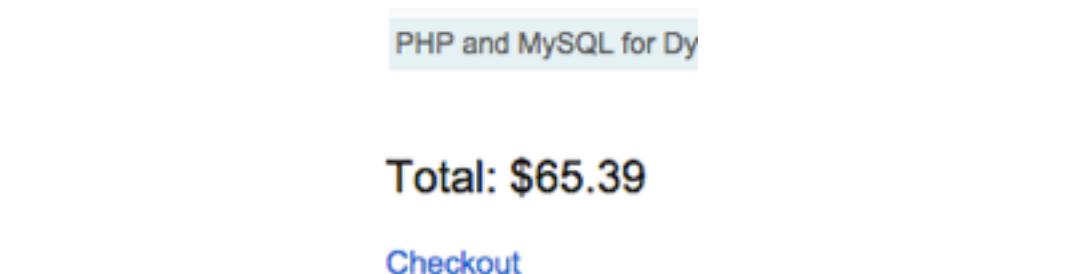
```
public function actionDeleteItem($id) {
    $model=CartContent::model()->findPk($id);
    $model->delete();
    if(!isset($_GET['ajax']))
        $this->redirect(isset($_POST['returnUrl'])
            ? $_POST['returnUrl'] : array('admin'));
}
```

This is a mild variation on the `actionDelete` method defined in the class, with the difference being that an instance of `CartContent` is being deleted, not a `Cart` instance.

With all this in place, users can now add items to their cart, view the cart, remove items from the cart, and quickly go to view an item in their cart. There’s still room for improvement, of course: the ability to update quantities within the cart would be necessary, and a quick one-click “clear cart” option would make sense.

Finally, the view cart page shows the order total and creates a link to checkout (**Figure 23.4**):

```
<h3>Total: <?php echo Utilities::formatAmount($model->getTotal()); ?></h3>
<?php echo CHtml::link('Checkout', array ('/pay')); ?>
```



PHP and MySQL for Dynamic Web Applications

Total: \$65.39

[Checkout](#)

**Figure 23.2:** The end of the cart display.

## Adding the Payment Module

To complete the checkout process, the customer has to pay. To do that, I've integrated the Stripe payment module developed and explained in Chapter 19, “[Extending Yii](#)”. In the process of doing so, I've slightly modified the original extension I created. True, one shouldn't have to modify extensions, but the changes were really to fill in some gaps from its original incarnation, and this modified version is what I'd turn into a real extension in any case.

{NOTE} For detailed explanation on the extension itself, see Chapter 19. For ethical reasons, I should also point out that I work for Stripe. Also, while Stripe is currently only available for live usage in about 20 countries, you can use a non-active test account in any country, free of charge.

The extension is first dropped into **protected/modules**, under the name “pay”. (You should get the extension from this chapter's [GitHub repository](#), if you haven't already.)

Next, there's a migration to be run that creates the **payment** database table:

```
$this->createTable('payment', array
    'id' => 'pk',
    'charge_id' => 'string NOT NULL',
    'email' => 'string NOT NULL',
    'amount' => 'integer UNSIGNED NOT NULL',
    'date_added' => 'timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP'
);
$this->createIndex('charge_id', 'payment', 'charge_id', true);
$this->createIndex('email', 'payment', 'email');
```

This table is particular to the extension, storing the pertinent details about the Stripe charge.

For the migration to work, you have to edit the `console.php` file to set the database, and then run this command from within the `protected` directory:

```
yiic migrate --migrationPath=application.modules.pay.migrations
```

Next, the module is configured in the main configuration file:

```
'modules'=>array(
    'pay'=>array(
        'public_key' => 'pk_test_XXXX',
        'private_key' => 'sk_test_XXXX',
        'redirectOnSuccess' => '/order/create'
    )
),
```

The public and private keys are required, and comes from the Stripe account. The `redirectOnSuccess` property is a new one I've added that I thought would be nice to have in the extension. It allows you to set a new destination for the user upon a successful charge.

The module now has these properties:

```
class PayModule extends CWebModule {
    public $public_key;
    public $private_key;
    public $redirectOnSuccess;
    public $amount;
```

The amount needs to be set dynamically. The `PayModule` class's `beforeControllerAction()` method is where that should happen, as it's designed for code that customizes the extension:

```
public function beforeControllerAction($controller, $action) {
    if($parent::beforeControllerAction($controller, $action)) {
        // Get the cart:
        $cart = Utilities::getCart();
        $this->amount = $cart->getTotal();
        if($this->amount < 50) {
            throw new CException('You have nothing to purchase.');
        }
        return true;
    } else {
        return false;
    }
}
```

With this code setting the amount dynamically, the extension has been updated to use the `amount` property (this value was hard-coded in the Chapter 19 version):

```
# pay/controllers/DefaultController.php
public function actionIndex() {
    $model=new Payment;
    $model->amount = Yii::app()->controller->module->amount;
```

The last changes to the extension include storing the payment ID in the session and then allowing the module to redirect to another page, if `redirectOnSuccess` is set:

```
if ($charge->paid == true) {
    $model->charge_id = $charge->id;
    $model->save();

    // Store values in session:
    Yii::app()->session['payment_id'] = $model->id;

    // Redirect:
    if (!empty(Yii::app()->controller->module->redirectOnSuccess)) {
        $this->redirect(
            array(Yii::app()->controller->module->redirectOnSuccess));
    } else {
        $this->redirect(array('thanks', 'amount' => $model->amount));
    }
}
```

Now, if the Stripe charge succeeds, the payment details are stored in the database, the payment ID is stored in the session, and the customer is redirected to “order/create”.

As a reminder, the extension itself creates a simple form for taking the customer’s details (**Figure 23.5**). The page also defines the proper JavaScript to send the payment details securely to Stripe, receiving a representative token in return. That token is then used by the server-side code to actually make the charge request.

(With this better version of the extension, usage of it would only require configuration of the extension in the main configuration file and, optionally, setting the amount dynamically in the controller.)

## Creating Customers

After the customer has paid, a few things have to happen:

1. The customer record needs to be created in the database.

# Pay

Fields with \* are required.

Email \*

Credit Card Number \*

Credit Card Expiration (MM/YYYY) \*

 / 

Credit Card CVC \*

**Pay \$65.39**

**Figure 23.3:** The payment form.

2. The order itself needs to be recorded in the database.
3. The actual customer needs to be able to download the purchase(s).

Upon purchase, the customer is sent to “order/create”, so the code for the above steps goes within that method. The first thing the method needs to do is retrieve the payment details:

```
public function actionCreate() {

    // Must have processed a charge before coming here:
    if (!isset(Yii::app()->session['payment_id'])) {
        throw new CHttpException(400, 'You have not made a purchase.');
    }

    // Get the payment info:
    $cmd = Yii::app()->db->createCommand('SELECT * FROM payment
    WHERE id=:id');
    $payment_id = Yii::app()->session['payment_id'];
    $cmd->bindParam(':id', $payment_id, PDO::PARAM_INT);
    $payment = $cmd->queryRow();
    if ($payment === null) {
        throw new CHttpException(404, 'You have not made a purchase.');
    }
}
```

Next, the customer record should be recorded in the database. Because the customer may have purchased something in the past, the code first checks for the customer’s

existence already, using the email address provided during the payment process:

```
// Fetch the customer, if existing:  
$customer=Customer::model()->find('email=:email',  
array(':email' => $payment['email']));  
  
// If no customer, create a new one:  
if($customer==null) {  
    $customer = new Customer;  
    $customer->email = $payment['email'];  
    $customer->save();  
}
```

With the customer record either created or retrieved, the next step is to store the order in the database.

## Recording Orders

Orders are stored in two tables: `order` and `order_content`. The “create” action of the “Order” controller will first create and save the `Order` instance:

```
$order=new Order;  
$order->customer_id = $customer->id;  
$order->payment_id = $payment['id'];  
$order->total = $payment['amount'];  
$order->date_entered = $payment['date_added'];  
$order->save();
```

Next, the order details—the contents of the order—need to be saved within the `order_content` table. The smoothest way to do that is to add code to the `Order` class that populates the related table:

```
# protected/models/Order.php  
public function afterSave() {  
    // Store the order contents in the order contents table:  
    $cart = Utilities::getCart();  
    $cmd = Yii::app()->db->createCommand('INSERT INTO order_content  
    (order_id, book_id, quantity, price_per)  
    SELECT :order_id, cc.book_id, cc.quantity, b.price  
    FROM cart_content AS cc, book AS b  
    WHERE (b.id=cc.book_id) AND (cc.cart_id=:cart_id)');  
    $order_id = $this->id;  
    $cart_id = $cart->id;
```

```
$cmd->bindParam(':order_id', $order_id, PDO::PARAM_INT);
$cmd->bindParam(':cart_id', $cart_id, PDO::PARAM_INT);
$cmd->execute();
$cart->clear();
}
```

The code itself should be pretty self-explanatory. The underlying SQL command is:

```
INSERT INTO order_content (order_id, book_id, quantity, price_per)
SELECT :order_id, cc.book_id, cc.quantity, b.price
FROM cart_content AS cc, book AS b
WHERE (b.id=cc.book_id) AND (cc.cart_id=:cart_id)
```

This is an example of a `INSERT INTO...SELECT` query, which is an SQL syntax I rather like. The values selected are used for the insert. Those values need to be the:

- Order ID
- Book ID
- Quantity
- Price per book

The order ID is available within the method through `$this->id`. That value is selected as a static value (as opposed to selecting it from a table). The other values come from the `cart_content` and `book` tables.

The final step in the method is to clear out the cart's contents (since the items have all been purchased): `$cart->clear()`. The `clear()` method is defined within the `Cart` class:

```
public function clear() {
    $cmd = Yii::app()->db->createCommand('DELETE FROM cart_content
WHERE cart_id=:cart_id');
    $cart_id = $this->id;
    $cmd->bindParam(':cart_id', $cart_id, PDO::PARAM_INT);
    $cmd->execute();
    $cmd = Yii::app()->db->createCommand('DELETE FROM cart
WHERE id=:cart_id');
    $cmd->bindParam(':cart_id', $cart_id, PDO::PARAM_INT);
    $cmd->execute();
}
```

The method has to clear out both the `cart` and `cart_content` tables.

*{TIP}* If you want to create a “clear cart” functionality, it just needs to link to an action that calls this the `clear()` method of the `Cart` class.

Returning to the “Order” controller, the last step is to render the view file that shows the order:

```
$this->render('view',array(  
    'model'=>$order  
)  
)  
}
```

The view file starts with a `CDetailView` widget to display the order details (**Figure 23.6**).

## View Order #14

Order Number	14
Email	larryullman@gmail.com
Total	\$65.39
Date Entered	Dec. 20, 2014

**Figure 23.4:** The start of the view order page.

```
<?php $this->widget('zii.widgets.CDetailView', array(  
    'data'=>$model,  
    'attributes'=›array(  
        'id',  
        'customer.email',  
        array(  
            'name' => 'total',  
            'value' => Utilities::formatAmount($model->total)  
        ),  
        array(  
            'name' => 'date_entered',  
            'value' => Utilities::formatDate($model->date_entered)  
        ),  
    ),  
) ; ?>
```

Next, a link to download each book is shown (**Figure 23.7**).

## Download Purchases

- [The Yii Book](#)
- [Effortless E-commerce with PHP and MySQL \(2nd Edition\)](#)
- [PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide \(4th Edition\)](#)

```
<h2>Download Purchases</h2>
<ul>
<?php $items = $model->orderContents;
foreach ($items as $item) {
    echo '<li>' . CHtml::link($item->book->title,
        array ('/book/download', 'id' => $item->book_id)) . '</li>';
}
?>
</ul>
```

The link goes to “book/download”, passing along the book ID.

## Downloading Books

The “download” action of the “book” controller sends the PDF files to the user through the browser. It requires a book ID. The trick to sending the PDF is that the file itself is stored in another directory, presumably not within the Web root. To send the PDF to the browser, the right header lines have to be sent. I’ll explain the method a few bits at a time:

```
public function actionDownload($id) {
    $book = $this->loadModel($id);
```

First the method loads the book using the provided ID. If no such book exists, the `loadModel()` method would throw an exception.

In the complete example, I’d add code here that checks that the current user has the right to download the given book. Likely this would be accomplished by storing the customer ID in the session (after completing a purchase or logging in), and then checking the `order_content` table for the combination of that customer ID and this book ID. If no match was found, an exception would be thrown.

Next, information about the book file has to be retrieved:

```
$file = Yii::getPathOfAlias('application') . '/books/' . $id . '.pdf';
$fs = filesize($file);
$title = $book->filename;
```

The book uses the name “X.pdf”, where X is the book’s ID (similar to how the book’s images are named). The PDFs are stored in the `protected/books` directory. The file’s size is determined, as is the book’s title, from the database. This will allow the code to change the downloaded file to use, for example, “yiibook.pdf”, as the file’s name, instead of “1.pdf”.

Next, the method sends all the necessary headers:

```
header("Pragma: public");
header("Expires: 0");
header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
header("Cache-Control: private",false);
header("Content-Description: File Transfer");
header("Content-Transfer-Encoding: binary");
header("Content-Type: application/pdf");
header("Content-Disposition: attachment; filename=\"$title\"");
header("Content-Length: $fs");
```

You can look these up for yourself to see what you do, but the long and the short of it is that this combination works across the browsers I've tested in.

Finally, the file itself is sent to the browser, and the application is terminated (so nothing is rendered):

```
    readfile($file);
    Yii::app()->end();
}
```

## Things to Possibly Add

This is the heart of a mostly complete e-commerce site, with shopping cart functionality, actual payments, customer creation, and delivery of goods. Still, it's not 100% complete (nor was it intended to be). Just a few things I know you'd want to do implement still are:

- Removal of all un-used controllers and actions
- Fixing of permissions on all controller actions
- Ability to update the quantity of items in the cart
- Complete customer account management (registration after purchase, ability to change a password, forgotten password tool, etc.)
- Deny access to books unless the user has purchased them
- Handle all exceptions nicely
- A custom layout and design

The SQL commands also create definitions for two more tables: `download` and `password_token`. The former can be used to track the downloads of books, which may be useful. The latter would be used as part of a “forgot password” logic.

## Selling Physical Goods

In the introduction to this chapter, I explain how a digital e-commerce site differs greatly from one that sells physical goods. From a programming perspective, there are different steps you'll need to take.

First, every e-commerce site needs unique identifiers—call them “product IDs” or “SKUs”—for each item sold. Unique identifiers is the only way you can track shopping carts, manage inventory, and deliver goods.

Second, with physical goods, you need to track inventory:

- Increase inventory when new stock is received
- Decrease inventory when items are sold
- Display inventory unavailability (i.e., “out of stock”) on the site

This is not hard to do, but you'll need to make some policies. For example, if putting something in a cart decreases the inventory on hand, you'll never have the bad experience of a user suddenly not being able to buy something because the last one was sold while it was in their cart. On the other hand, you'll lose some sales, as the reduced inventory would reflect what people *think* they're going to buy, not what has actually been purchased. How you handle this will differ from one business model to another.

Similarly, just being out of stock of something may not rule out it being purchased. It could be an item that gets replenished on a frequent and reliable basis.

With physical goods, you also have to handle shipping and delivery. This means knowing the:

- Origination and destination addresses
- Package's size and weight
- Shipping costs by service
- Approximate shipping time

For the package's size and weight, you'd want to store this in the database, along with the other product details. That way, when a user buys 2 of Widget A and 1 of Widget D, you can know how much it weighs in total.

For calculating the costs, the clear best approach would be to use a third-party service, such as a national or international carrier. They all have APIs through which you send all of the above information, and the API returns the cost options. There's no better way to handle this.

Finally, a site that sells physical goods needs people to actually do all of the above: receive inventory, package up outgoing parcels, order more inventory, etc. This would likely not be your problem (if you're just the developer), but it does mean you'll need to create administrative interfaces to help in this area:

- Show orders to be shipped
- Create shipping labels (perhaps)
- Show which items need to be re-ordered (i.e., low inventory)

Obviously there are many differences between selling virtual products and physical goods. But even within each broad category, a multitude of differences exist: virtual products that require licenses (e.g., software) need extra code; physical goods that are perishable have to be handled more cautiously; for other products, different local, state, and national laws may apply. But hopefully through this chapter, you have a good taste as to some of the components of implementing e-commerce in Yii.

## Chapter 24

# SHIPPING YOUR PROJECT

You've done it: you've wrapped up your Yii project! You've tested it, gone through reviews and edits with your client (or by yourself), and polished it thoroughly. The project is done. Well, the *development* version is done. Now you have to take the site live. How do you do that?

In this chapter, I'll walk through many of the steps you should take in order to ship your project. By "ship", I mean: take the site from a development status to a production one.

Admittedly, some of these ideas could be implemented from the onset. If so, that's something you can apply to your next Yii project!

### The Framework Installation

For security purposes, the framework itself should always be installed outside of the application directory. If you have a server that runs multiple Yii-based sites, it's best to place the framework files in a directory that's commonly accessible, such as `/var/www/yii-version`.

Once you've placed the framework files in a new location, or anytime you're moving the Yii site to a production server with a different relationship between the site files and the framework files, you'll need to change this line in your bootstrap `index.php` file:

```
$yii = dirname(__FILE__).'/../framework/yii.php';
```

For example, you could change that to an absolute reference to the communal framework directory:

```
$yii = '/var/www/yii-1.1.15.022a51/framework/yii.php';
```

However, having the version number in the path is tedious and ugly. It'd be clearer if you create a symbolic link to that directory:

```
ln -s /var/www/yii-1.1.15.022a51 /var/www/yii
```

Now the **index.php** file can use:

```
$yii = '/var/www/yii/framework/yii.php';
```

A nice benefit of this approach is that the Yii framework version is still preserved in the original directory name (MySQL does a similar thing on Mac OS X, symbolically linking the most current version to `/usr/local/mysql`).

Further, when you go to install a new version of the framework, you can install the framework in its own new directory. Then you can use a test version of the bootstrap file that uses that new Yii version:

```
$yii = '/var/www/yii-1.1.16.044bc7/framework/yii.php';
```

If everything works correctly—and your unit and functional testing could be of great help here, then recreate the symlink to the newer version and your default bootstrap file will start using it.

## Setting Permissions

A common misstep when taking a site live is not having the proper permissions set on the various directories. This can lead to problems with your logs and assets, as Yii needs to write data to those directories (**protected/runtime**, in the case of log files).

In Unix permissions terms, you should establish 0755 (directory read + write) permissions on:

- **protected/runtime**
- **web root/assets**

You may also find you'll want to restrict the permissions on directories that were previously more open (all in **protected**):

- **controllers**

- **data**
- **models**
- **views**

Those should all be fine with 0644 permissions.

## Transferring Assets

You may be tempted to copy over the *contents* of your **assets** folder when you take your site live. Do not do this! The Yii asset manager, see Chapter 19, “[Extending Yii](#),” will take care of your assets for you. Just make sure the **assets** folder exists on your live server and that it has writable permissions by the Web server. That’s all you need to do; Yii will take care of the rest.

If you’d rather store your assets in a different directory, you just need to configure the assets manager. This, of course, is done in the primary configuration file. Set a new base path and base URL:

```
'assetManager' => array(
    'basePath' => 'fullbasepath' . DIRECTORY_SEPARATOR . 'a',
    'baseUrl' => DIRECTORY_SEPARATOR . 'a',
),
```

That code tells Yii to publish and serve the assets from the **a** directory instead.

## Moving the Application Directory

When you’re taking your site live, it’s a good time to re-evaluate moving the application directory (i.e., **protected**) outside of the Web directory, if you have not already.

To do so, just move your **protected** directory to a place outside of the Web directory. Then set the application’s **basePath** in the main configuration file to point to it:

```
'basePath' => 'new/path/to/protected'
```

*{TIP}* Moving the protected folder outside of the Web root directory is just an extra security precaution. It’s not required, and you may not want to bother with the change, especially as you’re just getting started.

Next, you’ll need to edit the bootstrap file so that it properly references the configuration:

```
$config=dirname(__FILE__).'/../protected/config/main.php';
```

And that's it!

If it's not an option to move the **protected** directory outside of the Web directly, make sure that the password protection on it works. The following **.htaccess** file should already be in the **protected** directory, created by **yiic**:

```
deny from all
```

You can test this is working by trying to access anything within **protected** directly in your browser (e.g., head to <http://example.com/protected/config/main.php>).

If you'd rather break up your application further, you can specifically set the locations for the following:

- `controllerPath`
- `extensionPath`
- `layout` (primary layout file)
- `layoutPath`
- `viewPath`

## Taking the Bootstrap File Live

The default bootstrap file (**index.php**), created by the **yiic** command, is coded for easier development. Specifically, it's written such that debugging is enabled. That's not something you'll want on a live site, though, so you'll need to delete these lines:

```
defined('YII_DEBUG') or define('YII_DEBUG',true);
defined('YII_TRACE_LEVEL') or define('YII_TRACE_LEVEL',3);
```

Removing the first line disables debugging mode. You'll want to do that for both performance and security reasons.

The second line identifies the level of stack tracing to perform—how many steps of execution should be logged. Again, having this kind of logging in place on a live site will hurt performance.

## Temporary Debugging

If you're lucky, or just well organized, you have a development installation of your site and a production version. If so, then you'll never end up making code changes

or (again, hopefully) debugging a live site. But if, for whatever reason, you do have that need, there's a trick to enable debugging for only yourself.

The bootstrap file originally has this line:

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

You could replace it with something like this:

```
if (isset($_GET['debug'])) define('YII_DEBUG', true);
```

If you do that, then to debug any page on the fly, change the URL from:

**www.example.com/index.php/site/contact**

To:

**www.example.com/index.php/site/contact/debug/true.**

If you do go this route, do yourself a favor and make the term less obvious than “debug”, and remove that line (or comment it out) once you’re done.

The only downside to this approach is that it won’t allow you to test the submission of forms (because `$_GET['debug']` won’t be passed upon submission). If that’s a need, check for the presence of a cookie to enable debugging, and then set that cookie for yourself.

## Using Multiple Configuration Files

By default, Yii creates three configuration files for an application:

- **main.php**, the primary one
- **console.php**, for use with console applications
- **test.php**, used for unit tests

You may find this arrangement to be sufficient. However, in more complex applications, having all of your primary application configuration be addressed in a single file can become unwieldy. Further, if you tend to use common configurations among all your Yii projects, having a new default configuration file, to be amended and appended by the main configuration file, may make more sense. You might also need multiple configuration files for different environments:

- Development
- Staging
- Production (aka live)

There are a couple of ways you can make use of multiple configuration files in Yii.

For the environment situation, a common solution is to watch for a server variable in the bootstrap file and include the appropriate configuration file based upon that variable's value:

```
switch ($_SERVER['HTTP_HOST']) {
    case 'localhost':
        $config = 'localhost';
        break;
    case 'dev.example.com':
        $config = 'development';
        break;
    default:
        $config = 'production';
        break;
}
$config = dirname(__FILE__).'/protected/config/' . $config . '.php';
```

{TIP} Watching for a server variable can also be used to dynamically update the bootstrap file for different locations of the framework directory relative to the application.

The above approach dynamically switches the primary configuration file based upon context. You can also break the single configuration file into multiple (which can be used with or without the above technique). This can be simply done once you remember that the configuration file is one big array, and each configuration item is often an array, too.

For example, you may want to break out the database configuration so that the main and console configurations can both use it. Here's how that might be configured within **main.php**:

```
'db'=>array(
    'connectionString' => 'mysql:host=localhost;dbname=yii_cms',
    'emulatePrepare' => true,
    'enableParamLogging' => true,
    'username' => 'root',
    'password' => 'password',
    'charset' => 'utf8',
),
```

To break this out, first create a file, named **db.php**, stored within the **protected/-config** directory. That file would contain:

```
<?php
return array(
    'connectionString' => 'mysql:host=localhost;dbname=yii_cms',
    'emulatePrepare' => true,
    'enableParamLogging' => true,
    'username' => 'root',
    'password' => 'password',
    'charset' => 'utf8',
);
```

To use this file within another configuration file, you'd write:

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

That code requires the **db.php** file found within the same directory as the current configuration file.

That's all there is to it! You can use this to separate out anything that you're starting to find is cluttering up your configuration:

- URL routing rules
- Application parameters (i.e., not specific to components)
- Module inclusions and configurations
- Logging settings

## Better Logging

During development of a project, you'll normally want to enable Web logging, so that Yii immediately displays the details of any problems to you. On a live site, however, you can't do that (for many obvious reasons). But using logging is still important on live sites; you just need to be smarter in how you approach it.

The most important thing to keep in mind is that logging will hurt the performance of your site, but when things go wrong—bugs or security breaches, having good logs will make your life vastly easier. Towards that end, you'll need to make good decisions as to:

- What occurrences you log
- What data you log
- Where that log goes

For the first question, you can choose to log or not log:

- All access (as your Web server already does)
- Changes to user accounts
- Warnings
- Errors

For the second question, you'll need to specifically decide if you'll want to log some or all existing variables (and their values).

For the third question, you can choose to have the log item:

- Emailed
- Displayed in the Web browser (which you don't want to do on a production site)
- Written to a file
- Outputted such that it's visible in Firebug

There's no one right answer, nor one right solution. You could choose to write some logs to files, but be emailed for serious occurrences.

You can write logs yourself using `Yii::log()`. It takes three arguments: the log message, an error level, and a "category". I'll return to "category" shortly.

The available levels are:

- trace
- info
- profile
- warning
- error

For example, to write an "info" log, you'd do this:

```
Yii::log('Somebody did something.', 'info');
```

So what happens with that log? Well, it depends upon your configuration!

That line of code adds the log item to memory. Yii will then output it to the proper destination, which is to say, Yii will route the log message. Message routing is handled via the `CLogRouter` component, configured in your application:

```
'log'=>array(
    'class'=>'CLogRouter',
    'routes'=>array(
        array(
            'class'=>'CFileLogRoute',

```

```
    'levels'=>'error, warning',
),
// uncomment the following to show log messages on web pages
// array(
//   'class'=>'CWebLogRoute',
// ),
),
),
),
```

That's the default Yii configuration for **CLogRouter**. Note that the "log" component is the only preloaded Yii component, by default:

```
'preload'=>array('log'),
```

You configure all of your log handling by adding routes to the "log" configuration. This is an array of one or more elements. The above creates one log route: writing all errors and warnings to a file. If you uncomment the second array, that would create a second route: logging *everything* to the Web browser.

Each destination is identified by a class:

- **CDbLogRoute**
- **CEmailLogRoute**
- **CFileLogRoute**
- **CProfileLogRoute**
- **CWebLogRoute**

Each route is further customized by assigning properties to the class's attributes. Each class extends **CLogRoute**, which defines the **levels** property, among others.

With that in mind, to have yourself be emailed for errors, you'd configure a route like so:

```
'log'=>array(
  'class'=>'CLogRouter',
  'routes'=>array(
    array(
      'class'=>'CEmailLogRoute',
      'levels'=>'error',
      'emails'=>array('me@example.edu'),
      'subject'=>'Site Error!',
      'sentFrom'=>'site@example.com'
    ),
  ),
),
```

As another example, the `CWebLogRoute` class has a `showInFirebug` property. When set to true, Yii will output the log message such that it only shows in Firebug. Thus you can have the convenience of Web logging without affecting the user experience.

The log routing classes also have `categories` and `except` properties that can be used to create more nuanced logging. For example, say that you want to handle problems with the database different from any other problem. The following code logs most errors to files, but sends emails for the database category:

```
'log'=>array(
    'class'=>'CLogRouter',
    'routes'=>array(
        array(
            'class'=>'CFileLogRoute',
            'levels'=>'error',
            'except'=>'system.db.*',
        ),
        array(
            'class'=>'CEmailLogRoute',
            'levels'=>'error',
            'categories'=>'system.db.*',
            'emails'=>array('me@example.edu'),
            'subject'=>'Database Error!',
            'sentFrom'=>'site@example.com'
        ),
    ),
),
```

The categories should be strings in the format `xxx.yyy.zzz`, like path aliases. They are made up aliases, but will be treated hierarchically, allowing you to have a logging approach for `views.*` but exempt `views.site.*`, if you so choose.

*{TIP}* Logs are stored in memory and outputted when the application ends. Avoid using `die()` or `exit()` in your Yii applications, or else Yii won't be able to do the necessary clean up, such as outputting logs.

The last thing to know about logging is how to add contextual information such as the variables that exist at the time of the logging. Contextual data is added via the `CLogFilter` class. To simply enable it, provide it as a "filter" configuration for any log route:

```
'log'=>array(
    'class'=>'CLogRouter',
    'routes'=>array(
        array(
```

```
'class'=>'CFileLogRoute',
'levels'=>'error',
'filter'=>'CLogFilter',
),
),
),
),
```

You can further customize how the contextual logging works by providing values for the various `CLogFilter` class:

```
'log'=>array(
    'class'=>'CLogRouter',
    'routes'=>array(
        array(
            'class'=>'CFileLogRoute',
            'levels'=>'error',
            'filter'=> array(
                'class' => 'CLogFilter',
                // Include username & ID involved:
                'logUser' => true,
                // Log $_REQUEST:
                'logVars' => array('REQUEST')
            ),
        ),
    ),
),
),
```

## Things to Do

To complete this chapter (and the book!), here's a quick checklist of things you'll want to do as part of shipping a project:

- Make sure all your tests pass, if you've created them
- Disable debugging
- Disable Gii and any other modules or extensions that are no longer needed
- Disable any back-doors or potential security holes you may have created for “convenience” while developing
- Check the permissions on the directories
- Properly configure logging
- Minify your CSS and JavaScript
- Use CDN-hosted versions of libraries, if possible
- Hammer your site with problematic input to confirm nothing terrible happens or is shown

- Cache, cache, cache
- Create a back up plan!
- Start making a list of things to change, add, and fix in the next revision
- Celebrate!