

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.

Revision: 0.61

Copyright © 2013 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

| | |
|---|-----------|
| Introduction | 1 |
| Why Frameworks? | 1 |
| Why Yii? | 3 |
| What You'll Need | 7 |
| About This Book | 8 |
| Getting Help | 11 |
| 1 FUNDAMENTAL CONCEPTS | 13 |
| Object-Oriented Programming | 13 |
| The MVC Approach | 19 |
| Using a Web Server | 25 |
| Command Line Tools | 26 |
| 2 STARTING A NEW APPLICATION | 31 |
| Downloading Yii | 31 |
| Testing the Requirements | 32 |
| Installing the Framework | 34 |
| Building the Site Shell | 34 |
| Testing the Site Shell | 36 |
| 3 A MANUAL FOR YOUR YII SITE | 38 |
| The Site's Folders | 38 |
| Referencing Files and Directories | 40 |
| Yii Conventions | 41 |
| How Yii Handles a Page Request | 42 |

| | |
|--|------------|
| 4 INITIAL CUSTOMIZATIONS AND CODE GENERATIONS | 47 |
| Enabling Debug Mode | 47 |
| Moving the Protected Folder | 48 |
| Basic Configurations | 49 |
| Developing Your Site | 60 |
| Generating Code with Gii | 70 |
| 5 WORKING WITH MODELS | 77 |
| The Model Classes | 77 |
| Establishing Rules | 79 |
| Changing Labels | 96 |
| Watching for Model Events | 99 |
| Relating Models | 103 |
| 6 WORKING WITH VIEWS | 107 |
| 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 |
| 7 WORKING WITH CONTROLLERS | 131 |
| 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 |

| | |
|---|------------|
| 8 WORKING WITH DATABASES | 161 |
| Debugging Database Operations | 161 |
| Database Options | 163 |
| Using Active Record | 164 |
| Using Query Builder | 179 |
| Using Database Access Objects | 184 |
| Choosing an Interface Option | 187 |
| Common Challenges | 189 |
| 9 WORKING WITH FORMS | 192 |
| Understanding Forms and MVC | 192 |
| Creating Forms without Models | 193 |
| Using CHtml | 195 |
| Using “Active” Methods | 196 |
| Using CActiveForm | 197 |
| Using Form Builder | 199 |
| Common Form Needs | 205 |
| 10 MAINTAINING STATE | 224 |
| Cookies | 224 |
| Sessions | 228 |
| 11 USER AUTHENTICATION AND AUTHORIZATION | 233 |
| Fundamentals of Authentication | 233 |
| Authentication Options | 244 |
| The UserIdentity State | 250 |
| Authorization | 254 |
| Working with Flash Messages | 268 |
| 12 WORKING WITH WIDGETS | 271 |
| Using Widgets | 271 |
| Basic Yii Widgets | 274 |
| Presenting Data | 280 |
| The jQuery UI Widgets | 302 |

| | |
|---|------------|
| 13 USING EXTENSIONS | 309 |
| The Basics of Extensions | 309 |
| The bootstrap Extension | 312 |
| The giix Extension | 317 |
| Validator Extensions | 320 |
| Auto-Setting Timestamps | 322 |
| Using a WYSIWYG Editor | 323 |
| 14 JAVASCRIPT AND JQUERY | 327 |
| What You Must Know | 327 |
| Adding JavaScript to a Page | 328 |
| Using JavaScript with CActiveForm | 333 |
| Implementing Ajax | 337 |
| Common Needs | 349 |

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 1.1.13. It is expected that version 2 of the Yii framework will have its alpha release in early 2013.

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 Will Be New in Yii 2?

In early 2013, the alpha release of Yii 2 should come out, with the general release coming later in the year. At the time of this writing, the major changes for Yii 2 are still under consideration, but it is known that Yii 2 will:

- Use namespaces for its classes (in keeping with more recent PHP adoption of namespaces)
- Have a more logical structure for its MVC components
- Do more for creating console applications
- Make improvements for working with databases, including Active Record changes
- All in all, be even more beautiful (truly!)

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

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.

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**).

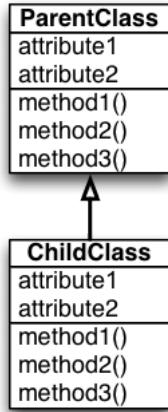


Figure 1.1: The child class can inherit members from the parent class.

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 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.

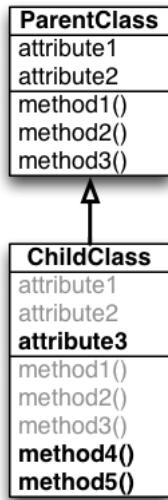


Figure 1.2: The child class can add new members to the ones it inherited.

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**).

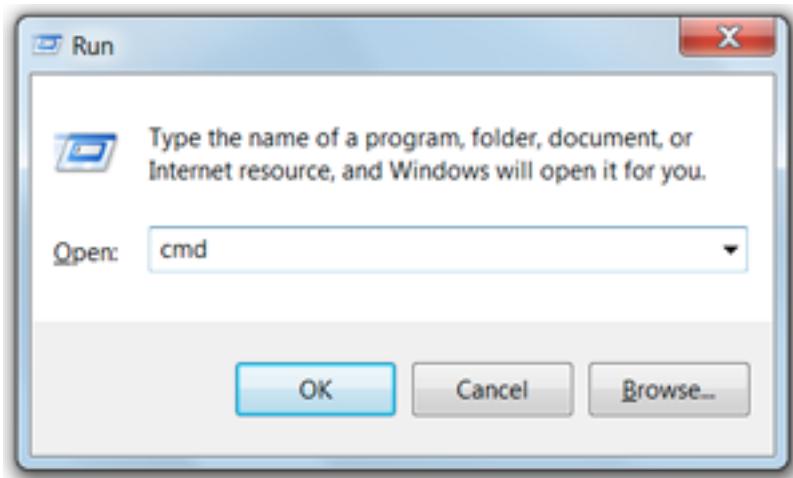


Figure 1.3: The cmd prompt.

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.

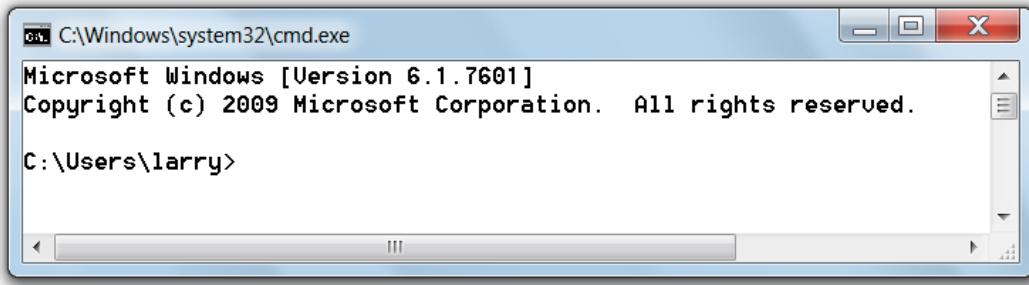


Figure 1.4: The command line interface on Windows.

{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.

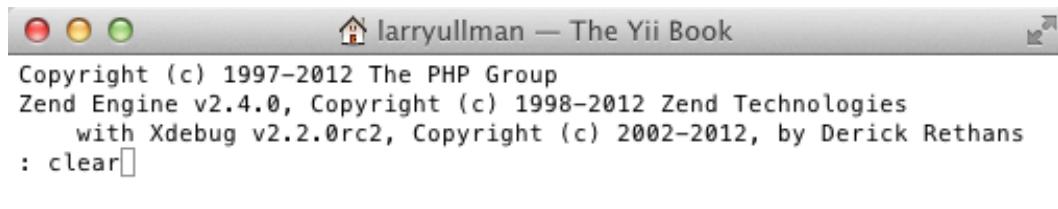
On versions of *nix, including Mac OS X, referencing the PHP executable is rarely a problem. On Windows, it might be. To test your setup, type the following in your command-line interface and press Enter/Return:

```
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*.



A screenshot of a Windows command-line interface window titled "larryullman — The Yii Book". The window shows the output of the PHP executable, which includes copyright information for The PHP Group and Zend Technologies, and a command prompt line starting with "clear".

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 semicolon 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

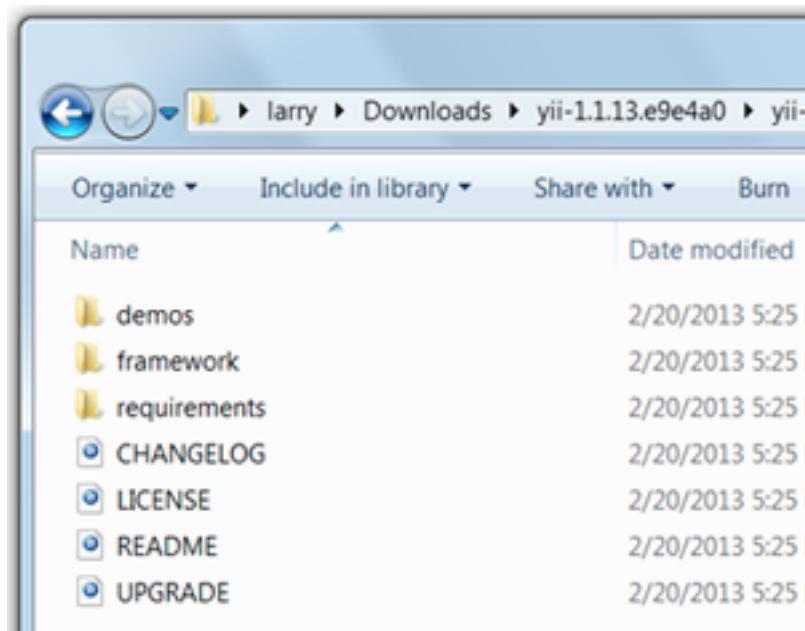


Figure 2.1: The contents of the downloaded Yii framework folder.

Yii: a blog, the game Hangman, a basic “Hello, World!”, and a phone book. The **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 5.3). 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](#)).

- 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>CWsdlGenerator</code> | |
| PDO extension | Passed | All DB-related classes | |
| PDO SQLite extension | Passed | All DB-related classes | This is required if you are using SQLite database. |
| PDO MySQL extension | Passed | All DB-related classes | This is required if you are using MySQL database. |
| PDO PostgreSQL extension | Passed | All DB-related classes | This is required if you are using PostgreSQL database. |
| PDO Oracle extension | Warning | All DB-related classes | This is required if you are using Oracle database. |
| PDO MSSQL extension (<code>pdo_mssql</code>) | Warning | All DB-related classes | This is required if you are using MSSQL database from MS Windows. |
| PDO MSSQL extension (<code>pdo_iblib</code>) | Passed | All DB-related classes | This is required if you are using MSSQL database from GNU/Linux or other UNIX. |
| PDO MSSQL extension (<code>pdo_sqlsrv</code>) | Warning | All DB-related classes | 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

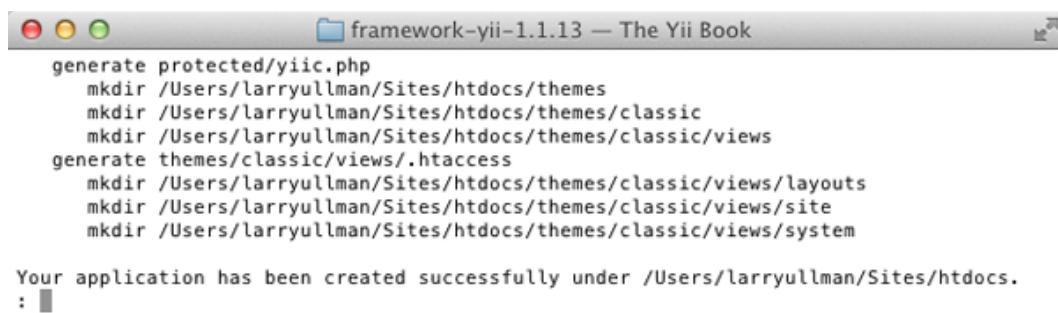
```
./yiic webapp ../htdocs
```

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**).



```
framework-yii-1.1.13 — The Yii Book
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.
:
```

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**).

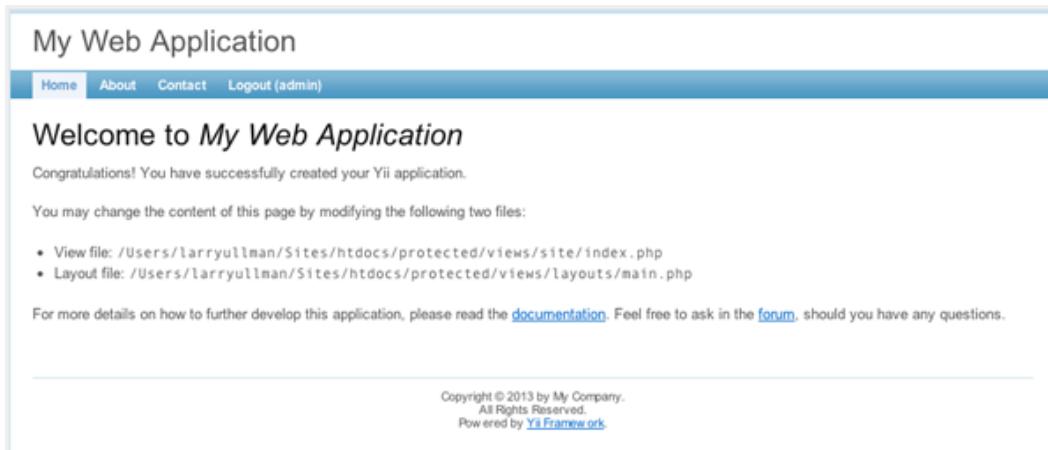


Figure 2.4: The shell of the generated site.

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 personal 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 | framework folder |
| zii | Zii library location |
| application | protected folder |
| webroot | directory where you can find the index file |
| ext | protected/extensions |

If you're using modules in your site (to be covered in Chapter 15, "Working with Modules"), 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).

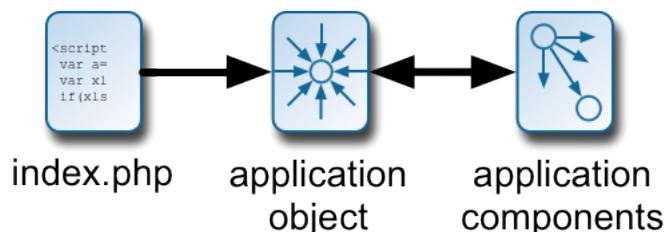


Figure 3.1: The index page creates an application object which loads the application components.

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](#)).

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:

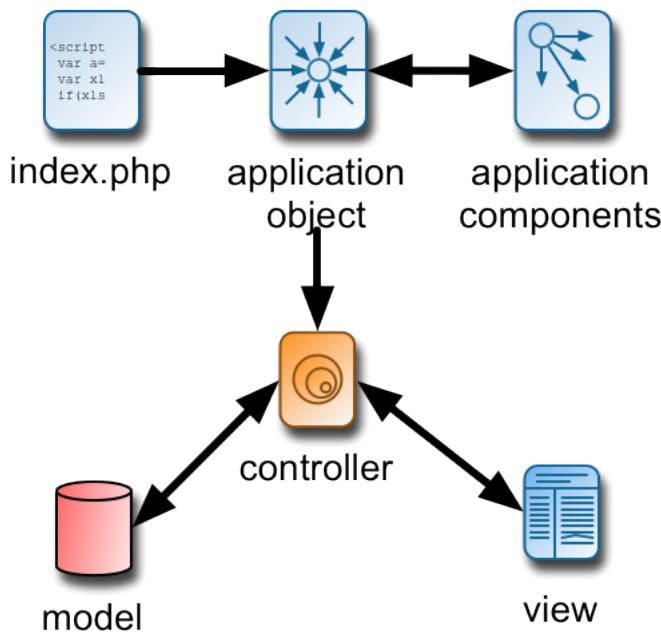


Figure 3.2: Subsequent steps involve the correct controller being called, accessing models, and rendering views.

<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 26, "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 20, “Implementing Unit Tests,” 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 Customing 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**).

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.

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**).



```
application.log
1 2013/02/22 14:43:31 [error] [exception.CHttpException.404] exception 'CHttpException' with message
2 'The system is unable to find the requested action "contacts".' in
3 /Users/larryullman/Sites/framework-yii-1.1.13/web/CController.php:483
4 Stack trace:
5 #0 /Users/larryullman/Sites/framework-yii-1.1.13/web/CController.php(270):
6 CController->missingAction('contacts')
7 #1 /Users/larryullman/Sites/framework-yii-1.1.13/web/CWebApplication.php(282):
8 CController->run('contacts')
9 #2 /Users/larryullman/Sites/framework-yii-1.1.13/web/CWebApplication.php(141):
10 CWebApplication->runController('site/contacts')
11 #3 /Users/larryullman/Sites/framework-yii-1.1.13/base/CApplication.php(169):
12 CWebApplication->processRequest()
13 #4 /Users/larryullman/Sites/htdocs/index.php(13): CApplication->run()
14 #5 {main}
15 REQUEST_URI=/index.php/site/contacts
16 ---
```

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

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',
            '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**.

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

| 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) |
| 14:43:31.191114 | error | exception.CHttpException404 | 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 Stack trace: #0 /Users/larryullman/Sites/framework-yii-1.1.13/web/CController.php(270): CController->missingAction('contacts') #1 /Users/larryullman/Sites/framework-yii-1.1.13/web/CWebApplication.php(202): 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 ... |

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

essentially mini-applications within a site. You might create an administration module or a forum module. Chapter 15, “Working with Modules”, 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 lets 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](#).

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.

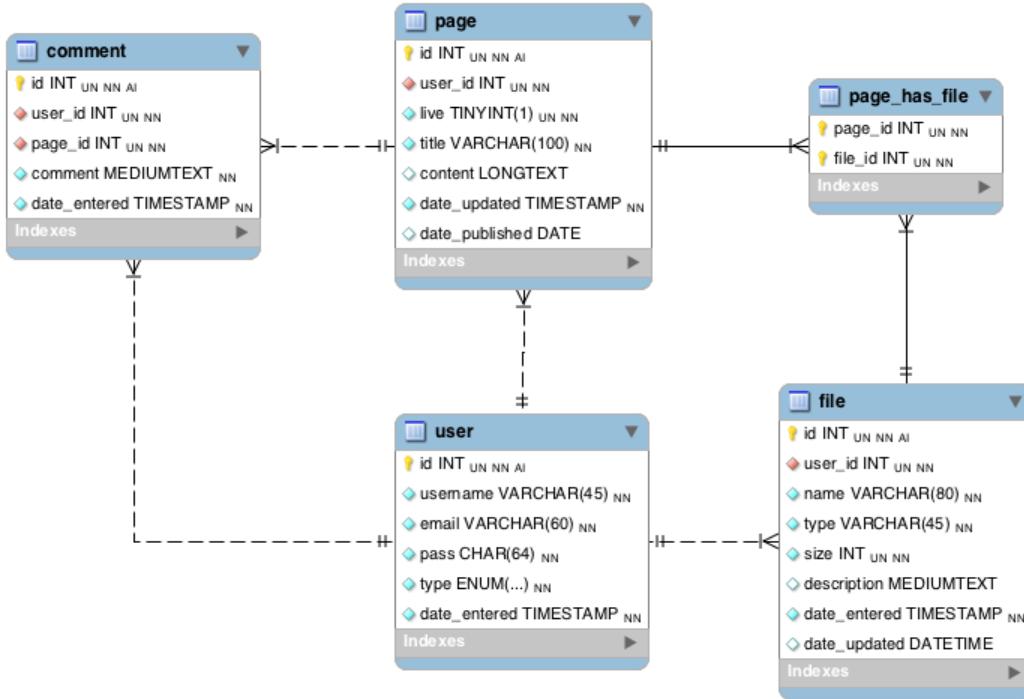


Figure 4.6: The CMS database schema.

{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 23, "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, the Gii tool is at 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 instead.

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

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:

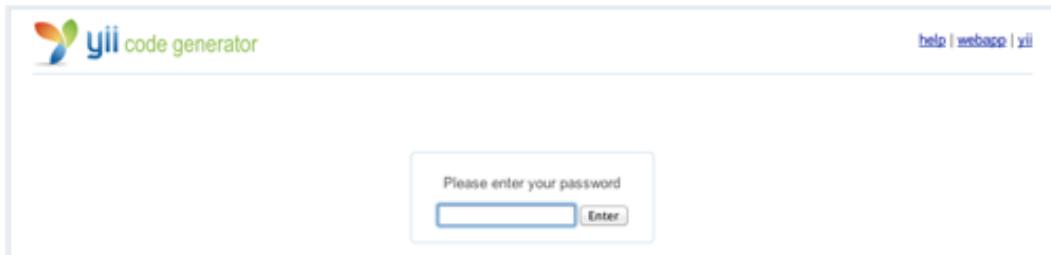


Figure 4.7: The Gii login page.

A screenshot of a web browser showing the Gii home page. The page features the Yii logo and "yii code generator" text at the top. Below this, a large heading says "Welcome to Yii Code Generator!". A subtext below the heading reads "You may use the following generators to quickly build up your Yii application:". A bulleted list follows, each item containing a blue hyperlink: "Controller Generator", "Crud Generator", "Form Generator", "Model Generator", and "Module Generator".

Figure 4.8: The Gii home page.

- Controllers
- CRUD functionality
- Forms
- Models
- Modules

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

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**):

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.

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.

| Preview | Generate |
|--|---|
| Code File | Generate <input type="checkbox"/> |
| models/Comment.php | new <input checked="" type="checkbox"/> |
| models/File.php | new <input checked="" type="checkbox"/> |
| models/Page.php | new <input checked="" type="checkbox"/> |
| models/PageHasFile.php | new <input type="checkbox"/> |
| models/User.php | new <input checked="" type="checkbox"/> |

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

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**).

[Preview](#)

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

```
Generating code using template "/Users/larryullman/Sites/framework-yii-1.1.14"
generating models/Comment.php
    Unable to write the file '/Users/larryullman/Sites/htdocs/protected/models/Comment.php'
generating models/File.php
    Unable to write the file '/Users/larryullman/Sites/htdocs/protected/models/File.php'
generating models/Page.php
    Unable to write the file '/Users/larryullman/Sites/htdocs/protected/models/Page.php'
skipped models/PageHasFile.php
generating models/User.php
    Unable to write the file '/Users/larryullman/Sites/htdocs/protected/models/User.php'
done!
```

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

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

- **Comment.php**
- **File.php**
- **Page.php**
- **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](#)):

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

| | |
|-------------------------|---|
| Model Class * | Page |
| Controller ID * | page |
| Base Controller Class * | Controller |
| Code Template * | default (/Users/larryullman/Sites/framework-yii-1.1.13/gii/gene |

[Preview](#)

Controller ID is case-sensitive. CRUD controllers are often named after the model class name that they are dealing with. Below are some examples:

- `post` generates `PostController.php`
- `postTag` generates `PostTagController.php`
- `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.

Figure 4.12: Generating CRUD functionality for pages.

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`

The screenshot shows a user interface for generating code. At the top, there are two buttons: "Preview" and "Generate". Below them is a table with two columns: "Code File" and "Generate". The "Code File" column lists various PHP files, and the "Generate" column contains checkboxes, all of which are checked.

| Code File | Generate <input checked="" type="checkbox"/> |
|--|--|
| controllers/PageController.php | new <input checked="" type="checkbox"/> |
| views/page/_form.php | new <input checked="" type="checkbox"/> |
| views/page/_search.php | new <input checked="" type="checkbox"/> |
| views/page/_view.php | new <input checked="" type="checkbox"/> |
| views/page/admin.php | new <input checked="" type="checkbox"/> |
| views/page/create.php | new <input checked="" type="checkbox"/> |
| views/page/index.php | new <input checked="" type="checkbox"/> |
| views/page/update.php | new <input checked="" type="checkbox"/> |
| views/page/view.php | 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 are going to be of 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'=>'/^[\w-]{6,20}$/i'),
```

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



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

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 lets 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_entered to NOW() every time:
array('date_entered', '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'),  
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', '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—aren’t 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',
```

Create File

*Fields with * are required.*

Uploaded By

File Name *

File Type *

File Size *

Description

Date Entered

Date Updated

Create

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

```
'pass' => 'Password',
'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**)

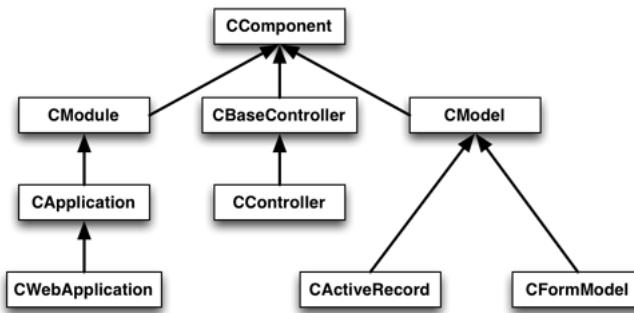


Figure 5.4: Part of Yii's class inheritance structure, with `CComponent` at the top.

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 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 19, “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**).

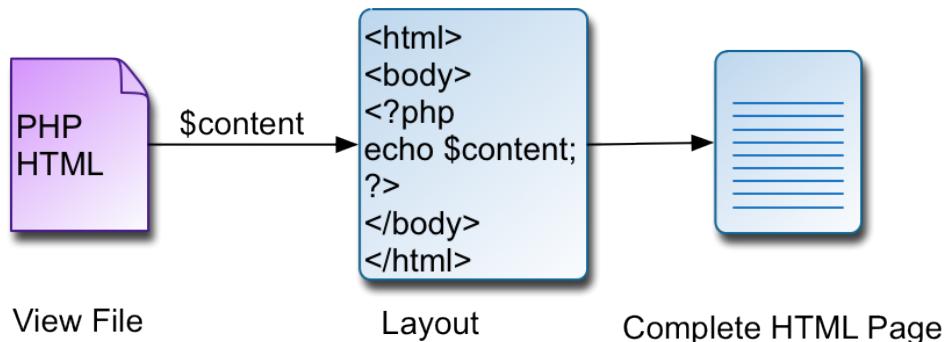


Figure 6.2: How Yii renders a complete page by pulling an individual view file into the layout.

{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> | | |
| \$text | string | link body. It will NOT be HTML-encoded. Therefore you can pass in HTML code such as an image tag. |
| \$url | mixed | a URL or an action route that can be used to create a URL. See normalizeUrl for more details about how to specify this parameter. |
| \$htmlOptions | array | additional HTML attributes. Besides normal HTML attributes, a few special attributes are also recognized (see clientChange and tag for more details.) |
| {return} | 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 for the create a user page (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 http://www.example.com/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="http://www.example.com/index.php?  
    r=pageview&id=23">Something</a>  
<a href="http://www.example.com/index.php/  
    page/view/id/23">Something</a>  
<a href="http://www.example.com/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="http://www.example.com/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 secure 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 <code>Controller</code> |
| controllers/SomeController.php | Every view in <code>SomeController</code> |
| SomeController::actionSomething() | The view rendered by <code>actionSomething()</code> |

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**).

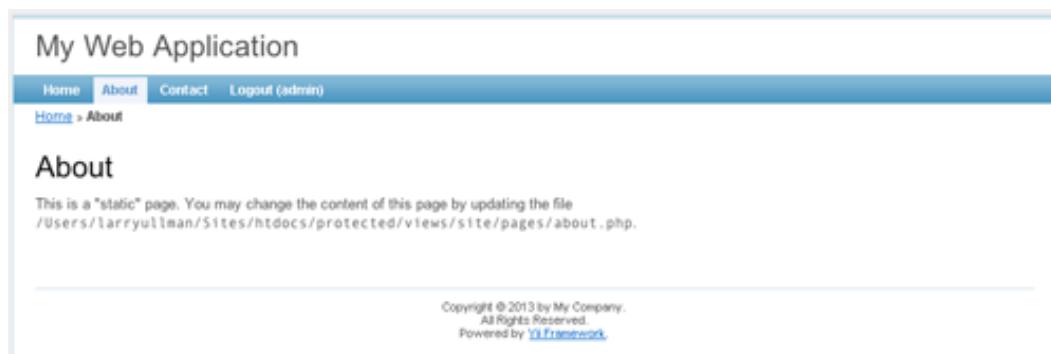


Figure 6.4: The page-specific content goes across the entire width of the browser.

But some pages should have a sidebar (**Figure 6.5**).

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`



Figure 6.5: The page-specific content shares the browser width with a sidebar.

- `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., it does not have properties that can 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'),
    'user'=>'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.
```

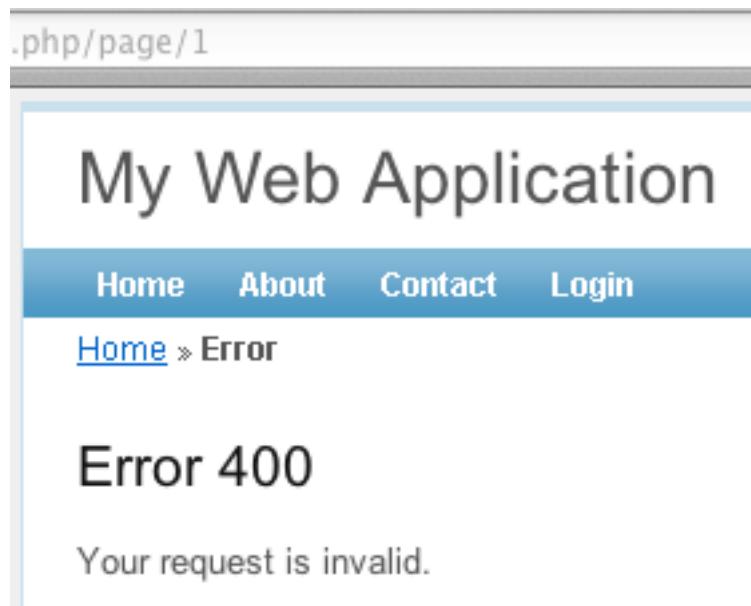


Figure 7.1: This exception is actually caused by a misnamed method parameter.

{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 route, 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, if 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.');
```

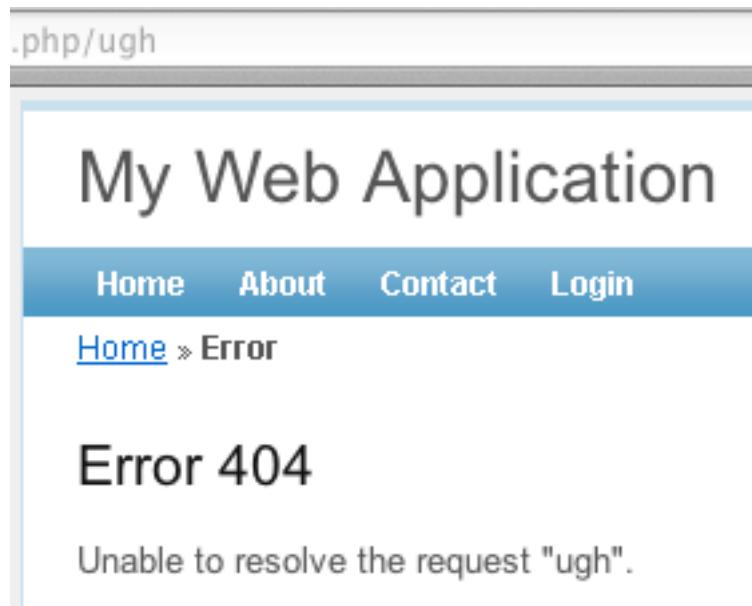


Figure 7.2: Because the "ugh" controller does not exist, a 404 exception is thrown.

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 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:

```
# protected/config/main.php "components" section
'db'=>array(
```

```
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.

```
'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:

```
$model = Page::model()->findPk($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**).

| find() method | | |
|---|---------------|---|
| <pre>public CActiveRecord find(mixed \$condition='', array \$params=array())</pre> | | |
| \$condition | 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 CDbCriteria object; Otherwise, it should be an instance of CDbCriteria . |
| \$params | 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 CDbCriteria::params to set parameters. |
| {return} | 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). Then 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 parameter array. 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 clause |

| | |
|--------|---|
| 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 `findById()` 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 `findById()` 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**).

findAll() method

| public array findAll(mixed \$condition='', array \$params=array()) | | |
|---|-------|---|
| \$condition | mixed | query condition or criteria. |
| \$params | array | parameters to be bound to an SQL statement. |
| {return} | 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()` return 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:

```
$model = Page::model()->findByPk($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:

```
'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.

```
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.

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:

```
$page = Page::model()->with(array('pageComments' =>  
    array('order'=>'date_entered')))->findPk($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:

```
$page = Page::model()->with(array('pageComments' =>  
    array('scopes'=>'recent')))->findPk($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:

```
$page = Page::model()->with(array(  
    'pageComments' => array('scopes'=>'recent',  
        'condition' => 'approved=1')))->findPk($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()`:

```
$page = Page::model()->scopeName()->with(array(  
    'pageComments' => array('scopes'=>'recentComments',  
        'condition' => 'approved=1')))->findPk($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:

```
$page = Page::model()->with(array(  
    'pageUser' => array('joinType' => 'INNER JOIN'),  
    'pageComments',  
    'pageComments.commentUser' =>  
        array('joinType' => 'INNER JOIN')))->findPk($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:

```
$page = Page::model()->with('commentCount')->findPk($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 |
|---------------------|---|
| <code>from</code> | The tables to be used |
| <code>join</code> | A JOIN |
| <code>limit</code> | A LIMIT clause without an offset |
| <code>offset</code> | A LIMIT clause with an offset |
| <code>order</code> | The ORDER BY clause |
| <code>select</code> | The ¹⁸⁰ columns to be selected |
| <code>where</code> | 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 existing 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 | | |
|---|--------|--|
| <pre>public static string label(string \$label, string \$for, array \$htmlOptions=array())</pre> | | |
| \$label | string | label text. Note, you should HTML-encode the text if needed. |
| \$for | 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. |
| \$htmlOptions | array | additional HTML attributes. The following HTML option is recognized: <ul style="list-style-type: none">• 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>. |
| {return} | 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 `CCController::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 controller (e.g., “search”) 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 ContactModel;
    $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 ContactModel;
    $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 ContactModel;
    $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 ContactModel;  
    $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 ContactModel;
    $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 delmmas.

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 screenshot shows a registration form with two password input fields. The first field is labeled "Password" and contains six red asterisks. Below it is an error message: "Password must be repeated exactly.". The second field is labeled "Pass Compare" and is currently empty.

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 18, "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).

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,
      'mimeTypes' => '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**).

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',
```

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.

```
'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**).

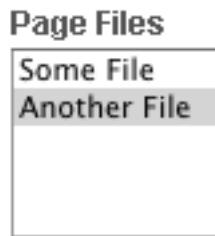


Figure 9.7: The file associated with this page is selected in the drop down box.

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 (:page_id, :file_id)";
    $cmd = Yii::app()->db->createCommand($q);
    foreach ($_POST['Page']['pageFiles'] as $file_id) {
        $cmd->bindParam(':page_id', $model->id,
                         PDO::PARAM_INT);
        $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:
`http://www.example.com/page.php?action=this`
- 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 performed, 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`
- `sessionName`, 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',
        'sessionName' => '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 sever, 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 TEXT,
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.

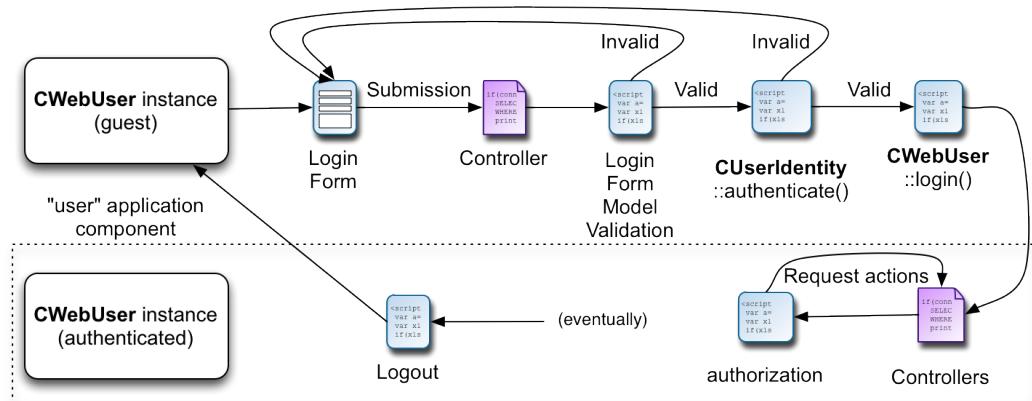


Figure 11.1: The authentication and authorization process.

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 4). 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.

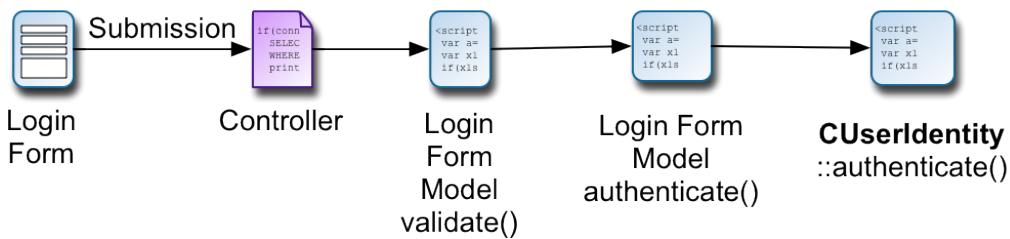


Figure 11.2: How the Yii app gets to the `UserIdentity::authenticate()` method.

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 |

ERROR_UNKNOWN_IDENTITY 100

If that first conditional is false—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 6). 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 8). 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()
$duratio=$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 `yiic`, 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>
```

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.

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 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', 'pass', '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>
```

My Web Application

[Home](#) [About](#) [Contact](#) [Logout \(text@example.com\)](#)

Hello, text@example.com!

Figure 11.4: Greeting the user by login name.

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 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).

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 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;
19     }
20     public function getId() {
21         return $this->_id;
22     }
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), 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($user->type) &&
        ($user->type=="author") ||
        ($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).

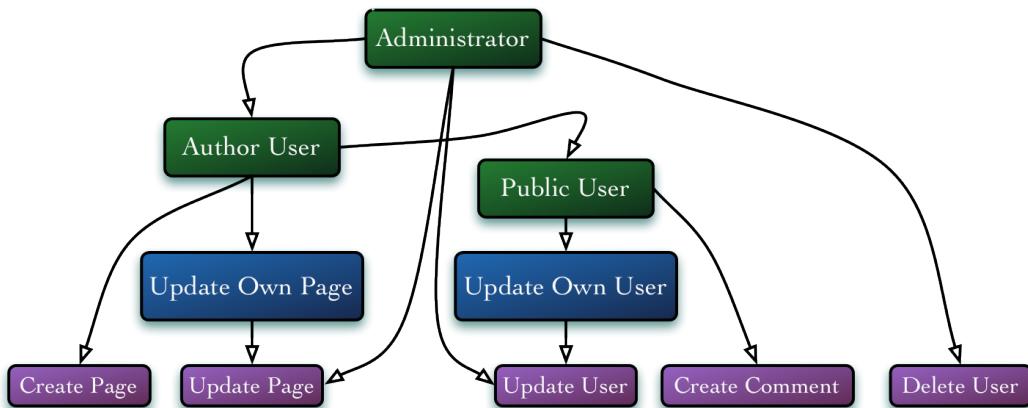


Figure 11.5: Some authorization items for the CMS example.

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 18, “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.

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. 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.

{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 an 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.

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() {
    $model=$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('oUpdatePage');
```

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() {
    $model=$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**).

The screenshot shows the documentation for the `CActiveForm` class. At the top, it says "CActiveForm" with a "View" button and a "23" link. Below that is a navigation bar with "All Packages", "Properties", and "Methods". A table provides detailed inheritance information:

| | |
|-------------|--|
| Package | <code>system.web.widgets</code> |
| Inheritance | class <code>CActiveForm</code> » <code>CWidget</code> » <code>CBaseController</code> » <code>CComponent</code> |
| Subclasses | <code>CCodeForm</code> |
| Since | 1.1.1 |
| Source Code | framework/web/widgets/CActiveForm.php |

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 Turning 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**).

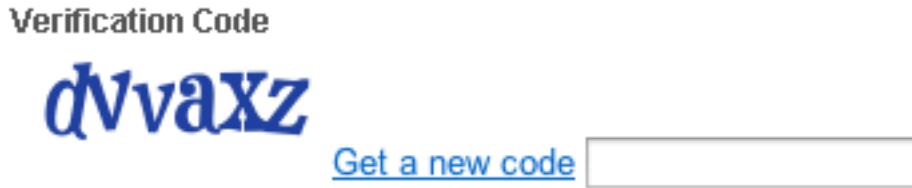


Figure 12.2: The CAPTCHA widget displays the image and creates the “Get a new code” link.

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 `yic` 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**):

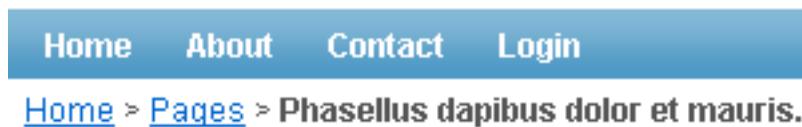


Figure 12.6: The same trail with a different separator.

```
# 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 sources 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} As using an array as a data source is less common when working with a database, so I won't discuss it in this chapter. But you'll see an example in Chapter 18, "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.

CListView

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 `CListView`. The `CListView` 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: Username ▾ Email |
| ID: | 5 | Username: Administrator Email: admin@example.com Password: 21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6 Type: admin Date Entered: 2013-02-17 15:59:24 |
| ID: | 6 | Username: Another Admin Email: admin2@example.net Password: 21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6 Type: admin Date Entered: 2013-02-10 15:59:24 |
| ID: | 4 | Username: Another Author Email: blah@example.org Password: 21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6 Type: author Date Entered: 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

| | |
|----------------|--|
| ID | 1 |
| User | 3 |
| Live | 1 |
| Title | Aliquam malesuada, ligula sit amet. |
| Content | Lore 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. |

Figure 12.8: The view display for a single Page record.

Unlike `CListView` and `CGGridView`, `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_entered',  
        '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. Read hiphound's laundry account autem Nonnulla. |

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_entered',  
        'date_published',  
    ),  
) ; ?>
```

View Page #1

| | |
|----------|--|
| ID | 1 |
| Username | Some Author |
| Live | Yes |
| Title | Aliquam malesuada, ligula sit amet. |
| Content | 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. |

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_entered',
'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'=gt;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_entered',
        '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**).

The CGridView presents a series of records in a table and provides the following functionality:

| Displaying 1-10 of 16 results. | | | | | | |
|--------------------------------|-------------|------------------|--|--|--------|---------------------|
| ID | Username | Email | | Password | Type | Date Entered |
| 1 | Test | text@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 16:59:24 |

Figure 12.11: The admin grid of users.

- 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 submissions of the grid form via Ajax uses 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 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 `CGGridView` 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_entered',
        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.

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 `emptyDisplay` values:

| 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.

```
# protected/views/page/admin.php
<?php $this->widget('zii.widgets.grid.CGridView', array(
    'id'=>'page-grid',
    'dataProvider'=>$model->search(),
    'filter'=>$model,
    'nullDisplay' => 'N/A',
    'emptyDisplay' => '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/view.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 GridView with three columns: 'Email', 'type', and 'date_published'. A dropdown menu is open over the second column, showing options '(Select)', 'Public', 'Author' (which is selected), and 'Admin'. The grid displays three rows of data:

| Email | Type | Date Published |
|-------------|--------|----------------|
| example.net | Author | 2013 |
| example.org | Author | 2013 |
| example.org | Author | 2013 |

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**.

| Username | (select) | |
|----------------|----------|---------|
| 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**).

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

| 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.

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.

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

| 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.

- 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**).

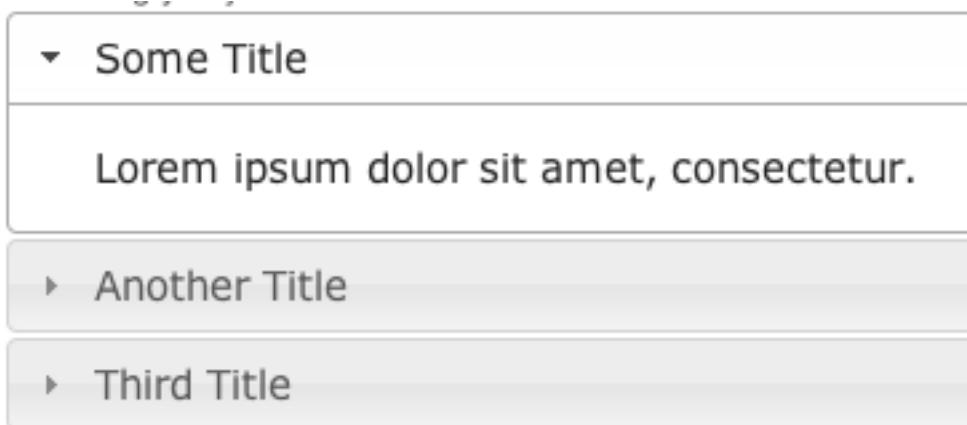


Figure 12.17: A *jQuery UI accordion*.

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.'
    ),
    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 21, “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:

- 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.clevertech.biz/index.html>. Then expand the downloaded file to create a folder called something like *clevertech-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**).

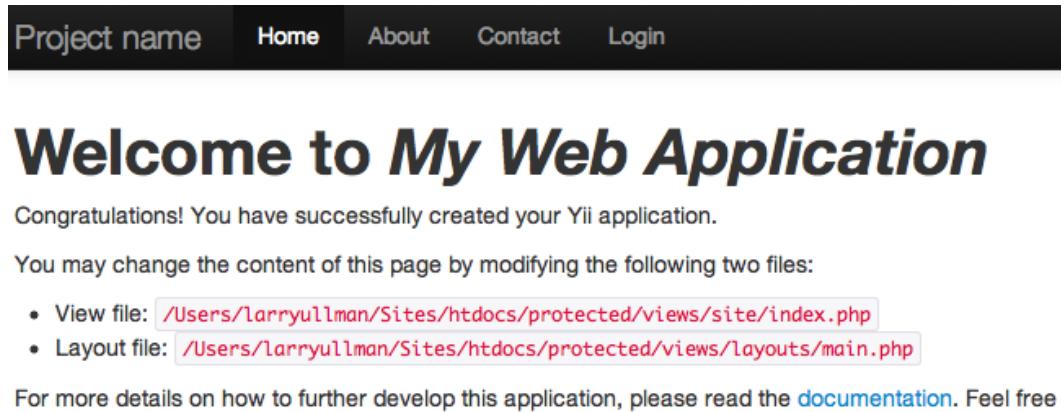


Figure 13.1: The Bootstrap version of the basic site design.

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:

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}',
),
),
));

```

| 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.

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
- 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 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**).

| 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.

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**).

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

View Page Aliquam malesuada, ligula

| | |
|-------|-------------------------------------|
| ID | 1 |
| User | Some Author |
| 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.

(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](#)).

Payment

Credit Card Number *

4485587139555400

Credit Card Number is not a valid Credit Card number.

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**).

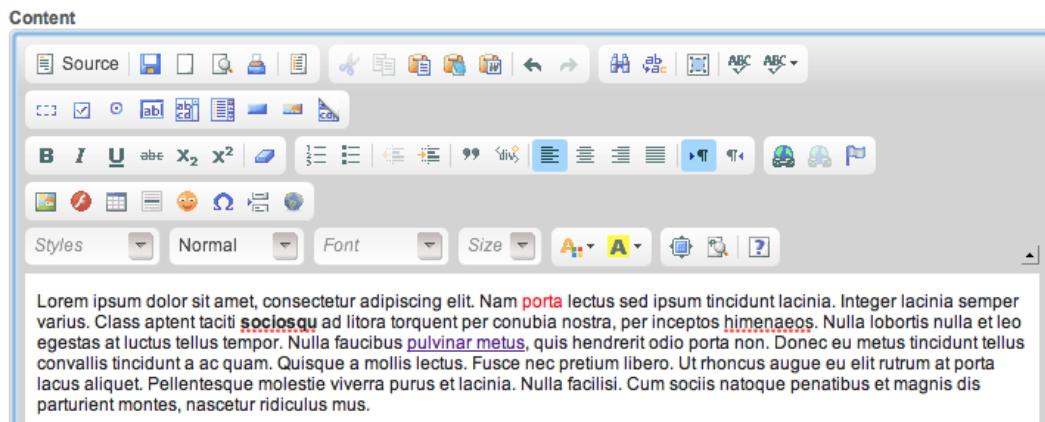


Figure 13.9: The CKEditor instance used to create and edit a Page record.

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'=gt;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.

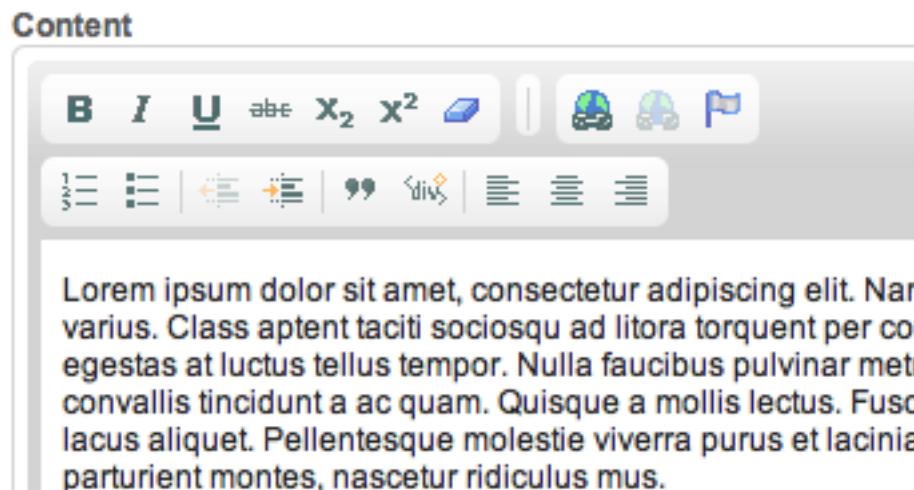


Figure 13.10: A customized toolbar.

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:

```
# 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::app()->getClientScript()->registerScript()` 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.yiGridView.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 `yiGridView.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.
- CClientScript::POS_END, at the end of the BODY.
- CClientScript::POS_LOAD, within a window.onload event handler
- CClientScript::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 CClientScript::POS_READY. It’s slightly faster than the standard JavaScript window.onload option. If you’re not using jQuery, then use CClientScript::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.

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:

```
{"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.

```
# 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'=gt;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 may 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>
```

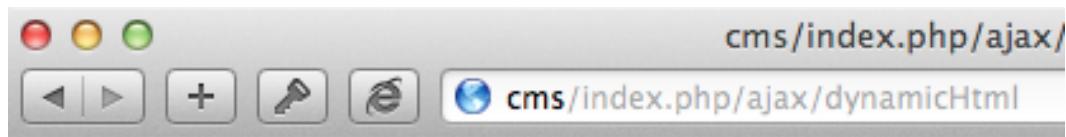
The received `$data` array's pieces are placed within a context of HTML (**Figure 14.3**). Changing the data values in the controller therefore changes the output.

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:



Dynamic!

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 animus.

Figure 14.3: The “dynamic” HTML response.

- 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**).

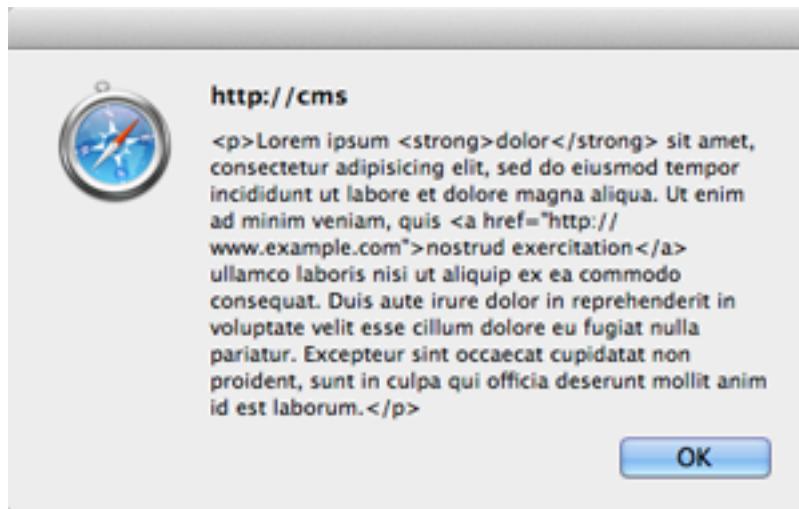


Figure 14.4: The response from the HTML action.

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

| | | |
|---|--------|---|
| <code>public static string ajaxButton(string \$label, mixed \$url, array \$ajaxOptions=array(), array \$htmlOptions=array())</code> | | |
| <code>\$label</code> | string | the button label |
| <code>\$url</code> | mixed | the URL for the AJAX request. If empty, it is assumed to be the current URL. See <code>normalizeUrl</code> for more details. |
| <code>\$ajaxOptions</code> | array | AJAX options (see <code>ajax</code>) |
| <code>\$htmlOptions</code> | array | additional HTML attributes. Besides normal HTML attributes, a few special attributes are also recognized (see <code>clientChange</code> and <code>tag</code> for more details.) |
| <code>{return}</code> | 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()`.

Testing Ajax

[Get Content](#)

Figure 14.6: The page when the user first sees it.

Testing Ajax

Dynamic!

Consectetur

Get Content

Figure 14.7: The same page after the user has clicked the button.

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 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) {
9             $('#selectedTitle').text(ui.item.value);
```

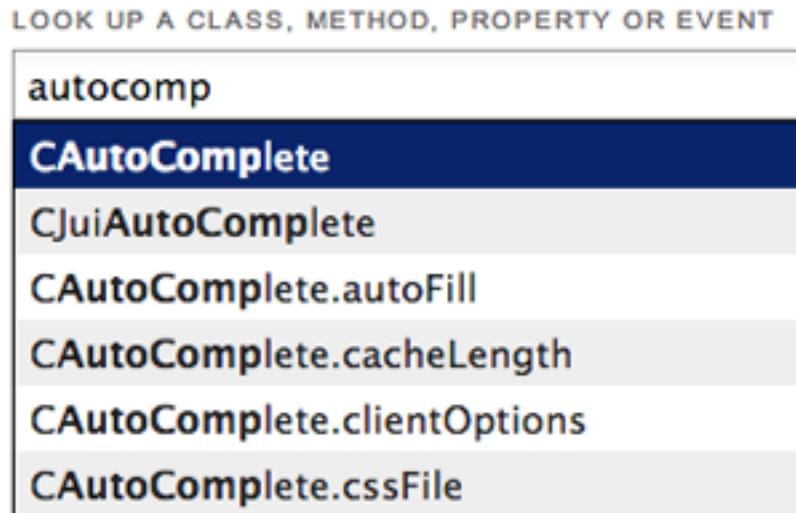


Figure 14.10: Autocomplete functionality in the Yii class reference.

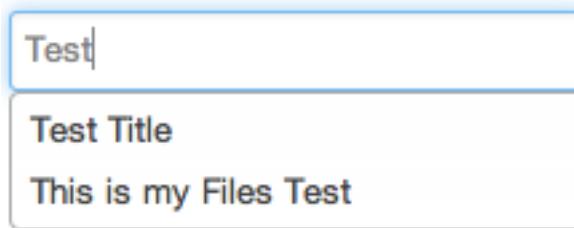


Figure 14.11: Autocompletion of page titles.

```
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 provided 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 `CGGridView` 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.