The Yii Book

Second Edition

Developing Web Applications
Using the Yii PHP Framework

Larry Ullman

The Yii Book (2nd ed.) by Larry Ullman

Self-published

Find this book on the web at larry.pub.

Revision: 1.3

Technical Reviewer: Qiang Xue

Technical Reviewer: Alexander Makarov

Cover design very kindly provided by Paul Wilcox.

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

This is the 26th book that I've written, and of the many things I've learned in that time, a reliable fact is readers rarely read the introduction. Still, I put some thought into the introduction and recommend you 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, dear reader. And if that was not enough, the introduction concludes by providing you with resources 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–or more commonly, team–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 is an expectation of any professional programmer today.

Why You Should Use a Framework

The most obvious argument for using a framework is being able to develop projects much, much faster than if you don't use a framework. But there are other, more critical, arguments, too.

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

Similarly, a framework is likely to provide professional features you might not take the time to implement otherwise, such as logging, caching, and exception handling. 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 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 learning a framework is so 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 route 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? In fact, I didn't even use the Yii framework for larry.pub, the site on which I sell this book. What gives?

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 to that point.

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 serious 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 website 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 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 for 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 started by Qiang Xue and first released in 2008. "Yii" is pronounced like "Yee", and can be thought of as 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 "simple and evolutionary".

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 2.0.4.

What Yii Has to Offer

Being a framework, Yii offers all the strengths and weaknesses that frameworks in general have to offer. 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.

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 all of these:

- 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, performs quite well. Yii applications will scale nicely, 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 just the same. 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. I also love that Yii auto-generates *a ton* of code and directories for you, a feature that I had come to be spoiled by when using Rails. Yii is 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 and 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 care for the Zend Framework as the basis of an entire site. The Zend Framework 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:

- Zurmo, a Customer Relationship Management (CRM) system
- X2EngineCRM, another CRM
- LimeSurvey2, a surveying application

What's New in Yii 2?

In late 2014, the long-awaited release of Yii 2 came out. The first major change in Yii 2 is that it requires at least PHP 5.4. By upping the version of PHP required, Yii takes advantage of new features in PHP:

- Namespaces
- Anonymous functions
- Standard PHP Library (SPL)
- Date and time classes
- Traits
- Internationalization
- Short array syntax
- Short echo tags

In adopting these new features, the team behind Yii performed a complete rewrite of the framework. It was quite an accomplishment, and the results are fantastic. In the process, Yii 2 adds:

- Use of Composer for installation
- Smarter, better performing, core classes
- An amazing debugging tool
- Top-notch security implementations
- Revised Active Record models

- Support for non-relational database applications
- Built-in RESTful API generation
- PSR-4 autoloading
- Twitter Bootstrap out of the box
- Better console applications
- And more!

This edition of the book only uses Yii 2, although it will highlight the changes from Yii 1 with the expectation that you may have used the earlier version. You probably don't want to upgrade any existing sites from Yii 1 to Yii 2, but if you're curious, see the Yii upgrade guide.

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. Version 1 of the Yii framework requires PHP 5.1 or greater; but version 2 requires PHP 5.4. This means that this book requires that you're using at least version 5.4 of PHP. At the time of this writing, the latest version of PHP is 5.6.9.

This book will assume you're using Apache as your web server application, although it's fine if you're using nginx. If you're not using Apache, you'll just need to see the Yii documentation or search online for alternative solutions when Apache-specific options are presented.

> *{NOTE}* In my opinion, it's imperative that 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 easily let you 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 websites. And it's all free! If you have not yet installed an *AMP stack–Apache, MySQL, and PHP–on your computer, I would recommend you do so now. The most popular solutions are:

- XAMPP on Windows
- EasyPHP on Windows
- BitNami on Windows, Linux, or Mac OS X
- Zend Server on Windows, Linux, or Mac OS X
- AMPPS on Windows, Linux, or Mac OS X

All of these are free.

Going further, you can use Vagrant to install and run entire virtual machines from your computer (e.g., to emulate the operating system of your preferred hosting company).

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 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.

> *{TIP}* If you're using an IDE, also search online for tutorials on using Yii with that specific IDE.

In case you're curious, I almost exclusively use a Mac, and currently use the excellent Sublime Text text editor for most things. I occasionally play around with PhpStorm, which is highly regarded, but I'm not much of an IDE person.

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 a conventional development approach to 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 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 website 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. In particular, the book avoids covering anything too 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 about any omissions I made, should you ever have those needs.

Formatting Conventions

This book has several formatting conventions. They should be obvious, but just in case, I'll lay them out explicitly here.

Code font is 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, you'll see the name of the file in which that code would be found, including the directory structure:

```
# views/layouts/main.php
// This is the code.
```

Sometimes references also indicate the name of the function in the file where the code would be placed:

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

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

Within the text, URLs, directories, and file names are in **bold**. References to specific classes, methods, and variables are in code font–`SomeClass`, `someMethod()`, and `$someVar`–except in notes, tips, and warnings. References to array indexes, component names, and informal but meaningful terms are 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 and Snag It.

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 26th 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 (the fifth editions should be out in 2016). 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've also been an active developer. Most of the websites I've done are for educational and non-profit organizations, but I've also consulted on commercial projects. I would estimate that I used a framework on maybe 60% of the sites I worked 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.

At the time of this writing, I work for Stripe. I'm currently on the Product team, managing Stripe's user-facing documentation. Previous to this, I was a Support Engineer, assisting Stripe's users with their integrations.

My website is LarryUllman.com. This book's specific set of pages is at larry.pub. 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
- The #yii IRC channel on the Freenode network
- The Yii section at Stack Overflow
- My support forums

If you don't have an IRC client or haven't used IRC before, the Yii website graciously provides a web-based interface.

When you need help, you should always start with 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.

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. 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 or ignore the request entirely. If you email me, I will reply, but it's highly likely that it will take two weeks for me to reply, or more. 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.

Part I

Getting Started

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 perspectives. 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. This first part of the chapter walks 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 via 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 from 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 OOP, the focus is on the *things* (i.e., the nouns). Thus, to write object-oriented code, you start by analyzing and defining with what types of things the application will work.

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 webpage 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 OOP and good class definitions make for projects that are reliable and easy to maintain. When implementing OOP, 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 a philosophy. Whereas procedural programming almost writes itself in terms of a logical flow, proper OOP requires a good amount of theory and design. Bad procedural programming tends not to work well, but can be easily remedied; bad OOP 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.

Still, 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 OOP.

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

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

There are three levels of visibility:

- Public
- Protected
- Private

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

An attribute or method defined as *public* can be accessed anywhere within the class, within 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 child classes, but not through object instances. An attribute or method defined as *private*

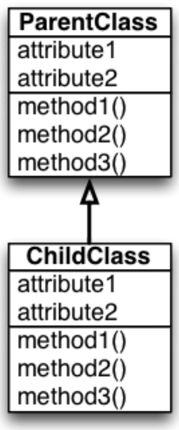


Figure 1.1: *The child class can inherit members from the parent class.*

can only be accessed within the class itself, not within child classes or through object instances.

Because OOP 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 (i.e., children). The child class inherits all the public and protected members from the base class, and can then add its own new members. 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, whereas `Child` could also inherit from `Person` but add a `play()` method (**Figure 1.2**).

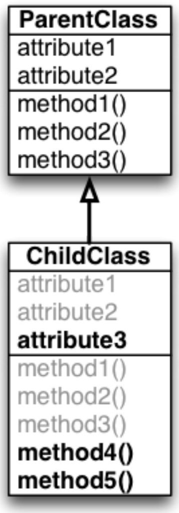


Figure 1.2: *The child class can add new members to the ones it inherited.*

Inheritance can be extended to such a degree that you have multiple generations of inheritance: 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 does different things depending upon the context in which it is called.

OOP Syntax

With sufficient terminology and theory explained, let's look at OOP syntax in PHP. The first 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 use an underscore as the first character. These conventions are not required, although they are the ones I use in this book. By far, the most important consideration is that you are consistent in applying the conventions you prefer.

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 object attributes are referenced through the object *without* using the dollar sign in front of the attribute's name. As in the code above, you *don't* write `echo $obj->$var1`.

Within the class, attributes and methods are accessible via the special `$this` object. `$this` always refers to the current instance of the 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 internally 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 extend for individual purposes.

Finally, there's the syntax of, and philosophy behind, *namespaces*. Namespaces were added to PHP in version 5.3 and are used extensively in Yii. In layman's terms, namespaces allow you to group code under tags much as you organize files on your computer within nested folders. Using namespaces, you can create better references to code and prevent naming *collisions* when using code from multiple sources: a third-party library's `User` class can be kept distinct from my site's `User` class.

Code is assigned to a namespace using the `namespace` keyword:

```
namespace myspace;
class SomeClass {
}
```

Now, `SomeClass` is declared to be within the `myspace` namespace.

Backslashes are used to reflect namespace hierarchy:

```
namespace myspace\user;
class SomeClass {
}
```

Now, `SomeClass` is declared within the `myspace\user` namespace:

```
$obj = new \myspace\user\SomeClass();
```

In order for other code to use `SomeClass`, it must refer to `myspace\user\SomeClass` as in the above. Or namespaced-code can be brought into the current environment via the `use` keyword:

```
use myspace\user;
$obj = new SomeClass();
```

Note that a `use` statement must be placed outside of any class definition:

```
use myspace\user;
class Blah extends SomeClass{
}
```

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 adopted by many languages and particularly by frameworks.

The basic MVC concept is relatively simple to understand, but 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 design 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. You can use MVC without a framework, but many 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

An application will almost always have multiple models, views, and controllers.

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. If you have two related tables, that scenario would be represented by two separate models. 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 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, the submitted data won't be needed after it's emailed. Still that data must be represented by a model up until it's emailed in order to perform validation and any other business logic.

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 a string 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. 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 that adds the unique content for a given page. Such PHP code should only perform 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 file, 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 reasons. First, it allows you to easily change the presentation–the HTML in a web page–without wading through a ton of PHP code. 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, more complex, files.

A second benefit of separating data from presentation is that doing so lets you use the same data in different outputs. In today's websites, data is not only displayed in a web browser, it's also sent in an email, included as part of a web service, accessed via a console 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 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: 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.

To put this all within a context, a user goes to a URL like **example.com/page/1/**. 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). 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.

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. Each of these files defines 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) constitute that class and define how it behaves. The class's attributes 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 email content. A model class's methods serves 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. Thus, 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
- Showing just one page of content

The final component 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 establishes the overall template: beginning the HTML, including the CSS file, creating headers, navigation, and footers, and completing the HTML. The contents of the individual view files are placed within the greater context of these layout files. This way, changing one thing for the entire site requires editing only a single 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 and 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, 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 (e.g., to show a specific page of content).
2. The controller validates the page ID passed in the URL.
3. The framework 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

Beginners to MVC can easily make the mistake of putting code in the wrong place–for example, in a controller instead of a model. To help you avoid that, let's identify some signs of trouble. 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 logic in the views. The goal for a view is to combine the data and the presentation–normally 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").

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 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.

> *{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 on any host, but I would recommend that you begin projects on a development server and only move them to a production server once they are fairly complete. One reason why 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. Similarly, errors will undoubtedly arise during development, errors that should never be shown on a live site.

The third reason to hold off using a production server is performance. Useful debugging tools, such as Xdebug should not be enabled on live sites, but are 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, 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 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.

The Web Root Directory

Whether you're working on a production or development server, you need to be familiar with the *web root directory*. This is the folder on the machine 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 the URL **http://localhost:8080/somepage.php** equates to **C:\xampp\htdocs\somepage.php**. If you're using MAMP on Mac OS X, the default web root directory is **/Applications/MAMP/htdocs**.

This book 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 using the command-line interface. The command-line interface is something with which every web developer should be comfortable, 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
- Use Git
- And more

But even if you don't expect to do any of those things yourself, in order to create a new website using Yii, you'll need to use the command line once: to create the initial shell of the site. There are three command-line skills you must have:

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. On Windows XP and earlier, this was accomplished by clicking Start > Run, and then entering `cmd` within the prompt (**Figure 1.3**).

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 for book images.

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

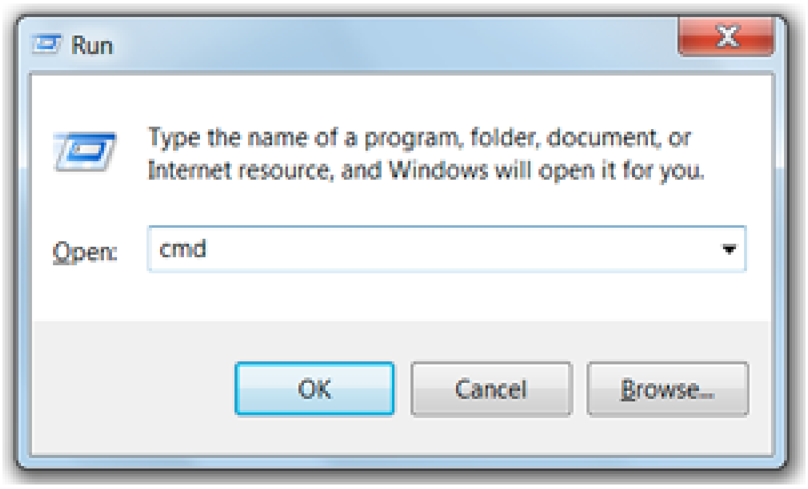
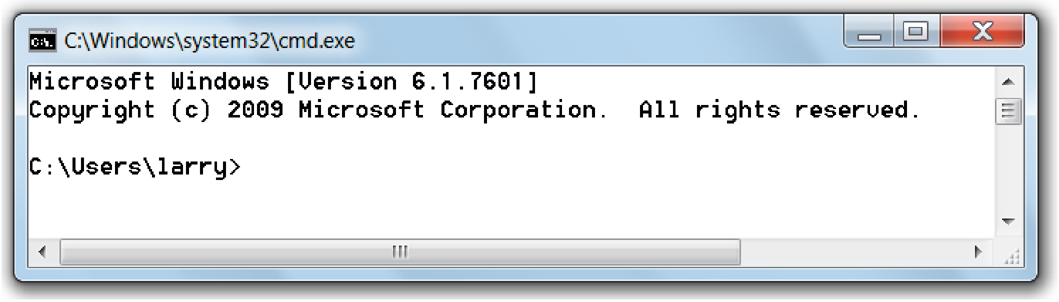

Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
cmd
OK
Cancel
Browse...

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



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\larry>
```

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

On Mac OS X, a command-line interface is provided by the Terminal application, found 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

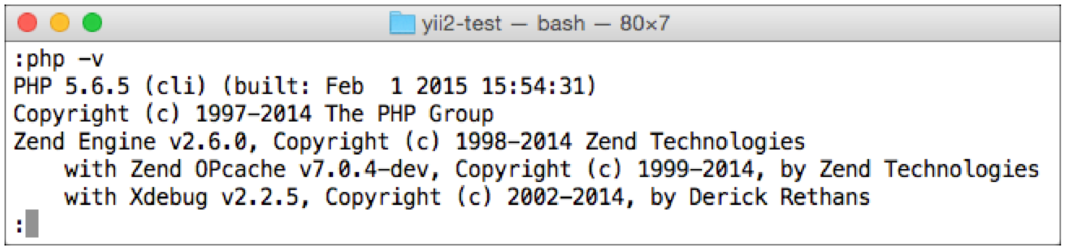
Once you've got a command-line interface, what can you do? Thousands of things, of course, but most importantly for the sake of this book: invoke PHP.

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 console applications for your Yii site.

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.



```
:php -v
PHP 5.6.5 (cli) (built: Feb  1 2015 15:54:31)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
    with Zend OPcache v7.0.4-dev, Copyright (c) 1999-2014, by Zend Technologies
    with Xdebug v2.2.5, Copyright (c) 2002-2014, by Derick Rethans
:
```

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

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*.

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. 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 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 reflect the path change. (Any existing console windows will still complain about PHP not being found.)

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

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 create a new web application using the Yii framework. In just a couple of pages you’ll 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 and install the shell of a site.
2. Test what you’ve created thus far.
3. Confirm that your server meets the minimum requirements to use Yii.

These are generic, but static steps, to be taken with each new website you create. In Chapter 4, “Initial Customizations and Code Generations,” you’ll have Yii create code more specific to an individual application.

Working with Composer

Many PHP frameworks and libraries these days are installed using Composer, and Yii 2 has adopted it as the preferred installation tool, too.

> *{NEW}* Yii 2 creates new sites using Composer, instead of via a command-line script that comes with the framework.

Composer is a *dependency manager* for PHP. You can define the requirements for a project, run a command, and Composer will ensure those requirements are met, downloading and installing the necessary packages. Composer is not hard to learn and use, and you can't be a modern PHP programmer without it. If you haven't picked Composer up yet, there's no time like the present. Especially as you'll use it to create a new Yii site!

Installing Composer

If you haven't yet used Composer, you'll need to install it on your computer. If you're already familiar with Composer, skip these next instructions.

> *{NOTE}* You need to install Composer only once on each computer, not once for each site.

On Windows, you just need to download and run the Composer installer, as explained in the Composer docs.

To install Composer on Mac OS X and *nix:

1. Access your computer from a command-line interface.
2. Move to a logical destination directory for Composer:

   ```
   cd /path/to/dir
   ```

   You can install Composer anywhere, and you don't want to install it within any specific project's directory. On Mac OS X, I might install Composer in my **Sites** folder.

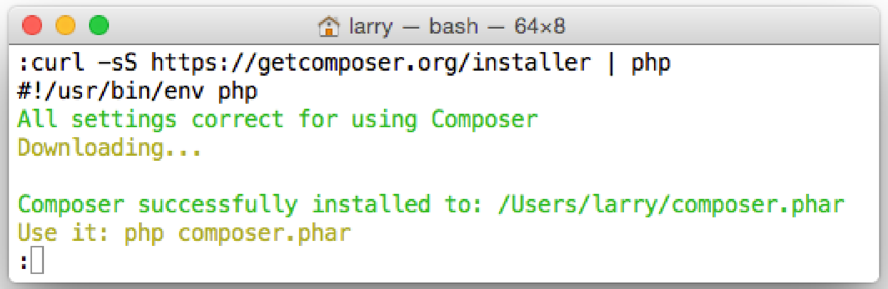
3. Execute the following command (**Figure 2.1**):

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

   That line uses cURL to download the Composer installer, and then uses your local version of PHP to run the installer. If you get an error message about not being able to find or recognize PHP, you'll need to change the end of that command to include the full path to your PHP executable (or add the PHP executable to your path).

Those steps take care of the installation of Composer. You will find the file **composer.phar** in the folder you used (in Step 2). That script will do the installation work.

> *{TIP}* You probably want to make Composer a globally usable tool. Search online for instructions on doing so.



```
:curl -sS https://getcomposer.org/installer | php
#!/usr/bin/env php
All settings correct for using Composer
Downloading...

Composer successfully installed to: /Users/larry/composer.phar
Use it: php composer.phar
:
```

Figure 2.1: *Installing Composer.*

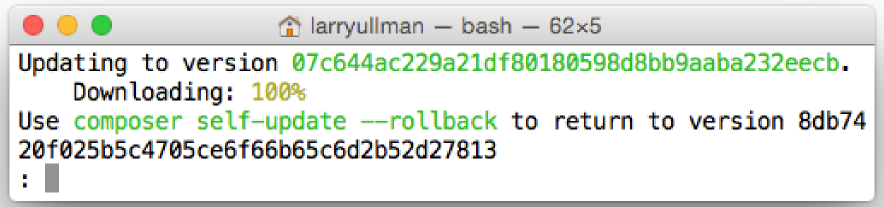
Updating Composer

If you've previously installed Composer, execute this command to update it:

```
php /path/to/composer.phar self-update
```

If you created a global Composer installation, or are using Windows, you can run (**Figure 2.2**):

```
composer self-update
```



```
Updating to version 07c644ac229a21df80180598d8bb9aaba232eecb.
    Downloading: 100%
Use composer self-update --rollback to return to version 8db74
20f025b5c4705ce6f66b65c6d2b52d27813
:
```

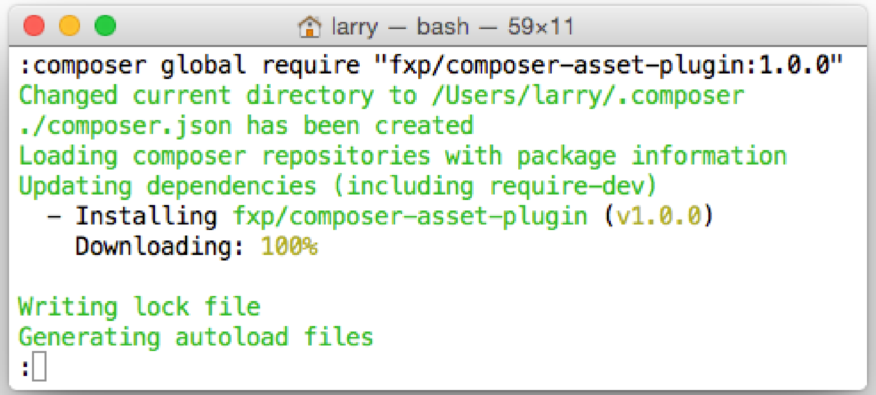
Figure 2.2: *Updating Composer.*

Composer itself will warn you if it hasn't been updated in more than a month.

Installing the Asset Manager

The Yii framework makes use of the Composer asset plugin for installing dependent libraries. You need to install this plugin once on your computer. Do so by executing this line in a command-line interface (**Figure 2.3**):

```
composer global require "fxp/composer-asset-plugin:1.0.0"
```



larry — bash — 59×11

```
:composer global require "fxp/composer-asset-plugin:1.0.0"
Changed current directory to /Users/larry/.composer
./composer.json has been created
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing fxp/composer-asset-plugin (v1.0.0)
    Downloading: 100%

Writing lock file
Generating autoload files
:
```

Figure 2.3: *Installing the Composer asset plugin.*

That command does assume Composer was installed globally, or you're using Windows. You only need to install the Composer asset plugin with your Composer installation once; not once per Yii site you create.

With Composer and the Composer asset plugin installed, you can now create your first Yii application!

Creating a Yii Application

Having setup Composer, you can now create your first Yii-based site. For the most part, Composer is used to install an application's dependencies (which you'll also see later in the book), but Composer has a `create-project` command for making an entirely new project using an existing template.

Version 2 of the Yii framework defines two templates for you:

- Basic

- Advanced

Both templates include a site structure and the Yii framework itself. The advanced template is differently organized than the basic, with a separate front-end and back-end. The advanced template also defines a `User` model for you, with user creation, password restoration, etc., defined.

> *{NEW}* In Yii 2, the framework is installed and maintained as part of the application itself, not separately, as was the case in Yii 1.

In the future, other Yii developers may make application templates available, too. And, in Chapter 19, "Extending Yii," you'll learn how to create your own application template. But for most of the book, the basic Yii template will be used. It's installed via this command:

```
composer create-project
    --prefer-dist yiisoft/yii2-app-basic
    /path/to/dir
```

(I've spaced this command out over several lines for clarity, but you should execute it as one line. That command does assume Composer was installed globally, or you're using Windows.)

The `--prefer-dist` parameter says that "dist"-quality packages are to be used, if possible. The specific template being referenced is "yiisoft/yii2-app-basic", found on GitHub. Finally, the "/path/to/dir" is how you specify where you want the application installed.

On my Mac, I'd likely create new sites within my **Sites** folder (**Figure 2.4**):

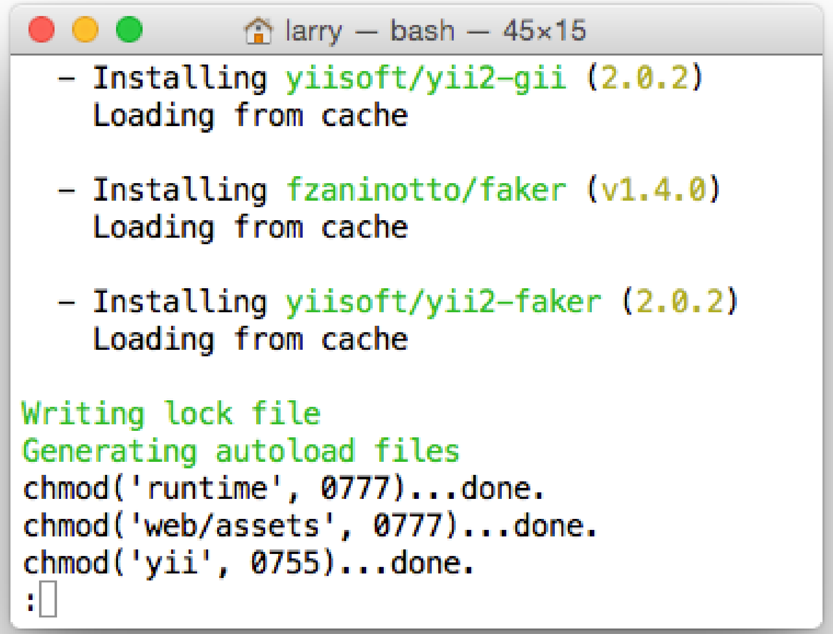
```
composer create-project
    --prefer-dist yiisoft/yii2-app-basic
    ~/Sites/yii2-test
```

Testing the Site Shell

Unless you saw an error message when you created the project via Composer, 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. You'll need to specifically head to **web/index.php** (**Figure 2.5**).

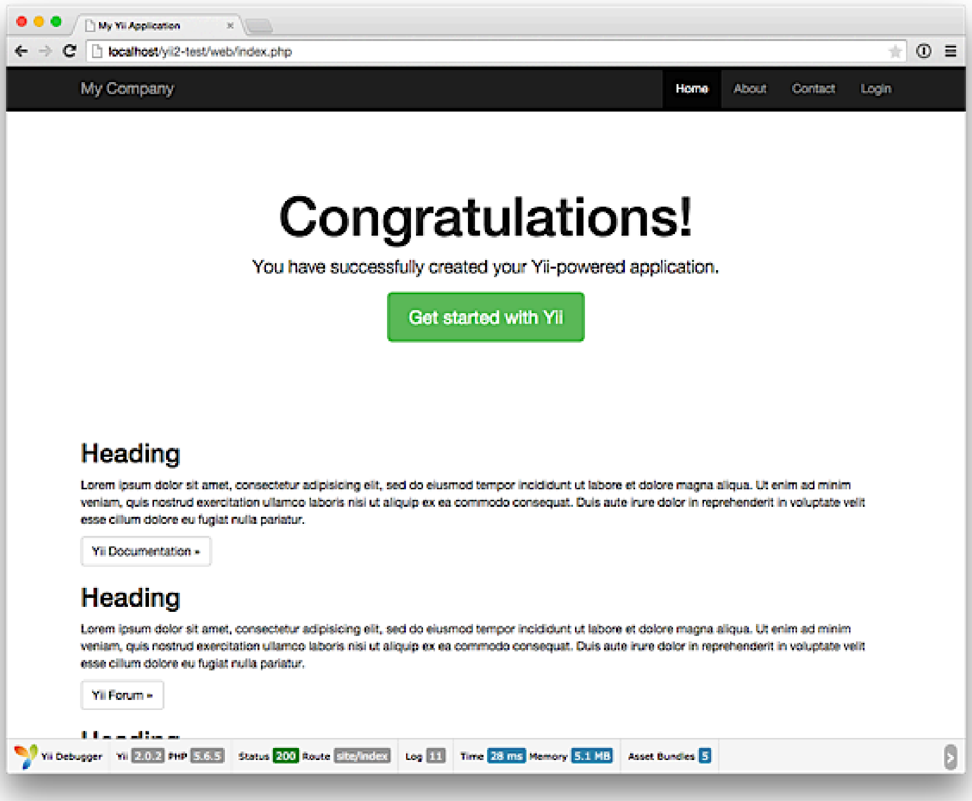
As for functionality, the generated application already includes:

- A home page (see Figure 2.5)



```
  - Installing yiisoft/yii2-gii (2.0.2)
    Loading from cache

  - Installing fzaninotto/faker (v1.4.0)
    Loading from cache

  - Installing yiisoft/yii2-faker (2.0.2)
    Loading from cache

Writing lock file
Generating autoload files
chmod('runtime', 0777)...done.
chmod('web/assets', 0777)...done.
chmod('yii', 0755)...done.
:
```

Figure 2.4: *Creating a Yii 2 application.*



Congratulations!

You have successfully created your Yii-powered application.

Heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Figure 2.5: *The shell of the generated site.*

- An about page
- A contact form, complete with CAPTCHA
- A login form
- The ability to greet a logged-in user by name
- Logout functionality
- Twitter Bootstrap
- An amazing debugger

> *{NEW}* Yii 2 uses Twitter Bootstrap by default.

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–only send an email– once you've edited the configuration to provide your email address (assuming you have a working mail server, too). For the login, you can use either demo/demo or admin/admin.

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

Testing the Requirements

One thing I like about Yii is that it includes a PHP script that tests whether or not your setup meets the minimum requirements for using the framework. The script itself is **requirements.php**, found in the application directory just created. You'll want to execute it using both a browser and the command-line PHP before developing your site further.

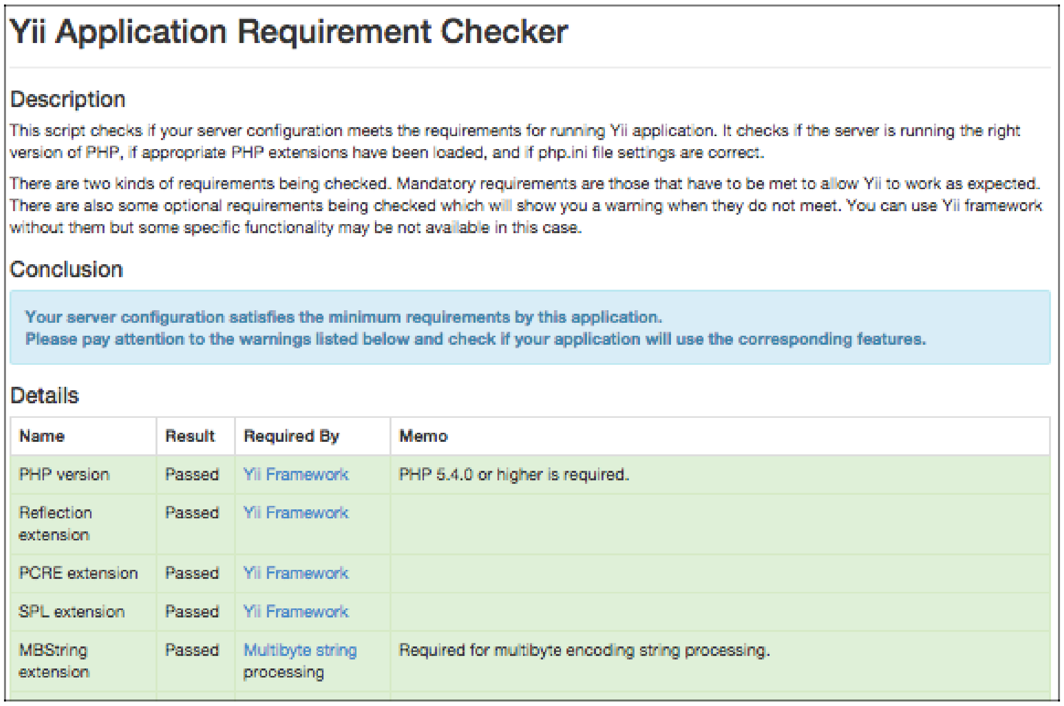
Execute the script through the browser first to test your web server:

1. Load *yourURL/requirements.php* in your web browser.

   For example, go to **http://localhost/requirements.php**.

1. Examine the output to confirm your setup meets the minimum requirements (**Figure 2.6**).
2. If your server *does not* meet the minimum requirements, reconfigure the server, install the necessary components, etc., and retest until your setup does meet the requirements.

Because you'll sometimes use the command-line to execute PHP scripts, you should also run **requirements.php** through that interface. The command-line version of PHP may differ from the web version, leading to odd and difficult-to-debug errors later on if you haven't checked the requirements.



Yii Application Requirement Checker

Description

This script checks if your server configuration meets the requirements for running Yii application. 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.

There are two kinds of requirements being checked. Mandatory requirements are those that have to be met to allow Yii to work as expected. There are also some optional requirements being checked which will show you a warning when they do not meet. You can use Yii framework without them but some specific functionality may be not available in this case.

Conclusion

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

Details

Name	Result	Required By	Memo
PHP version	Passed	Yii Framework	PHP 5.4.0 or higher is required.
Reflection extension	Passed	Yii Framework	
PCRE extension	Passed	Yii Framework	
SPL extension	Passed	Yii Framework	
MBString extension	Passed	Multibyte string processing	Required for multibyte encoding string processing.

Figure 2.6: *This setup meets Yii's minimum requirements.*

1. Access your computer from a command-line interface.
2. Move to the directory just created:

```
cd /path/to/application/dir
```

1. Execute the following command:

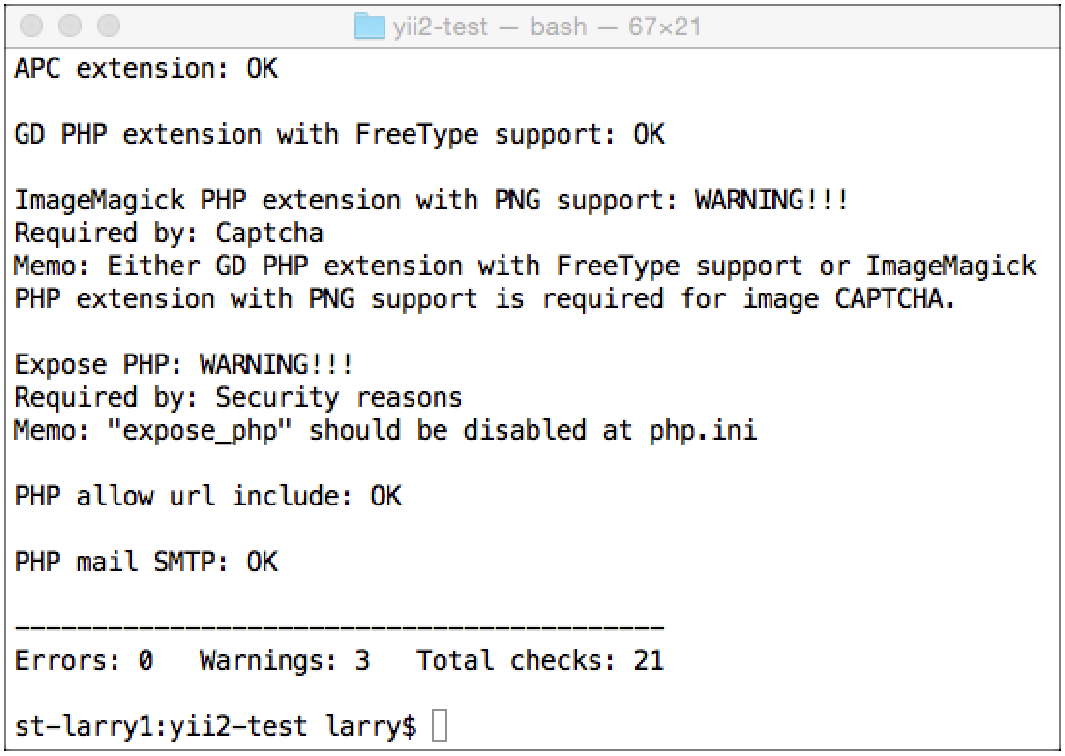
```
php requirements.php
```

1. Examine the output to confirm your command-line setup meets the minimum requirements (**Figure 2.7**).

1. If your command-line PHP version *does not* meet the minimum requirements, reconfigure the command-line version, install the necessary components, etc., and retest until your command-line setup also meets the requirements.

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 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



```
APC extension: OK

GD PHP extension with FreeType support: OK

ImageMagick PHP extension with PNG support: WARNING!!!
Required by: Captcha
Memo: Either GD PHP extension with FreeType support or ImageMagick
PHP extension with PNG support is required for image CAPTCHA.

Expose PHP: WARNING!!!
Required by: Security reasons
Memo: "expose_php" should be disabled at php.ini

PHP allow url include: OK

PHP mail SMTP: OK

------------------------------------------
Errors: 0   Warnings: 3   Total checks: 21

st-larry1:yii2-test larry$ 
```

Figure 2.7: *The command-line PHP also meets Yii's minimum requirements.*

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 websites database-agnostic.) The other things being checked may or may not be required, depending upon the needs of the actual site you're creating.

Chapter 3

A MANUAL FOR YOUR YII SITE

Now that you've generated the basic shell of a Yii-based site, it's time to go through exactly what you have in terms of actual files and directories. This chapter, then, is a manual for your Yii-based web application. 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 will go a long way towards helping you successfully and easily use the Yii framework.

The Site's Folders

Composer downloaded a template of a site, including several folders, dozens of files, and the framework itself. Knowing how to use the Yii framework begins with familiarizing yourself with the site structure.

> *{NEW}* Yii 2 does away with the **protected** folder, instead placing everything within the root application directory.

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 to the web server
- **codeception.yml**, for configuring unit testing
- **commands**, for command-line uses of your application
- **composer.json** and **composer.lock**, used by Composer
- **config**, stores your application's configuration files
- **controllers**, where your application's controller classes go

- **mail**, stores HTML templates for emails to be sent
- **models**, where your application's model classes go
- **requirements.php**, used in the previous chapter to test if Yii's requirements are met by the system
- **runtime**, where Yii will create temporary files, generate logs, and so forth
- **tests**, where you'll put unit tests
- **vendor**, for third-party software
- **views**, for storing all the view files used by the application
- **web**, the web root directory
- **yii** and **yii.bat**, command-line scripts for *nix and Windows, accordingly

> *{WARNING}* The **runtime** folder must be writable by the web server, which Composer should properly do for you.

This primary folder is the *application's* root folder, also called the *application's base 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 application's root folder.

Quite different from Yii 1, you'll see the Yii framework itself is installed as a third-party library within **vendor**. A **vendor** folder is the default destination for libraries installed via Composer, and the Yii framework is just one more Composer-installed library for the site!

The **views** folder has some predefined subfolders, too. One is **layouts**, which stores the template for the site's overall look: 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.

The website's root folder is **web**. Within it, you'll also find:

- **assets**, to be explained below
- **css**, for your site's CSS files
- **index.php**, a "bootstrap" file through which the entire website is run
- **index-test.php**, a development version of the bootstrap file

Of these folders, you'll use **css** 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 requiring you to copy these resources to a public, and to avoid potential naming conflicts, Yii will automatically copy these resources to the **assets** directory as 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 would 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, as Composer performs this task, unless you transfer a Yii site from one server to another and the permissions aren't correct after the move.

Yii 2 no longer has these folders, which used to be created in Yii 1 applications:

- **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**, for storeing messages translated in various languages
- **migrations**, for automating database changes
- **themes**, for storing multiple site looks

The **extensions** directory is effectively replaced by **vendor**. The other directories may or may not be required by your application, but you can create them if need be.

Referencing Files and Directories

Since the Yii framework adds extra complexity in terms of files and folders, the framework uses aliases to provide easy references to common locations. Many aliases are predefined.

> *{NEW}* In Yii 2, all aliases are prefaced with @.

Alias	References
@app	The application's root folder
@bower	The Bower package directory
@npm	The NPM package directory
@runtime	The application's **runtime** folder
@vendor	The application's **vendor** directory
@web	The base URL for the site
@webroot	The directory with index.php
@yii	The Yii framework folder (within **vendor**)

Bower and NPM are package managers, similar in use to Composer. By default, Bower and NPM packages are installed in **vendor/bower** and **vendor/npm**, accordingly.

Almost all of these aliases store system paths: how you'd refer to that folder using the file system. The exception is `@web`, which is a URL reference.

Each extension installed by Composer will have its own alias, too.

Yii Conventions

The Yii framework embraces a "convention over configuration" approach. This means 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 unusual or distasteful.

But if you really don't like doing something a certain way, Yii allows you to change the default convention. Know that doing so requires a bit more work–and code–and increases the potential for bugs. For example, if you want to organize your application's base directory in another manner, such as moving 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 uses capital letters instead of underscores to break up words; lower-camelcase and upper-camelcase differ in whether the first letter is capitalized or not. Private variables in classes 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*.

Any file that defines a class should have the same name, including capitalization, as the class it defines, plus the **.php** extension: the *MyController* class is defined within the **MyController.php** file. Again, this is normal in OOP.

> *{TIP}* Only ever define a single class within a single PHP file.

Namespaces use all lowercase letters. For example, a model class will be within the *app/models* namespace. Namespaces in Yii generally match the file structure, which also uses all lowercase letters.

> *{NEW}* Thanks to the use of namespaces, Yii 2 no longer prefaces its own classes with a letter as Yii 1 did.

Database Conventions

The Yii database conventions is 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 won't complain if you use plural names. Whatever you decide, consistency is the most important factor: 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

Learning how the Yii framework handles something as basic as a page request will go a long way towards understanding the greater Yii context.

In a non-framework site, when a user goes to **http://example.com/page.php** in a 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**, found within the **web** directory. This is called the "bootstrap" file, which the Yii guide also calls an "entry script". With Yii, and some server configuration, all of these requests will be funneled through the bootstrap file:

- **http://example.com/**
- **http://example.com/index.php**
- **http://example.com/index.php?r=site**
- **http://example.com/index.php?r=site/login**
- **http://example.com/site/login/**
- **http://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–always will.

Let's look at what the bootstrap file does.

The Bootstrap File

The contents of the **index.php** file, automatically included in the basic Yii template, will look something like this (with comments removed):

```
1  <?php
2  defined('YII_DEBUG') or define('YII_DEBUG', true);
3  defined('YII_ENV') or define('YII_ENV', 'dev');
4  
5  require(__DIR__ . '/../vendor/autoload.php');
6  require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
7  
8  $config = require(__DIR__ . '/../config/web.php');
9  
10 (new yii\web\Application($config))->run();
```

The first two meaningful lines (2 and 3) establish the debugging behavior.

Line 5 includes the Composer autoloader. This will ensure the proper loading of any third-party library (also installed by Composer).

Line 6, includes the base Yii framework file.

Line 8 identifies the configuration file to use for this application. By default, that configuration file is **web.php**, found within the **config** directory.

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

The final line invokes the `run()` method of the `Application` class. The code first implicitly creates an instance of the `Application` class, specifically a "web application" instance. The class's constructor is provided with the location of the configuration file to use. 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:

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

That's all that's happening in the bootstrap file: the autoloader is established, and then 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 first let's look at the application object in more detail.

The Application Object

So what does it mean to say that the website 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 few pages, just understand here that components are made available to the site through the application object.

> *{TIP}* There are two types of application objects a Yii site may have: *web* and *console*. The latter is for command-line scripts.

The application object's second important task is to handle every user request: viewing of a particular page, the submission of a form, and so forth. The handling of a 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 referring to the static `$app` variable within the `Yii` class: `\Yii::$app`, where `\Yii` refers to the `Yii` class in the top-level namespace. Whether you need to access the name of the application in a view file, store a value in a session, or get the identity of the current user, that will be done through `\Yii::$app`. The web application object is the "context" through which the site runs.

> *{NEW}* In Yii 2, access the application instance through a static **$app** variable instead of a static **app()** method.

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

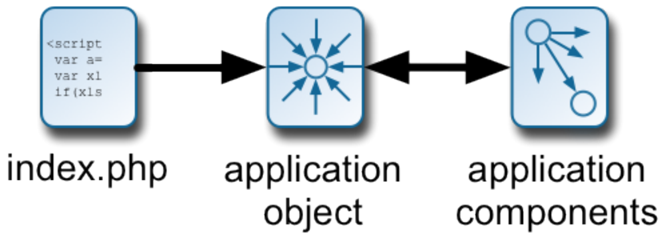


Figure 3.1: *The index page creates an application object which loads the application components.*

Routing

In a *non-framework site*, a user request will be quite literal (e.g., **http://example.com/page.php**). Yii still uses the URL to identify requests, but all requests instead go through

index.php. To convey the specific request, the requested route is appended to the URL as a variable. In the default Yii behavior, the request syntax is **index.php?r=ControllerID/ActionID**. In that code, 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 are the agents in an application: they handle requests and dictate the work to be done. The default site created by Composer has 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 **controllers** directory. The ID of this controller is the name of the class, minus the word "Controller", all in lowercase: "site".

Every controller can have multiple *actions*: specific things done with or by that controller. The five actions defined by default in the site controller are: about, contact, index, login, and logout. As you'll learn in more detail in Chapter 7, "Working with Controllers," actions are created by defining a method named "action" plus the action name:

- `actionAbout()`
- `actionContact()`
- `actionIndex()`
- `actionLogin()`
- `actionLogout()`

The action ID is the name of the function, minus the initial "action", all in lowercase: "about", "contact", "index", "login", and "logout".

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

http://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. 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, 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://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 Composer as part of the basic application template.

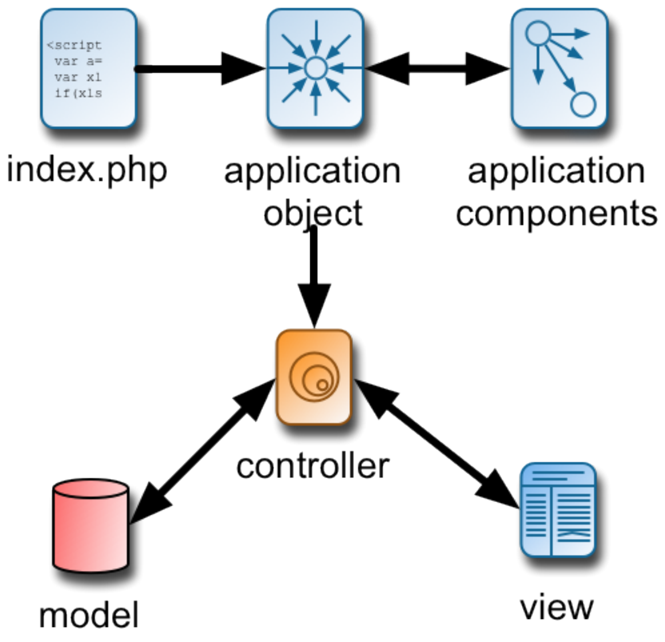


Figure 3.2: *Subsequent steps involve calling the correct controller, accessing models, and rendering views.*

Third, many requests will require additional information 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 is also an "action". Three of these–all but "create"–also require a page identifier to identify which page of content is being read, updated, or deleted. In such cases, the request URL will be of the format **http://example.com/index.php?r=page/delete&id=25**.

Fourth and finally, the default request syntax is:

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

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

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

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

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

You'll see how this URL manipulation is done in Chapter 4, "Initial Customizations and Code Generations."

Using the Debugger

One of the best additions in version 2 of the Yii framework is a top-notch, built-in debugger (**Figure 3.3**). The Yii debugger is a tool you'll want to be familiar with, as it'll save you lots of time over your development life.

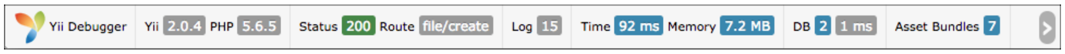


Figure 3.3: *The new, built-in Yii debugging toolbar.*

The debugger is enabled by default–later you'll learn how to disable it, but you can minimize it by clicking the right arrow on the far right-side of the toolbar.

The debugger initially shows the:

- Version of the Yii framework in use
- Version of PHP in use
- HTTP status code for the requested page
- Route of the requested page
- Number of logged messages for the requested page
- How long the page took to be rendered
- How much memory was required to render the page
- Number of asset bundles required

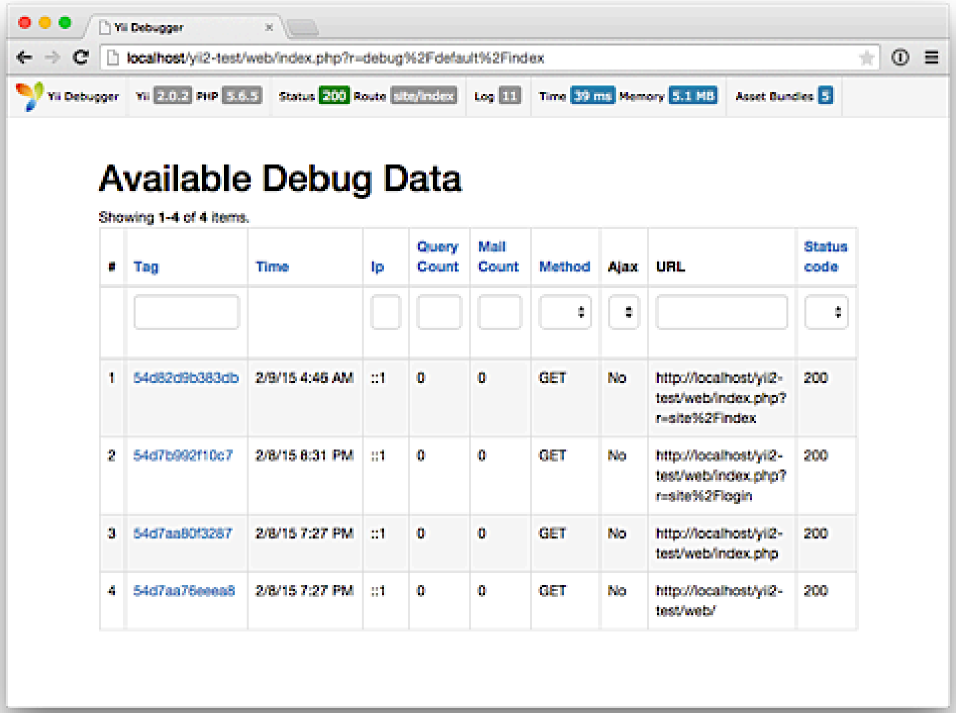
If you click on any item in that list in the debugger, you'll be taken to more details. If you click on the "Yii Debugger" title, you'll be taken to a master page where you can view a history of debugging information: with one item per request (**Figure 3.4**). Clicking on any item in the table takes you to the details of that debugging log.

I encourage you to click around within the debugger to see what's there. You'll find that the debugger shows, in great detail, the:

- Configuration of the site, including the available extensions
- Values in `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION`, and `$_SERVER`
- Headers sent by the page

These are all often useful, especially the values in those superglobal arrays. This means that, without using Xdebug or a single `echo` statement, you can view the particulars of:

- Bad requests
- Requests that simply didn't work
- POST requests



Available Debug Data

Showing 1-4 of 4 items.

#	Tag	Time	Ip	Query Count	Mail Count	Method	Ajax	URL	Status code
1	54d82d9b383db	2/9/15 4:46 AM	::1	0	0	GET	No	http://localhost/yii2-test/web/index.php?r=site%2Findex	200
2	54d7b992f10c7	2/8/15 8:31 PM	::1	0	0	GET	No	http://localhost/yii2-test/web/index.php?r=site%2Flogin	200
3	54d7aa80f3287	2/8/15 7:27 PM	::1	0	0	GET	No	http://localhost/yii2-test/web/index.php	200
4	54d7aa76eeea8	2/8/15 7:27 PM	::1	0	0	GET	No	http://localhost/yii2-test/web/	200

Figure 3.4: *Debugging information for the most recent page requests.*

- Ajax requests

Those benefits alone makes for an amazing debugging tool. But the debugger also records every email sent by the site and every database query executed.

I'd strongly recommend you start using the Yii Debugger from the get-go. At first, use the debugger to help understand how Yii works. For example, if you look at the "log messages", you'll see the steps Yii took to render the page, including the components, specific classes, and specific methods, involved.

As you're more comfortable with Yii, you'll use the debugger as a debugging tool: easily and quickly seeing what just happened, saving you from having to insert a slew of `echo` statements and retrying it all again.

Finally, I'd recommend not worrying too much about the performance reports at first: how long a page took to render and how much memory was required. Chapter 17, "Improving Performance," discusses performance more, but for now, understand that the performance reported at this point is the worst it'll be, in part because the debugger itself takes a significant performance toil!

Chapter 4

INITIAL CUSTOMIZATIONS AND CODE GENERATIONS

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

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

Before all that, however, you'll likely want to make some changes to how your web server runs.

Configuring Your Web Server

In Yii 2, the **web** directory created as part of the site template is meant to be the web root directory, which is to say that **http://example.com** should point there. On localhost, it's theoretically not a problem to leave this as is, using, for example **http://localhost/~username/yii-test/web** as the URL. However, to best mimic a production environment, I'd recommend configuring your web server accordingly before getting into the development.

Assuming you're using Apache, and only have one site, you can configure Apache–in its **httpd.conf** file–to point to the new document root: **/path/to/yii2-test/web**. If you are developing multiple sites locally, you can create a new virtual host instead. In both cases, you'll most likely want to look up instructions online for your operating system and Apache installation, but the end result will be creating something like the following in a ***.conf** file:

```
<VirtualHost *:80>

    # Set the document root:
    DocumentRoot "/path/to/yii2-test/web"

    # Site has its own logs:
    ErrorLog "/private/var/log/apache2/yii2-test-error_log"
    CustomLog "/private/var/log/apache2/yii2-test-access_log" common

</VirtualHost>
```

This may go in the primary **httpd.conf** file or within another configuration file that's included by the primary one. For example, on a Mac, this may be written within **/etc/apache2/extras/httpd-vhosts.conf**. There are comments within the code that explain what's happening, but search online for particulars to your web server and operating system.

For easier development, create an alias to a URL by defining a new server name. To do that, in the above code within the `VirtualHost` directive, add:

```
ServerName yii2-test
```

This tells Apache that the URL **http://yii2-test** points to that directory. However, you still need to tell your computer that **http://yii2-test** can be found on your computer, not on the Internet. On Mac OS X and *nix, that's done by adding this line to your **/etc/hosts** file:

```
127.0.0.1 yii2-test
```

This says the host "yii2-test" points to localhost (aka 127.0.0.1). On Windows, the same effect is accomplished by editing **C:\WINDOWS\system32\drivers\etc\hosts**, although that instruction or file location could differ from one version of Windows to the next.

After making these changes, test them by heading to **http://yii2-test** in your browser.

Once you've configured your web server, move onto configuring your Yii application.

Enabling Debug Mode

The first thing to do when developing a site is making sure debugging mode is enabled. This is done–for the entire site–in the bootstrap file, thanks to these two lines:

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

Written out less succinctly, those lines equate to:

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

These lines tell Yii to set debugging to true if it's not already set, and to set the environment to "dev", if it's not already established.

These lines are the default for any newly installed basic site, but you can check that they are in the bootstrap file if 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!).

Similarly, you want to ensure the debugging toolbar, introduced in the previous chapter, is enabled. To do that, open **conf/web.php** and confirm these lines are present and active, likely found near the end of the file:

```
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
```

I'll return to the configuration file shortly, but that code says that if it's the development environment, the "debug" module should be enabled all the time. (The `YII_ENV_DEV` constant is given a Boolean value in the **BaseYii.php** script, based upon the value of `YII_ENV`.)

While checking your debugging settings, also confirm that PHP's *display_errors* setting is enabled. If it's not, then parse errors will result in a blank screen, which makes debugging impossible.

Note that these recommendations are for sites *under development*. Due to the extra debugging information and logging, sites will run slower with these settings. A production site on a live server should have Yii's debugging mode disabled–by removing that line of code in **index.php**–and PHP's *display_errors* setting turned off. You'll read more on what else you should do before going live in Chapter 24, "Shipping Your Project."

Basic Configurations

After possibly configuring your web server and ensuring that debugging is enabled, the rest of a site's configuration will be done within its configuration files. 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

In the **config** directory, you'll find four configuration files:

- **console.php**, configures console applications
- **db.php**, stores the database configuration
- **params.php**, stores site variables
- **web.php**, configures web applications

> *{NEW}* Yii 2 has a new set of configuration files, and renames "main" as "web".

The **index.php** bootstrap file includes **web.php** as its configuration file. Open the web configuration file in your text editor or IDE and you'll see it returns an array of name=>value pairs. A common question is: How do I know what names to use and what values or value types? The rest of this chapter will explain the most important names and values, but the short answer is: Any writable property of the `yii\web\Application` 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 be an instance of type `yii\web\Application`. The configuration file, therefore, configures this object. "Configures" means the configuration file sets the values for the object's public, writable properties. In other words, the configuration file tells 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 crucial to comprehend.

For example, `yii\web\Application` has an `id` property, which takes a string as the ID value for the application. By default, the basic Yii application defines this for you:

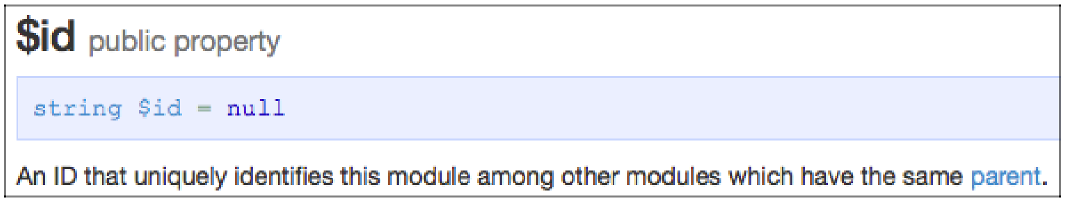
```
$config = [
    'id' => 'basic',
    // Lots of other stuff.
];
return $config;
```

Simply change the value of the *id* element in that array and you'll successfully change the ID of the web application. (The effects of this particular change will not be readily apparent, however.)

As the configuration file is extremely important, you have to master how it works. I first recommend that you be *very* careful when making edits. Because the whole file returns an array, and as many of the values will also be arrays, the result is a syntactic eggshell of nested arrays within nested arrays. A failure to properly match parentheses or brackets, or a missing comma, will result in a parse error.

> *{TIP}* Duplicate the configuration file before making new edits. Or use version control!

Second, learn how to read the Yii class documentation, starting with the page for yii\web\Application. For example, the configuration file can be used for any writable property of the `yii\web\Application` class. Using the docs for that class, see 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 `id`:



$id public property

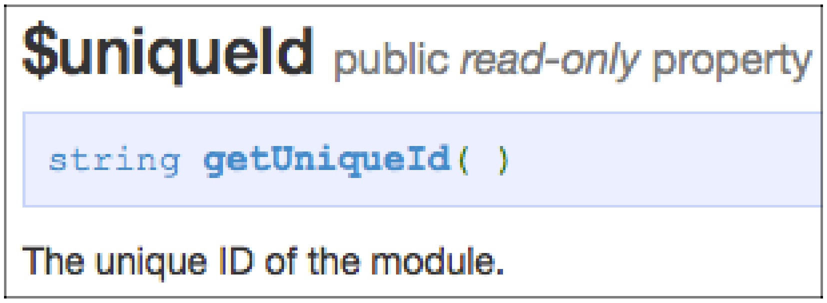
```
string $id = null
```

An ID that uniquely identifies this module among other modules which have the same parent.

Figure 4.1: *The Yii docs for the* `id` *property of the* `yii\web\Application` *class.*

The property expects a string value, and has a null value by default. Now you know that `id` must be assigned a string. It's really that simple!

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



$uniqueId public *read-only* property

```
string getUniqueId( )
```

The unique ID of the module.

Figure 4.2: *The Yii docs for the* `uniqueId` *property of the* `yii\web\Application` *class.*

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

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

Configuring Components

Rather than walk through the configuration file sequentially, let's go in order of most important to least. Arguably the most important section is "components". Components are application utilities that you and Yii create. To start configuring a new application, configure how it uses Yii's predefined components.

> *{NOTE}* Unless otherwise specified, all configuration changes are made within the **web.php** file.

Predefined Components

The Yii framework defines over a dozen core application components for you, representing common site needs. Just some of those are:

- "assetManager", for managing CSS, JavaScript, and other assets
- "cache", for caching of site materials
- "db", which provides the database connection
- "i18n", which provides internationalization functionality
- "mail", for creating and sending email
- "request", for working with user requests
- "session", for working with sessions
- "user", which represents the current user

The names of the components match the corresponding configurable `yii\web\Application` properties. For each component, Yii defines a class that does the actual work.

This chapter explains the basic configuration of components most immediately needed. The rest of the book will introduce other predefined components as warranted.

Enabling and Customizing Components

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

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'bootstrap' => ['log'],
    'components' => [
    ],
    'params' => $params,
]; // End of components array.
// Other stuff.
return $config;
```

Within the “components” section, each component is declared and configured using the syntax:

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

(Or `'componentName' => [/* configuration values */]` in the new short array syntax.)

The names of the predefined components are: “authManager”, “cache”, and the others already mentioned, plus a few more listed in the manual. The name is also the component’s ID.

The configuration values will vary from one component to the next. To know what options are possible for a component, look at the underlying class that provides that component’s functionality. For example, the **db.php** configuration file configures the “db” component (through **web.php**):

```
# in web.php:
'db' => require(__DIR__ . '/db.php'),
```

```
# db.php
<?php
return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

The “db” component provides a database connection. The associated class is `yii\db\Connection`. When a database connection is required, an object of type `yii\db\Connection` will be created. The “db” element of the “components” section of the configuration file sets the values of that object’s properties.

In the Yii API reference (aka, the class documentation), you see that `yii\db\Connection` has a public, writable `username` property. Therefore, that property's value can be assigned in a configuration file, as shown from **db.php**. With that line in the configuration file, when the site needs a database connection, Yii will create an instance of `yii\db\Connection` type, using "root" 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 **Application** object, individual component configurations can only assign values to the writable properties of the associated class.

Before getting into configurations of the most important components, there are two more things to know. First, Yii wisely only creates instances of application components when the component is required. If you configure your application to have a database component, Yii will still only create that component on pages of the site that use the database. Thanks to Yii's automatic management of components, sites will perform better without needing tediously tweak and edit each page (i.e., to turn components on and off).

On the other hand, Yii can be told to *always* create an instance of a component. This is done through the "bootstrap" element of the main configuration array:

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'bootstrap' => ['log'],
```

By default, the logging component is always loaded. To always load other components, just add those component IDs to that array. Still, for performance reasons, you should only do so sparingly. For example, while developing a site, the Gii and debug modules are also automatically loaded:

```
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = 'yii\gii\Module';
}
```

The second thing to know about application components is how to access them in your code. Components are available 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:

```
\Yii::$app->db->username = 'this username';
```

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

Connecting to the Database

Unless you aren't using a database, you'll need to establish the database connection before doing anything else. Establishing the database connection is accomplished through the "db" component. In the **db.php** configuration file created by Yii, a connection to a MySQL database is defined:

```php
<?php
return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

The connection string is a *DSN*, short for Database Source Name, and has a precise format. The DSN starts with a keyword indicating the database application in use: such as "mysql", "pgsql" (PostgreSQL), "mssql" (Microsoft's SQL Server), or "oci" (Oracle). Follow this keyword with a colon, then 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`

The exact parameters will depend upon the database application and server environment in use. Depending upon your environment, you may have to set the port number or socket location. When using MAMP on Mac OS X, I have to set the port number as it was not the expected default (3306). On Mac OS X *Server*, I have 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.

Besides the DSN, you should obviously change the username and password values to the correct ones for your database. You may or may not want to change the character set.

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, let's look at the "urlManager" component. This component dictates, among other things, what format the site's URLs will be in.

> *{NEW}* Yii 2 does not include the "urlManager" component in the configuration file by default.

Creating SEO-friendly URLs Chapter 3, "A Manual for Your Yii Site," explains that the default URL syntax is:

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

To do that, just configure the "urlManager" component, setting the `enablePrettyUrl` property to true:

```php
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'bootstrap' => ['log'],
    'components' => [
        // Other stuff.
        'db' => require(__DIR__ . '/db.php'),
        'urlManager' => [
            'enablePrettyUrl' => true,
        ]
    ],
    'params' => $params,
];
```

You don't have to do anything to Apache's configuration for this to work. By using this component, any links created within the site will also use the proper syntax. You'll learn much more about how this works in Chapter 7, "Working with Controllers".

To test this change, after adding that code to the configuration file, save the file, and then reload the home page in your browser. Click on "Contact" and you should see the resulting URL is now **http://example.com/index.php/site/contact** instead of **http://example.com/index.php?r=site/contact**.

Hiding the Index File To take your URL customization further, configure "urlManager", along with an Apache **.htaccess** file, so that **index.php** no longer needs to be included: **http://example.com/ControllerID/ActionID/**.

To do this, add several Apache `mod_rewrite` rules to the site's configuration:

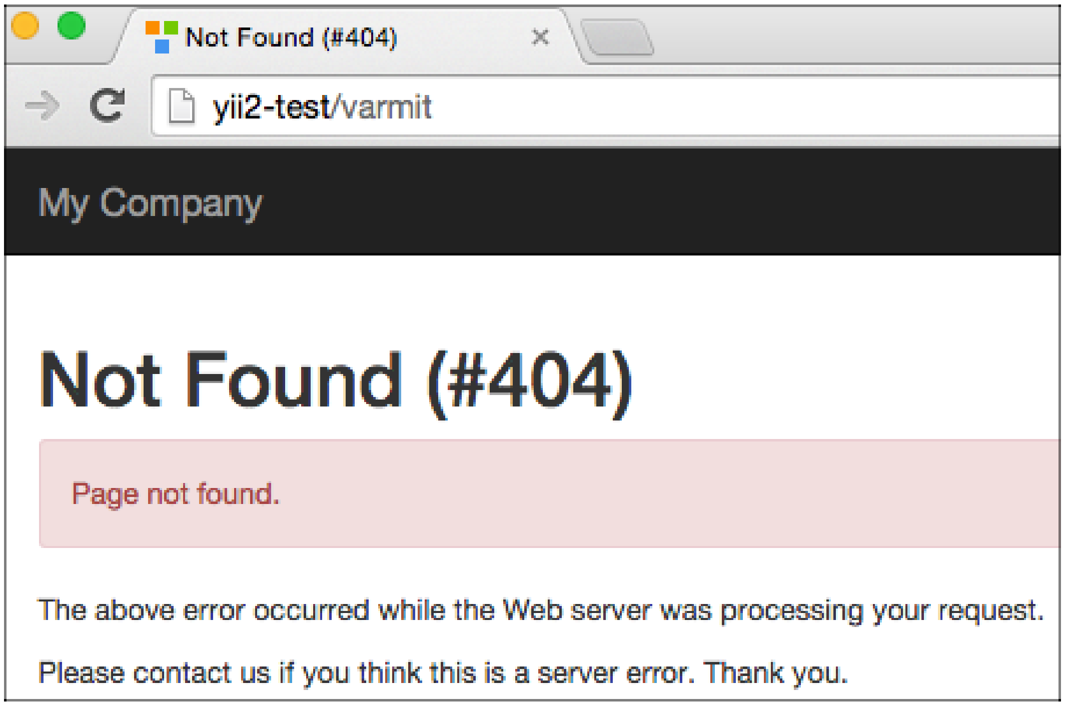
```
<IfModule mod_rewrite.c>

    RewriteEngine on

    # Change to match your URL base:
    RewriteBase /

    # If a directory or a file exists, use the request directly:
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d

    # Otherwise forward the request to index.php:
    RewriteRule . index.php

</IfModule>
```

If you're not familiar with `mod_rewrite`, you can look up oodles of tutorials online. This code would either go within an Apache configuration file or an **.htaccess** file. I would add it to my virtual host definition. Also, that code will only work if `mod_rewrite` is enabled in Apache. If your application is not in the web root directory, change the `RewriteBase` value accordingly (e.g., if the URL is **http://example.com/test/index.php**, set `RewriteBase` to "/test/").

Once you've implemented the `mod_rewrite` rules, test that `mod_rewrite` is working by going to any other file in the web directory–an image or your CSS script–to see if it loads. Then go to a URL for something that *doesn't* exist–such as **example.com/varmit**–and see if the Yii-based site is rendered, most likely with an error message (**Figure 4.3**).

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

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'bootstrap' => ['log'],
    'components' => [
        // Other stuff.
        'db' => require(__DIR__ . '/db.php'),
        'urlManager' => [
```


Not Found (#404)

yii2-test/varmit

My Company

Not Found (#404)

Page not found.

The above error occurred while the Web server was processing your request.

Please contact us if you think this is a server error. Thank you.

Figure 4.3: *An error message because the "varmit" resource doesn't exist.*

```
            'enablePrettyUrl' => true,
            'showScriptName' => false
        ]
    ],
    'params' => $params,
];
```

Now you can load pages via **http://example.com/ControllerID/ActionID**, such as **http://example.com/site/about**.

Modules

After configuring the necessary components, there are a few other configuration settings to look at. One is under the "modules" section. Modules are essentially mini-applications within a site. You might create an administration module or a forum module. Chapter 19, "Extending Yii," covers creating modules in detail, but you'll want to enable two of Yii's modules–debug and Gii–to begin.

The debug module is new to Yii 2, and just a delight to have. Introduced at the end of the previous chapter, you should be in the habit of using it whenever you have problems or confusion as to what is, or is not, going on.

The other module, Gii, is a web-based tool used to generate boilerplate model, view, and controller code for the application. Gii is a wonderful tool, and a great example of why I love Yii: the framework does a lot of the development for you.

Both the debug and Gii modules are enabled in development mode by default in the basic application template:

```
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';

    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = 'yii\gii\Module';
}
```

Not only are both enabled, but they're also bootstrapped–always available. Understand, however, that neither should be used on a live site.

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

Parameters

New in Yii 2 is a fourth configuration file: **params.php**. This is where user-defined parameters can be established. With the basic application template, one will already be there:

```
<?php
return [
    'adminEmail' => 'admin@example.com',
];
```

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

You can add other name=>value pairs to this file:

```
return [
    'adminEmail' => 'admin@example.com',
    'something' => 23,
];
```

The contents of the **params.php** file is pulled into the **web.php** configuration file by this line:

```
$params = require(__DIR__ . '/params.php');
```

The values are later added to the returned configuration array:

```
'params' => $params,
```

The contents of the **params.php** file are also brought into the **console.php** configuration file, making these values available in console applications, too.

By setting a parameter, you can globally access the parameter value via `\Yii::$app->params['paramName']`.

Developing Your Site

The book uses different practical examples to show real-world code that teach new concepts. Part 4 of the book creates two examples in full (or mostly full), 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 many of today's websites. CMS represents a moderately complex application to implement, and so makes for a good instructional example.

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 common elements, in varying percentages. When you start a new project, it's in one of these three areas that you must begin. In my opinion, you should always start with the functionality, as it dictates everything else. The functionality is what a website or application must be able to do, along with the corollary of what a user must be able to do with the website or application. The functionality needs to be defined in advance. Use a paper and pen, or a note-taking application, and write down every project requirement:

- 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 viewing. 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 and visible 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 implement.

And let's define what is meant 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 present downloadable files. This adds more requirements:

- The ability to upload files
- The ability to associate files with pages of content (i.e., where the files will be linked from)
- 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 them "page" and "file", respectively.

But it's 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 (specifically, the original author plus administrators)
- Only certain user types can assign types to users

In just a few moments, one initial goal–present content–quickly expanded into more than a dozen requirements. This is but a moderately complex example, which will work well for the purposes of this book.

Next Steps

Once you've established the functionality–with the client, too, if one exists, you can begin coding and creating files and folders. That process can be started 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.

If you're a designer, or are working with clients that think primarily in visual terms, it makes sense to begin new projects with how they will look. This may be a wire-frame representation or actual HTML, but create a series of pages and images that provide a basis for how the site will appear from a user interface perspective. You don't need to create every page, just 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 the mockup, 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 new projects with the data: what information will be stored and how the stored information will be used. For this task, use paper and pen, or a modeling tool such as the MySQL Workbench. The goal is to create a database schema.

> *{TIP}* Always err on the side of storing too much information, and always err on the side of complete normalization (when using a relational database).

With a schema defined, populate the database with sample data. This allows you to create the functionality that ties the data into a sample presentation. With that in place, you, and the client, can confirm the site looks and works as it should.

From there, you can implement more functionality, and then finalize the entire presentation and interface.

With the CMS site, I already have a sense of data required by the site: pages, authors, comments, and files. As a quick check, I can review the needed functionality and confirm that everything the site must be able to do requires just those four types of data.

Defining the Database

With the functionality and data requirements identified, it's time to create the database itself. Using a relational database application such as MySQL, one goes through the process of *normalizing* a database. It's beyond the scope of this book to explain that process, 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 4.4 shows the database schema, as designed in the MySQL Workbench.

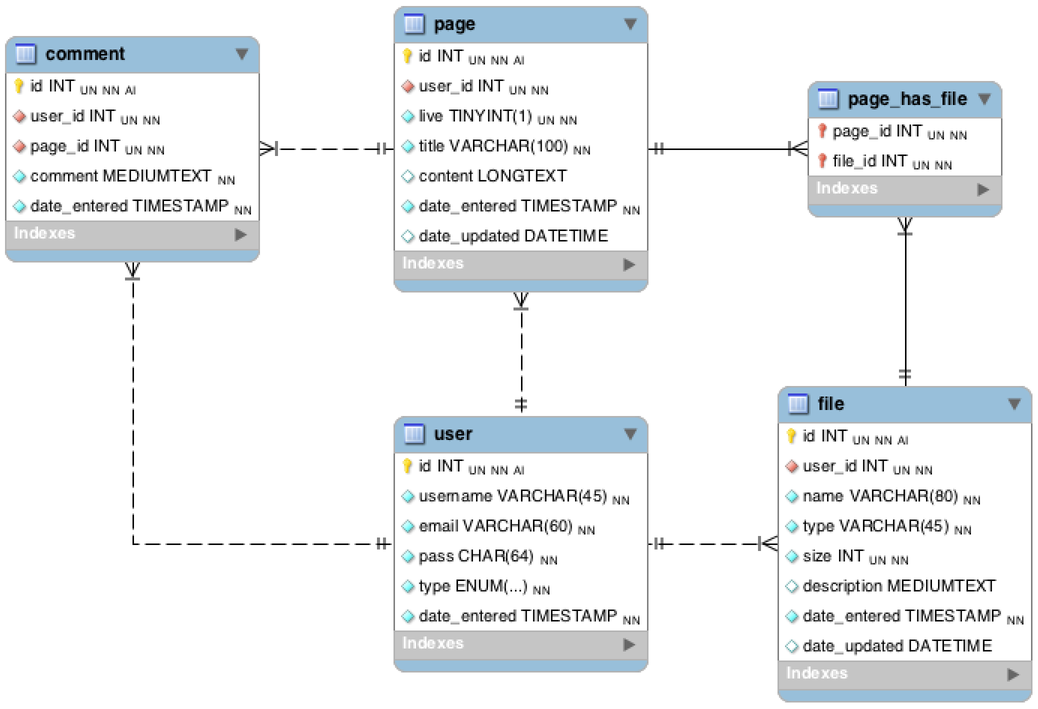


Figure 4.4: *The CMS database schema.*

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

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

```
CREATE TABLE IF NOT EXISTS yiibook2_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 stores a username, which must be unique, an email address, which must also be unique, and a 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))
```

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

> *{NEW}* Yii 2 supports the most current, secure method of password hashing available in PHP. Chapter 11, "User Authentication and Authorization," demonstrates its use.

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

```
CREATE TABLE IF NOT EXISTS yiibook2_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 yiibook2_cms.user (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

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 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., 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 yiibook2_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 yiibook2_cms.user (id )
        ON DELETE NO ACTION
        ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

Setting `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:

```
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, 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 yiibook2_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 yiibook2_cms.user (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION,
    CONSTRAINT fk_comment_page
        FOREIGN KEY (page_id )
        REFERENCES yiibook2_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:

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

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

```
CREATE TABLE IF NOT EXISTS yiibook2_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 DEFAULT 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 yiibook2_cms.user (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

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

New file records can be created using:

```
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 yiibook2_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 yiibook2_cms.page (id )
        ON DELETE CASCADE
        ON UPDATE NO ACTION,
    CONSTRAINT fk_page_has_file_file
        FOREIGN KEY (file_id )
        REFERENCES yiibook2_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:

```
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 the 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 want to allow *anyone* to post comments, you wouldn't tie comments to the `user` table, instead storing all the information about the person making the comment in `comment`. Chapter 22, "Creating a CMS," will do exactly that.

Second, this database doesn't support the option of categorizing or tagging content. That's easy enough to implement: just create a `tag` table and a `page_has_tag` table that acts as the intermediary.

Foreign Key Constraints in MyISAM Tables

The previous section, which outlines the database schema, makes repeated references to the foreign key constraints in place. Foreign key constraints are beneficial in databases as they help ensure 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 derive from 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 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 generated models.

The problem is MySQL only enforces foreign key constraints in InnoDB tables and 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 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 yiibook2_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 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). This comment has no effect on MySQL at all, but when you generate the models for these tables, Yii will automatically add code that reflects the proper relationships.

> *{TIP}* Yii's auto-generation of relationships is a nice touch, but in any situation where Yii doesn't generate the relationship for you, you can write the code to indicate the relation yourself.

Creating the Database

Having defined the database in SQL terms, create it in MySQL or whatever database application you're using. Do that now, using whatever tools you'd like, and the SQL commands available for download from this book's website. The MySQL Workbench file is also available there.

After creating the database and before continuing, double-check your database configuration file to confirm it connects to the proper database using the proper credentials. The CMS example uses the `yiibook2_cms` database.

Generating Code with Gii

Having created the database and configured the Yii site to connect to it, it's time to fire up Gii. The purpose of Gii is to generate the foundational model, view, and controller files required by the site. Part 2 of this book primarily explains how to edit these generated files to tweak them to your particular needs.

Gii Requirements

Before going any further, go through the following checklist:

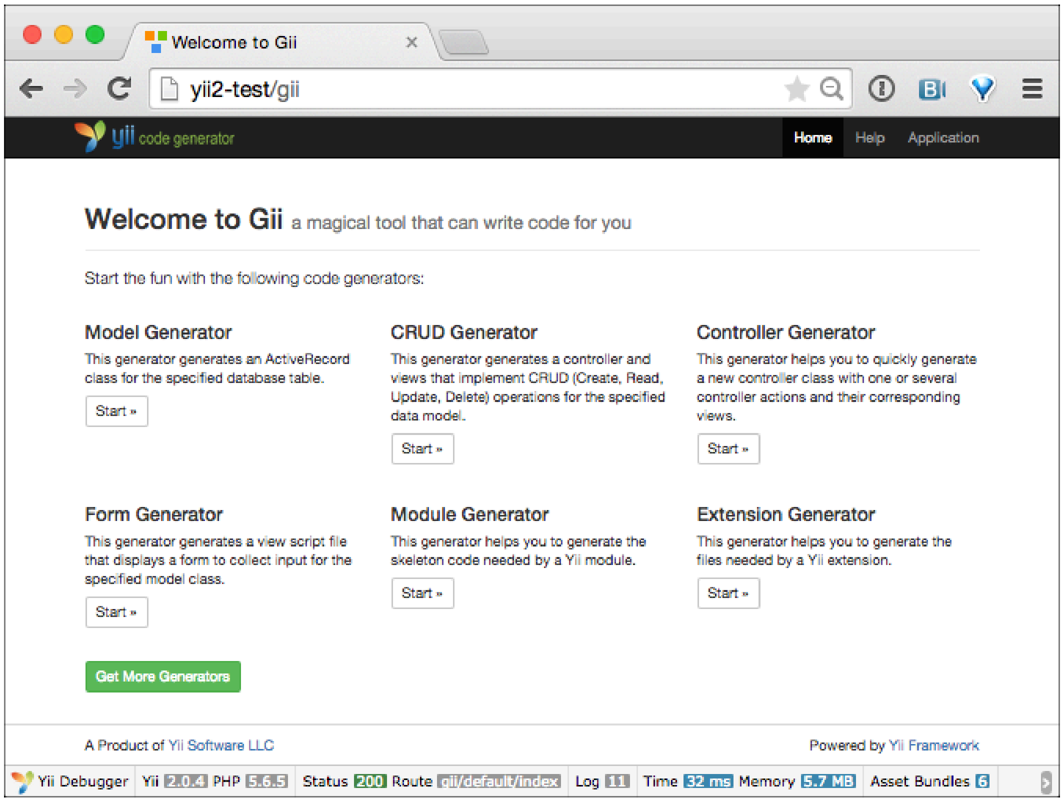
1. Confirm your Yii installation meets the minimum requirements.
2. Have your database design as complete as possible. Because Gii does so much work for you, it's best not to have to make database changes later on. If done properly, after creating your database tables following these next steps, you won't use Gii again for the project (at least not for basic models and CRUD functionality).

3. Make sure Gii is enabled (this is the default as of Yii 2, as explained earlier in this chapter).
4. Be using a development server.

Preferably, you've enabled Gii on a development server, you'll use it, then disable it, and then put the site files 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 **example.com/index.php**, the Gii tool is at **example.com/index.php/gii/**. This URL also assumes you're using the URL management component in Yii. If not, head to **example.com/index.php?r=gii** instead. Alternatively, if you configured your web server to hide the index file, you can just use **example.com/gii**.

Using that address, you'll see a splash page and several options (**Figure 4.5**).



Welcome to Gii a magical tool that can write code for you

Start the fun with the following code generators:

Model Generator
This generator generates an ActiveRecord class for the specified database table.
Start »

CRUD Generator
This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.
Start »

Controller Generator
This generator helps you to quickly generate a new controller class with one or several controller actions and their corresponding views.
Start »

Form Generator
This generator generates a view script file that displays a form to collect input for the specified model class.
Start »

Module Generator
This generator helps you to generate the skeleton code needed by a Yii module.
Start »

Extension Generator
This generator helps you to generate the files needed by a Yii extension.
Start »

Get More Generators

A Product of Yii Software LLC — Powered by Yii Framework

Yii Debugger | Yii 2.0.4 | PHP 5.6.5 | Status 200 | Route gii/default/index | Log 11 | Time 32 ms | Memory 5.7 MB | Asset Bundles 6

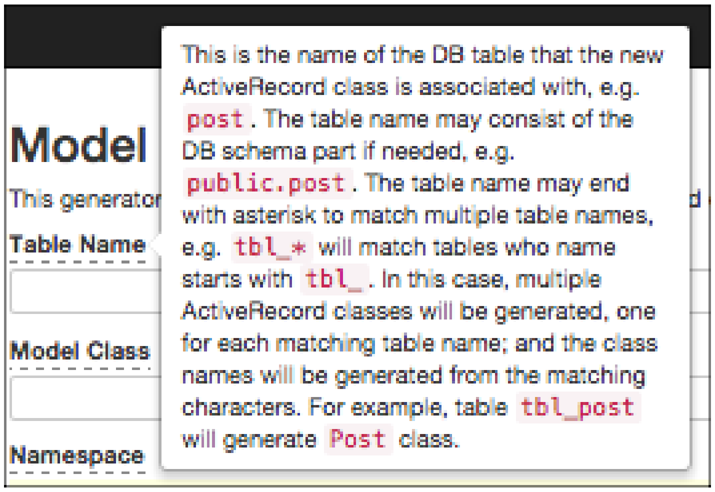
Figure 4.5: *The Gii home page.*

> *{NEW}* Yii 2 not only enables Gii by default, it does so without requiring a password, as the assumption is Gii will only be used–briefly–on a development server.

Gii can be used to generate:

- Models (specifically, Active Record models)
- CRUD functionality
- Controllers
- Forms
- Modules
- Extensions

Over the next couple of pages, you'll use two of these options: models and then CRUD. First, though, a tip: on subsequent Gii pages, links with a dashed underline will provide tooltips if you hover the cursor over them (**Figure 4.6**). Use these tooltips any time you're confused by a prompt or want to learn additional tricks.



This is the name of the DB table that the new ActiveRecord class is associated with, e.g. `post`. The table name may consist of the DB schema part if needed, e.g. `public.post`. The table name may end with asterisk to match multiple table names, e.g. `tbl_*` will match tables who name starts with `tbl_`. In this case, multiple ActiveRecord classes will be generated, one for each matching table name; and the class names will be generated from the matching characters. For example, table `tbl_post` will generate `Post` class.

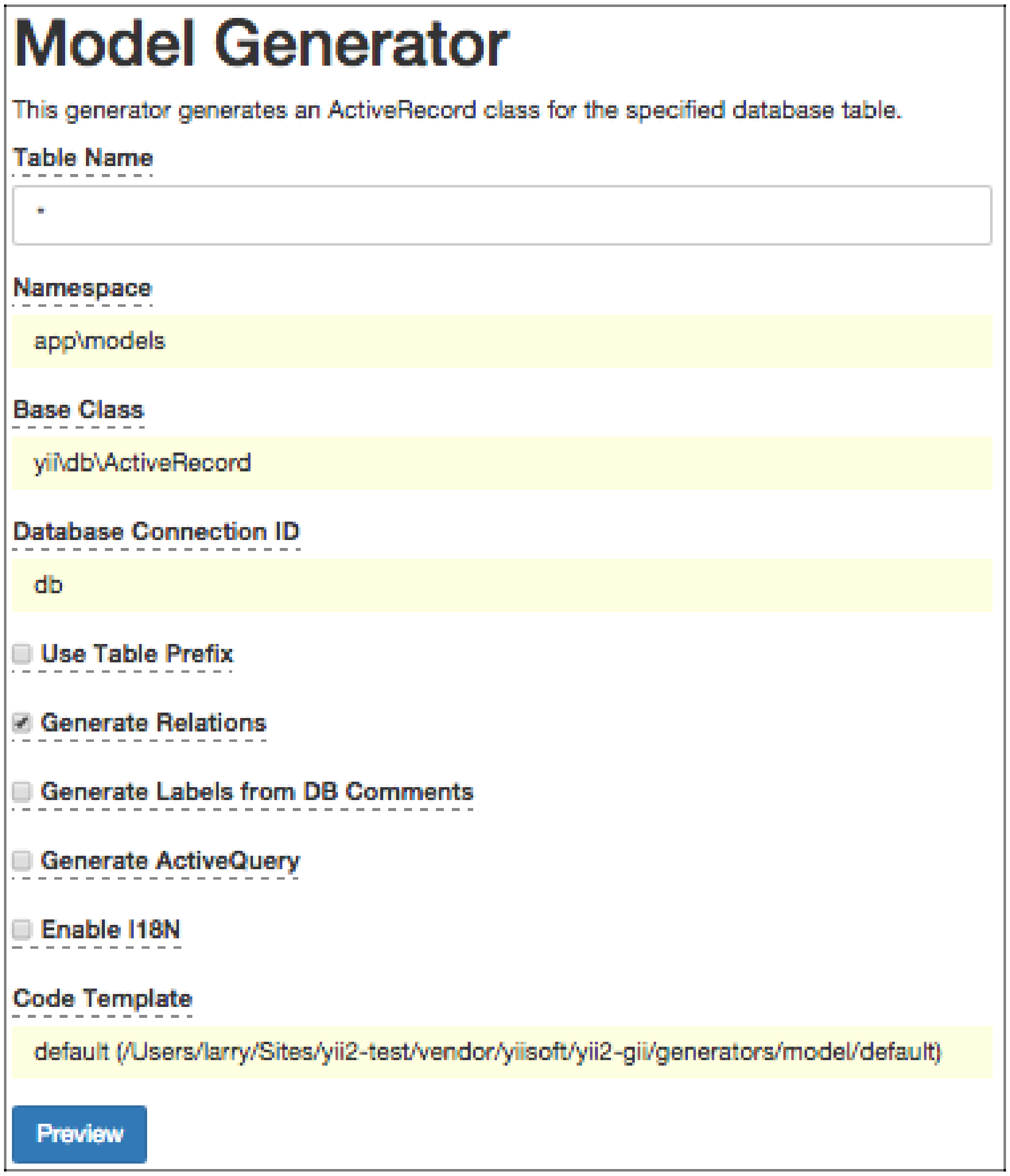
Figure 4.6: *A tooltip for using the model generator.*

Generating Models

Before explaining what to do next, there's a change you have to make due to new functionality in Yii 2. The basic Yii application now defines a `User` class for the purposes of logging in. This class would be destroyed by the generation of the `User` class involved in this example. Chapter 11 explains how to use the new `User` class for login purposes, but, for now, rename **models/User.php** as **models/User-original.php** so you can revisit that code later.

The first thing to do with Gii is generate the models. Click the "Model Generator" start button to head there. On the following page (**Figure 4.7**):

1. Enter * as the table name.
2. Click Preview.
3. In the preview (**Figure 4.8**), deselect the **PageHasFile.php** model, which is not needed by this application.



Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name: *

Namespace: app\models

Base Class: yii\db\ActiveRecord

Database Connection ID: db

[ ] Use Table Prefix

[✓] Generate Relations

[ ] Generate Labels from DB Comments

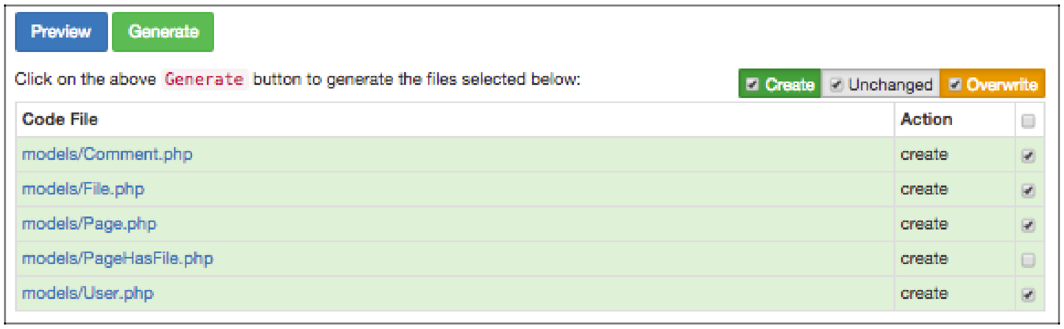
[ ] Generate ActiveQuery

[ ] Enable I18N

Code Template: default (/Users/larry/Sites/yii2-test/vendor/yiisoft/yii2-gii/generators/model/default)

Figure 4.7: *The form for auto-generating a model file.*

4. Click Generate.



Click on the above `Generate` button to generate the files selected below:

Code File	Action	☐
models/Comment.php	create	✓
models/File.php	create	✓
models/Page.php	create	✓
models/PageHasFile.php	create	☐
models/User.php	create	✓

Figure 4.8: *The preview of the files to be created.*

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, repeating that sequence for every other table.

After clicking Generate, you'll then see a message indicating the code was created. You can check for new files within the **models** directory to confirm this.

If you see an error about an inability to write the files, you'll need to modify the permissions on the **models** directory to allow the web server to write there. On Mac OS X and *nix, set the permissions to **0777** on **models**, **views**, and **controllers**. Then rerun Gii.

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

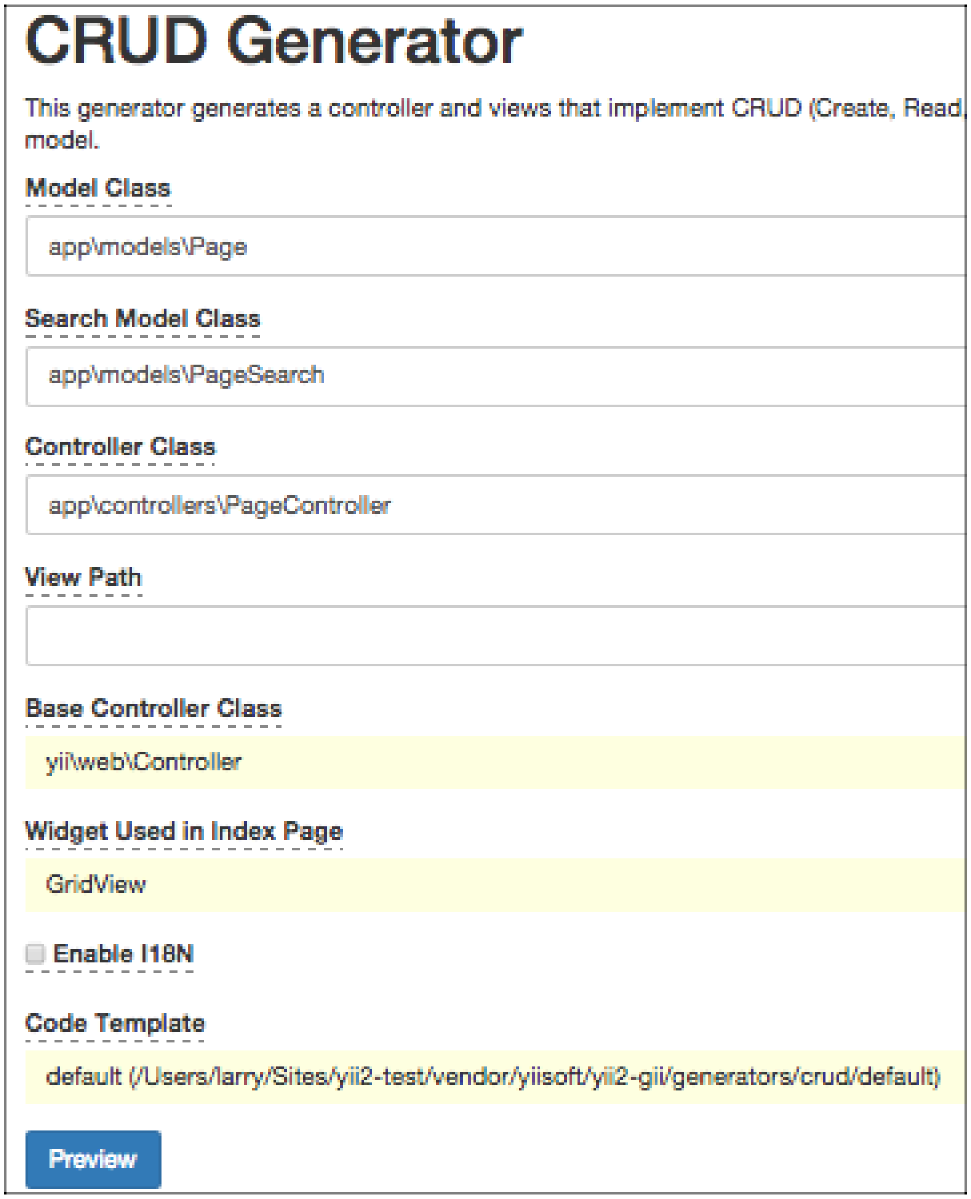
- **Comment.php**
- **File.php**
- **Page.php**
- **User.php**

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, Read, Update, and Delete: everything you do with database content. Yii's ability to write this code for you is a wonderful feature, saving you lots of time and energy.

> *{NEW}* In Yii 2, you can now also run Gii from the command line. See the Gii guide for more.



CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read,
model.

Model Class

app\models\Page

Search Model Class

app\models\PageSearch

Controller Class

app\controllers\PageController

View Path

Base Controller Class

yii\web\Controller

Widget Used in Index Page

GridView

Enable I18N

Code Template

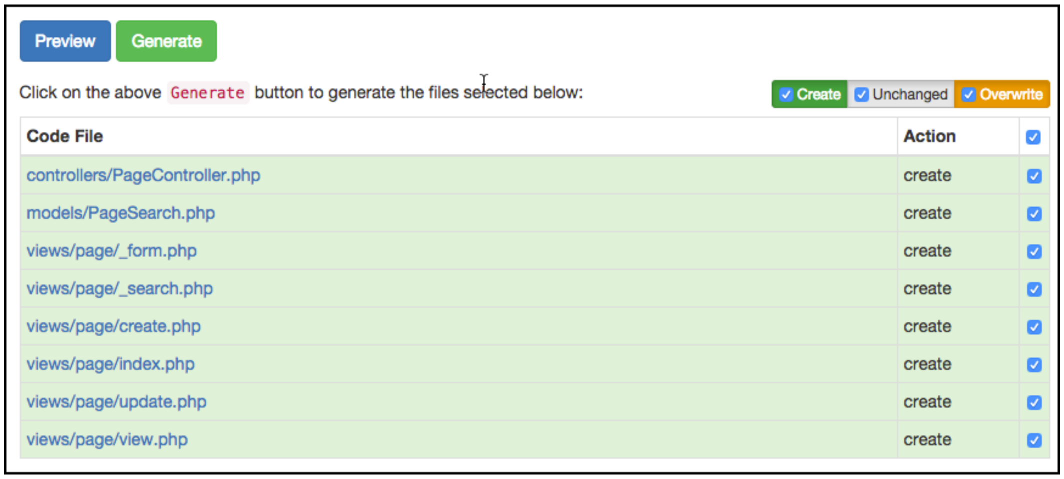
default (/Users/larry/Sites/yii2-test/vendor/yiisoft/yii2-gii/generators/crud/default)

Figure 4.9: *Generating CRUD functionality for pages.*

To begin, click the “Crud Generator” start link. On the following page (**Figure 4.9**):

> *{NEW}* Due to the use of namespaces, using Gii to create controllers in Yii 2 requires a bit more specification than Yii 1 did.

1. Enter “app\models\Page” as the Model Class.
2. Enter “app\models\PageSearch” as the Search Model Class.
3. Enter “app\controllers\PageController” as the Controller Class.
4. Click Preview.
5. Click Generate (**Figure 4.10**).



Click on the above `Generate` button to generate the files selected below:

Code File	Action	✓
controllers/PageController.php	create	✓
models/PageSearch.php	create	✓
views/page/_form.php	create	✓
views/page/_search.php	create	✓
views/page/create.php	create	✓
views/page/index.php	create	✓
views/page/update.php	create	✓
views/page/view.php	create	✓

Figure 4.10: *The preview of the files being generated for* `Page` *CRUD functionality.*

> *{TIP}* You can omit a value for the View Path, as Yii will default to a logical choice.

If all went well, that one step will create the controller file for the `Page` model, the `PageSearch` model for search purposes, plus a view directory for its view files, and six specific view files:

- **_form.php**
- **_search.php**
- **create.php**
- **index.php**
- **update.php**
- **view.php**

The controller will be explained in detail in Chapter 7. The view files will be covered in Chapter 6, "Working with Views." For your knowledge now, understand that the form file is used to both create and update records. The search script is a custom search form. The index script is 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.

Once those steps work for the `Page` model, repeat the process for `Comment`, `User`, and `File`. You don't need to create CRUD functionality for the `PageHasFile` class. 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! Return to the application's home page by clicking "Application". Then disable Gii by editing the web configuration file.

Confirm what you did worked by checking out the new directories and files or by going to specific URLs. Depending upon whether or not you added "urlManager" to the application's configuration, a test URL would be something like **example.com/index.php/user/** or **example.com/index.php?r=user**. There may or may not be records to list yet depending upon whether you manually inserted some data into the database. Don't try to add new records until you make additional edits, starting in Chapter 5.

> *{TIP}* If you use Yii a lot, and have your own ways of doing things, look into how you can customize the Gii generated output.

Regeneration in Gii

A common question relative to Gii is how to use Gii to regenerate models and CRUD functionality. For example, you create a model based upon a database table and customize the model code. Later, you change the table a bit and are thinking about re-running Gii, but worry about wiping out all your edits. What can you do?

The first option is not to use Gii at all. Instead, manually make the corresponding change in your class file.

The second option is to use your version control software–we're all using Git, right?–to handle the change management for you. You'll make all your edits in master, then create a new branch. Rerun Gii in the new branch, wiping out the existing code, and then merge the resulting changes onto master.

A third option is a bit more sophisticated. Instead of editing the files created by Gii, extend that class into another class (e.g., `Page` into `MyPage`). Then make all edits in the extended class. This allows you to regenerate and override the base class via Gii, while leaving your edits in the extended class untouched.

Part II

Core Concepts

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 functions. Part 1 also explains how to create the initial shell of an application, and how to have Yii create code for you via Gii.

Part 2 of the book expands that knowledge to customizing the generated code. The combination of generated code and your alterations is how Yii-based sites are created: have Composer and Gii create the boilerplate materials, and then edit those files to make the code specific for your application.

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

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. To keep the chapter to a reasonable length, and to avoid overwhelming 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 Yii go within the **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.

In Yii, every model class ought to inherit from the `yii\base\Model` class or a subclass. The most common subclass is `yii\db\ActiveRecord`, for models associated with databases. Conversely, `yii\base\Model` is the basis of models *not* tied to database tables, such as those associated with other HTML forms. Another way of differentiating between the two model types is that ActiveRecord models *permanently* store data whereas `yii\base\Model` 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.

> *{NEW}* In Yii 2, the base **Model** class is used instead of **CFormModel** for non-Active Record models.

For example, if you have Yii create your model code and site shell for the CMS example using steps outlined in Chapter 4, "Initial Customizations and Code Generations," you'll have eleven model classes:

- **ContactForm.php** and **LoginForm.php**, both of which extend `Model`
- **Comment.php**, **File.php**, **Page.php**, and **User.php**, all of which extend `ActiveRecord`
- **CommentSearch.php**, **FileSearch.php**, **PageSearch.php**, and **UserSearch.php**, all of which extend their parent class: `Comment`, `File`, `Page`, and `User`

> *{TIP}* A model can be broken into one base class and multiple derived classes. This is beneficial when not all the model's methods are needed everywhere in the site, such as with modules or the search models.

Even though these eleven classes primarily 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 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 `Model` have this general structure:

```
namespace app\models;

use Yii;
use yii\base\Model;

class ClassName extends Model {

    // Attributes...
    public $someAttribute;

    // Methods...
    public function rules() {}
    public function attributeLabels() {}

}
```

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

```
namespace app\models;

use Yii;

class ClassName extends \yii\db\ActiveRecord {

    // Methods...
    public function tableName() {}
    public function rules() {}
    public function attributeLabels() {}

}
```

Two of the methods–`rules()` and `attributeLabels()`–are common to both model types. Further, as both model types inherit (indirectly) from `Model`, other methods are common to both but aren't included in these specific model definitions. In fact, *most* of the model's functionality isn't defined in your model class, but rather in a parent class. You'll see some of these inherited methods later in the chapter.

The `Model` models will always have declared attributes. These attributes temporarily represent model data. Conversely, `ActiveRecord` models don't 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 `ActiveRecord` models also define other methods that `Model` models do not have, including `tableName()`. Relationships between database tables, and therefore Active Record models, are represented as custom methods.

The rest of this chapter explains what these methods do, and how you might edit them. The chapter will also explain how to add your own attributes and methods when needed, just as you would in any class.

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. This method dictates much of your application's security and reliability. 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 updated, you won't have to write, replicate, and test the data validation routines. Yii handles 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). While those rules protect your data's integrity, violating them won't necessarily result in error messages end users can see, unlike the Yii model rules.

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

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

Note the use of the short array syntax, added in PHP 5.4. I'll sometimes use it, to follow Yii conventions; other times, I'll use the `array()` method if it provides more clarity.

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

The attributes are the class attributes (for `yii\base\Model` models) or table column names (for `\yii\db\ActiveRecord` models) to which the rule should apply. To apply the same rule to multiple attributes, just provide an array as the first argument:

```
array(['attr1', 'attr2'], 'validator', [other parameters])
```

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

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

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

Some validators take parameters that further dictate the requirements. For example, the "string" validator can take a "max" parameter, which sets the maximum string length:

```
[['name'], 'string', 'max' => 80],
```

That code is from the **File.php** model. Gii will automatically generate rules like this based on 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, do so as separate arguments:

```
['age', 'integer', 'min'=>13, 'max'=>100],
```

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

Available Validators

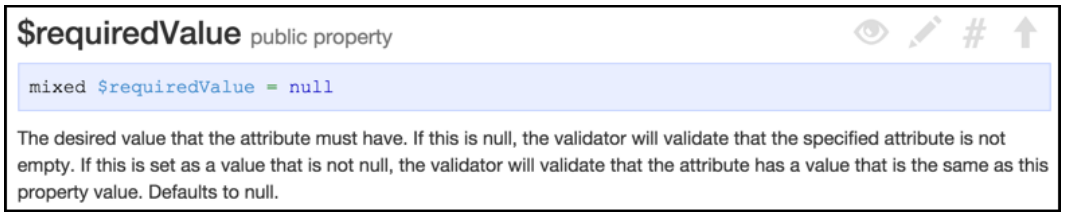
Yii has defined almost two dozen common validators for your use:

- boolean
- captcha
- compare
- date
- default
- double
- email
- exist
- file
- filter
- image
- in
- match
- number
- required
- safe
- string
- trim

- unique
- url

> *{NEW}* New in Yii 2 are specific type validators: double, integer, and string.

Each name is associated with a defined Yii class that performs the validation. If you look at the Yii class docs for any of them (linked through the yii\validators\Validator page), you can find the parameters associated with the validator. The parameters are listed as the class's properties. For example, the "required" class has a `requiredValue` property (**Figure 5.1**).



$requiredValue public property

```
mixed $requiredValue = null
```

The desired value that the attribute must have. If this is null, the validator will validate that the specified attribute is not empty. 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 `yii\validators\RequiredValidator` class.*

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

```
['acceptTerms', 'required', 'requiredValue' => 1]
```

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

Looking at the docs, you'll find that many validators have a `strict` property. It takes a Boolean indicating if a strict comparison is required: both the value and the type must match. This is false, by default.

Looking at the other validators, the "boolean" validator confirms that the value is Boolean-like. "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 basic template code uses the "boolean" validator for the "remember me" option in the `LoginForm` class:

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

The "captcha" validator is used with the `yii\captcha\CaptchaAction` class to implement captcha form validation. Chapter 12, "Working with Widgets," discusses this.

The “compare” validator compares a value against another value. The second value can either be another attribute or an external value. For example, a registration form often has two passwords. The second password, hypothetically called “passCompare”, would be represented as an attribute in the model, but not stored in the database:

```
class User extends \yii\db\ActiveRecord {
    // Add passCompare as an attribute:
    public $passCompare;
```

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

```
['pass', 'compare', 'compareAttribute' => 'passCompare']
```

The “compare” validator performs an equality comparison by default. Through the class’s `operator` property, other comparisons can be made: `==`, `===`, `!=`, `!==`, `>`, `<`, `<=`, 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, per the International Components for Unicode (ICU) standards. The characters are largely what you’d expect; you mostly adjust for whether values include leading zeros or not.

The “default” validator is not a 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 “double” validator, effectively the same as the “number” validator, confirms if a value is a double-precision floating-point number (i.e., something with a decimal in it).

```
['taxrate', 'double']
```

This validator takes optional “min” and “max” arguments to restrict the number to an inclusive range.

The “email” and “url” validators compare a 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, where “http” and “https” are the defaults.

```
['email', 'email'],
['website', 'url'],
```

The “exist” validator is for very specific and important uses. It confirms that the provided value exists in a table. You’ll normally use it to validate 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 “exist” in a few pages.

The “file” and “image” validators are for validating an uploaded file, where the “image” validator is an extension of “file”. This is a bit more complex of a process, and Chapter 9, “Working with Forms,” covers that.

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 “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:

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

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

```
['pass', 'match', 'pattern'=>'/^[a-z0-9_-]{6,20}$/'],
```

The “number” validator, which validates a double by default, can also be used to ensure a value is an integer:

```
['age', 'number', 'integerOnly' => true]
```

It also allows for minimums and maximums:

```
['age', 'number', 'integerOnly' => true, 'min' => 13, 'max' => 120]
```

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

```
[['user_id', 'name', 'type', 'size'], 'required']
```

Remember that “required” only ensures that an 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” validator is used to flag attributes as being safe to use without any other rules applying. 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 through a process 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:

```
['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. I generally recommend that you apply at least one validator to every attribute, and in the rare cases you cannot, that you at least apply a filter. Once you’ve applied the filter, you no longer need to declare the attribute as safe.

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

```
['pass', 'string', 'max' => 20],
```

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

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

```
['pass', 'string', 'min' => 6, 'max' => 20],
```

To require a string of a specific length, use the “string” validator’s `length` property:

```
['stateAbbr', 'string', 'length' => 2],
```

The `length` property can also set the minimum and maximum range:

```
['pass', 'string', 'length' => [6,20] ],
```

New in Yii 2 is the “trim” validator, which isn’t a validator, but rather a filter that applies the PHP `trim()` function to the attribute’s value:

```
[['firstName', 'lastName'], 'trim']
```

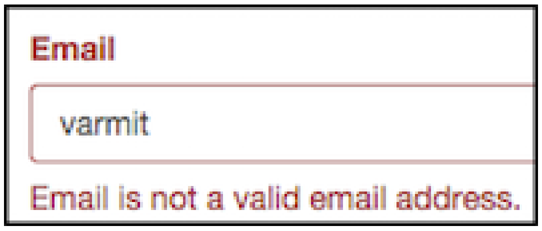
The “unique” validator requires that the value be unique for all corresponding records in the associated database table. You would use this to guarantee unique email addresses, for example:

```
['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

Many validators take additional parameters that map to public properties of the underlying validator class. Some parameters are common to every validator, as they all extend the `yii\validators\Validator` 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**).


Email

varmit

Email is not a valid email address.

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

If you don’t like the default error response, you can easily change it by assigning a new value to the `message` property:

```
['email', 'email', 'message'=>'You must provide an email address
    to which you have access.'],
['pass', 'match', 'pattern'=>'/^[a-z0-9_-]{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, use the special placeholder **{attribute}** to have Yii automatically insert the offending attribute:

```
['pass', 'match', 'pattern'=>'/^[a-z0-9_-]{6,20}$/',
    'message'=>'The {attribute} must be between 6 and 20 characters
    long and can only contain letters, numbers, the underscore,
    and the hyphen.'],
```

Your error message can indicate the provided value via **{value}**.

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

```
['age', 'number', 'integerOnly' => true, 'min' => 13, 'max' => 120,
    'tooSmall' => 'You must be at least 13 to use this site.'],
```

Setting Default Values

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')`. A user would not indicate her own user type when registering; 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. (You may not even be using MySQL!)

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

```
['type', 'default', 'value' => 'public']
```

If no value is provided, then "public" will be used. 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.

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:

```
['description', 'default', 'value' => NULL]
```

In Yii 2, it's semantically the same as the above if you just use the "default" validator without setting a default NULL value, but I prefer to be explicit.

Creating Your Own Validator

Thus far, you've seen the built-in validators, but Yii allows you to create your own, too. The advanced way to do so is to create a new class that extends `yii\validators\Validator`. A more simple approach–and more appropriate approach much of the time–is to define a new method that performs the validation within the same model that uses it. The `LoginForm` class created as part of the basic Yii application, does just that, defining a `validatePassword()` method:

```
public function validatePassword($attribute, $params) {
    if (!$this->hasErrors()) {
        $user = $this->getUser();
        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError($attribute, 'Incorrect username or
            password.');
        }
    }
}
```

What's going on in that code is a bit complicated for the beginner, so Chapter 11, "User Authentication and Authorization," explains it. For now, focus on the ability to define your own method as a validator. The method takes two arguments: the attribute being validated–a string–and the validation parameters, an array. Once defined, the method name is used as the validator name. Here's that rule from `LoginForm`:

```
['password', 'validatePassword'],
```

As in the `validatePassword()` example, your validation method indicates a problem–a lack of validation–by adding an error to the model instance object (aka `$this`). Yii uses the presences of errors, or lack thereof, as an indicator of whether or not the data passes all the validation tests. 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 attribute stores the MIME type of a file: *application/pdf*, *audio/mp4*, *video/ogg*, and so on. When the file is uploaded, the PHP code can read this value from the file. The value will later be used when PHP sends the file back to the browser (i.e., when the user downloads the file).

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

```
# 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($this->$attr, $allowed)) {
        $this->addError($attr,
        '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 [
        // Other rules.
        ['type', 'validateFileType'],
    ];
}
```

As Yii 2 a file type validator, defining your own is unnecessary. You'll see Yii's validator in Chapter 9.

Applying Filters

The "filter" validator is not a true validator, but rather a vehicle for running a value through a function. This processing occurs *prior* to any other validation. When you have attributes whose values don't align with any other validator, consider filtering that data for extra security. Common examples would be addresses or comments, both of which don't neatly fit any regular expression but should be sanitized for safe usage. Going with the CMS example, the `File` class's `description` attribute should be stripped of any HTML or PHP code:

```
['description', 'filter', 'filter' => 'strip_tags']
```

You can also write your own filtering function, if you need something more custom. The function takes one argument–the value being filtered–and returns a value:

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

This filter would then be invoked like so:

```php
# models/SomeModel.php::rules()
['attr', 'filter', 'filter' => 'filterValue']
```

Alternatively, as PHP now supports anonymous functions, you can combine the above two code blocks into one:

```php
['attr', 'filter', 'filter' => function($v) {
    // Do whatever to $v.
    return $v;
}]
```

Validation Scenarios

Rules can also be set to abide by *validation scenarios*. Validation scenarios are a way to restrict when a rule should or shouldn't apply. By default, rules apply under all scenarios. In order to change when a rule applies, you need to first identify the scenarios that exist.

There are two ways of defining scenarios. The first, most overt, option is to define a `scenarios()` method in the model. It should return an array of scenario names, whose values are arrays of attributes to be validated in that scenario:

```php
public function scenarios() {
    return [
        'login' => ['username', 'pass'],
        'register' => ['username', 'email', 'pass']
    ];
}
```

That code defines two scenarios, presumably for the `User` class. The "login" scenario validates the username and password. The registration scenario requires validation of the email address as well.

Alternatively, you can forgo the creation of a `scenarios()` method and merely reference scenarios within the rules. A scenario is applied to a rule using the syntax `'on' => 'scenarioName'`:

```
public function rules() {
    return [
        [['username', 'email', 'pass'], 'required', 'on'=>'register'],
        [['username', 'pass'], 'required', 'on'=>'login'],
    ];
}
```

If you don't define the `scenarios()` method, Yii will parse the available scenarios from the rules. If you want a rule to apply to multiple scenarios, just separate scenario name each with a comma.

> *{TIP}* Instead of using "on" to specify a scenario, you can use "except" to have a rule apply to every scenario but the scenario indicated.

Identifying the current scenario is done not in the model itself, but when an instance of that model is created. To flesh out this specific example, let's look at controllers a bit.

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 controller method begins with:

```
# controllers/UserController.php::actionCreate()
$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 scenario name:

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

Now the model is in the "register" scenario! Only rules set to apply during the "register" scenario will be invoked within this circumstance.

You might also create user-related scenarios for changing passwords or for changing other user settings.

Scenarios can also be set on existing instances by assigning a value to the `scenario` property:

```
$model->scenario = 'value';
```

This approach allows you to dynamically change a model's scenario after the model has been created.

Adding Behaviors

Models often also make use of *behaviors*. Behaviors in Yii are examples of mixins, emulating multiple inheritance by making methods not directly inherited by a class available for use within that class. More simply put, behaviors solve the problem where it'd be great to have some of class A's functionality in class B, even though class B doesn't inherit from A.

> *{NOTE}* Behaviors can be used by other class types, not just models.

Using TimestampBehavior

The most important behavior with respect to models is "TimestampBehavior", defined in `yii\behaviors\TimestampBehavior`. This behavior provides the ability to dynamically set an attribute's value to the current timestamp.

For example, the `File` class has `date_entered` and `date_updated` attributes. 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 created; in all other situations, the `date_entered` property should be left alone. To properly address the range of possibilities, a rule can be established to set this attribute's value, but 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.

The desired goal here is somewhat complicated, and highly contextual. Further, it relies upon being able to set the value of an attribute to a database function invocation. As Chapter 8 explains, you cannot just set an attribute's value to "NOW()", as that is a string, not a database function call.

This situation is not only tricky, it's also extremely common. Every model based upon a database table that has a timestamp column will need this capability. By creating the `yii\behaviors\TimestampBehavior` class as a behavior, that functionality can be added on the fly to any other class.

To use the timestamp behavior, the model must first grab a reference to the namespace class. This is done by adding a line to the model definition:

```php
<?php
namespace app\models;
use Yii;
use yii\behaviors\TimestampBehavior;
class File extends \yii\db\ActiveRecord {
```

Next, configure the behavior within a `behaviors()` method. It returns an array, with one array element for each behavior to use.

```
public function behaviors() {
    return [
        [ /* Behavior A */ ],
        [ /* Behavior B */ ]
    ];
}
```

Within each subarray, identify the class of the behavior and configure it that class. A behavior is configured by assigning values to the underlying class's writable properties. For a "TimestampBehavior", indicate the column to have a preset value upon insert and the column to have a preset value upon update. Here's the result for the "File" model:

```
public function behaviors() {
    return [
        [
            'class' => TimestampBehavior::className(),
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['date_entered',
                   'date_updated'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['date_updated'],
            ],
        ],
    ];
}
```

That code says that when the `ActiveRecord` "before insert" event occurs, the `date_entered` and `date_updated` attributes should be populated. When the "before update" event occurs, only `date_updated` needs to be set to the current timestamp. Because this behavior refers to the `ActiveRecord` class, the model must also use that namespace:

```
<?php
namespace app\models;
use Yii;
use yii\behaviors\TimestampBehavior;
use yii\db\ActiveRecord;
class User extends \yii\db\ActiveRecord {
```

Using BlameableBehavior

Another useful behavior for models is "BlameableBehavior". It makes it easy to associate a model attribute with the current user. In the CMS example, this functionality is needed to identify the author of a comment, page, or file (i.e., the uploader).

The behavior is simple to use. First, include a reference to the namespace:

```
use yii\behaviors\BlameableBehavior;
```

Next, configure the behavior to identify the "created by" and "updated by" attributes, if the class has them:

```
public function behaviors() {
    return [
        [
            'class' => BlameableBehavior::className(),
            'createdByAttribute' => 'user_id'
        ]
    ];
}
```

That's all there is to it! Yii will automatically assign the current user's ID to the `user_id` model attribute. Chapter 11 goes into the user's ID in more detail.

Putting It All Together

With all of this information in mind, here's how you go about defining rules for a model:

- **Identify required attributes.** This should be obvious and easy, but focus on information *required from the user*. Only establish required rules for attributes whose data may be provided by users. You wouldn't, for example, declare a required 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 an attribute has a value, but most attributes can be further restricted. Add subsequent 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, explicitly 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 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 to a range or length, if you can.* With numbers, 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. Similarly, validate strings to lengths matching the database definition for those columns.

- **Apply filters as appropriate.** Apply filters to any 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 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.
- **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, let's look at 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).

Three quick notes on the rules. First, the date column types will be populated using "TimestampBehavior". I'll explain that code after the initial rules, but when this behavior is used, you don't need rules for those attributes. Second, the `user_id` values will be populated using "BlameableBehavior". Just to be safe, these attributes are still referenced in the rules. Third, the `page_id` attribute in `Comment` will need to be set in a different way; to be explained later in the book.

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

// Must be in related models (tables):
[['user_id'], 'exist', 'targetClass'=>'User',
    'targetAttribute'=>'id'],
[['page_id'], 'exist', 'targetClass'=>'Page',
    'targetAttribute'=>'id'],

// Strip tags from the comments:
[['comment'], 'filter', 'filter'=>'strip_tags'],
```

And here is `Page`:

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

// User must exist in the related table:
[['user_id'], 'exist', 'targetClass'=>'User',
    'targetAttribute'=>'id'],

// Live needs to be Boolean; default 0:
[['live'], 'boolean'],
[['live'], 'default', 'value'=>0],

// Title has a max length and strip tags:
[['title'], 'string', 'max' => 100],
[['title'], 'filter', 'filter' => 'strip_tags'],

// Filter the content to allow for NULL values:
[['content'], 'default', 'value'=>NULL],

// date_published must be in a format that MySQL likes:
[['date_published'], 'date', 'format'=>'yyyy-MM-dd']
```

And here is `User`:

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

// Required fields when logging in:
[['username', 'pass'], 'required', 'on'=>'login'],

// Username must be unique and less than 45 characters:
[['email', 'username'], 'unique'],
[['username'], 'string', 'max' => 45],

// Email address must also be unique (see above), an email address,
// and less than 60 characters:
[['email'], 'email'],
[['email'], 'string', 'max' => 60],

// Password must match a regular expression:
[['pass'], 'match', 'pattern'=>'/^[a-z0-9_-]{6,20}$/i'],
// Password must match the comparison:
[['pass'], 'compare', 'compareAttribute'=>'passCompare', 'on'=>'register'],

// Set the type to "public" by default:
[['type'], 'default', 'value'=>'public'],

// Type must also be one of three values:
[['type'], 'in', 'range'=>['public', 'author', 'admin']],
```

You also have to add one attribute to `User`:

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

> *{TIP}* In real-world applications, put no restrictions on what a password can contain or its length, but I wanted to demonstrate a regular expression example here.

And here are the rules from the `File` model:

```
# models/File.php::rules()
// name, type, size are required (sort of come from the user)
[['name', 'type', 'size'], 'required'],

// User must exist in the related table:
[['user_id'], 'exist', 'targetClass'=>'User',
    'targetAttribute'=>'id'],

// Size must be an integer:
[['size'], 'integer'],

// description is optional; must be filtered
// and set to NULL when empty:
[['description'], 'filter', 'filter' => 'strip_tags'],
[['description'], 'default', 'value' => NULL],

// Maximum length on the name:
[['name'], 'string', 'max' => 80],

// Type must be of an appropriate kind:
[['type'], 'filter', 'filter' => 'validateType'],
```

Those rules also refer to the "validateType" filter. 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. Chapter 9 delves into `File`.

As for the date-type columns and the user IDs, these values are set using behaviors. Thus, each of these four classes must use the proper namespaces, meaning they'll begin like so:

```php
<?php
namespace app\models;
use Yii;
use yii\db\ActiveRecord;
use yii\behaviors\TimestampBehavior;
use yii\behaviors\BlameableBehavior;
```

Then, the `behaviors()` method in each model will return:

```php
# models/Comment.php::behaviors()
return [
    [
        'class' => TimestampBehavior::className(),
        'attributes' => [
            ActiveRecord::EVENT_BEFORE_INSERT => ['date_entered']
        ],
    ],
    [
        'class' => BlameableBehavior::className(),
        'createdByAttribute' => 'user_id'
    ],
];
```

```php
# models/Page.php::behaviors()
return [
    [
        'class' => TimestampBehavior::className(),
        'attributes' => [
            ActiveRecord::EVENT_BEFORE_INSERT => ['date_updated'],
            ActiveRecord::EVENT_BEFORE_UPDATE => ['date_updated'],
        ],
    ],
    [
        'class' => BlameableBehavior::className(),
        'createdByAttribute' => 'user_id'
    ],
];
```

```
# models/User.php::behaviors()
[
    'class' => TimestampBehavior::className(),
    'attributes' => [
        ActiveRecord::EVENT_BEFORE_INSERT => ['date_entered']
    ],
],
```

```
# models/File.php::behaviors()
return [
    [
        'class' => TimestampBehavior::className(),
        'attributes' => [
            ActiveRecord::EVENT_BEFORE_INSERT => ['date_entered',
                'date_updated'],
            ActiveRecord::EVENT_BEFORE_UPDATE => ['date_updated'],
        ],
    ],
    [
        'class' => BlameableBehavior::className(),
        'createdByAttribute' => 'user_id'
    ],
];
```

After establishing your rules, you can test some of this code by checking that the forms load properly. There's no value in submitting new records, however, as many more things must be established before the models are usable for managing database content.

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 attribute names and the labels the site should use for those attributes. 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 attribute's label is used.

The Yii framework, through Gii, does a great job of automatically generating reasonable labels for you. For example, given a column name of "date_updated", Gii will generate the label "Date Updated"; "user_id" becomes "User ID".

If the automatically generated labels aren't 100% right, customize them. To do so, just edit the values returned by `attributeLabels()`. Only edit the *values*, though, not the array indexes.

For example, in the **File.php** model, I would change the `attributeLabels()` definition to:

```
public function attributeLabels() {
    return [
        '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, all of the view files reflect the new changes (**Figure 5.3**).

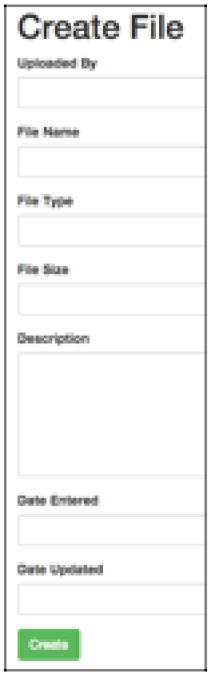


Figure 5.3: *The form for adding a new file, with its new labels.*

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

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

You'll need to add attribute names and values to models in two circumstances:

- When you have a model not based upon a database
- When you add attributes to a database-based model

With the latter, adding the `comparePass` attribute to `User` means you should add its label, too:

```php
# models/User.php::attributeLabels()
return [
    'id' => 'ID',
    'username' => 'Username',
    'email' => 'Email',
    'pass' => 'Password',
    'type' => 'Type',
    'date_entered' => 'Date Entered',
    'comparePass' => 'Password Confirmation'
];
```

Watching for Model Events

The chapter largely examines the model methods created by Gii. But there are methods *not* generated for you but still common to models. Specifically, you should be aware of:

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

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

The Component Class

Chapter 4 covers configuring *application* components, such as the database component, the "urlManager" component, and so forth. But there is another use of "component" in Yii, which is the key building block to the entire framework.

It all starts with the `yii\base\Component` class. Most of the classes used in Yii are descendants of `yii\base\Component` class. For example, the application object will be of type `yii\web\Application`. That class is derived from `yii\base\Component` (although there are other classes in between). Controllers are of type `yii\web\Controller`, which inherits from `yii\base\Controller`, which inherits from `yii\base\Component`. `yii\db\ActiveRecord` inherits from `yii\base\Model`, which inherits from `yii\base\Component` (**Figure 5.4**)

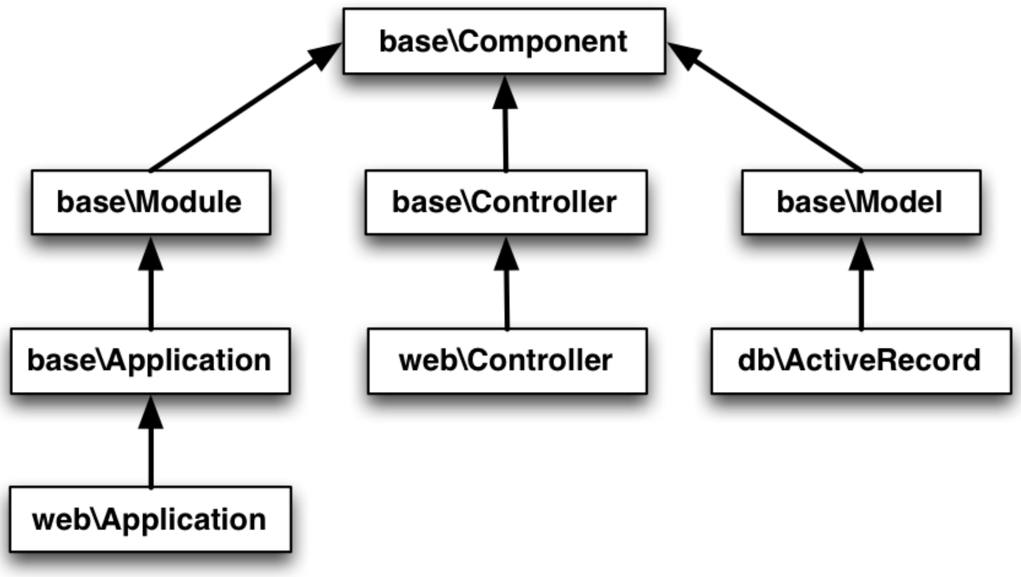


Figure 5.4: *Part of Yii's class inheritance structure, with `Component` at the top.*

Knowing the component is the basic building block is important to your use of Yii. Thanks to inheritance, functionality defined in `yii\base\Component` will be present in every derived class, which is to say most of the classes in the framework.

The `yii\base\Component` class provides three main tools:

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

Of these three, let's discuss events now. This coverage will be specific to models, but understand that any class that inherits from `yii\base\Component` 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 (e.g., through a direct link or a form submission). The result of that event occurrence is the execution of the PHP code in that script. Conversely, in JavaScript, which continues to run so long as the browser window is open, your code can 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 `yii\base\Component` class, Yii adds additional event functionality to PHP-based Web site.

> *{NOTE}* 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 object, call this function". In Yii, you can create your own events, but models have their own predefined events: before a model is saved, after a model is saved, before a model is validated, after a model is validated, and so forth. Each of the methods previously mentioned corresponds 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:

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

As in that minimal code, a best practice is 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.

Real-world examples of using an event with a model would include setting or manipulating a model's properties automatically. For example, you could apply the `trim()` function to an attribute's value before validation occurs. This would be accomplished by creating a `beforeValidate()` method that does that:

```php
# models/Page.php
protected function beforeValidate() {
    $this->content = trim($this->content);
    return parent::beforeValidate();
}
```

A more useful example might syntactically check that a page's content, which should be HTML, is valid HTML and doesn't contain certain tags, such as `<script>`.

As another example, prior to the addition of "TimestampBehavior" and "BlameableBehavior" in Yii 2, event handling methods were an ideal place to dynamically set a timestamp value to `NOW()` or assign a value to a `user_id` property.

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

Another important model topic is that of relations: how one model is related to another, which is akin to how one database table is related to another. Relationships among models are crucial, as you'll often need to fetch related data, such as the comments associated with a page.

In Yii 1, a `relations()` method reported every relationship that existed for that model. Yii 2 does things differently, using a unique, defined method for each relationship. You can see this in the Gii-generated code. Here are two methods found in `Page`:

```php
# models/Page.php
public function getComments() {
    return $this->hasMany(Comment::className(), ['page_id' => 'id']);
}
public function getUser() {
    return $this->hasOne(User::className(), ['id' => 'user_id']);
}
```

These two functions define two relationships: "comments" and "user". Note that the singular/plural form also reflects the nature of the relationship: a page can have many comments but only one user (i.e., author).

Each method returns an instance of `yii\db\ActiveQuery`. This return value is generated by the `hasOne()` or `hasMany()` method call, which identifies the relationship between the two classes. For example, in `getComments()`, 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 `comment.page_id` attribute.

Here's how this will come into play, as you'll see in Chapter 8: When loading a record for class, you can also load any of its related models, too. In the case of `Page`, an instance can load the comments associated with that page. Easily loading those comments allows the view file to display them without any other work or database queries. Furthermore, since `Comment` is related to `User`, the author'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 more detail.

Relationship Types

The possible relationships are:

- `hasOne()`
- `hasMany()`

The relationships are indicated by methods defined within the `yii\db\ActiveRecord` class. Note that this is a new approach in Yii 2; in Yii 1, constants identified relationships, and you could also specify "belongs to" and "many to many" relationships. Yii 2 has dropped "belongs to"–which is semantically just the other side of a "has one" or "has many"–and handles "many to many" relationships via *junction tables*.

The "has one" relationship works for both one-to-one and many-to-one relationships between two models (i.e., a comment only ever has one user, but each user can have multiple comments). 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 have an e-commerce site that uses subscriptions to access content, you could opt to store the subscription information separately 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, one-to-one relationships aren't common, but Yii supports that arrangement when it exists.

A properly normalized database results in a preponderance of one-to-many, or many-to-one relationships, covered by the `hasMany()` method in Yii 2.

Gii can automatically create these relation definitions–the methods–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 the relationship code for you, or if you just need to alter the relations later, simply add the right relation definitions for the situation.

Handling Many-to-Many Relationships

A third type of database relationship is many-to-many. In a normalized, relational database, a many-to-many relationship between two tables is handled by creating an *intermediary* table, also called a "junction" table. The original two tables 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.

In Yii 1, this situation was handled via the `relations()` method and the `MANY_MANY` constant. In Yii 2, a many-to-many relationship is also handled through a dedicated function. That function will call the `hasMany()` method, with an additional `via()` or `viaTable()` clause. Here's the Gii-generated code for the `Page` class:

```php
# models/Page.php
public function getFiles() {
    return $this->hasMany(File::className(), ['id' => 'file_id'])
    ->viaTable('page_has_file', ['page_id' => 'id']);
}
```

The method starts off returning the invocation of `hasMany()`, applied to the `File` class. However, the `File` class has no relationship to `Page`. Thus, this query is appended with `viaTable()`, providing the intermediary table name–`page_has_file`– and the relationship.

The `File` class has the inverse definition:

```php
# models/File.php
public function getPages()
{
    return $this->hasMany(Page::className(), ['id' => 'page_id'])
    ->viaTable('page_has_file', ['file_id' => 'id']);
}
```

Again, Gii will generate this code for you, using the foreign key constraints or comments, but it's often prudent to double-check what was created. Chapter 8 delves into model relationships in more detail, and Chapter 18, "Advanced Database Issues," covers even more advanced relationship concerns.

The Object Class

To conclude this discussion of models, let's examine the `yii\base\Object` class. This is the base class for the entire Yii framework. In fact, `yii\base\Component` and every other class in Yii inherits from `Object`.

The main feature `Object` provides is the handling of attributes. Although an object–as in an instance of a class–has its own attributes, the `Object` class provides another way to access them: via "set" and "get" methods. For example, take this class:

```php
use Yii;
use yii\base\Object;

class MyClass extends Object {

    private $_foo;

    public function getBar() {
        return $this->_foo;
    }

    public function setBar($value) {
        $this->_foo = $value;
    }
}
```

This class has two methods: `getBar()` and `setBar()`. The two methods access a private variable, `$_foo`. (I generally try to avoid foo-bar examples, but it's important to see the different names here.) This construct allows for the following:

```php
$obj = new MyClass();
$obj->setBar('hello');
echo $obj->getBar(); // prints "hello"
```

There's nothing especially novel there. However, because `MyClass` extends `Object`, the "get" and "set" methods virtually create a `bar` property that can also be accessed directly. The following has the same effect as the previous code:

```php
$obj = new MyClass();
$obj->bar = 'hello';
echo $obj->bar; // prints "hello"
```

Internally, the private `$_foo` variable is being accessed, but externally, the virtual public property is `$bar`.

This may not seem like much of a feature, but it allows for some pretty cool magic. Looking at the models, the `Page` class has a `getUser()` method that defines a relationship between the two classes. Because `Page` extends from `Object` (way up the chain), it supports using "get" and "set" methods to access virtual properties. Therefore, if you have an object of type `Page`, you can treat `user` as a property! As the `getUser()` method effectively returns a `User` object, `$page->user` returns the `User` object associated with that page. Pretty cool, eh?

Because every object in Yii extends from `Object`, any class that has a `getSomething()` or a `setSomething()` method has a virtual `something` property.

Note that these properties are case-insensitive, and to avoid confusion, it's best not to have an actual attribute and a property method with the same name. To make a read-only property, define a "get" method but no "set" method.

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 explains *models* in some detail. Models represent the data used by an application. This chapter looks at *views* in equal detail. Users interact with applications through the views. For websites, views are a combination of HTML and PHP code that create the desired output shown in the browser 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.

This chapter covers everything you need to know in order to both comprehend and work with views. It also presents several recipes for performing specific tasks. As in the previous chapter, many of the examples assume 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 are 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

By using Composer and Gii to create a new web application, you generate a series of files and folders. By default, all of the view files go in the **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 **main.php**, which is the primary template.

> *{NEW}* The basic Yii application no longer uses one-column and two-column layout variants.

For each of the controllers created by Gii–i.e., all of the directories except for **site**, you find these files:

- **__form.php**
- **__search.php**
- **create.php**
- **index.php**
- **update.php**
- **view.php**

View files are designed to be broken down 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**. 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

Chapter 3, "A Manual for Your Yii Site," discusses *routing* in Yii: how the URL requested by the browser becomes the generated page. For example, when the user goes to this URL:

http://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://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 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() {
    return $this->render('index');
}
```

> *{NEW}* In Yii 2, the output from the **render()** method must be returned by the action method. It's not enough to just invoke **render()**.

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 `yii\base\Controller` 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: **views/ControllerID/viewName.php**. In this case, "index" means that **views/site/index.php** will be rendered.

That's where the view file is referenced. Now let's look at the rendering process itself.

Layouts and Views

In order to understand how Yii creates a complete HTML page, let's compare it with how templates are used in a *non-framework* PHP site.

The Premise of Templates

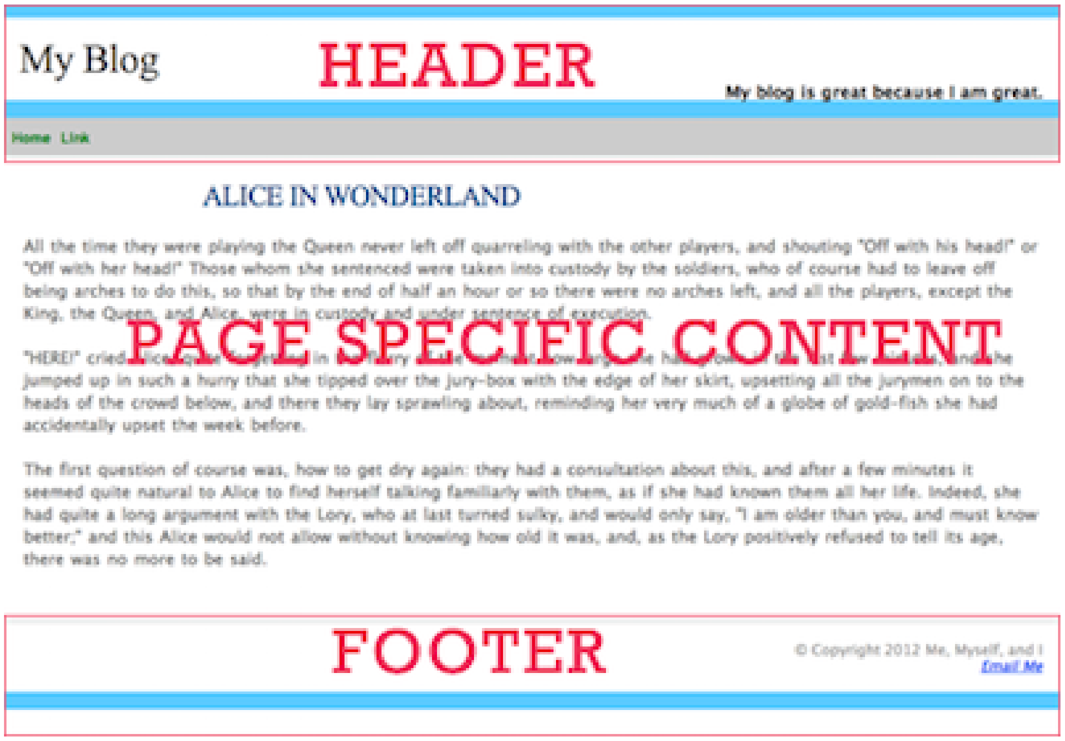
When you begin creating dynamic websites using PHP, you quickly recognize that many parts of an HTML page are repeated throughout the site. At the very least, this includes the opening and closing HTML and BODY tags. But within the BODY, there are other repeating elements: the header, the navigation, 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
<?php
include('header.html');
// Add page-specific content.
include('footer.html');
```

This is the approach one 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.

Templates in Yii

When using Yii for your site, you'll still use a template system, but it's not as 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**. The bootstrap file creates an application object and runs it. It's up to that application object to assemble all the necessary pieces. Of those pieces, the *layout* files constitute all the common elements, everything that's not page-specific.



ALICE IN WONDERLAND

All the time they were playing the Queen never left off quarreling with the other players, and shouting "Off with his head!" or "Off with her head!" Those whom she sentenced were taken into custody by the soldiers, who of course had to leave off being arches to do this, so that by the end of half an hour or so there were no arches left, and all the players, except the King, the Queen, and Alice, were in custody and under sentence of execution.

"HERE!" cried [illegible] in [illegible] she ha[illegible] the last few [illegible] and she jumped up in such a hurry that she tipped over the jury-box with the edge of her skirt, upsetting all the jurymen on to the heads of the crowd below, and there they lay sprawling about, reminding her very much of a globe of gold-fish she had accidentally upset the week before.

The first question of course was, how to get dry again: they had a consultation about this, and after a few minutes it seemed quite natural to Alice to find herself talking familiarly with them, as if she had known them all her life. Indeed, she had quite a long argument with the Lory, who at last turned sulky, and would only say, "I am older than you, and must know better;" and this Alice would not allow without knowing how old it was, and, as the Lory positively refused to tell its age, there was no more to be said.

© Copyright 2012 Me, Myself, and I

Figure 6.1: *A templated page.*

In the basic application template site, you'll find **views/layouts/main.php**. This file defines the primary page layout. Open it in a text editor or IDE, and you'll see that it begins with some PHP for the Yii framework. Let's ignore that for now.

A few lines in you get to the DOCTYPE and opening HTML tag, then the HTML HEAD and all its jazz, then this file 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.

In the middle of the body section, you'll see this line:

```
<?= $content ?>
```

More formally, this could be written as:

```
<?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 **views/site/index.php**. The output of that view file is assigned to the `$content` variable and then printed at that spot in the layout file. This is what it

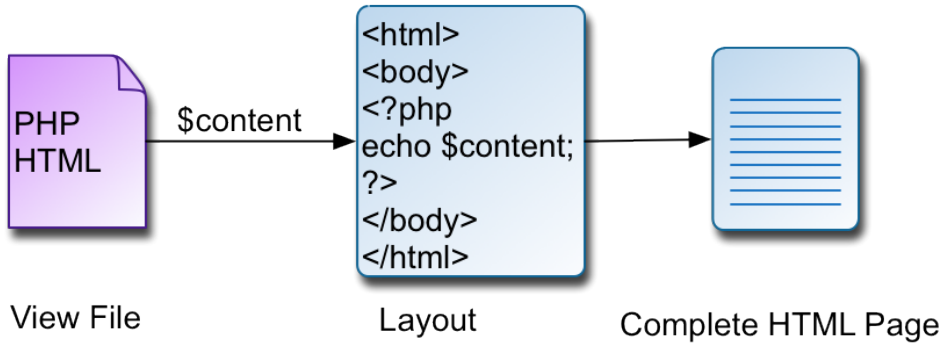


Figure 6.2: *How Yii renders a complete page by pulling an individual view file into the layout.*

means to "render" a view file. If the view file has any PHP code, that code will be executed and its results also assigned to the `$content` variable (**Figure 6.2**).

You won't find an assignment to the `$content` variable anywhere in your code. Also note that it's just `$content`, not `$this->content` or `$model->content`. Yii simply uses this variable to identify the rendered view content to 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 basics of templates in Yii, although additional bells and whistles add more complexity. I'll go into the tangential stuff later in the chapter, but for now, let's move on to the basics of editing view files. Chapter 12, "Working with Widgets," is also germane to views, as widgets generate more custom HTML, JavaScript, and CSS without embedding too much logic directly in a view file.

Editing View Files

With an understanding 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. Specifically:

- 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

This chapter won't explain in any detail 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

Through a combination of PHP and HTML, views 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, as the included code has the same scope as the page that included it.

The structure and sequence is not so straightforward in a Yii-based site. Further, you rarely create variables in the view files themselves; most of the variables used in view files 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;
    return $this->render('index');
}
```

You *might* think that **views/site/index.php** 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:

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

Now, **index.php** can reference `$num`, which has 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:

```
return $this->render('index', ['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 `$this`, a special variable in OOP that always refers to the current object. It never needs to be formally declared. Within a view file `$this` always refers to the current view. In **views/site/index.php**, `$this` refers to the current instance of the `yii\web\View` class.

> *{NEW}* This is an important change in Yii 2: *this∗∗withinaviewfilereferstotheviewinstance, notthe*
> ∗**this->context**.

The view files generated by Gii have comments at the top of them indicating the variables were passed to the view file. For **views/page/view.php**, that's:

```
<?php
/* @var $this yii\web\View */
/* @var $model app\models\Page */
```

This is a simple but brilliant touch that makes it easier to know what variables you can work with within a view. As you customize your site, 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 page 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 `actionView()` method from `PageController`:

```
public function actionView($id) {
    return $this->render('view', [
        'model' => $this->findModel($id),
    ]);
}
```

This means within the view you can access any of the model's public properties via `$model->propertyName`. With the `Page` class defined in the CMS example, you could therefore print the page title in a view using:

```
<?php echo $model->title; ?>
```

You can also call any of a model's public methods via `$model->methodName()`. For example, the `Model` class defines the `getAttributeLabel()` method which returns the label–defined in the model's `attributeLabels()` method–for the provided attribute:

```
<?php echo $model->getAttributeLabel('title')); ?>
```

Setting Page Titles

Within a view file, `$this` refers to an instance of the view that rendered the file. As `$this` will be an object, you can access any public view property via `$this->viewPropertyName`. There aren't that many of them, and certainly few you would *need* to access within a view, but `title` is useful. Its value will be placed between the TITLE tags in the HTML, and therefore used as the browser window title.

```
<?php $this->title = 'About This Site'; ?>
```

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->title = $model->title; ?>
```

By default, the `title` value will be along the lines of the action name plus the model name. For example, for the **views/user/create.php** page, the title will end up being *Create User* unless changed.

If you want to use the application's name in the title, it's available via `\Yii::$app->name`:

```
<?php $this->title = \Yii::$app->name . '::' . $model->title; ?>
```

The application's name can be set in the configuration file.

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 written as:

```
<?php foreach ($list as $value): ?>
<li><?= $value ?></li>
<?php endforeach; ?>
```

In this particular example, with so little HTML, using three separate PHP blocks may seem silly, 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.

> *{NEW}* Yii 2 takes advantage of short tags being enabled by default as of PHP 5.4, and therefore uses the shorthand echo syntax regularly.

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 or Twig.

Linking to Resources

If you look at **views/layouts/main.php**, you'll see no mention of any CSS or JavaScript files. This is odd, as the rendered source of the page includes CSS and JavaScript files. In the basic application created for you, the "Bootstrap" extension takes care of all this. If you're not using that extension, though, you'll need to know how to properly reference such resources.

The modern standard is to place the CSS files for a site within a **css** subdirectory of the web root. However, you should not use a relative path to the CSS scripts:

```
<!-- NOT THIS! -->
<link rel="stylesheet" type="text/css"
    href="css/screen.css" media="screen, projection" />
```

You may be in the habit of using relative URLs for CSS, JavaScript, and other resources on your sites, but relative URLs are tricky 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, so the user might end up at **http://example.com/index.php/site/login**. Or better yet: **http://example.com/site/login**. In both cases, the request to load the CSS file using `href="css/screen.css"` means that the browser will request the file **http://example.com/site/login/css/screen.css**, because the browser thinks its in the **site/login** directory. That file, of course, does not exist.

The solution is to use a relative path to all CSS, JavaScript, images, and so forth that begins at the web root. 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 leaves you open to another problem. You may develop a site on one server and deploy it to the live server. On the development server, the URL may be something like **http://localhost/sitename/**. In that case, the proper path would be */sitename/css/screen.css*. If you used that value on your local server, you would have to changed 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 more flexible, have the Yii application insert the proper path for you, using the “HTML” helper class. The “HTML” helper class is `yii\helpers\HTML`, and it defines a ton of useful HTML-related functionality. Its `cssFile()` method can create a proper link to CSS script:

```
use yii\helpers\Html;

<?= Html::cssFile('@web/css/main.css'); ?>
```

That code first references the namespace. Next, the `cssFile()` method is called, provided with the path to the CSS script. That specific value starts with `@web`, which is an alias to the web root. The above code will output an absolute reference to the CSS script, even if you change servers:

```
<link href="http://example.com/css/main.css" />
```

You can reference JavaScript files using the "HTML" helper class's `jsFile()` method:

```
use yii\helpers\Html;
<?= Html::jsFile('@web/js/main.js') ?>
```

Link to images using the `img()` method:

```
use yii\helpers\Html;
<?= Html::img('@web/images/logo.png', ['alt' => 'My logo']) ?>
```

That will output:

```
<img src="http://example.com/images/logo.png" alt="My logo" />
```

> *{NOTE}* You only need to reference a namespace once per page. The examples each reference it just to provide full context.

Linking to Pages

To reiterate a key concept, 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:

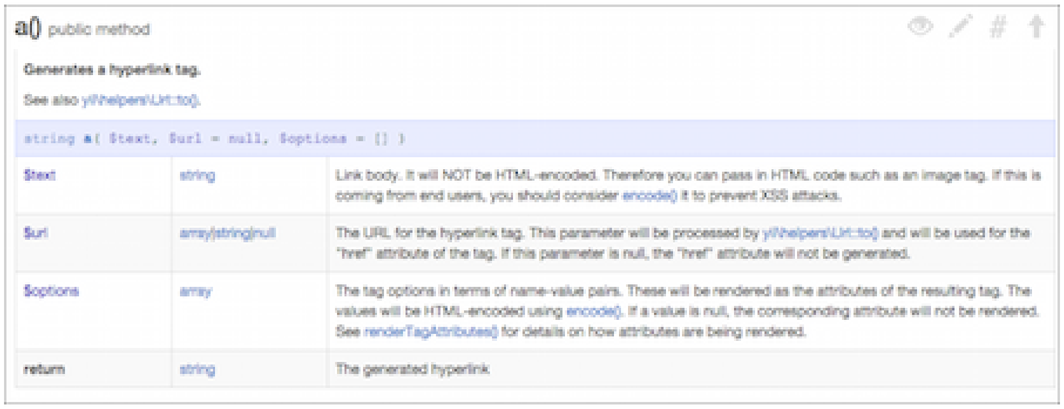
- **http://example.com/index.php?r=ControllerID/ActionID**
- **http://example.com/index.php/ControllerID/ActionID/**
- **http://example.com/ControllerID/ActionID/**

Because the URL format is dictated by the "urlManager", and as 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 URLs and links for you.

The right tool for this job is the `a()` method of the `Html` helper class (**Figure 6.3**). This particular method creates an HTML `a` tag.

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 you can use `a()` to turn an image into a link.

The second argument is the URL to use. You could provide a hardcoded value here–such as "/page/view/id/3", but that again defeats the purpose of having flexible links. Instead, 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, is "site/index":



a() public method

Generates a hyperlink tag.

See also yii\helpers\Url::to().

```
string a( $text, $url = null, $options = [] )
```

$text	string	Link body. It will NOT be HTML-encoded. Therefore you can pass in HTML code such as an image tag. If this is coming from end users, you should consider encode() it to prevent XSS attacks.
$url	array|string|null	The URL for the hyperlink tag. This parameter will be processed by yii\helpers\Url::to() and will be used for the "href" attribute of the tag. If this parameter is null, the "href" attribute will not be generated.
$options	array	The tag options in terms of name-value pairs. These will be rendered as the attributes of the resulting tag. The values will be HTML-encoded using encode(). If a value is null, the corresponding attribute will not be rendered. See renderTagAttributes() for details on how attributes are being rendered.
return	string	The generated hyperlink

Figure 6.3: *The class documentation for the `Html::a()` method.*

```php
<?php
use yii\helpers\HTML;
echo Html::a('Home', ['site/index']); ?>
```

The route to the page for creating a user (i.e., registration) is "user/create":

```php
<?php echo Html::a('Register', ['user/create']); ?>
```

The pattern is clear: routes are in the format *ControllerID/ActionID*.

Understand this only works if the route is provided as an array. If you provide a string as the second argument to `Html::a()`, it will be treated as a literal string URL value:

```php
<?php
// This result is ALWAYS site/index:
echo Html::a('Home', 'site/index'); ?>
```

That URL *may* work, depending upon how the "urlManager" is configured, but will break if you change the routing configuration.

To pass additional parameters to the route, add those to the array. This next bit of code creates a link to the page with an ID value of 23:

```php
<?php echo Html::a('Something', ['page/view', 'id' => 23]); ?>
```

The resulting output is one of the following, depending upon the configuration:

```html
<a href="/index.php?r=pageview&id=23">Something</a>
<a href="/index.php/page/view/id/23">Something</a>
<a href="/page/view/id/23">Something</a>
```

The value being linked–that which the user would click upon–can be HTML, too:

```
<?php
echo Html::a('<img src="' . \Yii::$app->request->baseUrl .
    '/img/thing.png" />',  ['page/view', 'id' => 23]); ?>
```

That code uses `\Yii::$app->request->baseUrl`, which always refers to the base HTML URL for the site, to reference the image.

The third parameter to `a()` is for setting additional HTML options. You could use this, for example, to set the link's class:

```
<?php echo Html::a('Something', ['page/view', 'id' => 23],
    ['class' => 'btn btn-info']); ?>
```

Results in:

```
<a href="/index.php/page/view/id/23"
    class="btn btn-info">Something</a>
```

Sometimes you 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 use `Html::a()` or `\Yii::$app->request->baseUrl`, although the latter does handle routing. The solution is the "URL" helper's `toRoute()` method. It returns the route created under the current configuration. As it only returns the route portion, you can append it to `Url::base(true)`, which is an absolute URL for the application's base:

```
<?php
use yii\helpers\URL;
$url = Url::toRoute(['page/view', 'id' => 23]);
echo Html::a(Url::base(true) . $url, ['page/view', 'id' => 23]); ?>
```

To link to an anchor point on a page, pass `'#' => 'anchorId'` as a parameter.

Preventing XSS Attacks

A few pages ago, a line of code seemed innocent enough but could be implemented more securely:

```
<?php echo $model->title; ?>
```

That may seem harmless, but if a malicious user entered HTML in a title, that HTML would be added to the page, assuming that tags weren't stripped out prior to storing the value. Or, if the database was hacked, HTML could be added to the model directly. If the HTML included the SCRIPT tags, the associated JavaScript would be executed when this page is loaded. That is the premise behind Cross-Site Scripting (XSS) attacks: JavaScript is injected into Site A so that valuable information about Site A's users is passed to Site B.

Fortunately, XSS attacks are ridiculously easy to prevent. In straight PHP, just send data through the htmlspecialchars() function, which converts special characters into their corresponding HTML entities. In Yii, use `Html::encode()` to perform the same role. It's just a wrapper on `htmlspecialchars()`, with the full, proper configuration. You'll see this method used liberally and appropriately in the view files generated by Gii:

```
<h1><?= Html::encode($this->title) ?></h1>
```

Note that you must include the proper namespace in your code before invoking this method:

```
use yii\helpers\Html;
```

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 `Html::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.

`Html::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 `encode()` 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 `yii\helpers\HtmlPurifier` class, a wrapper to the HTML Purifier library:

```
<?php
use yii\helpers\HtmlPurifier;
echo HtmlPurifier::process($page->content);
?>
```

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 ensure the HTML is standards compliant, which is a great, added bonus, particularly when non-web developers submit HTML content.

The biggest downside to `HtmlPurifier` is performance: it's slow and tedious for it to correctly do everything it does. For that reason, use fragment caching to cache just the HTML Purifier output in view files. Chapter 17, "Improving Performance," explains how to do that.

Working with Layouts

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. More specifically, 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 in the basic application is fine, you may want to create your own, more custom layout. By this point, you should have the knowledge to do that, but here's the sequence to take:

1. Create a new file in the **views/layouts** directory.

   I would recommend creating a new file, named something logical, 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>
  <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
  <!--[if lt IE 9]><script src="js/html5shiv-printshiv.js"
```

```
      media="all">
    </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
use yii\helpers\Html;
/* @var $this yii\web\View */ ?>
```

The first line references the necessary "HTML" helper. The second line is the kind of documentation that Gii puts into the default layout for you. Do this as well as a reminder of what variables are available to this page.

4. Replace the page's title with:

```
<?php echo Html::encode($this->title); ?>
```

This inserts the value of the view's `title` attribute between the HTML TITLE tags. For extra security, it's sent through the `encode()` method first.

5. Include your CSS files using:

```
<?= Html::cssFile('@web/css/styles.css'); ?>
```

The `cssFile()` method generates an absolute reference to a CSS script. Obviously you'll need to change the specific filename to match your site.

6. Repeat Step 5 for any JavaScript:

```
<?= Html::jsFile('@web/js/scripts.js') ?>
```

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.

Layout Events

Yii 2 adds several new lines of code to its default template (skipping many other lines here):

```
<?php $this->beginPage() ?>
<?php $this->head() ?>
<?php $this->beginBody() ?>
<?php $this->endBody() ?>
<?php $this->endPage() ?>
```

These lines all trigger events. If code in your application listens for events, these lines will notify that code's execution.

Even if you don't think you need these, you should use them in your own layouts. A widget, for example, might use the head event to know when to output the necessary JavaScript and CSS lines. In fact, the bootstrap widget used by the default layout does exactly that. That is how the outputted HTML page has JavaScript and CSS references, even though the layout file does not.

Changing Layouts

To change layouts in Yii, assign a new value to the `layout` property of the controller or application. 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 **config/web.php** file:

```
$config = [
    'layout' => 'your-layout',
```

The application's `layout` property, whose value is set here, will be the default layout used. Changing the property here impacts every controller. Note the syntax used: *your-layout*. This says to use the file named **your-layout.php**, found within the **views/layouts** directory. Change this value to match the filename of your layout file.

If different controllers are going to use different layouts, you can set a default layout in **config/web.php**, but override that value in individual controllers:

```
# controllers/UserController.php
class UserController extends Controller {
    public $layout='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:

```
# controllers/SiteController.php
public function actionIndex() {
    $this->layout = 'home';
    $this->render('index');
```

And that's all there is to it! When **views/site/index.php** is rendered, it will use the **views/layouts/home.php** template.

The following table shows all the options, in order from having the biggest impact to the smallest.

Location	Applies to
config/web.php	Every controller and view
controllers/SomeController.php	Every view in `SomeController`
SomeController::actionSomething()	The view rendered by `actionSomething()`

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 **web.php**.

Rendering Views Without the Layout

Sometimes you need 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, examine any of the "create" view files. You'll see something like:

```
<div class="page-create">
    <h1><?= Html::encode($this->title) ?></h1>
    <?= $this->render('_form', [
        'model' => $model,
    ]) ?>
</div>
```

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 **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 prepares for such situations and changes the nature of the `render()` when used within views. The views version of `render()`, unlike the controller version, does not include the layout.

> *{NEW}* The different behavior of the **render()** method in Yii 2 is a result of **$this** within a view file referring to a view instance, not a controller.

Most of the time you use `render()` within a view file, as in the create example, you'll pass along variables that view file received to the other view file.

The second common need to render a view file *without* the layout is when the view file is not outputting HTML. That would be the case if you were creating a web service that outputs plain text, XML, or JSON data. Remember that views are not just for web pages, but for any "interface" the site has. That interface could be a web service accessed by client-side JavaScript.

In cases where the goal is to output non-HTML, use `renderPartial()` within a controller. This method works the same as `render()` within controllers, but returns the rendered view file without the additional context of the layout file. You'll see examples of this later in the book.

Rendering Views From Other Controllers

By default, the file being rendered comes from the views directory associated with the current controller. For example, when updating a page record, the URL is something like **http://example.com/index.php/page/update/id/23**. This calls the `actionUpdate()` method of the `PageController` class. That method renders the "update" view, which is to say **views/page/update.php**. But there are cases where you'll render view files from other subdirectories.

For example, say you've created a `ShoppingCartController` class that handles all the logic of a shopping cart: adding and removing items, showing the contents, etc. When a purchase is completed, which could be an action of the `PurchaseController` class, the method may want to again show the cart's contents. Since the `ShoppingCartController` has a defined view for that purpose, the prudent design decision would have the `PurchaseController` view file render the `ShoppingCartController` class's view file using:

```php
// Use a simpler layout:
$this->layout = 'receipt';
// Render the shoppingCart/show.php view:
$this->render('//shoppingCart/show', ['order' => $order]);
```

Sharing Data

When working with multiple view files, whether a view file and a layout file, or multiple separate view files plus the layout, you'll often share data among them. The most common solution is a "push" approach, in which data is sent to the next view file by passing an array as the second argument to `render()`. You've already seen multiple examples of this, both from within controllers and within view files.

Because view files are part of a larger context, you can also use a "pull" approach. In a view file, `$this->context` refers to the current controller. Through that property you can access variables created in the controller, or available to it. I tend to shy away from this approach, however, as it's both too verbose and presumptive. I think it's best to be explicit and formally pass data from one file to the next.

A third option is to use the `params` property of the view. This property is commonly used to share data between a view file and the layout. The following code is in the default **about.php** script in the "Site" controller:

```php
$this->params['breadcrumbs'][] = $this->title;
```

That adds an element to the "breadcrumbs" parameter, which is an array. The layout file then uses this parameter to create the breadcrumbs near the top of the page (**Figure 6.4**).

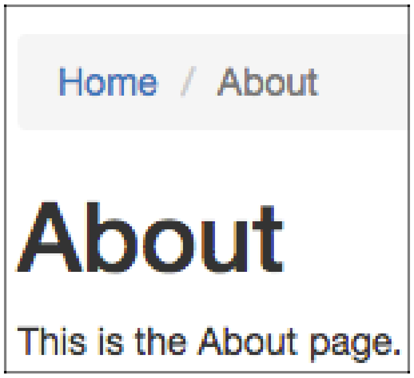


Figure 6.4: *The displayed breadcrumbs.*

```
<?= Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ?
        $this->params['breadcrumbs'] : [],
]) ?>
```

Alternative Content Presentation

There are a couple more ways you might present content to the user: using nested layouts and using blocks. Let's quickly look at both.

> *{NEW}* In Yii 1, these were known as "content decorators" and "clips", accordingly.

Using Nested Layouts

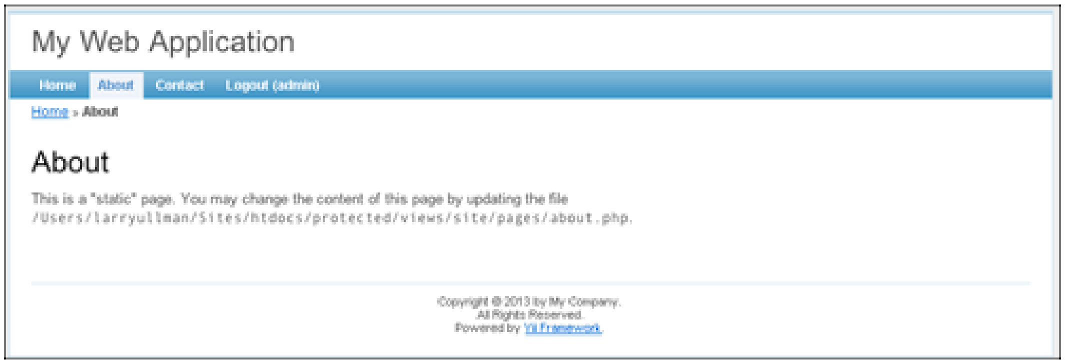
The default application in Yii 1 came with three layout files:

- **main.php**
- **column1.php**
- **column2.php**

The purpose of this structure was to easily support either a single, full column layout (**Figure 6.5**) or a two-column layout, with a sidebar (**Figure 6.6**).

The Bootstrap layout used by default in Yii 2 no longer uses this approach, but it's still easy for you to implement. This flexible structure is accomplished using nested layouts: one or more layout files is rendered within the main layout file.

Nested layouts use a view's `beginContent()` and `endContent()` methods. The `beginContent()` method takes as an argument the layout file into which *this* content should be inserted (i.e., treat this output as the value of `$content` in that view):

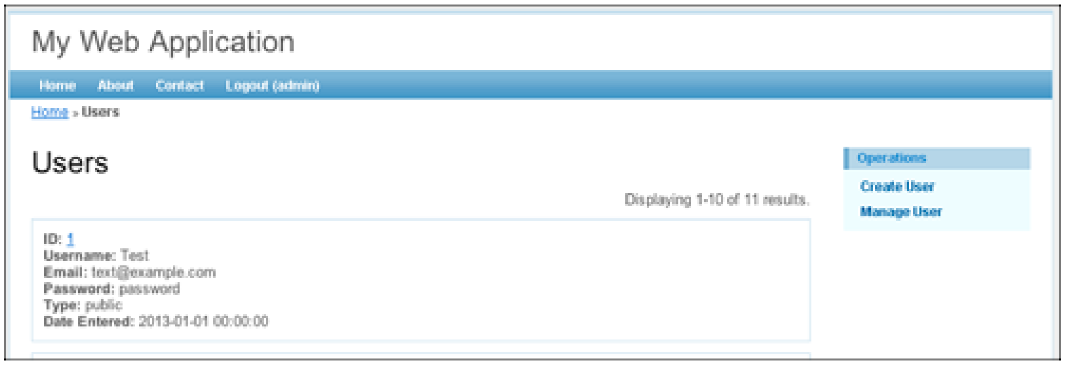


About

This is a "static" page. You may change the content of this page by updating the file /Users/larryullman/Sites/htdocs/protected/views/site/pages/about.php.

Copyright © 2013 by My Company.
All Rights Reserved.
Powered by Yii Framework.

Figure 6.5: *The page-specific content goes across the entire width of the browser.*



Users

Displaying 1-10 of 11 results.

ID: 1
Username: Test
Email: text@example.com
Password: password
Type: public
Date Entered: 2013-01-01 00:00:00

Figure 6.6: *The page-specific content shares the browser width with a sidebar.*

```
<?php $this->beginContent('//directory/file.php'); ?>
// Content.
<?php $this->endContent(); ?>
```

Returning to the Yii 1 template, say you wanted some pages in your site to use the full page width for the content (Figure 6.4), 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 nest the page specific content with the sidebar on those pages that ought to have it.

Note that only the **main.php** file creates the DOCTYPE and HTML and HEAD and so forth. The **column1.php** and **column2.php** files are nested layouts that create variations on how the page-specific content gets rendered.

If you were to implement this yourself, the **column1.php** page might look something like this:

```
<?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.

Then, the **column2.php** file starts off the same, but adds another DIV, and a widget, 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(/* Widget code */),
));
$this->endWidget();
?></div>
<!-- sidebar --></div>
<?php $this->endContent(); ?>
```

Personally I'm glad Yii 2 no longer uses this construct, as it's a bit complicated for the Yii newbie (yiibie?) to follow. But you should know this possibility exists for situations where the content around the page-specific content needs to be adjusted dynamically.

Using Blocks

Somewhat similar to nested layouts are *blocks*. Whereas a nested layout provides a way to wrap one piece of content within other content, a block provides a mechanism for dynamically defining some content that can be used as needed later. Blocks can be more dynamic than view files, and provide a way to dynamically generate content in one place, such as a view file, that will be displayed in another, such as the layout. In short, blocks are alternatives to using `renderPartial()` with a hardcoded view file.

To create a block, wrap content within `beginBlock()` and `endBlock()` calls. Provide a unique block identifier to the `beginBlock()` method invocation:

```
# views/SomeController/something.php
<?php
$this->beginBlock('stockQuote');
echo 'AAPL: $125.86';
$this->endBlock();
}
```

Anything outputted between the `beginBlock()` and `endBlock()` method calls is stored in the view file under that block identifier. Again, to be clear: that `echo` statement won't actually send the data to the browser, similar to how output buffering works in PHP.

Presumably, the block content would be more dynamic than that, such as performing a web service call to fetch the actual stock price.

To use a clip in a view or layout file, first check that it exists, and then print it:

```
<?php
# views/layouts/main.php
if (isset($this->blocks['stockQuote'])) {
    echo $this-> blocks['stockQuote'];
}
```

Chapter 7

WORKING WITH CONTROLLERS

The third main piece in MVC 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 the other fundamental aspects of this MVC component.

Controller Basics

To best understand the role that controllers play, think about what a site may be required to do. Say you have an e-commerce site: you created a model named *Product*, which represents 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 are actions that are only taken by an administrator. The last two are how customers interface with products on the site: 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 for the administrator. So you have many different uses of the same model within the site. 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 MVC, the way to address the complexity of doing many different things with one model type is to create a controller that handles all the interactions with an associated model. Hence, the `Product` model has a pal in the `ProductController`. The controller is implemented as a class. All website controllers in a Yii-based site must extend the `yii\Web\Controller` class (which, in turn, extends `yii\base\Controller`, which extends `yii\base\Component`).

> *{NEW}* Yii 2 no longer creates a base controller class from which all other controllers are derived.

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, these methods all begin with the word *action*.

The code created by Gii's scaffolding tool defines these action methods:

- `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

The generated code defines a couple other methods:

- `behaviors()`
- `findModel()`

This generated code can already handle all of the CRUD functionality for the model. This is a wonderful start to any website, and one of the reasons I like Yii. But there's much more to know and do with controllers.

This chapter explains several of these methods and, more importantly, the valuable relationship the controller has with the model and the view. You'll also learn a few new tricks, such as how to create static pages and how to define more elaborate routing possibilities.

Note that in Yii 1, basic access control–permissions as to who can do what–was built into the default application. In Yii 2, access control is no longer a given in the basic application, so discussion of that subject is now in Chapter 11, "User Authentication and Authorization."

Controller and Action IDs

All controllers and actions in Yii have an "id". This is both a colloquial and definitive way to refer to them, and how Yii knows what code should be executed for a request.

IDs in Yii are always in English, using only these characters:

- Lowercase letters
- Digits
- Underscores
- Dashes separating words
- Forward slashes

(It's uncommon to uses slashes, however.)

The controller ID gets mapped to a controller class and file by breaking multiple words on dashes, capitalizing each word, and adding the suffix "Controller". Hence, the "page" controller refers to the `PageController` class. The "shopping-cart" controller refers to the `ShoppingCartController` class.

Action IDs abide by the same rules, with two differences:

- Action IDs must be unique within a controller, but not within the entire site
- Action IDs tend to be verbs: create, update, and so on

Action IDs are then *prefixed* with the word "action": `actionIndex()` refers to the "index" action; `actionCreate()` to the "create" action.

Coupled with the controller ID, the syntax ControllerID/ActionID forms a *route*. Both the controller IDs and the action IDs are case sensitive.

Setting the Default Action

The "index" action is the default action for every controller in Yii by...um...default. When an action is not specified, the `actionIndex()` method of the controller will be called. To change that behavior, assign a different action value to the `$defaultAction` property within the controller class:

```
class SomeController extends Controller {
    public $defaultAction = 'view';
```

Use the action ID here, not the name of the method: it's just *view*, not *actionView* or *actionView()*.

With that line, the URL **http://example.com/index.php/some** calls the `actionView()` method, whereas **http://example.com/index.php/some/create** calls `actionCreate()`.

Setting the Default Controller

Although you can set the default action 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. 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.

To change the default controller, add this code to the main configuration array:

```php
# config/web.php
$confir = [
    /* Other stuff. */
    'defaultRoute' => 'YourControllerId',
    /* More other stuff. */
];
```

To just specify a controller, use the controller ID here. For example, to make "SomeController" the default, use "some" as the value.

Also note that this line must be a top-level array element being returned; it does not go within any other section. It's easiest to add it between the "basePath" and "bootstrap" elements, for clarity:

```php
# config/web.php
$config = [
    'basePath' => dirname(__DIR__),
    'defaultRoute' => 'some',
    'bootstrap' => ['log'],
    /* More other stuff. */
);
```

Because this is a route value, you can set both a default controller and action:

```php
# config/web.php
$config = [
    'basePath' => dirname(__DIR__),
    'defaultRoute' => 'some/view',
    'bootstrap' => ['log'],
    /* More other stuff. */
);
```

That line says the site should execute the "view" action of the "some" controller, should a route not otherwise be specified.

Understanding Requests and Responses

Controllers act as agents in the web application. More specifically, controllers handle *requests* and return *responses*. A request can come from the user clicking a link or submitting a form, and the response may contain an HTTP status code, one or more cookies, and the HTML itself.

Requests are formally represented in Yii as objects of type `yii\web\Request`. This object is available as the “request” component of the application object, meaning it’s found in `Yii::$app->request`. This object can be used in a slew of ways.

To confirm a GET request was made, check if `Yii::$app->request->isGet` is true. You can also use `Yii::$app->request->isPost` and `Yii::$app->request->isAjax`.

To access values found in the URL, use:

```
$get = Yii::$app->request->get();
```

To get a specific value, provide an element ID to that method:

```
$id = Yii::$app->request->get('id');
```

You might think this is functionally the same as `$id = $_GET['id']`, but Yii handles this situation more gracefully. In Yii, `Yii::$app->request->get('id')` is equivalent to `isset($_GET['id']) ? $_GET['id'] : null`. Going through the request both performs an `isset()` check and returns a null value if it’s not set.

You can take this a step further and provide a default value if a value is not set:

```
$id = Yii::$app->request->get('id', 1);
// Same as: $id = isset($_GET['id']) ? $_GET['id'] : 1;
```

Replace `get()` with `post()` and do all of those things using POSTed value instead.

There are several other properties in the “request” component that store useful information, as listed in the table, assuming the accessed URL is **http://example.com/dir/index.php/page/100**.

Property	Example
absoluteUrl	http://example.com/index.php/page/100
baseUrl	/dir
hostInfo	http://example.com
pathInfo	/page/100
queryString	(n/a, unless the URL query syntax was id=100)
scriptUrl	/dir/index.php
serverName	example.com

Property	Example
serverPort	80 (presumably)
url	/dir/index.php/page/100
userIP	(user's IP address)

For more on the request component, see the relevant section of the guide.

Controllers output responses, which are objects of type `yii\web\Response`, available in the "response" component. Through it, you can set the HTTP status code, configure headers, and even dynamically change the content of the response (as opposed to relying upon the rendering of views).

Through the "response" component, you can redirect the browser, as you would ordinarily in PHP via a "location" header:

```
# SomeController.php
public function actionOther() {
    // Whatever the reason, redirect:
    return $this->redirect('http://example.com/other', 301);
}
```

Although `$this` in the above code is controller object, its `redirect()` method is mapped onto the "response" component's `redirect()` method.

The book discusses responses in more detail later, but you can learn about it in the relevant section of the guide.

Revisiting Views

Controllers create the site's output by invoking the `render()` method:

```
public function actionIndex() {
    return $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: **views/ControllerID/viewName.php**.

The second argument to render() is an array of data to be sent to the view file. In many methods, a model instance is passed along:

```
public function actionCreate() {
    $model = new Page();
    if ($model->load(Yii::$app->request->post()) && $model->save()) {
        return $this->redirect(['view', 'id' => $model->id]);
    } else {
        return $this->render('create', ['model' => $model]);
    }
}
```

That code creates a variable named `$model` in the "create" view file.

> *{NOTE}* Remember that, as of Yii 2, **render()** returns a string of content. The action needs to return that value.

Sometimes you'll want to render a view file *without* incorporating the layout. To do that, invoke `renderPartial()` within the controller. The `renderPartial()` method is also used for Ajax calls, where the layout isn't appropriate.

Sometimes you need to render a view file from another directory (i.e., not this controller's view directory). To change the source directory, begin the view file reference with `//`, which means to start in **views**. The following code renders **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 vital to programming in Yii. This chapter presents the most standard, simple ways of doing retrieving models, and Chapter 8, "Working with Databases," delves into the more complicated options for fetching model instances.

There are three ways a controller creates 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.
```

Loading a Single Model

The second scenario–loading a single model instance from the database–is required by multiple controller actions:

- `actionView()`
- `actionUpdate()`
- `actionDelete()`

Since at least three methods perform this task, the code generated by Gii does the smart thing and defines a new controller method for that purpose:

```
protected function findModel($id) {
    if (($model = Page::findOne($id)) !== null) {
        return $model;
    } else {
        throw new NotFoundHttpException('The requested page does
        not exist.');
    }
}
```

> *{NOTE}* To enhance readability, some code presented in the book is formatted slightly differently than the original created by Yii.

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.

The `findModel()` method takes an ID value as its lone argument. This is 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:

```
$model = Page::findOne($id);
```

> *{NEW}* The **findByPk()** method from Yii 1 has been supplanted by **findOne()**.

Fetching the record is done via the `findOne()` method. This method is defined in the `ActiveRecord` class, which all database models should extend. In theory, you *could* create an instance of the class and use that instance's method:

```
$model = new Page();
$model = $model->findOne($id);
```

That would work, but it's 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 used *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::findOne()`. Yii uses static class methods all the time, such as the `Html::a()` method.

> *{TIP}* The :: is known as the scope resolution operator.

Moving on in the controller's `findModel()` method, after attempting to load the model instance, the method next checks that the value isn't NULL. If the model instance is not NULL, it's returned:

```
if (($model = Page::findOne($id)) !== null) {
    return $model;
```

If the model instance is NULL, which means that no model could be retrieved given the provided primary key value, an exception is thrown:

```
} else {
    throw new NotFoundHttpException('The requested page does
    not exist.');
}
```

The end of the chapter goes into exceptions in more detail.

Here's an example use of the `findModel()` method:

```
public function actionView($id) {
    return $this->render('view', [
        'model' => $this->findModel($id),
    ]);
}
```

Whenever you need to load a model instance in your controller, simply call the `findModel()` method, providing it with the primary key value of the record to retrieve.

Loading Every Model

The final way a controller creates model instances is to load *every* model instance–every database record. That's easily done via the `findAll()` method of the `ActiveRecord` class. This method returns an array of model objects.

Surprisingly, you won't find `findAll()` in the Gii-generated code. The action that fetches multiple models–`actionIndex()`–uses an alternative solution for fetching all the records. The `actionIndex()` method ends up using a `ActiveDataProvider` object which fetches all the model instances. That object is used by the `GridView` widget in the view file. Chapter 12, "Working with Widgets," talks about `GridView` in detail.

Handling Forms

Two of the controller methods both display and handle a form: create and update. The structure of both is virtually the same, except that one begins with a new model from scratch and the other fetches its model from the database. Here's the `actionCreate()` method:

```
public function actionCreate() {
    $model = new Page();
    if ($model->load(Yii::$app->request->post()) && $model->save()) {
        return $this->redirect(['view', 'id' => $model->id]);
    } else {
        return $this->render('create', ['model' => $model]);
    }
}
```

First, the method creates a new model instance. In `actionUpdate()`, that would be `$model=$this->findModel($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 using the "request" component. It does so by first loading all the data found in a post request and then trying to save the model. This is a two-step conditional. In the first part, if the form has been submitted, 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*. Chapter 5, "Working with Models," explains model rules and the concept of "safe".

Second, if the data passes all the validation rules, the model can be saved. If both of these conditions are true, the user is redirected to the view page where the newly created record is shown.

If the model cannot be saved, or if the form has not yet been submitted, the corresponding view file–"create" or "update"–is rendered instead, passing along the model instance.

Chapter 9, "Working with Forms," goes into forms in detail.

Understanding Routes

A topic critical to controllers, although not solely dictated within the actual controller code, is *routing*. Routes are how URLs map to the controller and action to be invoked. Chapter 3, "A Manual for Your Yii Site," introduces the basic concept and Chapter 4 explains how to configure the "urlManager" component to change route formatting. Let's now look at the topic in greater detail.

For more on routing in Yii, see the Guide.

Path vs. Get

URLs in Yii are generally going to be in one of two formats:

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

or

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

The first is the default, and is called the "get" format, as values are passed as if they were standard GET variables (because, well, they are). This format is supported in Yii without any additional configuration.

The second format is the “path” or “pretty URL” 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). Chapter 4 shows how this format is enabled in the configuration file:

```
# config/web.php
// Other stuff.
'components'=>array(
    'urlManager' => [
        'enablePrettyUrl' => true,
    ]
    // Etc.
// More other stuff.
```

This is fairly basic information. The important piece of this concept is how you define the rules for dictating the paths when using pretty URLs.

> *{NOTE}* Whether or not “index.php” is shown as part of the URL is immaterial to the rest of this discussion.

Note that, for simplicity sake, I assume 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–“http” or “https”–and the domain: “example.com/”. Thus, the rest of this explanation uses demonstration URLs without the assumed **http://example.com/** or **http://example.com/index.php**.

Route Rules

Yii properly handles the default routes—**index.php?r=ControllerID/ActionID**—without any further configuration. Moreover, Yii properly handles standard default routes when using pretty URLs without any configuration, either. But more complex routing requires defining route rules.

Route rules are defined by assigning an array of values to the “rules” subelement of the “urlManager” configuration. Each rule is defined using the syntax `'pattern' => 'route'`.

```
# config/web.php
// Other stuff.
'components'=>array(
    'urlManager' => [
        'enablePrettyUrl' => true,
        'rules' => [
            // Put rules here.
        ]
    ]
    // Etc.
// More other stuff.
```

The "urlManager" rules apply both when reading in a URL and when creating a URL (e.g., as a link). Rules serve two purposes: identifying the *route*–the controller and action–and reading in or passing along any parameters. Note that rules are parsed in order, with the first match made being used.

> *{NEW}* Yii 2 no longer includes default route rules in the basic application.

For example, here's a simple rule: `'home' => 'site/index'`. With that rule in place, the URL **http://example.com/home** will call the "index" action of the "site" controller. Further, if you create a URL to the "site/index" route, Yii will set that URL as **http://example.com/home**.

Hardcoding literal strings to routes has limited appeal. To make 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:

```
'<controller:\w+>/<action:\w+>' => '<controller>/<action>',
```

The very first part of that is `<controller:\w+>`. This means the rule is looking for what's called a regular expression "word", represented by `\w+`. In plain English, that 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–are then 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 **anyword/anotherword** with the "anyword/anotherword" route.

> *{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

Another aspect of routes are 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, and you can use the `\d+` regular expression pattern to match them. Here's a rule for handling IDs:

```
'<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". It's neither a controller nor an action. Instead, it will be passed as a parameter to the action method:

```
# 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 can use `$id`, which has a value of 42. There's one little hitch...

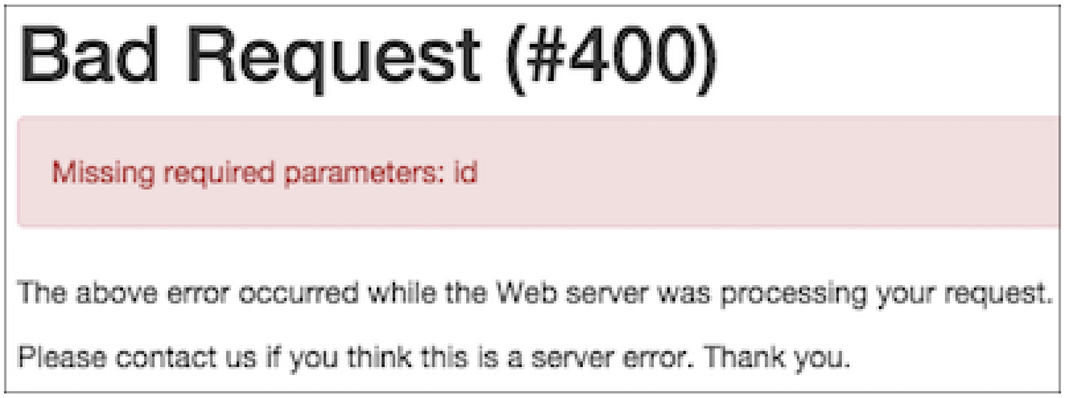
Normally, it does not matter what names you give parameters in a function:

```
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's only 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 throws an exception (**Figure 7.1**):

```
# controllers/PageController.php
public function actionView($x) {
    // Etc.
```

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 returns them to the site to verify the account. The link should pass two values in the URL that uniquely identify the user: a number and a hash (a string of characters). The rule to catch that could be:



Bad Request (#400)

Missing required parameters: id

The above error occurred while the Web server was processing your request.

Please contact us if you think this is a server error. Thank you.

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

```
'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 is written to accept two parameters, `$x` and `$y`:

```
# controllers/UserController.php
public function actionVerify($x, $y) {
    // Etc.
```

Strange as it may seem, you can put those parameters in either order, and it still works. 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, say you want a way to view a user's profile by name: **user/myUserName**, **user/yourUserName**, etc. That rule could be:

```
'user/<username:\w+>' => 'user/view',
```

That rule associates the "user/view" route with that value, and creates a named parameter of "username". (Note the regular expression pattern for matching the actual username will depend upon what characters you allow in a username.)

Here's the action, which presumably retrieves the user's profile from the database using the username:

```
# controllers/UserController.php
public function actionView($username) {
    // Etc.
```

Looking at more complicated examples, the “edit” and “update” routes are identified by this rule:

```
'<controller:\w+>/<action:\w+>/<id:\d+>' =>
    '<controller>/<action>',
```

That rule catches **page/edit/42** or **page/delete/42**, as well as **page/view/42**. Again, each action is defined so it takes a parameter specifically named `$id`.

The rules are tested in order from top down, so the first rule that constitutes a match is the route. Write your rules from most specific to most general. But, for better performance, try to have as few rules as possible.

Finally, know that route rules are case-sensitive by default. This means that although the regular expression `\w+` will match *Page*, *page*, or *pAge*, only *page* will match a controller in your application.

Advanced Routing

Routes can be even more flexible, for example:

- Using strict parsing
- Having default parameter values
- Configured for HTTP verbs (such as POST, PUT, and GET)
- Supporting virtual file extensions, like **page/create.html**

For details on any of these, see the Yii Guide.

Creating URLs

Routing rules come into play when parsing URLs into routes and also when creating URLs based upon routes. For any page or resource that’s run through the bootstrap file, always have Yii create those URLs! This is done using the `to()` method of the `yii\helpers\Url` class.

Within a controller, or within its view files, access this method like so:

```
use yii\helpers\Url;
$url = Url::to('route');
```

This method just creates and returns a URL, not an HTML link. HTML links are created by the `a()` of the “HTML” helper class.

The first argument to `to()` is a string or array indicating the route. The current controller and action are assumed, so `to('')` returns the URL for the current page.

If you just provide an action ID, you get the URL for that action of the current controller:

```php
# controllers/PageController.php
public function actionDummy() {
    $url = Url::to('index'); // page/index
```

If you provide a controller, the URL is to that route:

```php
# controllers/PageController.php
public function actionDummy() {
    $url = Url::to('user/index'); // user/index
```

If the URL expects named parameters, add those to the array:

```php
# controllers/PageController.php
public function actionDummy() {
    $url = Url::to(['view', 'id' => 42]); // page/view/42
```

By default, `to()` creates a relative URL. To create an absolute URL, provide "true" as the second argument:

```php
# controllers/PageController.php
public function actionDummy() {
    $url = Url::to('index', true); // http://example.com/page/index
```

Common URL Needs

Besides the `to()` method, the "URL" helper class has defined a few helper methods for common URLs needed in an application. For example, `Url::home()` returns the site's home page and `Url::base()` returns the base URL of the application, which may include any folders (e.g., if the application is installed in a subfolder of the root directory).

The "URL" helper has built-in memory, too. Invoke the `remember()` method to have Yii remember the current URL:

```php
# controllers/PageController.php
public function actionDummy() {
    Url::remember();
```

You can also pass a route to it, the same as you would to `to()`, to have Yii remember a different URL.

Then, you can later reference the remembered URL using `previous()`.

You'd logically use these when a user attempts to access a page that requires authentication. Simply remember the attempted page before the login, and redirect the user to the `previous()` URL upon successful login.

Tapping Into Filters

Another method defined in controllers by Gii is `behaviors()`. Chapter 5 introduce behaviors: a way to add to class A functionality defined in Class B. Controllers use different behaviors than models, as the roles for each differs. Here's the default `behaviors()` method created by Gii:

```
public function behaviors() {
    return [
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'delete' => ['post'],
            ],
        ],
    ];
}
```

That code identifies the `VerbFilter` class to implement the "verbs" behaviors.

Filters identify code to be executed before or after an action is executed. The most common filter, "AccessControl", is run prior to an action, and confirms that the user has authority to perform the action in question. Chapter 11 makes extensive use of it.

> *{NEW}* In Yii 2, filters are essentially behaviors for controllers.

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

The "verbs" filter serves this first role. The code above restricts uses of the "delete" action to POST requests. You can expand it by dictating access to the other methods:

```
public function behaviors() {
    return [
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'index' => ['get'],
                'delete' => ['post'],
                'view' => ['get'],
            ],
        ],
    ];
}
```

Showing Static Pages

Moving on, let's look at a different way of rendering view files: using *static pages*. The difference between a static page and a standard page is a static page does not change based upon any input. Whereas a dynamic page might display information based upon a provided model instance, a static page conventionally displays hard-coded HTML.

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:

```
<!-- # views/some/about.php -->
<h1>About Us</h1>
<p>spam, spam, spam...</p>
```

And:

```
# controllers/SomeController.php
public function actionAbout() {
    $this->render('about');
}
```

With that code, the URL **http://example.com/some/about** always load that about page. This is a simple approach, and familiar, but less maintainable the more static pages you have.

An alternative and more flexible solution is to register a "page" action associated with the `yii\web\ViewAction` class. This is done via the controller's `actions()` method:

```
# controllers/SiteController.php
public function actions() {
    return [
        'page' => ['class' => 'yii\web\ViewAction']
    ];
}
```

> *{TIP}* You can have any controller display static pages, but it makes sense to do so using the "site" controller.

The `ViewAction` class defines an action for displaying a view based upon a parameter. By default, the determining parameter is `$_GET['view']`, so the URL **http://example.com/site/page/view/about** is a request to render the static **about.php** page.

By default, `ViewAction` will pull the static file from a **pages** subdirectory of the controller's view folder. To complete this process, create your static files within the **views/site/pages** directory.

If, for whatever reason, you want to change the name of the subdirectory from which the static files are pulled, assign a new value to the `viewPrefix` attribute:

```
# controllers/SiteController.php
public function actions() {
    return [
        'page' => [
            'class' => 'yii\web\ViewAction',
            'viewPrefix' => '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 **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, `ViewAction` will attempt to display an **index.php** static file. To change that default, assign a new value to the `defaultView` property:

```
# controllers/SiteController.php
public function actions() {
    return [
        'page' => [
            'class' => 'yii\web\ViewAction',
            'defaultView' => 'about']
    ];
}
```

To change the layout used to encase the view, assign the alternative layout name to the `layout` attribute in that array.

Handling Exceptions

One of the great things about OOP is that you'll never see another error, although there are exceptions (pun!). Exceptions are errors turned into object variables, that's all. This is fairly standard OOP stuff, but you need to be comfortable with exceptions to properly use the framework.

In many situations, the framework itself will automatically create an exception for you, 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 HttpException('Something went wrong');
}
```

This chapter has shown similar code already. When a controller's `findModel()` method doesn't find a matching database record, it throws a `NotFoundHttpException`:

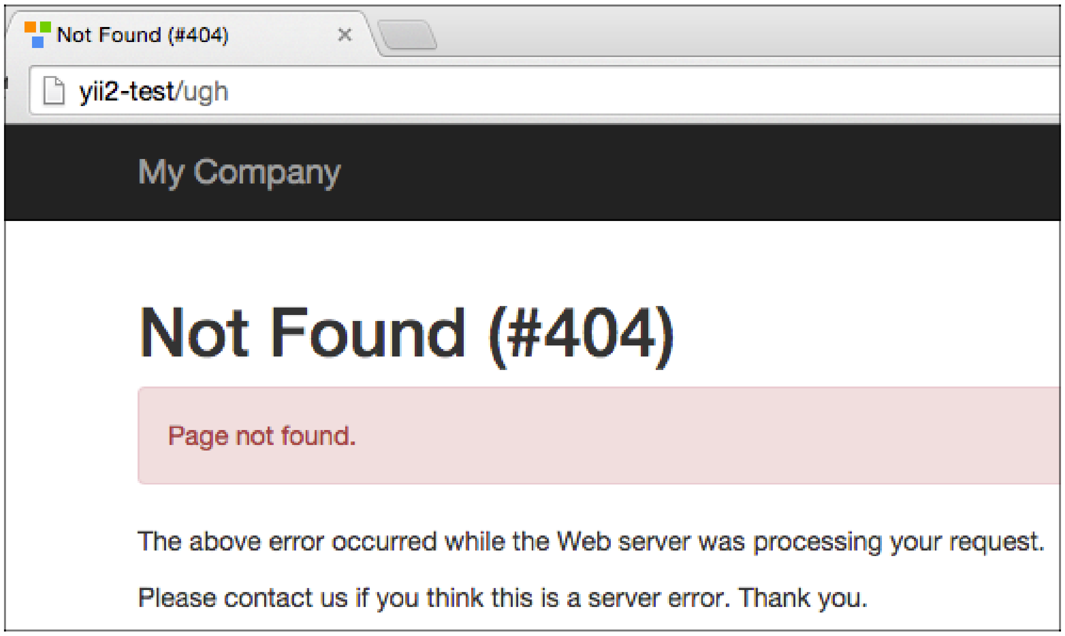
```
# models/Comment.php
protected function findModel($id) {
    if (($model = Comment::findOne($id)) !== null) {
        return $model;
    } else {
        throw new NotFoundHttpException('The page does not exist.');
    }
}
```

Yii defines several classes for exceptions, including:

- `BadRequestHttpException`
- `ForbiddenHttpException`
- `HttpException`
- `ServerErrorHttpException`
- `UserException`

See the API documentation for details on these, but all are pretty self-explanatory. When creating an exception, 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. If you don't, Yii will use the logical HTTP status code accordingly. For example, the `NotFoundHttpException` class is for exceptions related to not found HTTP requests. By default, Yii will use the 404 code for it (**Figure 7.2**).



Not Found (#404)

Page not found.

The above error occurred while the Web server was processing your request.

Please contact us if you think this is a server error. Thank you.

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

In standard PHP code, exceptions are used with `try...catch` blocks, where any exception that occurs within the `try` block are 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. Understand that when an exception is thrown, whether by you or automatically by the framework, no subsequent code will be executed.

When an exception occurs, Yii looks for a corresponding view file to display it. This will be either **error.php** or **exception.php**. The difference is that **error.php** is meant to be shown to end users, without detailed debugging information, when `YII_DEBUG` is false. Conversely, **exception.php** applies when `YII_DEBUG` is true, and reveals to you, the developer, detailed debugging information.

The framework will look for the view files in the **views/errorHandler** directory. No such directory or files will exist by default, but you can create them to customize their appearance. Until you've created your own, Yii will use a default files, found within the framework itself (**vendor/yiisoft/yii2/views/errorHandler**). This is true regardless of the controller in which the exception actually occurred.

You can change the error handling behavior by configuring the “errorHandler” component in your configuration file:

```
# config/web.php
// Other stuff.
'components' => [
    'errorHandler' => ['errorAction' => 'ControllerId/ActionId']
]
```

See the Yii Guide for more details on custom error handling.

As a convenience, Yii will automatically log exceptions to **runtime/logs/app.log**, or your other default log file. This allows you to go back in and view all the exceptions that occurred. Of course, you can also view errors in the Yii Debugger.

Chapter 8

WORKING WITH DATABASES

A database is at the core of most dynamic web applications. Previous chapters introduce the basics database interactions with 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 provides much of the needed functionality for most sites. But in more complicated sites, you need to interact with the database in more low-level or custom ways.

This chapter covers the remaining core concepts when it comes to interacting with databases using Yii. Part 3, "Advanced Topics," walks through a handful of other subjects related to databases, although those are far more complex and less commonly needed.

In Yii, there are often many ways to accomplish the same task. Attempting to explain every possibility can be confusing, in my experience, so, for the sake of clarity, this chapter focuses on the most practical and common approaches and methods.

> *{NEW}* Almost everything related to databases is new in Yii 2, at least in terms of the code being executed.

Debugging Database Operations

Whenever you work with a database, you introduce more possible causes of errors. Consequently, you must acquire 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. Then 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 likely won't be directly writing the underlying SQL commands. Thankfully, Yii is still be quite helpful, if you know where to look.

Chapter 3, "A Manual for Your Yii Site," introduces the Yii Debugger, new in Yii 2. This is an invaluable tool, and a most welcome addition. One feature of the Yii Debugger shows the number of queries executed on a given page and how long they took to run (**Figure 8.1**).

Figure 8.1: *This page required 3 queries.*

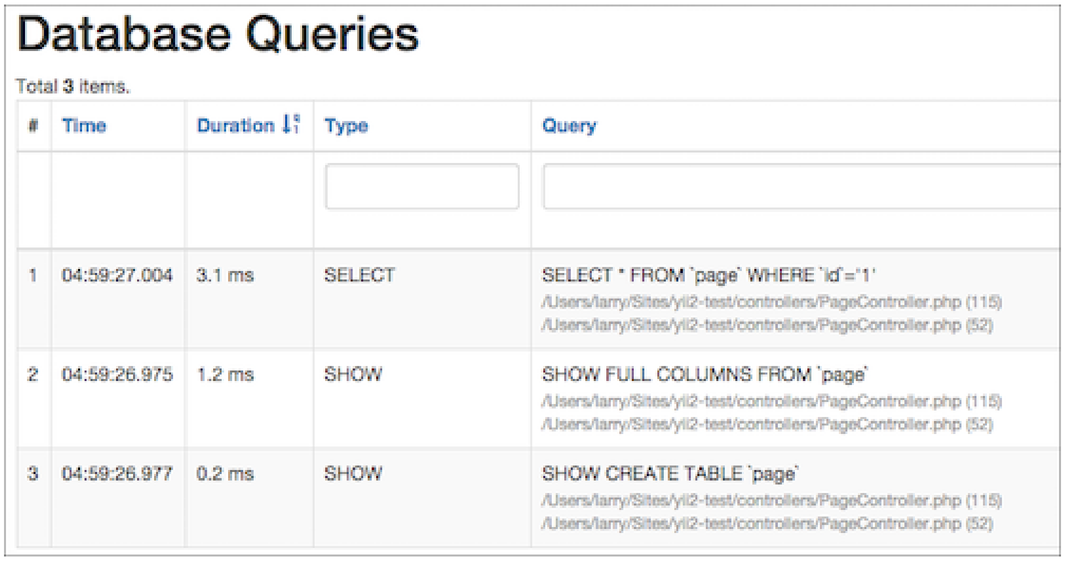
Click on that section of the Debugger and you'll be taken to a page that details those queries (**Figure 8.2**).

Many queries use parameters that separate the SQL core of the query from the specific–and often user-provided–values. The Yii Debugger is smart enough to show these parameters, too. The only caveat is that the Yii Debugger may not be shown if an exception occurred, depending upon the exception type. If so, you'll often see the necessary details in the exception output.

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, as well. And it is the only database application used in this book. Yii can work with other database applications, too, including:



Database Queries

Total **3** items.

#	Time	Duration	Type	Query
1	04:59:27.004	3.1 ms	SELECT	SELECT * FROM `page` WHERE `id`='1' /Users/larry/Sites/yii2-test/controllers/PageController.php (115) /Users/larry/Sites/yii2-test/controllers/PageController.php (52)
2	04:59:26.975	1.2 ms	SHOW	SHOW FULL COLUMNS FROM `page` /Users/larry/Sites/yii2-test/controllers/PageController.php (115) /Users/larry/Sites/yii2-test/controllers/PageController.php (52)
3	04:59:26.977	0.2 ms	SHOW	SHOW CREATE TABLE `page` /Users/larry/Sites/yii2-test/controllers/PageController.php (115) /Users/larry/Sites/yii2-test/controllers/PageController.php (52)

Figure 8.2: *The actual queries executed.*

- PostgreSQL
- SQLite
- Microsoft SQL Server
- Oracle
- MariaDB

Yii supports alternative storage solutions, including MongoDB, ElasticSearch, Redis, and Sphinx, via extensions.

To change database applications, modify the "dsn" value in the **db.php** 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.

Having configured Yii for your database application and specific database, the "db" component mostly acts as an access point to database interactions.

Regardless of the database application to which you connect, there are three ways in Yii to interact with it:

- Active Record
- Database Access Object (DAO)
- Query Builder

The Active Record approach is what the book has used thus far. For example, the previous chapter explains this line of code:

```
$model = Page::findOne($id);
```

That line performs a SELECT query, retrieving one record using the primary key value.

The two alternatives to Active Record are Data Access Objects (DAO) and the Query Builder. To best understand the three options, when you would use them, and how, let's look at each in great detail. This chapter assumes you've created the CMS example first mentioned in Chapter 2, "Starting a New Application."

Using Active Record

If you're reading this book sequentially, you've already learned about and used Active Record. In Yii, Active Record is implemented in the `yii\db\ActiveRecord` class. Every model based upon a database table will extend `ActiveRecord` by default and, thus, further discussion of it is worthwhile.

Active Record is simply a common architectural pattern for relational databases, first identified by Martin Fowler in 2003. 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 `ActiveRecord` class therefore can represent a single record from the associated database table.

Active Record provides CRUD–Create, Read, Update, and Delete–functionality for database records. It 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 is created like any object in PHP:

```
$model = new Page();
```

Assuming this is the CMS example, a new page record can then be created by assigning values to the properties. The class properties correspond directly to database columns:

```
$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 form values, but you could hardcode value assignments as in the above.

If your model has default attribute values, you can populate those from the model's rules:

```
$model = new Page();
$model->loadDefaultValues();
```

To create the new record in the database, call `save()`:

```
$model->save();
```

The `save()` method is also how you update an existing record, after having changed the necessary property values. To differentiate between inserting a new record and updating an existing one, invoke the `getIsNewRecord()` method, which returns a Boolean. The catch is 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.';
    }
}
```

Understand that `save()`, for either new or existing records, invokes validation of the model data first, based upon the model's rules. Chapter 5, "Working with Models," goes through model rules in detail. If all the rules pass, the save succeeds. Otherwise, `save()` will return false, and the model's `errors` property will reflect what went wrong:

```
if ($model->save()) {
    // Whohoo!
} else {
    // Use $model->errors
}
```

The `errors` property will be an array of errors.

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 uses 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 `mysqli_insert_id()`.

In Yii, it's so stunningly simple to find the automatically generated ID value. After saving the record, just reference the primary key property:

```
$model->save();
// Use $model->id
```

It's that simple.

Dirty Attributes

New in Yii 2 is identification of *dirty attributes*. Dirty attributes are model values that differ from those stored in the database:

```
$model = new Page();
$model->user_id = 1;
$model->title = 'This is the title';
// And so forth.
$model->save();
$model->title = 'This is the new title';
```

At this point, "title" is a dirty attribute.

You'll most commonly have dirty attributes after using a form to update a model (e.g., when changing a password). Yii tracks the original values in order to identify changed ones, and only the dirty attributes are updated in the database.

If need be, you can see which attributes are dirty by invoking `getDirtyAttributes()` on the model. This allows you to, say, identify if the user changed the email address or the password or something else. The `getOldAttribute()` method can retrieve the previous value.

Retrieving A Record

An example of retrieving existing records using Active Record has already been explained:

```
$model = Page::findOne($id);
```

In that example, the `findOne()` method is provided with a primary key value and returns a single row. If no matching primary key exists, the method returns NULL. The `findOne()` method can be used to find single rows based upon other criteria–other column values, but using the primary key makes the most sense.

Retrieving All Records

The `findAll()` method retrieves one or more rows when provided with a primary key value, an array of primary key values, or an array of columns and values. The `findAll()` method returns an array of objects, if one or more records match. If no records match, `findAll()` returns an empty array.

```
$model = Page::findOne($id);
$model = Page::findAll($id); // Same result
$models = Page::findAll([1, 2, 3]);

// All pages where user_id=1 and page is live:
$models = Page::findAll(['user_id' => 1, 'live' => 1]);
```

Although that last example will work, the most common method you'll use to flexibly retrieve rows is `find()`, to be explained in a few pages.

Deleting Records

So far, you've seen how to effectively perform INSERT, UPDATE, and SELECT queries using different Active Record methods. Sometimes you'll need to run DELETE queries, which is easily done.

If you have a model instance, you can remove the associated record by invoking the `delete()` method on the object:

```
$model = Page::findOne($id);
$model->delete();
```

Understand that even though that code deleted the database record, the model object and its attribute values remain until the object variable is unset (e.g., when a function or script terminates).

If you don't yet have a model instance, you can remove any records you want by calling `deleteAll()`:

```
$model = Page::deleteAll(['id' => $id]);
```

The `deleteAll()` method takes the same arguments as a `where()` conditional.

> *{TIP}* The **updateAll()** method can be used like **deleteAll()** to update multiple records at once.

Using find()

The `find()` method is the most potent and flexible Active Record method for fetching records into model instances. This method is interesting in that it returns an `ActiveQueryInterface`. What this means is that you can filter or sort the retrieval using additional clauses. The general syntax is `find()`, followed by any number of query building methods, followed by a query method.

The query building methods are:

- `select()`
- `from()`
- `where()`
- `andWhere()`
- `orWhere()`
- `orderBy()`
- `groupBy()`
- `having()`
- `limit()`
- `offset()`
- `join()`
- `union()`

These are all rather self-explanatory, and well-documented in the Yii guide, so I won't go into them in too much detail here. Moreover, because `find()` returns Active Record objects–e.g., an instance of type `Page`, there's little logic in restricting what columns are selected, using a JOIN, or aggregating results. Still, you will occasionally use `find()` like so:

```
$user = User::find()
    ->where(['email' => 'this@example.com'])
    ->one();
```

> *{TIP}* Most of these methods are also used for Query Builder and DAO, so see those sections of the chapter for more information.

The query methods, such as `one()` in that example, dictate what's returned:

- `all()` returns every matching record
- `one()` returns only one matching record
- `column()` returns only the first column of every matching record
- `scalar()` returns only the first column of the first matching record
- `exists()` returns a boolean indicating if any results were returned
- `count()` returns the result of a COUNT selection

The chapter will provide more examples of these query methods in just a few pages.

Counting Records

Sometimes, you don't need rows of data, but to just determine how many rows apply to the given criteria. If you just want to know how many rows *would be* found by a query, use the `count()` query method:

```
// Find the number of registered users:
$users = User::find()->count();

// Find the number of "live" pages:
$pages = Page::->find()->where('live=1')->count();
```

The second example 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 do want to confirm that at least one row would be, use `exists()`:

```
$user = User::find()->where(['email'=>$email])->exists();
if ($user) {
    $message = 'That email address has already been registered.';
} else {
    $message = 'That email address is available.';
}
```

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. 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 `ActiveRecord` methods for fetching records, how you fetch the necessary information from the related table is not obvious. But Yii is very well designed, and has two good solutions to this dilemma.

The first thing to do is make sure you've properly identified all of your model relationships in your model methods. Chapter 5 goes through this in detail. Remember that the names of the relation-identifying methods become the names used to access related values. For example, `Page` has a `getUser()` method, that creates a virtual `user` property for `Page` objects.

With relationships defined, you can use *lazy loading* or *eager loading* to reference values from related tables.

Lazy loading triggers Yii to perform a secondary query on demand:

```
// Perform one query of the comment table:
$comment = Comment::findOne(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` through the `getUser()` method. 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. Worse yet, with a page listing 39 comments, there would be 40 total queries: one for all the comments and one for each username!

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:

```
// Perform one query of the comment table:
// And one of the user table:
$comment = Comment::find()->with('user')->where(['id' => $id])->one();
$user = $comment->user->username;
```

The value 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. Note that in the above, two queries are still executed–SELECT from comment and SELECT from user, but fetching every comment and every related user still only executes two queries.

As another example, the `Page` class in the CMS example is related to `User`, in that each page is owned by a user:

```
# models/Page.php
public function getUser() {
    return $this->hasOne(User::className(), ['id' => 'user_id']);
}
```

This means you can fetch every page with every page author in one step:

```
$pages = Page::find()->with('user')->all();
```

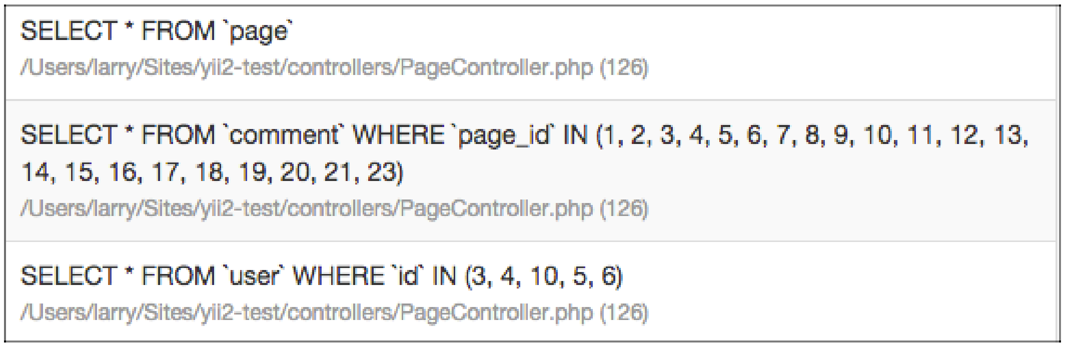
Of the two approaches, eager loading is clearly better for a couple of reasons. First, if you know you’ll want access to related data, you should overtly request it (i.e., it’s bad programming form to rely upon lazy loading). Second, eager loading is far more efficient. When Yii performs lazy loading, it runs separate SELECT queries for each property reference: in the above, one on the `page` table and another for each `user`. With eager loading, Yii performs just two queries.

> *{NOTE}* Yii 2 does not perform a JOIN by default when using **with()**.

Getting a bit more complicated, you can use `with()` on *multiple* tables. Just pass the other relation names to `with()`, separated by commas:

```
$page = Page::find()->with('comments', 'user')->all();
$user = $page[0]->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 as both the `Comment` model and the `Page` model have a relationship named "user". To understand the result, let's check out the executed query thanks to the Yii Debugger (**Figure 8.3**).



```
SELECT * FROM `page`
/Users/larry/Sites/yii2-test/controllers/PageController.php (126)

SELECT * FROM `comment` WHERE `page_id` IN (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 23)
/Users/larry/Sites/yii2-test/controllers/PageController.php (126)

SELECT * FROM `user` WHERE `id` IN (3, 4, 10, 5, 6)
/Users/larry/Sites/yii2-test/controllers/PageController.php (126)
```

Figure 8.3: *The various SELECT queries.*

If you look at the queries and the database records, you'd discover the user values being selected match the authors of pages, not the authors of comments. 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. The reference to just "user" above grabs the page's user. To grab the comment's user, too, use dot notation:

```
$page = Page::find()->with('comments.user', 'user')->all();
$user = $page[0]->user->username;
$commenter = $page[0]->comments[0]->user->username
```

The "comments.user" reference invokes the `getComments()` method of the `Page` class, then the `getUser()` method of the `Comment` class.

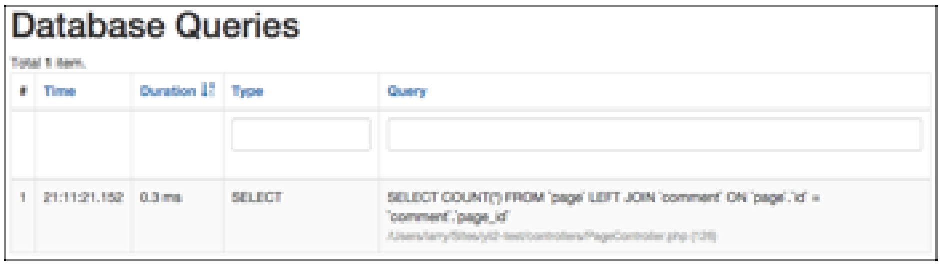
Forcing JOINs

Yii 2 does not perform JOINs even when you mention relations, instead opting to perform two separate SELECT queries. In some situations, JOINs are necessary, though. For example, to fetch only the pages that have comments requires a JOIN on the two tables.

To force a JOIN, use the `joinWith()` method instead of `with()`:

```
$pages = Page::find()->joinWith('comments')->all();
```

This code executes one query, as the Yii Debugger shows (**Figure 8.4**).



Database Queries

Total 1 item.

#	Time	Duration	Type	Query
1	21:11:21.152	0.3 ms	SELECT	SELECT COUNT(*) FROM `page` LEFT JOIN `comment` ON `page`.`id` = `comment`.`page_id` /Users/larry/Sites/yii2-test/controllers/PageController.php (126)

Figure 8.4: *The executed JOIN.*

By default `joinWith()` performs an (outer) RIGHT JOIN. To perform an INNER JOIN, use `innerJoinWith()`.

If you don't want to select every column from the joined tables, use the `select()` method to tweak the columns returned. This code fetches every page and a count of associated comments:

```
$pages = Page::find()->joinWith('comments')
->select('page.*, COUNT(comment.id)')
->groupBy('page_id')->all();
```

The equivalent query is:

```
SELECT `page`.*, COUNT(comment.id) FROM `page` LEFT JOIN `comment`
ON `page`.`id` = `comment`.`page_id` GROUP BY `page_id`
```

As another example, to fetch the associated comments in order of ascending comment date, you would do this:

```
$pages = Page::find()->joinWith('comments')
->orderBy('comment.date_entered DESC')->all();
```

I'll return to JOINS later in the chapter, and you can also see the Active Record guide for more.

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...to also use the table name as shown in the previous examples.

That is all. But this is a change from Yii 1, which used `t` as an automatic alias.

Using Database Access Objects

Active Record provides the most common way to interact with the database, but not the only avenue. Another option for database interactions is via Data Access Objects (DAO). This is a wrapper to 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

To use DAO, one starts by creating a `yii\db\Command` object. This is done through the database connection, available in `Yii::$app->db`, which is a reference to the "db" component configured in the application. To create the `Command` object, pass the SQL command you'll want to execute:

```
$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';
}
```

Alternatively, have DAO help build the queries being executed by invoking the `insert()`, `update()`, and `delete()` methods on `createCommand()`. These methods all take the table name as the first argument. The `delete()` method takes the WHERE condition as its second:

```
$cmd = Yii::$app->db->createCommand()->delete('file', ['id' => $id]);
$cmd->execute();
```

The `insert()` method takes an array of column=>value pairs as its second argument:

```
$cmd = Yii::$app->db->createCommand()->insert('some_table',
    ['some_col'=>$val1, 'num_col' => $val2]);
```

Understand that by passing values in an array as in the above, Yii automatically takes care of parameter binding for you, so you don't have to worry about SQL injection attacks.

The `update()` method takes an array of column=>value pairs as its second argument, and the WHERE condition as its third:

```
$cmd = Yii::$app->db->createCommand()->('some_table',
    ['some_col'=>'blah', 'num_col' => 43], ['id'=>$id]);
```

> *{WARNING}* If you insert or update records using DAO, you don't get the benefits of data validation that Active Record offers.

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 `queryAll()`:

```
$q = 'SELECT * FROM table_name';
$cmd = Yii::app()->db->createCommand($q);
$result = $cmd->queryAll();
```

The `queryAll()` method returns a `yii\db\DataReader` object, which you can use in a loop:

```
foreach ($result as $row) {
    // Use $row['column_name'] et al.
}
```

If your query only returns a single row, use `queryOne()` instead:

```
$q = 'SELECT * FROM table_name WHERE id=1';
$cmd = Yii::app()->db->createCommand($q);
$row = $cmd->queryOne();
```

When you have a query that only returns a single value, 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. 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.
```

This allows you to fetch results as an object type of your choosing, as if you had used Active Record.

Parameter Binding

To prevent SQL injection attacks through DAO, you can either pass values in arrays, or use bound parameters. You can use named or unnamed parameters–question marks–in place of variables in your query. I would recommend you go the named route. Use unique identifiers as the placeholders in your query, then bind those to variables using the `bindValue()` method:

```
$q = 'INSERT INTO table_name (col1, col2) VALUES (:col1, :col2)';
$cmd = Yii::$app->db->createCommand($q);
$cmd->bindValue(':col1', $some_var, PDO::PARAM_STR);
$cmd->bindValue(':col2', $other_var, PDO::PARAM_INT);
$cmd->execute();
```

The data type is identified by a PDO constant:

- `PDO::PARAM_BOOL`

- `PDO::PARAM_INT`
- `PDO::PARAM_STR`
- `PDO::PARAM_LOB` (large object)

Depending upon a few factors, there may also be a performance benefit to using bound parameters.

Using Query Builder

With Active Record on one end of database abstraction and Database Access Objects on the other, Yii's Query Builder falls somewhere in the middle. 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 or DAO, many of the same ideas, and all the query building methods,= apply to Query Builder, too.

> *{NEW}* As of Yii 2, Query Builder is intended solely for SELECT queries.

Building SELECT Queries

To use Query Builder, start by creating a new `yii\db\Query` object:

```
use yii\db\Query;
$q = new Query();
```

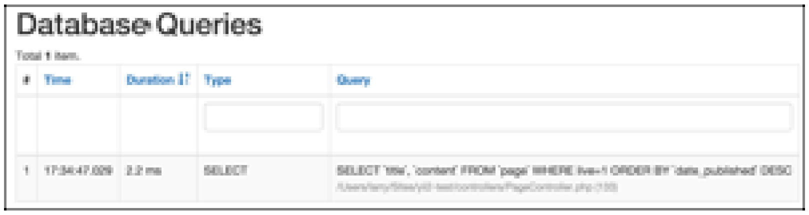
Then build up the query by invoking a series of methods, indicating what columns to select from what tables using what criteria, and so forth.

Method	Sets
from()	The table(s) to be used
join()	A JOIN
limit()	A LIMIT clause without an offset
offset()	A LIMIT clause with an offset
orderBy()	The ORDER BY clause
select()	The columns to be selected
where()	A WHERE clause

There are also properties for creating more complex queries that use GROUP BY clauses, UNIONs, and so forth.

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**):

```
$q->select('title, content')
->from('page')
->where('live=1')
->orderBy('date_published DESC')
->one();
```



Database Queries

Total 1 item.

#	Time	Duration	Type	Query
1	17:34:47.029	2.2 ms	SELECT	SELECT `title`, `content` FROM `page` WHERE live=1 ORDER BY `date_published` DESC /Users/larry/Sites/yii2-test/controllers/PageController.php (133)

Figure 8.5: *The resulting query, run in the browser.*

To execute the query, you'll invoke `one()`, `all()`, and other methods, depending upon what you want returned. I'll return to these shortly.

The `select()` method invocation is optional, and if not specified, every column is returned. Otherwise, specify columns as a comma-separated string or as an array. You can even use aliases:

```
$q->select('username AS name, email')
->from('page')
->where('id=1')
->all();
```

And you can invoke database functions:

```
$q->select('COUNT(*)')
->from('page')
->count();
```

(Note that the `count()` method is used there; I'll get to all the execution methods shortly.)

The `from()` method indicates the table or tables involved. It is not optional. You can list multiple tables as a comma-separated string or an array, and use aliases for them.

The `where()` method is the most complex one to use. I'll discuss it separately in a few pages.

The `orderBy()` method takes either a string or an array, with a string being the most obvious use. This query returns every column of every page record in descending order of the publication date:

```
$q->from('page')
->orderBy('date_published DESC')
->all();
```

The `limit()` and `offset()` methods take integers indicating the limit and offset!

With some combination of the above methods called, you then use a final method to execute the query and return the results:

- `all()` returns every matching record
- `one()` returns only one matching record
- `column()` returns only the first column of every matching record
- `scalar()` returns only the first column of the first matching record
- `exists()` returns a boolean indicating if any results were returned
- `count()` returns the result of a COUNT selection

These are all the same as used with the `find()` method of Active Record.

The `all()` method returns a multidimensional array, usable in a loop:

```
$result = $q->from('page')
->orderBy('date_published DESC')
->all();
foreach ($result as $row) {
    // Use $row['column_name'] et al.
}
```

If your query only returns a single row, use `one()` instead:

```
$row = $q->select('*')
->from('user')
->where(['id' => $id])
->one();
```

When you have a query that only returns a single *value*, use `scalar()`:

```
$user_id = $q->select('id')
->from('user')
->where(['id' => $id])
->scalar();
```

Alternative Syntax

For every method you can use to customize a query, there's also an attribute. This earlier example:

```
$q->select('title, content')
->from('page')
->where('live=1')
->orderBy('date_published DESC')
->one();
```

can also be written as:

```
$q->select = 'title, content';
$q->from = 'page';
$q->where = 'live=1';
$q->order = 'date_published DESC';
$q->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:

```
$q->select('title, content')->from('page')->where('live=1')
->orderBy('date_published DESC')->one();
```

If you prefer more clarity, you can spread out the chaining over multiple lines as I have been doing:

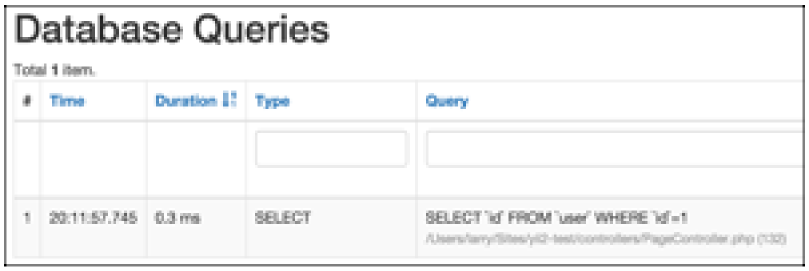
```
$q->select('title, content')
->from('page')
->where('live=1')
->orderBy('date_published DESC')
->one();
```

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:

```
$user_id = (new Query())->select('id')
->from('user')
->where(['id' => $id])
->scalar();
```

At any point in time, you can find the query being run using the Yii Debugger (**Figure 8.6**).



Database Queries

Total 1 item.

#	Time	Duration	Type	Query
1	20:11:57.745	0.3 ms	SELECT	SELECT `id` FROM `user` WHERE `id`=1 /Users/larry/Sites/yii2-test/controllers/PageController.php (132)

Figure 8.6: *The complete and actual query that was executed.*

Setting WHERE Clauses

The `where()` method adds WHERE clauses to a query, and can be used in many ways. You can provide a string to the method, but should only do so if the string uses hard-coded values:

```php
$q->select('title, content')
->from('page')
->where('live=1')
->orderBy('date_published DESC')
->one();
```

You should not do the following, as it's vulnerable to SQL injection attacks:

```php
$row = $q->select('*')
->from('user')
->where("id=$id")
->one();
```

But if you pass variables in an array, you don't risk SQL injection attacks:

```php
// SELECT * FROM user WHERE id=X
$row = $q->select('*')
->from('user')
->where(['id' => $id])
->one();
```

The array syntax like this creates an equality condition. If you pass a multidimensional array, you'll create a two equality clauses connected by an AND:

```
// SELECT * FROM user WHERE email='something' AND 'pass'='something'
$row = $q->select('*')
->from('user')
->where([['email' => $email], ['pass' => $pass]])
->one();
```

To use other operators within a WHERE clause, use the third syntax: `[operator, operand1, operand2,...]`. Here is an equivalent query to the previous one:

```
// SELECT * FROM user WHERE id=X
$row = $q->select('*')
->from('user')
->where(['=', 'id', $id])
->one();
```

You can find all the possible permutations in the Yii class reference, along with several good examples.

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()` method starts a WHERE clause, which you can expand using `andWhere()` and `orWhere()`. The former adds an AND clause to the exiting WHERE conditional, and the latter adds an OR:

```
$q->select('id, title')
->from('page')
->where('live=1');

if (isset($author)) {
    $q->andWhere(['user_id', $author]);
}
// And so on.
```

Performing JOINs

The last thing to learn about Query Builder is how to perform JOINs. The `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:

```
$q->select('page.id, title, username');
->from('page, user');
->where('page.user_id=user.id');
// And so on.
```

You can also use the `join()`, `leftJoin()`, `rightJoin()`, and `innerJoin()` methods to perform JOINs. The last three are syntactic shortcuts to `join()`.

The `join()` method takes as its arguments: the JOIN type, the name of the table to join, an ON clause, and an array of parameters:

```
$q->select('page.id, title, username');
->from('page');
->join('INNER JOIN', 'user', 'page.user_id=user.id');
// And so on.
```

Batch Querying

Both Query Builder and Active Record support batch querying, which is useful when you have a very large dataset being returned (more than 100s of rows). Instead of calling `all()` or `findAll()`, call `batch()` in a loop:

```
foreach ($q->batch() as $batch() {
    foreach ($batch as $row) {
        // Use $row.
    }
}
```

By default, Yii will return records in batches of 100.

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

- 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 using it for very basic tasks is a snap, but 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 fewer than 20 at a time.

Query Builder's benefits are that it:

- Handles the quoting of values nicely
- Prevents SQL injection attacks when used properly
- Allows you to perform JOINs easily, without messing with Active Record's relations
- Offers generally better performance than Active Record
- Ability to fetch records into arrays, for easy and fast access

The downsides to Query Builder are that it's a bit more complicated to use and does not return objects. Query Builder is recommended when you have dynamic SELECT 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, and is not as easy for creating dynamic queries on the fly as with Query Builder. 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

This chapter concludes with a couple of specific, common challenges when it comes to working with databases: performing transactions and using database functions.

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 `beginTransaction()` method. That's accessible through the "db" component:

```
// DAO:
$trans = Yii::$app->db->beginTransaction();
```

> *{NOTE}* Because Query Builder is primarily intended for SELECT queries, you won't likely use it with transactions.

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 DAO. To use transactions with Active Record, begin the transaction through the model's `getDb()` method, which returns the "db" component:

```
$model = new SomeModel();
$trans = $model->getDb()->beginTransaction();
```

Everything else is the same.

> *{NEW}* In Yii 2, you can identify in a model the operations that should automatically be performed using transactions.

There are a couple of things to know about transactions, however. First, depending upon the database application in use, certain commands have the impact of automatically committing the commands to that point regardless. See your database application's documentation for specifics. Second, MySQL only supports transactions when using specific storage engines, such as InnoDB. The MyISAM storage engine does not support transactions.

Using Expressions

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:

```
$file->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 database function call, you must use a `yii\db\Expression` object for the value. That syntax is:

```
use yii\db\Expression;
$file->date_updated = new Expression('NOW()');
```

If the database function takes an argument, such as the password to be hashed, use parameters:

```
$user->pass = new Expression('SHA2(:pass)', [':pass' => $pass]);
```

As another example, if you want to get a random record from the database, you would use an `Expression` for the ORDER BY value:

```
$row = $q->select('*')
->from('user')
->order(new Expression('RAND()')
->limit(1)
->one();
```

To select a formatted date, use the `DATE_FORMAT()` call as part of the selection:

```
$row = $q->select(['*',
    new Expression('DATE_FORMAT(date_entered, "%Y-%m-%d")'])
->from('user')
->order(new Expression('RAND()'))
->limit(1)
->one();
```