

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: 2.0

Copyright © 2024 by Larry Ullman

Technical Reviewer: Qiang Xue

Technical Reviewer: Alexander Makarov

Cover design very kindly provided by Paul Wilcox.

Notice of Rights

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

Notice of Liability

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of the book, the author shall not have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

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

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

ISBN-13:

ISBN-10:

This book is dedicated to:

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

Contents

Introduction	1
I Getting Started	12
1 Fundamental Concepts	13
2 Starting a New Application	30
3 A Manual for Your Yii Site	40
4 Initial Customizations and Code Generations	51
II Core Concepts	80
5 Working with Models	81
6 Working with Views	112
7 Working with Controllers	134
8 Working with Databases	155
9 Working with Forms	179
10 Maintaining State	206
11 User Authentication and Authorization	215

12 Working with Widgets	251
13 Using Extensions	284
14 JavaScript and jQuery	318
III Advanced Topics	338
15 Internationalization	339
16 Leaving the Browser	360
17 Improving Performance	376
18 Advanced Database Issues	404
19 Extending Yii	418
20 Working with Third-Party Libraries	461
21 Testing Your Applications	472
IV Completing Projects	488
22 Creating a CMS	489
23 Making an E-commerce Site	509
24 Shipping Your Project	536

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?

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.

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).

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.

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.

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. 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.

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

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](#).

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.

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.

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.

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.

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.

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.

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.

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](#).

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.

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.

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.

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* 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

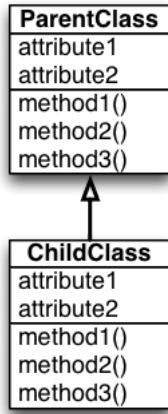


Figure 1.1: The child class can inherit members from the parent class.

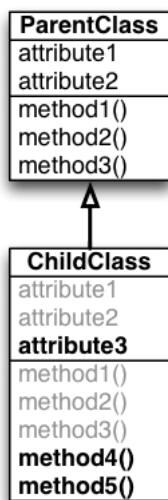


Figure 1.2: The child class can add new members to the ones it inherited.

from `Person` and add a `work()` method, whereas `Child` could also inherit from `Person` but add a `play()` method (**Figure 1.2**).

Inheritance can be extended to such a degree that you have multiple generations of inheritance: 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.

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{  
}
```

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.

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.

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

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

Each controller is also defined as a class. Within the class, different methods identify possible *actions*. The most obvious actions represent *CRUD* functionality: Create, Read, Update, and Delete. Yii takes this a step further by breaking "read" into one action for listing all of a certain model and another for showing just one. 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.

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.

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](#).”

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.

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.

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.

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.

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](#)).

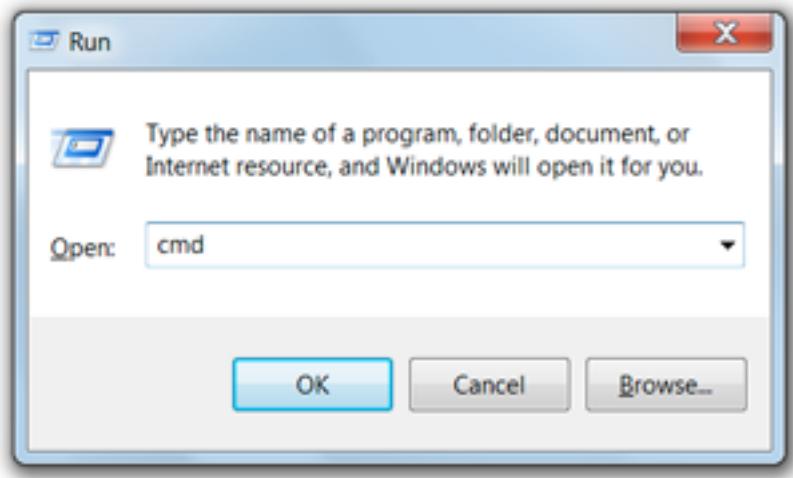


Figure 1.3: The *cmd* prompt.

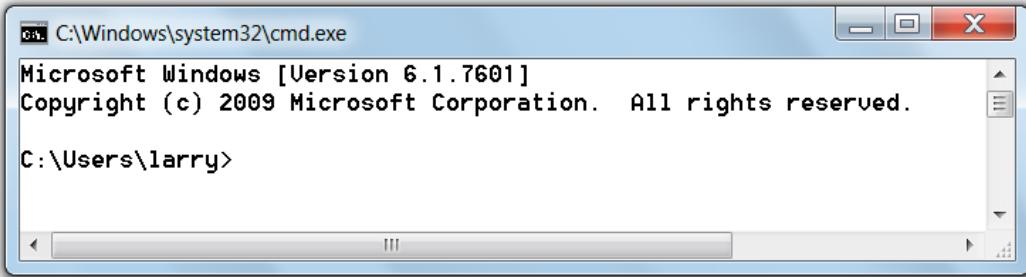


Figure 1.4: The command line interface on Windows.

Then click OK.

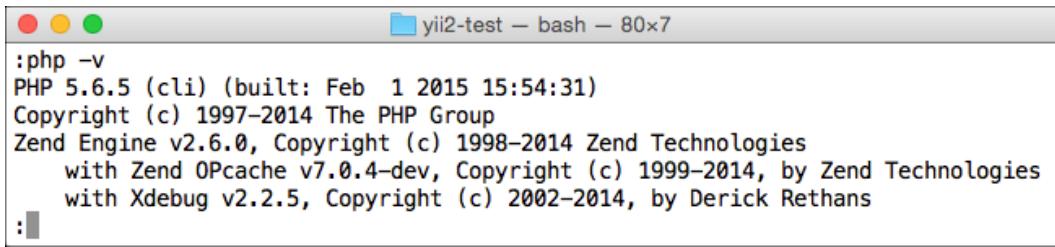
As of Windows 7, there is no immediate Run option in the Start menu, but you can find it under Start > All Programs > Accessories > Command Prompt, or you can press Command+R from the Desktop.

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”.

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.

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



A screenshot of a terminal window titled "yii2-test – bash – 80x7". The window shows the output of the command "php -v". The output displays the following information:

```
: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.

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.

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 semi-colon 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.

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.

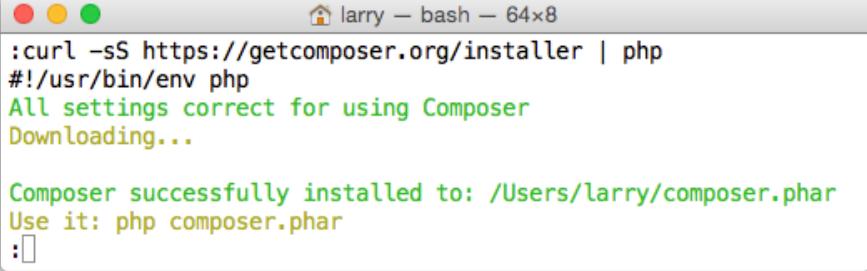
These are generic, but static steps, to be taken with each new website you create. In Chapter 4, “[Initial Customizations and Code Generations](#),” you’ll have Yii create code more specific to an individual application.

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!

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.



```
: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.*

{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).

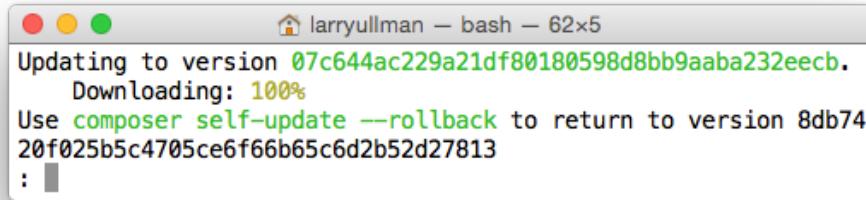


Figure 2.2: Updating Composer.

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.

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
```

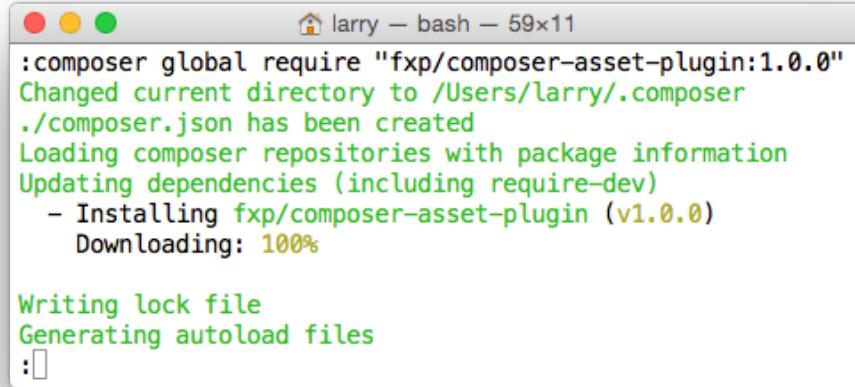
Composer itself will warn you if it hasn't been updated in more than a month.

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:1.1.1"
```

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!



```
: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.*

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:

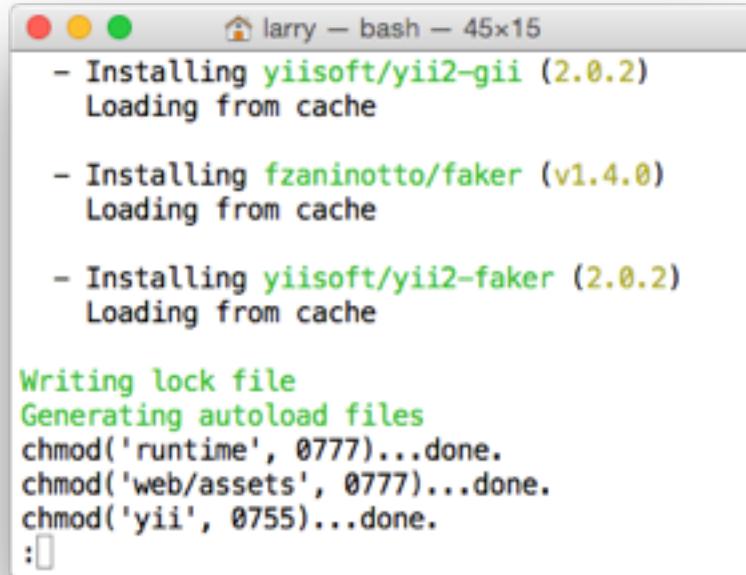
- Basic
- Advanced

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
```



```
- 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.

(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
```

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:

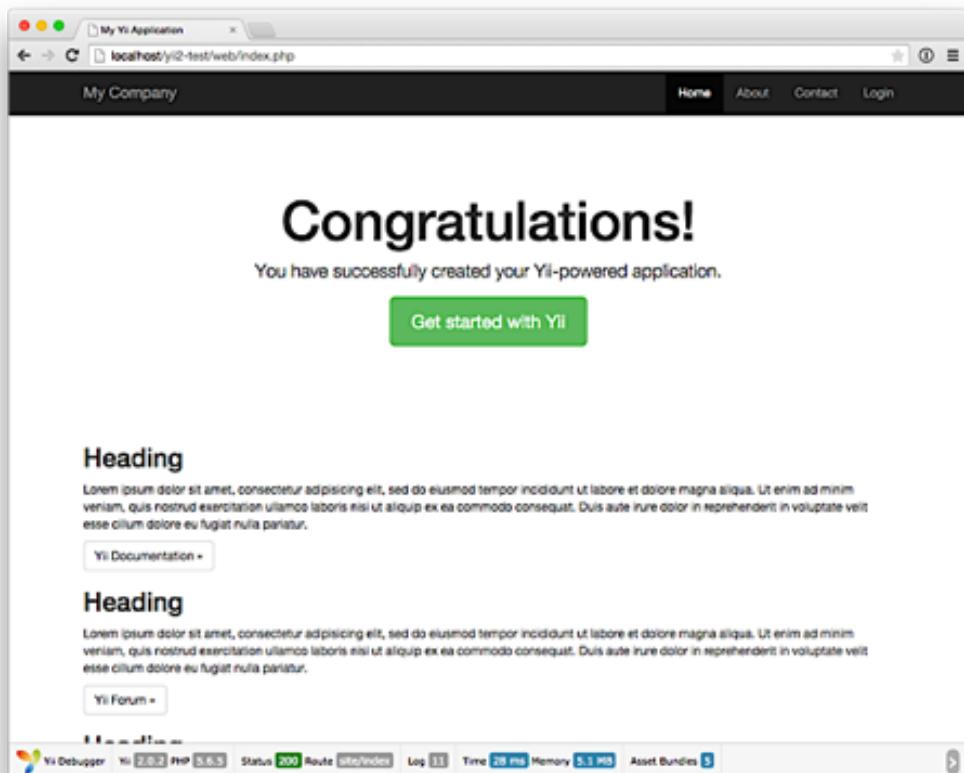


Figure 2.5: The shell of the generated site.

- A home page (see Figure 2.5)
- 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.

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.

1. Access your computer from a command-line interface.

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.

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

```
● ● ●   yii2-test — bash — 67x21
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.

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 storing 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.

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.

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.

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 prefixes its own classes with a letter as Yii 1 did.

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_*.

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 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.

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 `$app` variable within the `\Yii::$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.

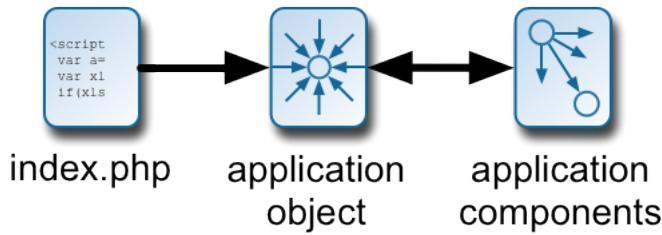


Figure 3.1: The index page creates an application object which loads the application components.

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

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,

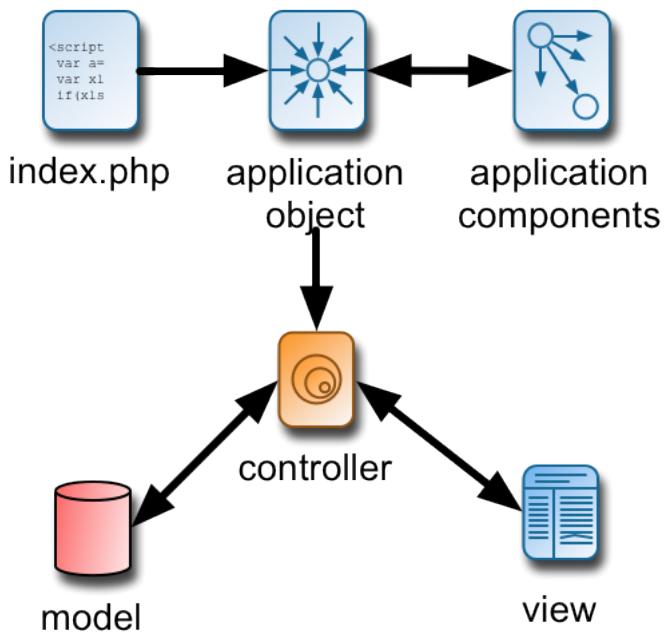


Figure 3.2: Subsequent steps involve calling the correct controller, accessing models, and rendering views.

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.

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`



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

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](#).”

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.

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:

The screenshot shows the Yii Debugger interface running on a Mac OS X system. The title bar says " Yii Debugger". The address bar shows the URL "localhost/yi2-test/web/index.php?r=debug%2fdefault%2fIndex". The top navigation bar includes "Yii Debugger", "Yi 2.0", "PHP 5.6.5", "Status 200", "Route", "Stacktrace", "Log", "Time 39 ms", "Memory 5.1 MB", and "Asset Bundles".

The main content area is titled "Available Debug Data" and displays a table of "Showing 1-4 of 4 items". The table has columns: #, Tag, Time, Ip, Query Count, Mail Count, Method, Ajax, URL, and Status code.

#	Tag	Time	Ip	Query Count	Mail Count	Method	Ajax	URL	Status code
1	54d82d9b383cb	2/6/15 4:46 AM	127.0.0.1	0	0	GET	No	http://localhost/yi2-test/web/index.php?r=debug%2fIndex	200
2	54d7aa76e007	2/6/15 8:31 PM	127.0.0.1	0	0	GET	No	http://localhost/yi2-test/web/index.php?r=site%2fLogin	200
3	54d7aa76e007	2/6/15 7:27 PM	127.0.0.1	0	0	GET	No	http://localhost/yi2-test/web/index.php	200
4	54d7aa76e008	2/6/15 7:27 PM	127.0.0.1	0	0	GET	No	http://localhost/yi2-test/web/	200

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

- 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
- 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 toll!

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.

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.

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](#)".

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.

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!

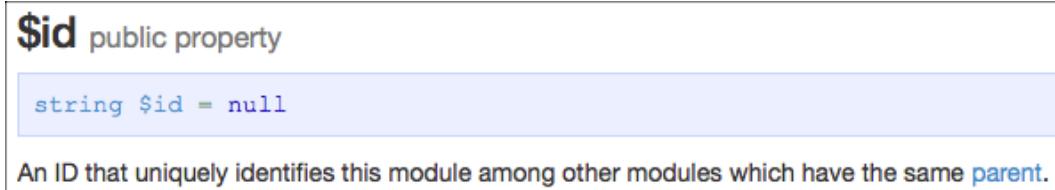


Figure 4.1: The Yii docs for the `id` property of the `yii\web\Application` class.



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

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`:

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**):

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 other configuration settings, as appropriate.

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.

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.

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.

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
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.

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.

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:

```
$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`.

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/`”).

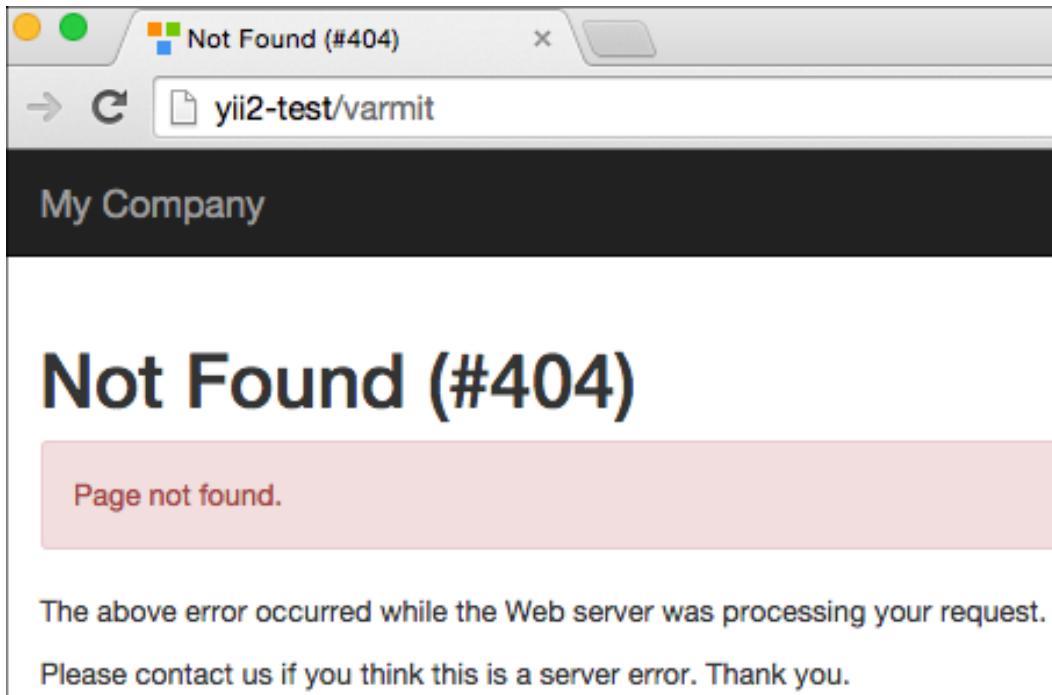


Figure 4.3: An error message because the “varmit” resource doesn’t exist.

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

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

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.

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']`.

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.

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.

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.

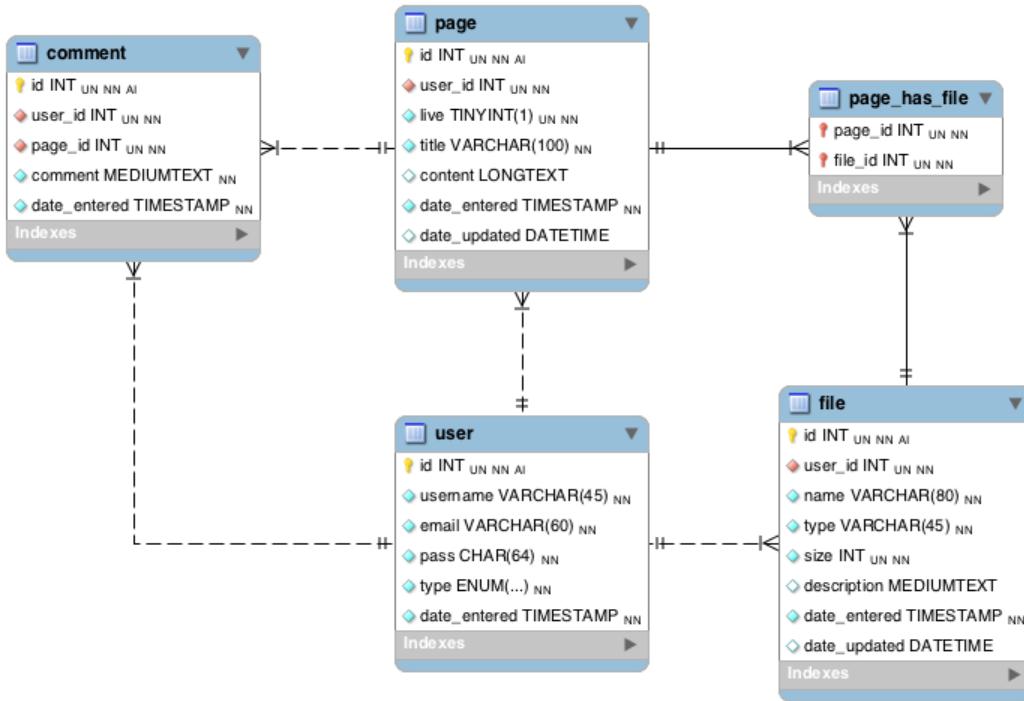
If you're a designer, or are working with clients that think primarily in visual terms, it makes sense to begin new projects with how they will look. This may be a wireframe representation or actual HTML, but create a series of pages and images that provide a basis for how the site will appear from a 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

**Figure 4.4:** The CMS database schema.

confirm that everything the site must be able to do requires just those four types of data.

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](#).

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 user-name, 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.

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.

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.

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.

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**).

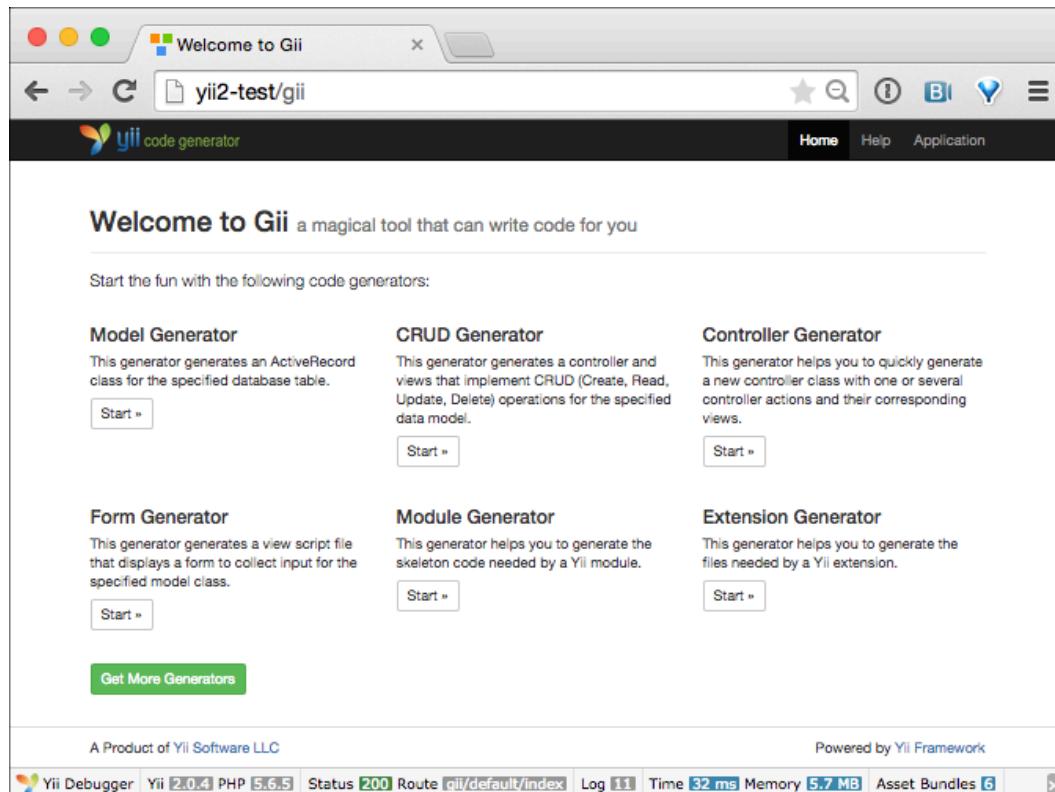


Figure 4.5: The Gii home page.

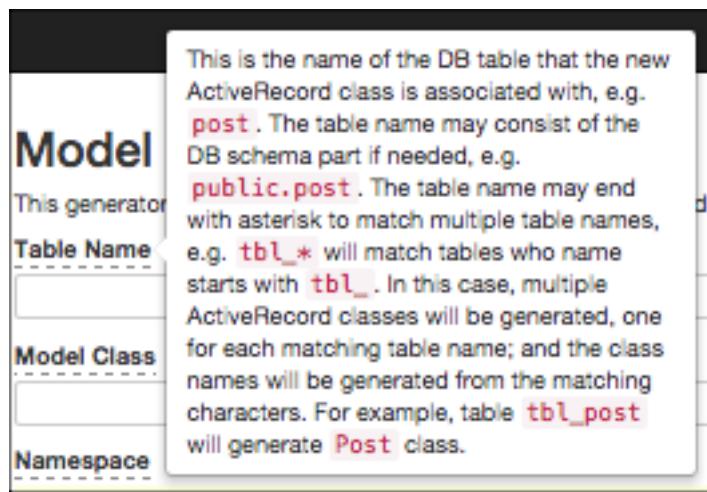


Figure 4.6: A tooltip for using the model generator.

{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.

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.

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)

Preview

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

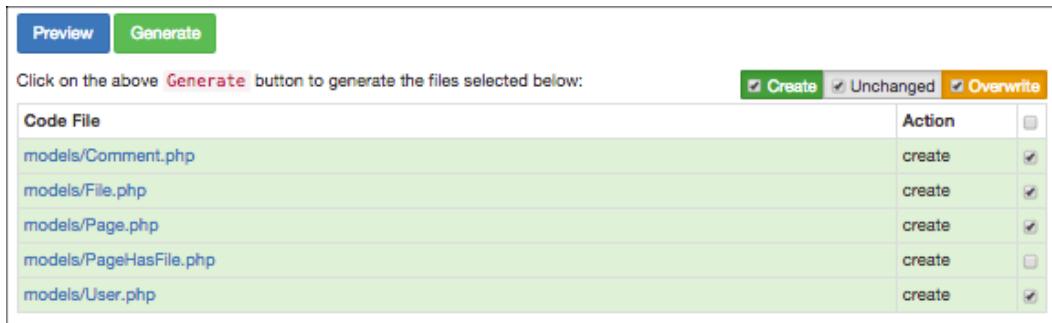


Figure 4.8: The preview of the files to be created.

2. Click Preview.
3. In the preview (**Figure 4.8**), deselect the **PageHasFile.php** model, which is not needed by this application.
4. Click Generate.

The * is a shortcut to have Gii automatically model every database table. If you'd rather generate a model at a time, you can enter a table name in the first field and work through Steps 2 and 3, 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.

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.

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**).

{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

CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) functionality for your model.

Model Class

app\models\Page

Search Model Class

app\models\PageSearch

Controller Class

app\controllers\PageController

View Path

(empty)

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)

Preview

Figure 4.9: Generating CRUD functionality for pages.

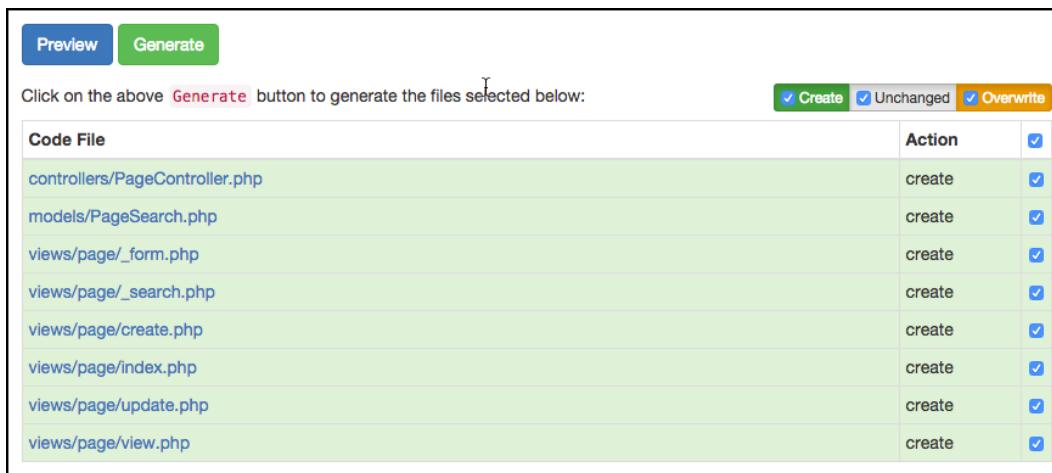


Figure 4.10: The preview of the files being generated for Page CRUD functionality.

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](#).

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](#).”

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.

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.

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**).

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

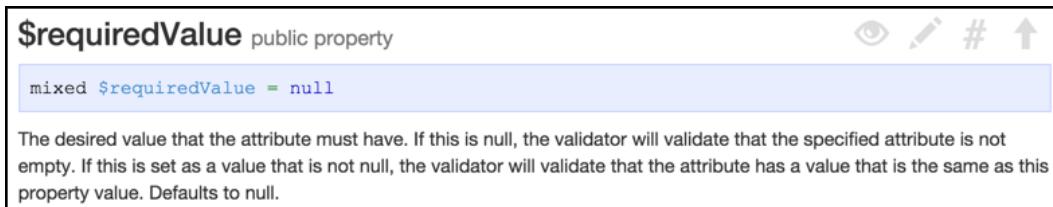


Figure 5.1: The details for the “requiredValue” property of the `yii\validators\RequiredValidator` class.

```
[ '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 “pass-Compare”, 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:

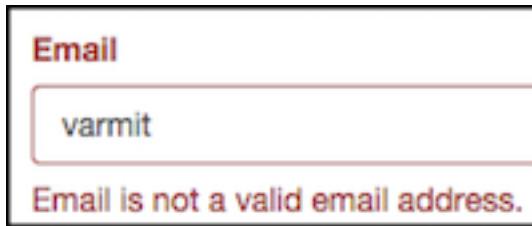


Figure 5.2: The default error message for an invalid email address.

```
['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!

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**).

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

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

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

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.'],
```

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.

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

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:

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

This filter would then be invoked like so:

```
# 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:

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

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:

```
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.

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.

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
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 {
```

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.

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 \yii\db\ActiveRecord {  
    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—are 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  
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:

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

```
    ],
];

# 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()
return [
    [
        '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.

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**).

{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

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

`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 `passCompare` attribute to `User` means you should add its label, too:

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

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.

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**)

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 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

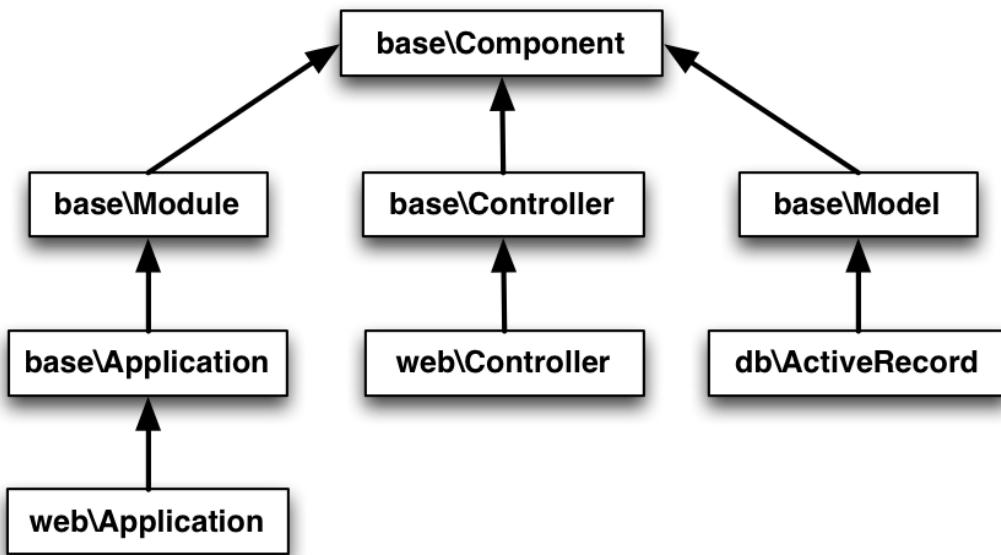


Figure 5.4: Part of Yii’s class inheritance structure, with `Component` at the top.

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:

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

The `beforeSave()` method must be defined to take an `$insert` argument. It will have a Boolean value indicating if this is a new record—an `INSERT`—or not.

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:

```
# 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.

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`:

```
# 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.

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.

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:

```
# 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:

```
# 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.

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:

```
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:

```
$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:

```
$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.

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.

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.



Figure 6.1: A templated page.

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.

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
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.

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

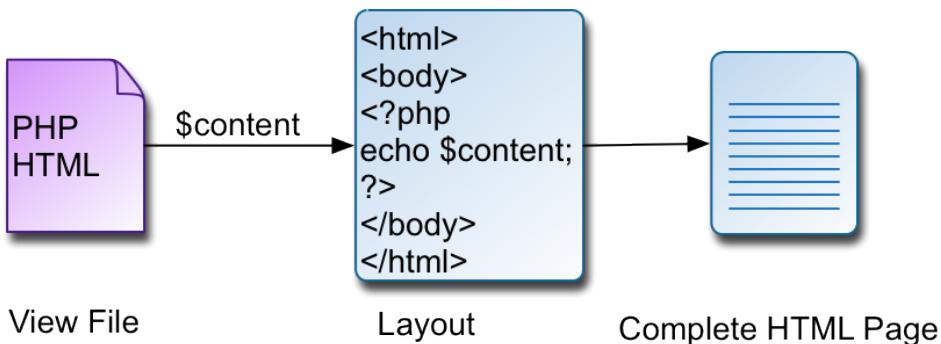


Figure 6.2: How Yii renders a complete page by pulling an individual view file into the layout.

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.

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 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.

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](#).”

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.

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'); ?>
```

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.

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 permanently enabled 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](#).

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:

```

```

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

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.

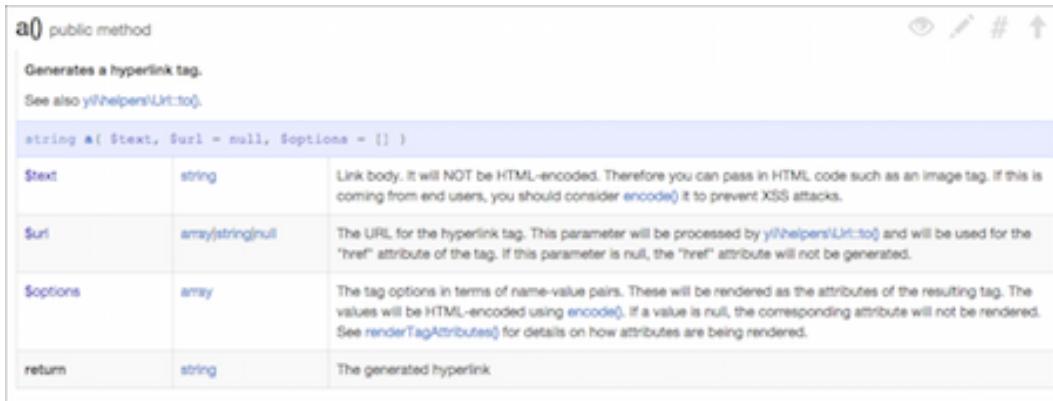


Figure 6.3: The class documentation for the `Html::a()` method.

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”:

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

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

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

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

```
<?php
echo Html::a('', ['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.

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.

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](#).

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.

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.

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**.

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.

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:

```
// Use a simpler layout:  
$this->layout = 'receipt';  
// Render the shoppingCart/show.php view:  
$this->render('//shoppingCart/show', ['order' => $order]);
```

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.

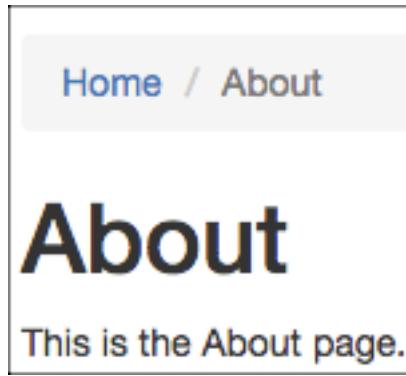


Figure 6.4: The displayed breadcrumbs.

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:

```
$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**).

```
<?= Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ?
        $this->params['breadcrumbs'] : [],
]) ?>
```

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.

The default application in Yii 1 came with three layout files:



Figure 6.5: The page-specific content goes across the entire width of the browser.

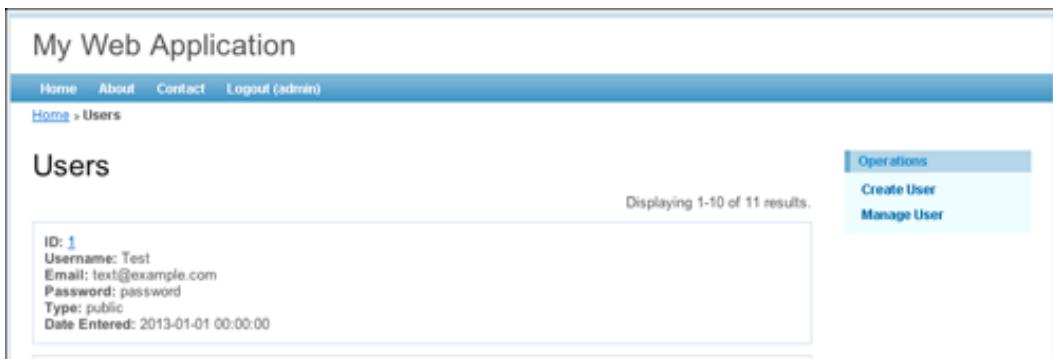


Figure 6.6: The page-specific content shares the browser width with a sidebar.

- **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):

```
<?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 6.5), but some pages should have a sidebar (Figure 6.6). 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('@app/views/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('@app/views/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.

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.

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](#).”

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.

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()`.

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:

```
# config/web.php  
$config = [  
    /* 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:

```
# 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:

```
# 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.

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	<code>http://example.com/index.php/page/100</code>
baseUrl	<code>/dir</code>
hostInfo	<code>http://example.com</code>
pathInfo	<code>/page/100</code>
queryString	(n/a, unless the URL query syntax was <code>id=100</code>)
scriptUrl	<code>/dir/index.php</code>
serverName	<code>example.com</code>
serverPort	80 (presumably)
url	<code>/dir/index.php/page/100</code>
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](#).

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]);
```

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)

The first way controllers create model instances is quite simple, and just uses standard object-oriented code:

```
public function actionCreate() {
    $model = new Page();
    // Etc.
```

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.

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.

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.

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**.

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](#).

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.

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?

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.

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:

```
'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.

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](#).

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:

```
# controllers/PageController.php
public function actionDummy() {
    $url = Url::to('index'); // page/index
```

If you provide a controller, the URL is to that route:

```
# controllers/PageController.php
public function actionDummy() {
    $url = Url::to('user/index'); // user/index
```

If the URL expects named parameters, add those to the array:

```
# 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:

```
# controllers/PageController.php
public function actionDummy() {
    $url = Url::to('index', true); // http://example.com/page/index
```

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:

```
# 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.

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'],
            ],
        ],
    ];
}
```

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 **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.

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) {
```

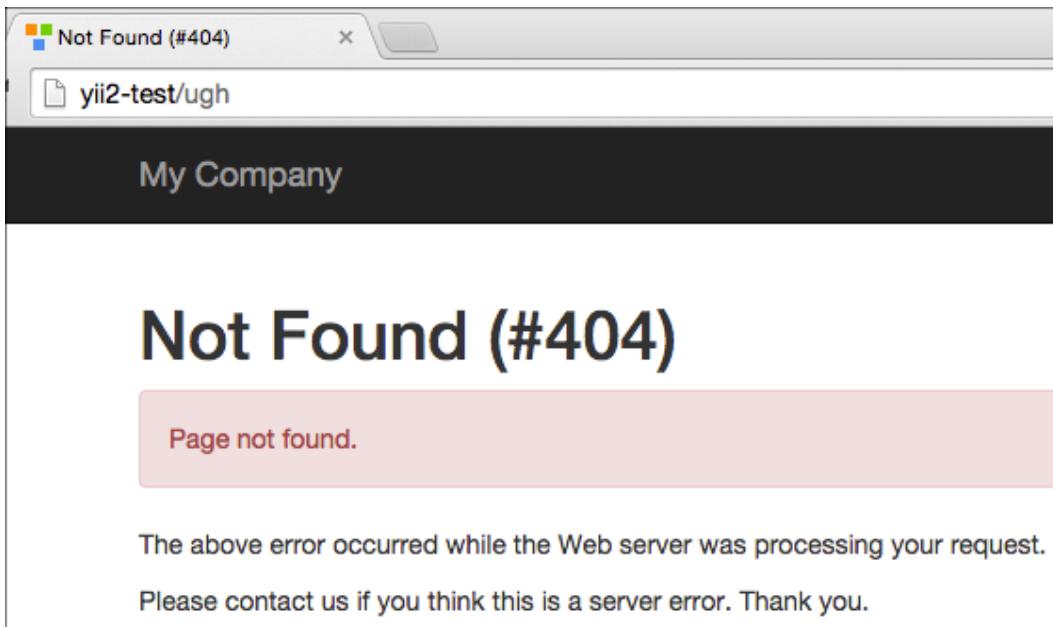


Figure 7.2: Because the “ugh” controller does not exist, a 404 exception is thrown.

```
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**).

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.

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



Figure 8.1: This page required 3 queries.

- 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 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**).

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.

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:

- [PostgreSQL](#)
- [SQLite](#)
- Microsoft [SQL Server](#)
- [Oracle](#)
- [MariaDB](#)

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.

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](#).”

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.

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()) {
    // Whoohoo!
} else {
    // Use $model->errors
}
```

The `errors` property will be an array of errors.

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.

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.

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.

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.

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.

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.

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.';  
}
```

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:

```
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.

```
# 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**).

Database Queries				
Total 1 item.	#	Time	Duration	
	Type	Query		
	1	21:11:21.152	0.3 ms	SELECT COUNT(*) FROM `page` LEFT JOIN `comment` ON `page`.`id` = `comment`.`page_id` it'senv/test/yii2-test/controllers/PageController.php (129)

Figure 8.4: The executed JOIN.

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.

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**).

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.

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.

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!

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()->update('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 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();
```

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.

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.

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.

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')
```

Database Queries			
#	Time	Duration ↑	Type
1	17:34:47.029	2.2 ms	SELECT
SELECT `title`, `content` FROM `page` WHERE live=1 ORDER BY `date_published` DESC			/Users/timmytunes/PycharmProjects/test/controllers/PageController.php:130

Figure 8.5: The resulting query, run in the browser.

```
->orderBy('date_published DESC')
->one();
```

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();
```

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/avery/Sites/yii2-test/controllers/PageController.php (132)

Figure 8.6: The complete and actual query that was executed.

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:

```
$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:

```
$row = $q->select('*')
->from('user')
->where("id=$id")
->one();
```

But if you pass variables in an array, you don't risk SQL injection attacks:

```
// 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
$q = $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.

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.
```

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.
```

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.

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.

This chapter concludes with a couple of specific, common challenges when it comes to working with databases: performing transactions and using database functions.

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.

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();
```

Chapter 9

Working with Forms

HTML forms are crucial to practically every website. As you'd expect, creating forms using a framework such as Yii is significantly different than creating forms using HTML alone. This chapter explains how to create HTML forms with Yii, the fundamentals of working with forms, and then walks through a few recipes for common, but more complex, form needs.

Before getting into code, let's take a moment to think about the MVC architecture. A form is part of the view component, as a form is an aspect of the user interface. Forms, though, are almost always associated with specific models. A contact form may have its own model, not tied to a database table, and a form for employees or departments are based upon a model tied to a database table. Whether the model is database-driven or not, the form is tied to a model. This is significant as it's the model that dictates what form elements should exist, controls validation of the form data, and even defines the form's labels (e.g., "First Name" for the `firstName` attribute).

{NOTE} There are situations where a form might not be associated with a model, but those are rare.

When Gii auto-generates CRUD functionality for a model, it creates a form in a file named `_form.php`. This file is included by other view files (any view file in Yii that starts with an underscore is intended to be an include). The same `_form.php` file is used whether the form is creating new records or updating existing ones, and the controller dictates the situation accordingly. In both situations, a model instance is provided by the controller to the form. If the model has populated attributes—the model represents an existing record, Yii populates the form's elements with the model's current values. This is the main distinction between adding a new record and updating an existing one.

Before getting to the view and its form, let's be clear as to how a view accesses a specific model. A controller may have this code:

```
public function actionCreate() {
    $model = new Page();
    // Code for validation and redirect upon save.
    // If not saved, render the create view:
    return $this->render('create', ['model' => $model]);
}
```

The **create.php** view file includes **_form.php**, passing along the model instance in the process:

```
<?= $this->render('_form', ['model' => $model]) ?>
```

Thanks to that code, **_form.php** has access to the model instance and can create a form tied to that model. Once the form view file has access to the model, it can create form elements in one of two ways:

- Invoking the `Html` helper class methods directly
- Using the `ActiveForm` widget

This chapter explains both options, but focuses more on the later, which is the default approach in Yii.

Of course, to be fair, you *could* create an HTML form using raw HTML without Yii at all. The downside is doing so creates no tie-in between the model's validation rules, errors, labels, etc., and the form. By creating the form using Yii, labels will be based upon the model definitions, invalid form values can automatically be highlighted, and much, much more. Plus, it's not hard to use Yii to create a form, once you understand how.

{NEW} Yii 2 no longer has a built-in form builder, although there are extensions that serve that role.

One approach for creating forms in Yii is simply a matter of invoking the appropriate `Html` methods. Chapter 6, “[Working with Views](#),” introduces this helper class, as it's also used for creating links. This class is used statically—not through object instances, and defines everything needed to make form elements and other HTML tags.

To start, you must include the `Html` helper class's namespace:

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

Next, create the opening FORM tag:

label() public method		
Generates a label tag.		
<pre>string label(\$content, \$for = null, \$options = [])</pre>		
\$content	string	Label text. 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 encode() it to prevent XSS attacks.
\$for	string	The ID of the HTML element that this label is associated with. If this is null, the "for" 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 label tag

Figure 9.1: The Yii class docs for the `Html::label()` method.

```
<?= Html::beginForm(['search'], 'get'); ?>
```

The method’s first argument is the tag’s “action” attribute value. Yii turns this into an appropriate route. The above, for example, will be submitted to one of the following, depending upon the application’s URL configuration:

- `http://example.com/index.php?r=search`
- `http://example.com/index.php/search`
- `http://example.com/search`

Thus, the form submission is handled by the search controller, and the default action of that controller.

The second argument to `beginForm()` sets the value of the form’s “method” attribute. You’d use “get” or “post” there, with “post” being the default. Search forms normally use GET.

Next, start adding form elements. There’s a method for each type. For example, the `label()` method creates an HTML LABEL (**Figure 9.1**).

The `textInput()` method creates a text input. Toss in a submit button, and you’ve got yourself a search box:

```
<?= Html::label('Search', 'terms'); ?>
<?= Html::textInput('terms'); ?>
<?= Html::submitButton('Go!'); ?>
```

Finally, close the form:

```
<?= Html::endForm(); ?>
```

The end result is the following HTML:

```
<form action="/index.php/page/search" method="get">
<label for="terms">Search</label>
<input type="text" name="terms">
<button type="submit">Go!</button>
</form>
```

It's not practical, or a good use of book space, to explain each `Html` method in detail in this book. Instead, this chapter covers how to create forms in practice, and leaves it up to you to look up the details and options in the [Yii class reference](#) for the `Html` helper class. Later, the chapter specifically walks through a few of the more common but difficult form needs. But rather than leave you entirely on your own, there are a few things you ought to know up front about the `Html` methods.

The “action” attribute for a form needs to be mapped to proper routes for the site and its controllers. To do so, Yii uses the `Url::to()` method, which does the following:

- Uses the current URL when an empty string is provided
- If a non-empty string is provided, that string is used as a URL without change
- If an array is provided, the array’s values are used as in the `Url::toRoute()` method, which Chapter 7, “[Working with Controllers](#),” explains

The short version on using `toRoute()` is the first element in the array is treated as the action (e.g., “view”) or controller and action (e.g., “search/view”). Any other array elements will be used as parameters passed in the URL.

You should also know that many of the `Html` methods take a final argument for providing additional HTML attributes. For example, to apply a class to an input:

```
<?= Html::textInput('terms', '',
    ['class' => 'input-medium search-query']); ?>
```

(The second argument to the `textInput()` method is its value.)

Instead of using `textInput()`, you could invoke the more generic `input()` method, passing the input type as the first argument: “text”, “password”, “email”, “search”, and so on. This is equivalent to the previous code example:

```
<?= Html::input('text', 'terms', '',
    ['class' => 'input-medium search-query']); ?>
```

The `Html` helper has methods for creating useful elements that don’t correspond to a single HTML form element. For example, the `checkBoxList()` method creates a sequence of checkboxes and `radioButtonList()` does the same thing with radio buttons (**Figure 9.2**):

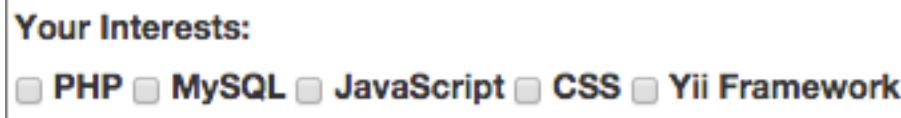


Figure 9.2: The `checkBoxList()` method creates multiple checkboxes from an array.

```
<?= Html::label('Your Interests:', 'interests'); ?>
<?= Html::checkBoxList('interests', '', ['PHP', 'MySQL',
    'JavaScript', 'CSS', 'Yii Framework']); ?>
```

Finally, as mentioned in other chapters, `Html` defines methods related to other HTML aspects, not just forms. For example, the `cssFile()` method creates a link to a CSS file:

```
<?= Html::cssFile('css/mycss.css'); ?>
```

Or, the `image()` method creates an HTML IMG tag and `a()` creates an HTML A tag.

Most forms are tied to model instances. When a view file is provided with a model, it then creates an associated form using either the “active” methods of the `Html` class or the `ActiveForm` widget.

Remember that tying forms to models means the form will automatically:

- Be pre-populated with the model’s values
- Show error messages inline
- Highlight invalid fields
- Create JavaScript for client-side validation

Each form method in the `Html` helper class is repeated with a version prefaced with “active”: `activeLabel()`, `activeTextInput()`, `activeSubmitButton()`, and so forth. The non-active versions are used *without* a model reference. The active versions all take a model instance as their first argument:

```
<div class="row">
    <?= Html::activeLabel($model, 'username'); ?>
    <?= Html::activeTextField($model, 'username') ?>
</div>
```

Each element’s name—such as “username” in the above—needs to be exactly the same as a corresponding attribute in the model. Doing so ties the model’s definition—its labels, validation routines, and so forth—to the form elements. Attempting to create

Unknown Property – yii\base\UnknownPropertyException

Getting unknown property: app\models\Page::userid

Figure 9.3: An exception is thrown by attempting to create a form element without a matching model attribute.

an element that doesn't correlate to a model attribute results in an error (**Figure 9.3**).

There's another benefit of tying a model to a form: if the form is being used for an update, the values will automatically be pre-populated/pre-selected/pre-checked based upon the existing model instance! As you should know, that alone requires a lot of code and logic in traditional programming.

Just using the “active” methods to tie a form to a model is great, but there's a more preferred route. The code Gii generates uses the `yii\widgets\ActiveForm` class. Although widgets aren't formally covered until Chapter 12, “[Working with Widgets](#),” using `ActiveForm` as a widget is easy enough to grasp and implement that it's covered now.

To use `ActiveForm`, first reference the necessary namespaces:

```
use yii\helpers\Html;
use yii\widgets\ActiveForm;
```

The `ActiveForm` class is required, but you'll also use `Html` to create individual elements, such as the submit button.

Start using `ActiveForm` by invoking the class's `begin()` method. Assign the result of the call to a new variable:

```
<?php $form = ActiveForm::begin(); ?>
```

Now `$form` is an object of the `ActiveForm` type, and it can be used to generate the form itself. For `ActiveForm`, this code also creates the initial FORM tag in the HTML page. (That code is from the `_form.php` file.)

From there, use the `field()` method to create form elements. Its first argument is a model instance and its second is the model attribute for which the form element is being created. The `field()` method returns an `ActiveField` instance. On the `ActiveField`, invoke a method corresponding to the type of element to be created:

```
$pass = $form->field($model, 'pass');
echo $pass->passwordInput();
?>
```

These two lines are normally condensed into one. Here are two examples from the `Page_form.php` view file:

```
<?= $form->field($model, 'user_id')->textInput() ?>
<?= $form->field($model, 'live')->textInput() ?>
```

The above is the default form code created by Gii. Both examples need to be changed for a live site, to be explained shortly. Also note that the results of the method call must be printed.

This might all seem like a lot of work, but by using `ActiveField`, Yii creates not just the form element, but also the `LABEL`, a place for error messages, and any wrapping HTML. Here is how a text input is rendered from just the one line of code above:

```
<div class="form-group field-page-user_id">
<label class="control-label" for="page-user_id">User ID</label>
<input type="text" id="page-user_id" class="form-control"
       name="Page[user_id]">
<div class="help-block"></div>
</div>
```

{NEW} Yii 2 automatically creates the `LABEL`, error handling location, and wrapping HTML elements, when using `ActiveForm`.

By using `ActiveForm`, instead of just `Html`, you can now easily implement:

- Server-side validation
- Client-side validation via JavaScript
- Client-side validation via Ajax

The first two are implemented automatically for you. Test it for yourself to see. Disable JavaScript and test that, too. Chapter 14, “[JavaScript and jQuery](#),” discusses Ajax and Ajax form validation.

Returning to the widget itself, you can customize the behavior of the form by passing an array of values when creating it:

```
<?php $form = ActiveForm::begin([
    'id'=>'page-form',
    'enableAjaxValidation'=>false
]); ?>
```

The various `ActiveForm` properties can be found in the documentation. Commonly you won’t need to customize any properties, but you can change the form’s “method”

and “action” attributes, or add additional HTML to the opening FORM tag (e.g., a class).

Finally, the form needs a submit button. Unlike the other form elements, the submit button isn’t tied to a model; it’s created by the `Html` class:

```
<div class="form-group">
    <?= Html::submitButton(
        $model->isNewRecord ? 'Create' : 'Update',
        ['class' => $model->isNewRecord ? 'btn btn-success' :
            'btn btn-primary']) ?>
</div>
```

That code creates a submit button with a label of “Create” or “Update”, depending upon how it’s being used. It also changes the applied class accordingly.

Then the form is closed by “ending” the widget:

```
<?php ActiveForm::end(); ?>
```

This code prints out the closing FORM tag.

Those are the basics for using `ActiveForm`. Once you’ve taken the above steps, form elements will be pre-populated when updating a record, errors will be clearly indicated upon form submission, and so forth.

By default, HTML forms are submitted back to the same page, unless a different ACTION value is provided. With Yii, this means a form on `/page/create` is submitted to `/page/create`. Here’s the definition of the `actionCreate()` method:

```
$model = new Page();
if ($model->load(Yii::$app->request->post()) && $model->save()) {
    return $this->redirect(['view', 'id' => $model->id]);
} else {
    return $this->render('create', ['model' => $model]);
}
```

The process is really straightforward. First, a new model instance is created. Then a conditional does two things:

1. Populates the model’s attributes with values posted to the page.
2. Attempts to save the model.

The first time the page is accessed, no values will be posted, and no validation occurs, so the conditional is false. Thus, the “create” page is rendered.

After the form has been submitted, values will be posted, and validation will take place. If all the validation rules—defined in the model—pass, the model can be saved. Then, the browser is redirected to a view page that shows the newly created model.

If any rule does not pass validation, a corresponding error is added to the model, and the model cannot be saved. Thus, the error will automatically be shown in the view form.

This is purposefully not intended as a recipe book, such as [Alexander Makarov's text](#), and my hope is that the material in this chapter to this point provides enough information that you could resolve whatever it is you need to do going forward. But there are still a few common needs and points of confusion that stand to be addressed.

{TIP} This chapter does not explain CAPTCHA, as it requires a widget.
You'll see how to use it in Chapter 12.

Due to the way they represent values, checkboxes can sometimes be challenging to work with. When you enter “lycanthropy” in a text box, you know that text box’s value will be “lycanthropy”. But when you check a checkbox, what value will it have? And, more importantly, how can that value be properly mapped to a model attribute?

The answer to the first question is simple: if a checkbox is checked, the resulting PHP variable will have the same value as the checkbox’s “value” attribute:

```
Receive Updates?  
<input type="checkbox" name="updates" value="yes">
```

If that box is checked, then `$_POST['updates']` will have a value of “yes”. If the checkbox is not checked, `$_POST['updates']` will not have a value (i.e., the variable won’t be “set”). This creates a problem as the database, and the corresponding model attribute, may use true/false, Y/N, or 1/0 to represent whether or not that checkbox was checked. You can easily set the affirmative value—true, Y, 1, but how do you set the negative (i.e., non-checked) value?

A second problem arises when updating models. You can pre-check a checkbox by adding the “checked” attribute to the element:

```
Receive Updates?  
<input type="checkbox" name="updates" value="yes" checked>
```

How do you make that happen when the model might use true, Y, or 1 for its affirmative value?

To answer all these questions, let’s run through some specific examples, starting by simply accessing checkboxes.

{TIP} Most of the information with respect to checkboxes equally applies to radio buttons.

The code included in the basic application template makes use of a checkbox already: the login form presents a “Remember Me?” element. This is an optional checkbox: the user can log in whether the box is checked or not. If the user does check the box, the login cookie’s expiration will be extended.

The checkbox is created in the view form using this code:

```
<?= $form->field($model, 'rememberMe',
    ['template' => "<div class=\"col-lg-offset-1 col-lg-3\">
        {input}</div>\n<div class=\"col-lg-8\">{error}</div>",
])->checkbox() ?>
```

This checkbox is mapped to a model attribute, `rememberMe`, so the controller can find the attribute’s value in `$model->rememberMe`. But what will that value be if checked? What will it be if not checked?

If a checkbox in Yii is *not* provided with a value, its default value will be 1. In other words, Yii will set a checkbox’s “value” attribute to 1, unless you specify otherwise. But Yii also does something rather clever: it creates a default value for the checkbox, too. The trick can be seen in the rendered HTML:

```
<input type="hidden" name="LoginForm[rememberMe]" value="0">
<input type="checkbox" id="loginform-rememberme"
    name="LoginForm[rememberMe]" value="1">
```

First, there’s a hidden input with a value of 0 and the name “LoginForm[rememberMe]”. Then comes the checkbox with a value of 1 *and the same name*. Whenever you have two form elements with the same name, the value of the second element will overwrite the value of the first. In this case, if the checkbox is checked, then `$_POST['LoginForm']['rememberMe']` ends up being 1, because the value of the checkbox overwrites the value of the hidden form element. If the checkbox is *not* checked, then `$_POST['LoginForm']['rememberMe']` ends up being 0, as the checkbox will not be set, leaving the hidden form element’s value intact.

To change the checked and unchecked values, pass corresponding arrays to the `field` and `checkbox()` methods:

```
<?= $form->field($model, 'rememberMe', [
    'template' => "<div class=\"col-lg-offset-1 col-lg-3\">{input}
        </div>\n<div class=\"col-lg-8\">{error}</div>",
    'value' => 'Y'])->checkbox(['uncheck'=>'N']); ?>
```

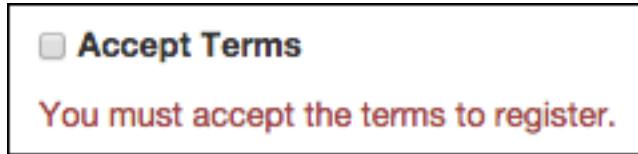


Figure 9.4: The resulting error message if the terms checkbox is not checked by the user.

The value for the checkbox is an HTML attribute, so it's set as part of the third argument to `field()`. The `uncheck` property is part of Yii's checkbox component, so it's set as an argument to `checkbox()`.

The “remember me?” example is one in which the checking of the box is optional, but what if it's required? A logical example is the pervasive “I agree to these terms (that I did not even consider reading)” checkbox.

Enforcing this requirement is simple, and is done in the same way you enforce any requirement on a model attribute: using the model rules. This code is presented in Chapter 4, “Initial Customizations and Code Generations”:

```
['acceptTerms', 'required', 'requiredValue'=>1,
    'message'=>'You must accept the terms to register.'],
```

Assuming the model has the `acceptTerms` attribute, and that the form creates that corresponding checkbox, the form now mandates that the user check the box (**Figure 9.4**).

The last remaining topic relative to checkboxes (and, implicitly, radio buttons) is how to handle updates. In *theory*, you could just add the “checked” attribute to the view code that creates the form. But you would have to do this conditionally, based upon what value the corresponding model attribute has, if any.

Yii has predicted and solved this dilemma, however: the framework will automatically check the box for you if the corresponding model attribute has a “true” value, in PHP terms. This includes the Boolean true, as well as any non-zero number, as well as any non-empty string. If your attribute has any of the following values, Yii will check the box for you on updates:

- true
- 1
- ‘Yes’
- ‘Y’

That's great, but the problem is that “N”, “No”, and “false” also qualify as “true” values.

If you're using values for your checkboxes that do not easily correlate to Booleans, such as Y/N or Yes/No, the solution is to convert the values from non-Booleans

to Booleans before rendering the form. You'll want to *perform* that conversion in the controller, before an update occurs, but the particular functionality should be defined within the model itself. Here's how I would do that...

To start, create a `convertToBooleans()` method. It should define an array of all attributes that need the conversion, and then loop through that array. Within the loop, the attribute's value should be converted to a Boolean:

```
# models/SomeModel.php
public function convertToBooleans() {
    $attributes = ['receiveUpdates', 'receiveOffers', ...];
    foreach ($attributes as $attr) {
        $this->$attr = ($this->$attr === 'Y') ? true : false;
    }
}
```

The list of strings must exactly match the names of the corresponding model attributes, but that's all you need to do to make this work.

Now the controller should invoke this method, but only when an update is being performed:

```
# controllers/SomeController.php
public function actionUpdate($id) {
    $model = $this->findModel($id);
    $model->convertToBooleans();
    // Rest of the method (show & handle the form).
```

Now the checkbox(es) in the form will be automatically checked as appropriate. Note that even with that code, the form should end up assigning Y/N or Yes/No to the model attributes, as that's what would be stored in the database:

```
<?= $form->field($model, 'attributeName', ['value' => 'Y'
])->checkbox(['uncheck'=>'N']); ?>
```

If you have any kind of user model, you presumably have a password attribute, required for logging in. This leads to two issues to be addressed:

- Confirming the password during registration
- Securely storing the password

Let's quickly look at handling both in Yii.

By now you've certainly been asked to confirm your password upon registration many thousands of times. The theory is the confirmation ensures the user knows

exactly what the password is because it was entered twice. That's a theory, anyway. Before showing you how to handle this situation in Yii, I'll first put forth the case that password confirmation isn't necessary.

Many sites are forgoing the password confirmation these days, and with good reason: it's an extra hassle that provides no extra security. Sure, in theory you'll know your password better because you entered it twice, but how many times have you done that just to forget it later anyway? A lot, right? A better solution is to just take the user's password once, and if the user forgets it, have a good "forget password" system in place. That's it. But if you still want to do a password confirmation...

Let's assume you have a `User` model that extends `ActiveRecord`. In the database, there's a `pass` column for storing the password. The first thing to do is add a password confirmation attribute to the model itself:

```
# models/User.php
class User extends \yii\db\ActiveRecord
    public $passCompare; // Needed for registration!
```

Then, add a rule that says the two password attributes must match:

```
# models/User.php::rules()
return [
    // Other rules.
    ['pass', 'compare', 'compareAttribute'=>'passCompare',
        'on'=>'register']
];
```

You'll notice that this rule only applies during the "register" scenario. This means that the rule will only apply when the model is being saved for the first time (in the `actionCreate()` method of the `UserController`):

```
public function actionCreate() {
    $model = new User(['scenario' => 'register']);
```

Next, your view file must also display the second password input:

```
<?= $form->field($model, 'passCompare')->passwordInput() ?>
```

And that will do it (**Figure 9.5**)!

{TIP} To support users changing their passwords, which naturally makes sense, change the password comparison rule to also apply to the "update" scenario.

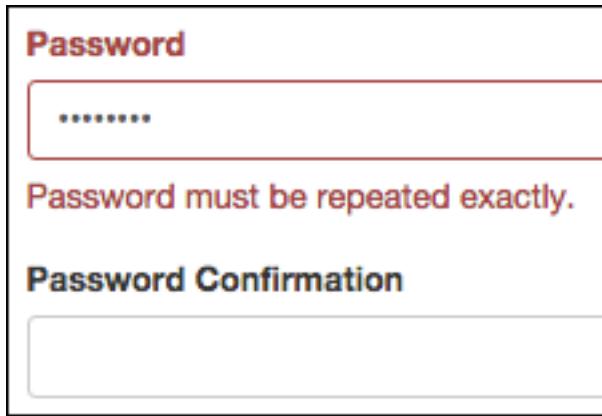


Figure 9.5: Password confirmation is now part of the registration form (for better or for worse).

Another common issue regarding passwords is how you secure them. Typically, one stores a *hashed* version of a password upon registration, not the actual password itself. Then, when the user logs in, the submitted login password is hashed using the same code, and the two hashes are compared. A MySQL function provides the simplest way to do all this:

```
INSERT INTO user (username, email, pass) VALUES
('trout', 't@example.com', SHA2('bypass', 512))
```

And:

```
SELECT * FROM user WHERE (email='t@example.com' AND
pass=SHA2('bypass', 512))
```

(Obviously those values would come in as PHP variables in a real site; the point is demonstrating the underlying commands).

The same approach can be taken using Yii, too. If just using a MySQL function as in the above, you can set the password using an Expression, as Chapter 8, “Working with Databases,” explains:

```
# models/User.php
public function beforeSave() {
    if ($this->isNewRecord) {
        $this->pass = new Expression('SHA2(:pass, 512)',
            array(':pass' => $this->pass));
    }
    return parent::beforeSave();
}
```

That code uses an `Expression` to set the value of the password via the `SHA2()` function but only for new records. That's obviously how the registration (i.e., `INSERT`) would work; you'd need to do a similar thing with `SHA2()` upon login.

That approach will work, and may be secure enough for some sites, but `SHA2()` has been cracked and it also requires a relatively current version of MySQL with SSL support enabled. An alternative, and more secure, approach is to hash the password in PHP.

Built into Yii 2 is the `generatePasswordHash()` method, defined within the `yii\base\Security` helper class. It's quite easy to use, only requiring the data to be hashed as its first argument. However, unlike some of the other helper classes, `Security` is an application component in Yii, accessible via `Yii::$app->getSecurity()`.

```
$pass = Yii::$app->getSecurity()->generatePasswordHash($pass);
```

The `generatePasswordHash()` method uses PHP's `password_hash()` function, currently the most secure way of hashing data. It does require PHP 5.5.0 or greater.

To use a PHP hashing function in your model upon registration, add a `beforeSave()` event handler, as Chapter 5, “[Working with Models](#),” explains:

```
# models/User.php
public function beforeSave($insert) {
    if ($insert) {
        $this->pass = Yii::$app->getSecurity()
            ->generatePasswordHash($this->pass);
    }
    return parent::beforeSave($insert);
}
```

(Of course, the model must also include the namespace.)

The method will securely generate a new, unique salt to use as part of the encryption. As the salt won't be known, or stored, another method validates the password upon login (being a hash, no decryption is possible):

```
if (Yii::$app->getSecurity()->validatePassword($pass, $storedHash)) {
    // Match!
} else {
    // Not a match!
}
```

Chapter 11, “[User Authentication and Authorization](#),” demonstrates this workflow.

Next, let's handle uploaded files in Yii using the MVC approach. In non-framework PHP, handling uploaded files, while not hard, is a different process than handling

other types of form data. Before going into the Yii approach, you should already know how to handle uploaded files using straight PHP, be able to set the correct permissions on folders, and do the other things that need to be in place for any PHP script to handle an uploaded file.

{NEW} In Yii 2, the “file” validator uses the file info extension by default, for better security and results.

The CMS project already has a good model to use for this example: `File`. But that model works a bit differently than many uses of uploaded files, so let’s come up with a new hypothetical: say users can upload an avatar image. It can be in JPG or PNG formats (no animated GIFs!), and must be smaller than 100KB in size. To start, create a new attribute for the model:

```
# models/User.php
class User extends \yii\db\ActiveRecord {
    public $avatar;
```

Next, set the rules in the model:

```
# models/User.php::rules()
// Other rules
['avatar', 'file', 'skipOnEmpty' => true,
 'maxSize' => 102400, 'extensions' => 'jpg, jpeg, png']
```

This rule uses the “file” validator, which in turn uses the `FileValidator` class. Its writable attributes include:

- `extensions`, the allowed file extensions (case-insensitive)
- `maxFiles`, the number of files that can be uploaded
- `maxSize`, the maximum number of bytes allowed
- `minSize`, the minimum number of bytes required
- `skipOnEmpty`, whether the file is required

Thus, the above code says that the file is optional, but if provided must be under 100KB, and has to use the `.jpg`, `.jpeg`, or `.png` extension.

Plus, you can set error messages using the `tooBig`, `tooMany`, `tooSmall`, and `wrongExtension` attributes:

```
# models/User.php::rules()
// Other rules
['avatar', 'file', 'skipOnEmpty' => true,
 'maxSize' => 102400, 'extensions' => 'jpg, jpeg, png',
 'tooBig' => 'The avatar cannot be larger than 100KB.',
 'wrongExtension' => 'The avatar must be a JPG or PNG.']}
```

And that takes care of all the validation!

{NOTE} More practically, one would use the “image” validator, which is an extension of “file”, but the larger goal of the chapter is to teach handling the uploading of any file type.

Next, let’s turn to the view file that displays the form. In order for PHP to be able to handle an uploaded file, the form must use the “enctype” attribute, with a value of *multipart/form-data*. To have a Yii form do that, set the “options” when invoking `beginWidget()`:

```
use yii\widgets\ActiveForm;
$form = ActiveForm::begin(
    [
        'enableAjaxValidation' => false,
        'options' => ['enctype' => 'multipart/form-data'],
    ]
);
```

{WARNING} Forgetting to set the “enctype” attribute is a common cause of problems when attempting to upload files.

Also remember that such forms must use the POST method, which is the default. And “enableAjaxValidation” is disabled because uploaded files cannot be validated via Ajax (more on Ajax validation in Chapter 14)."

Finally, create the file input in the form:

```
<?= $form->field($model, 'avatar')->fileInput() ?>
```

Other than those particular requirements, the rest of the form would be as normal, including a submit button and the ending of the ActiveForm widget.

Finally, there’s the controller that handles the uploaded file. The code created by Gii will look like this:

```
# controllers/UserController.php
public function actionCreate() {
    $model = new User();
    if ($model->load(Yii::$app->request->post())
        && $model->save()) {
        return $this->redirect(['view', 'id' => $model->id]);
    } else {
        return $this->render('create', [
```

```
        'model' => $model,
    ]);
}
}
```

That code takes care of everything except for the uploaded file. File uploads are handled using the `UploadedFile` class, the Yii equivalent to PHP's `$_FILES` array. Invoke its `getInstance()` method to access the uploaded file. This requires a bit of altered logic in the method:

```
$model = new User();
if (Yii::$app->request->isPost) {
    $model->load(Yii::$app->request->post());
    $model->avatar =
        UploadedFile::getInstance($model, 'avatar');
    if ($model->save()) {
        // All good!
        return $this->redirect(['view',
            'id' => $model->id]);
    }
}
return $this->render('create', ['model' => $model]);
```

This requires that the controller include the namespace, too:

```
use yii\web\UploadedFile;
```

As with any other model attribute, the model cannot be saved if it does not pass validation, including the validation of the file.

Finally, after saving the model, save the physical file to the server's file system:

```
if ($model->save()) {
    $model->avatar->saveAs('path/to/destination');
```

When the file is optional, as in my example, you should check that the attribute isn't null before attempting to save the file:

```
if ($model->save()) {
    if ($model->avatar !== null) {
        $model->avatar->saveAs('path/to/dest');
    }
}
```

Of course, you need to set the “path/to/destination” to something meaningful, explained next.

The “path/to/destination” value provided to the `saveAs()` method needs to indicate both the destination directory for the file *and* the file’s name.

With the destination, for security reasons, it’s best to store uploaded files outside of the web root directory, or at least in a non-public web directory. I always try to go outside of the web root directory, and with a Yii site, using a subfolder among the other application folders makes sense. For this example, let’s assume you’ve created an `avatars` directory in the application root directory (and set the permissions accordingly). Having done so, you can use this code to get a reference to that directory:

```
$dest = Yii::getAlias('@app/avatars');
```

{NOTE} Once you store an uploaded file so that it’s not directly available in the browser, you need to use a *proxy script* to display it in the browser. Chapter 16, “Leaving the Browser,” explains that.

Next, there’s the file’s name to determine. The `UploadedFile` object will have the following useful properties:

- `error`, an error code, if an error occurred
- `extension`, the file’s extension (from its original name)
- `name`, the file’s original name on the user’s computer
- `size`, the size of the uploaded file in bytes
- `tempName`, the path and name of the file as it was initially stored on the server
- `type`, the file’s MIME type

As the model attribute is assigned the value of the `UploadedFile` object, you could do this:

```
$model->avatar->saveAs($dest . '/' . $model->avatar->name);
```

For improved security, though, it’s also best to rename uploaded files. In this particular situation, where the uploaded file is directly related—in a one-to-one manner—with a user record, I’d use the unique user’s ID for the file’s name, although with its original extension intact:

```
$model->avatar->saveAs($dest . '/' . $model->id . '.' .
$model->avatar->extension);
```

Files that are uploaded in association with a model—such as an avatar for a user—pose a logical problem when it comes to updating the model. A user might change other pieces of information, such as the password, without touching the uploaded file.

To handle this situation, you must first address the validation rules in the model. If the file is required, you would want to make sure it's required only upon insertions of new records:

```
# models/User.php::rules()
// Other rules
['avatar', 'file', 'skipOnEmpty' => false,
 'maxSize' => 102400, 'extensions' => 'jpg, jpeg, png',
 'on' => 'register'],
['avatar', 'file', 'skipOnEmpty' => true,
 'maxSize' => 102400, 'extensions' => 'jpg, jpeg, png',
 'on' => 'update']
```

Next, you'll need to have the update form show the current value of the file. This could be the current image in the case of an avatar, or the original file name and size for something not visual. How you do this depends upon the file, your models, and so forth, but you should be able to figure that part out.

Then in the controller that handles the form's submission, you can use the same code already explained to handle the uploaded file. If the file was required, then it will only pass validation if a new file was provided. If the file was not required, then you don't want to blindly do this:

```
$model->avatar = UploadedFile::getInstance($model, 'avatar');
```

That would overwrite any existing file value with a NULL value (which would be bad). Instead, check for the presence of an uploaded file first:

```
$model->load(Yii::$app->request->post());
$upload = UploadedFile::getInstance($model, 'avatar');
if ($upload !== null) $model->avatar = $upload;
if ($model->save()) {
    if ($upload !== null) {
        $model->avatar->saveAs('path/to/destination');
    }
}
```

The `UploadedFile` instance is first assigned to a local variable. Then that local variable will be used as a flag to know whether or not a new file was uploaded.

Similar to the issue of handling partial model updates is how you handle incomplete form submissions. If the user submitted a form, including an uploaded file, but failed to complete the form, you'll need to make a decision as to how to handle that.

The simple option is to indicate to the user that they need to re-select the file to be uploaded. This isn't an uncommon approach, but it's suboptimal as a user experience.

A better solution is to save the uploaded file upon the first submission, and then note its presence in a session. You can then populate the attribute from the session upon subsequent form submissions.

If you have the need to upload multiple files with one model instance, there are a couple of ways you can do that. If the files are different, such as an avatar and a resumé, those would be two different attributes and you can handle each separately in the same way as you handle the one.

If the user could provide multiple files for the same attribute, things get more complicated but not that much more complicated. First, in the model, use the same validation rules you otherwise would, but also set the `maxFiles` attribute:

```
# models/SomeModel.php::rules()
// Other rules
['images', 'file', 'skipOnEmpty' => true,
 'maxSize' => 102400, 'extensions' => 'jpg, jpeg, png',
 'maxFiles' => 3]
```

Then, in the form, create the file input type, but allow for multiple uploads:

```
<?= $form->field($model, 'images[]')->fileInput(
    ['multiple' => true, 'accept' => 'image/*']) ?>
```

The “accept” parameter allows you to indicate the allowed MIME types.

Finally, in the controller, instead of calling `UploadedFile::getInstance()`, call `UploadedFile::getInstances()`, which will return an array of uploaded file instances, usable in a loop:

```
$files = UploadedFile::getInstances($model, 'images');
foreach ($files as $file) {
    if ($file !== null) {
        // Use $file->saveAs()
    }
}
```

There are several HTML elements that use lists of data:

- Check box list
- Drop down list
- List box

- Radio button list

These elements all take arrays for their data, but you'll often want those arrays to come from other models. For example, say you want to create a drop down list that allows an administrator to change the owner of a page (reflected in the `Page` model's `user_id` attribute). You'd want to pull all the `User` model instances from the database, and use the IDs as the drop down menu's values, and the names as the displayed labels.

You cannot just provide the results of a `User::findAll()` query to the `HTML` element, as `User::findAll()` returns an array of objects. Instead, use another helper class, specifically the “array” helper:

```
use yii\helpers\ArrayHelper;
```

This class, new in Yii 2, provides a series of methods for easily working with arrays.

For this example, you'll want to use the class's `map()` method to convert an array of objects into a more usable format. The method's first argument is the source, the second is the values to use as the keys, and the third is the values to use as the array values:

```
use app\models\User;
$data = User::find()->all();
$options = ArrayHelper::map($data, 'id', 'username');
echo Html::activeDropDownList($model, 'user_id', $options);
```

This will work for you just fine, and even pre-select the right option when performing an update. Still, there are two ways you could improve upon this:

- Only display the users that have the authority to be owner's of a page
- Only fetch the `User` class's `id` and `username` attributes, as those are the only two being used

Both issues are addressed by configuring the `findAll()` query.

Working with multiple models in the same form is the last subject covered in this chapter. In a specific way, an example of this was just presented: an attribute in one model is related to another model. Next, the chapter expands on that example in a couple of ways. Finally, the chapter explains how to create two model instances using one form.

The previous example demonstrated using one model to populate a drop down list for another model. In that situation, there is a one-to-many relationship between the two models. A more complicated situation exists when there's a many-to-many relationship between two models, such as `Page` and `File` in the CMS site. Because

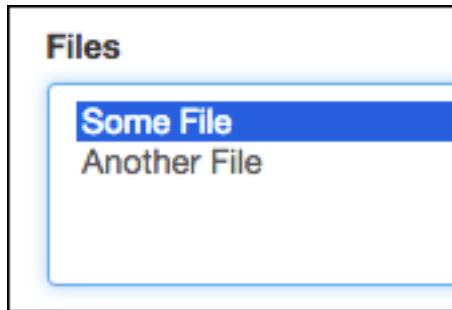


Figure 9.6: The drop down for selecting related files.

of the many-to-many relationship between these two, neither `Page` nor `File` has a foreign key to the other. Instead, a junction table is used: `page_has_file`. The table itself has only two columns: `page_id` and `file_id`. Each file associated with a page is represented by a record in this table. Still, it's not that hard to manage this relationship. One could, on the page management page, indicate the files that should be associated with—and presumably linked from—that page.

On the form for adding (or updating) a page, the user needs to be able to select multiple files to associate with the page:

```
use app\models\File;
use yii\helpers\ArrayHelper;
$data = File::find()->all();
$options = ArrayHelper::map($data, 'id', 'name');
echo $form->field($model, 'files')
    ->dropDownList($options, ['multiple' => 5]);
```

That `dropDownList()` method will create a drop-down of size 5 (five items will be shown), populated using the list of files, and the user can select multiple options (**Figure 9.6**).

This code works because `Page` has a `getFiles()` method, which effectively defines a `files` attribute. You can even add an attribute label that would be used in the form. You can even add a rule to the `Page` model to make sure this part of the form validates.

{TIP} When you create a relation, that relation becomes an attribute of the model.

Now that the form is working—in this case allowing you to select multiple files to be associated with a page, you'll need to update the controller to handle the file selections. Within the `actionCreate()` method (of `PageController`), after the page has been saved, loop through each file and add that record to the `page_has_file` database table. This is a great time to use a prepared statement and a transaction.

```
# controllers/PageController.php
public function actionCreate() {
    $model = new Page();
    if ($model->load(Yii::$app->request->post()) && $model->save()) {
        $files = Yii::$app->request->post('files')
        foreach ($files as $file_id) {
            // Save to the `page_has_file` table.
        }
        return $this->redirect(['view', 'id' => $model->id]);
    } else {
        return $this->render('create', [
            'model' => $model,
        ]);
    }
}
```

Personally, I would be inclined to use Direct Access Objects and prepared statements within the `foreach` loop to perform that task:

```
# controllers/PageController.php::actionCreate()
if ($model->save()) {
    $q = "INSERT INTO page_has_file (page_id, file_id)
          VALUES (:model->id, :file_id)";
    $cmd = Yii::$app->db->createCommand($q);
    $files = Yii::$app->request->post('files')
    $cmd->bindValue(':file_id', $file_id, PDO::PARAM_INT);
    foreach ($files as $file_id) {
        $cmd->execute();
    }
    // Et cetera
}
```

Now, when a new page is created, one new record is created in `page_has_file` for each selected file.

There's still one more consideration: the update process. The final step is to update the `page_has_file` table when the form is submitted (and the page is updated). This could be tricky, because the user could add or remove files, or make no changes to the files at all. The easiest way to handle all possibilities is to clear out the existing values (for this page) in the `page_has_file` table, and then add them in anew. To do that, in `actionUpdate()` of the `PageController`, you would have:

```
# controllers/PageController.php::actionUpdate()
if ($model->save()) {
    $q = "DELETE FROM page_has_file
          WHERE page_id={$model->id}";
```

```
$cmd = Yii::$app->db->createCommand($q);
$cmd->execute();
```

Then you use the same foreach loop as in `actionCreate()` to repopulate the table.

A more complicated but professional approach would be to use Yii 2's concept of "dirty attributes" to see which values changed. If the `files` attribute is dirty, you'd then update the `page_has_file` table accordingly. You could either use the above "delete and re-insert" approach, or add logic that specifically deletes the files that have been disassociated and inserts the new ones.

The previous example demonstrates how to create and handle a form in which multiple instances of a model is associated with a single instance of another. Sometimes, you may only have a one-to-one relationship but you'll actually want to create new instances of both models at one time. For example, in my *PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide* book I used an example in which an administrator could add works of art as products to be sold in an e-commerce site. In the form for adding a print, the administrator could select an existing artist or add a new artist while adding the print. How you do this is easier than you might think.

In your controller (for whichever model is primary), create instances of both objects and pass them to the view:

```
# controllers/ArtController.php
public function actionCreate() {
    $art = new Art();
    $artist = new Artist();
    return $this->render('create', [
        'artist'=>$artist,
        'art'=>$art,
    ]);
}
```

From there, it's rather simple:

- Create elements for both models in the view
- Back in the `actionCreate()` method, mass assign the *primary* model's attributes
- Then, mass assign the *secondary* model's attributes
- If the secondary model is not NULL, save it in the database and assign the new ID value as the foreign key in the *primary* model
- Save the primary model

And that should do it. For added reliability, you could use transactions here, as explained in Chapter 8.

Alternative, you can use the `Model::loadMultiple()` method, new in Yii 2, to populate the models. Its first argument are the models to be loaded and its second is the data source (e.g., a posted request):

```
# controllers/ArtController.php
public function actionCreate() {
    $art = new Art();
    $artist = new Artist();
    if (Model::loadMultiple([$art, $artist],
        Yii::$app->request->post())) {
        // Handle the form submission.
    }
    return $this->render('create', [
        'artist'=>$artist,
        'art'=>$art,
    ]);
}
```

Another practical use of this concept would be to allow an author to upload a file as part of a page creation. To do that, combine the information presented here with the earlier code and concepts for handling uploaded files.

The previous example sketches out how to create two different models from one form. Sometimes you'll want to support creating multiple instances of the same model type at once (e.g., for data entry purposes). The Yii manual refers to this as “[collecting tabular input](#)”, as this implicitly creates a table of data.

As a hypothetical, say you have a `Book` class, with `title` and `author` properties (the latter actually being linked to the `Author` model) and you want to create a page that allows for multiple books to be added at once. The `create` method first generates an array of model instances and passes them to the view file:

```
public function actionCreate() {
    $books = [];
    for ($i = 0; $i < 5; $i++) {
        $books[] = new Book();
    }
    $this->render('create', ['books' => $books]);
}
```

The view file needs to create the form elements, one set for each model instance. Logically, you'll want to create these in a loop. However, you cannot create multiple form elements with the same name. The trick is to prefix the element name with an index value:

```
# views/book/_form.php
foreach ($books as $index => $book) {
    echo $form->field($book, "[{$index}]title")->textInput();
}
```

This code creates form elements with the unusual name value “[0]title”, “[1]title”, and so on. There would also be elements named “[0]author”, “[1]author”, etc.

The code in the create method can use the `loadMultiple()` method just mentioned to quickly grab them all from the form data:

```
public function actionCreate() {
    $books = [];
    for ($i = 0; $i < 5; $i++) {
        $books[] = new Book();
    }
    if (Model::loadMultiple($books, Yii::$app->request->post())) {
        foreach ($books as $book) {
            $book->save();
        }
    }
    $this->render('create', ['books' => $books]);
}
```

The above code saves every model instance it can (i.e., that passes validation). You'd want to extend this example to decide what to do if every instance is validated, or if some aren't, etc. For example, you may choose to save and then unset the valid instances, and redisplay the form for error reporting for any remaining invalid instances.

Chapter 10

Maintaining State

Being able to maintain state is required to have any of the functionality expected on today's websites. Without maintaining state, users can't log in, shopping carts don't exist, and much, much more. Through cookies and sessions, server-side technologies such as PHP provide easy and reliable mechanisms for maintaining state in web applications. When using Yii, you could still use the standard PHP approaches, of course. But since you're already using the framework, you ought to use the framework for maintaining state as well.

This chapter explains everything you need to know to maintain state using Yii. The chapter does assume you already know the arguments for and against cookies vs. sessions, so no time is spent explaining when and why you would use one over the other, rather just the "how" is covered. You'll also learn about a Yii-specific option for maintaining some state for a short duration: flash messages.

The first subject in the chapter is cookies: how to create, read, delete, and customize them using Yii. You'll also see how to make your site more secure using cookies. Note that you'll learn here how to use Yii's methods, in lieu of PHP's standard `setcookie()` method for sending and the `$_COOKIES` array for reading.

Yii's `yii/web/Cookie` class defines the structure and functionality for working with an individual cookie, and `yii/web/CookieCollection` provides the interface for interacting with all of a site's cookies. You can access the collection via `Yii::$app->request->cookies`, as cookies are part of the HTTP request a browser makes of a web server. You'll want to grab a reference to this application property, even to send a cookie:

```
$cookies = Yii::$app->request->cookies;
```

{NEW} Yii 2 nicely simplifies working with cookies.

To create a cookie, add a new `Cookie` object:

```
$cookies->add(new \yii\web\Cookie([
    'name' => 'username',
    'value' => $model->username
]));
```

Remember that the cookie's name, and value, are visible to users in their browsers, so one ought to be prudent about what name you use and be extra mindful of what values are being stored.

Once you've created a cookie, you can access it like an array, using the cookie's name as the index:

```
Yii::$app->request->cookies['username'];
```

Or:

```
$cookies = Yii::$app->request->cookies;
echo $cookies['username'];
```

{NOTE} Remember that cookies are only readable on subsequent pages; cookies are never immediately available to the page that set them.

To test if a cookie exists, just use `isset()` on `Yii::$app->request->cookies['username']`, as you would any other variable:

```
if (isset(Yii::$app->request->cookies['username'])) {
    // Use Yii::$app->request->cookies['username']
}
```

To delete an existing cookie, just unset the element as you would any array element:

```
unset(Yii::$app->request->cookies['username']);
```

To delete all existing cookies (for a user on your site), call `removeAll()`:

```
Yii::$app->request->cookies->removeAll();
```

By default, cookies are set to expire when the browser window is closed. To change that behavior, modify the properties of the created cookie:

```
$cookies->add(new \yii\web\Cookie([
    'name' => 'username',
    'value' => $model->username,
    'expire' => time() + (60*60*24); // 24 hours
]));
```

You can manipulate other cookie properties using the above syntax, changing out the specific attribute: `domain`, `httpOnly`, `path`, and `secure`. Each of these correspond to the arguments to the `setcookie()` function, and are properties of the `yii\web\Cookie` class.

For example, if you want to limit a cookie to a specific domain, or subdomain, use `domain`. To limit a cookie to a specific folder, use `path`. To only transmit a cookie over SSL, set `secure` to true.

Cookie validation prevents cookies from being manipulated in the browser. To validate cookies, Yii stores a hashed representation of the cookie's parameters and values when the cookie is sent, and then compares the received cookie's parameters and values against that stored hash, ensuring they are the same. This is default behavior in Yii 2.

If the signature doesn't match, meaning the cookie has been modified since it was sent, attempts to access that cookie in Yii will fail:

```
if (isset(Yii::$app->request->cookies['username'])) {
    // Use Yii::$app->request->cookies['username']
} else {
    // Assume no safe cookie available for use!
}
```

The cookie is hashed using the “`cookieValidationKey`” value, set in the web configuration file:

```
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'request' => [
            'cookieValidationKey' => 'pdbr578XZJniv45FQReoZ3vEdeG073ao',
        ],
    // Even more other stuff.
]
```

For security purposes, be careful with the “`cookieValidationKey`” as it needs to be kept secret.

One thing to watch out for when using forms is the potential for Cross-Site Request Forgery (CSRF) attacks. A CSRF works like so:

- Site A does something meaningful by passing a value in a URL
- Site A requires that the user has a cookie from Site A in order to execute that action.

- Malicious site B has some code on it that unknowingly has users make that same request of site A. This request can even be made through an image tag's `src` attribute.
- If the user still has the cookie from site A, the request will be successful when the user loads the page on site B.

CSRF is a blind attack, in that the hacker cannot see the results of the request, but CSRF is amazingly easy to implement. As an example, let's say an administrator at your site logs in and does whatever but doesn't log out. The administrator therefore still has a cookie in the browser indicating access to the site (i.e., the user could open the browser and perform admin tasks without logging in again). Now let's say that the `src` attribute on malicious site B points to a page on your site that deletes a blog posting. If the administrator with the live cookie loads that page on site B, it will have the same effect as if that administrator went to your site and requested the blog deletion directly. This is not good.

To prevent a CSRF attack on your site, first make sure all significant form submissions use POST instead of GET. You should be using POST for any form that changes server content anyway, but a CSRF POST attack is a bit harder to pull off than a GET attack.

Second, set "enableCsrfValidation" to true in your configuration file, under the "request" component:

```
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'request' => [
            'enableCsrfValidation' => true,
        ],
    ],
    // Even more other stuff.
```

By enabling this setting this, Yii sends a cookie to the user containing a unique identifier. All forms then automatically store that same identifier in a hidden input. The form submission will only be handled then if the two identifiers match. With the case of a CSRF attack, the two identifiers will not match as the form's identifier will not be passed as part of the request. Note that this only works if you're using the `Html` or `ActiveForm` Yii class to create your forms; if you manually create the form tags, Yii won't insert the necessary code for preventing CSRF attacks.

The most important thing to remember about cookies is they are visible to the user in the browser. And unless you're using SSL for the cookies, they are also visible while being transmitted back and forth between the server and the client, which happens on every page request. Be careful of what gets stored in a cookie! If the data is particularly sensitive, use sessions instead of cookies.

With coverage of cookies completed, let's quickly look at sessions in Yii, which are equally easy to work with.

Sessions in Yii are accessed through the “session” application component:

```
$session = Yii::$app->session;
```

The returned object is of type `yii\web\Session`. To start a session, call the object's `open()` method:

```
$session->open();
```

Thanks to how HTML pages are rendered in the framework, you need not worry about those pesky “cannot modify headers” errors that commonly occur with non-framework sites. The `open()` method can be invoked wherever appropriate in your code.

{*NEW*} Yii 2 does not automatically start sessions as Yii 1 did.

You can confirm that a session is open by checking the session object's `isActive` property:

```
$session = Yii::$app->session;
if (!$session->isActive) $session->open();
```

You can store a value in a session by treating the session object like the `$_SESSION` array, or by more formally invoking the `set()` method on it:

```
$session = Yii::$app->session;
if (!$session->isActive) $session->open();
$session['key1'] = 'value';
$session->set('key2', 42);
```

Similarly, you can access an existing value by treating the session object like an array, or more formally using `get()`:

```
$session = Yii::$app->session;
if (!$session->isActive) $session->open();
echo $session['key1'];
echo $session->get('key2');
```

To remove a session variable, apply `unset()`, as you would to any other variable:

```
unset($session['key1']);
```

Or you can invoke `remove()`, if you prefer to use Yii's session methods:

```
$session->remove('key2');
```

Frequently, for debugging purposes or to store it in the database, it's helpful to know the user's current session ID. That value can be found in `Yii::$app->session->id`.

Those are the basics, and there's nothing really unexpected here once you know where to find the session data. The more complex consideration is how to configure sessions for your Yii application.

Change how your Yii site works with sessions using the primary configuration file. Within the file, add a "session" element to the "components" array, wherein you customize how the sessions behave. The key attributes are:

- `cookieParams`, for adjusting the session cookie's arguments, such as its lifetime, path, domain, and HTTPS-only
- `gcProbability`, for setting the probability of garbage collection being performed, with a default of 1, as in a 1% chance
- `name`, for setting the session's um, name, which defaults to `PHPSESSID`
- `savePath`, for setting the directory on the server used as the session directory, with a default of `/tmp` (on *nix systems)
- `timeout`, for setting after how many seconds a session is considered idle, which defaults to 1440
- `useCookies`, a Boolean indicating if a session cookie should be used or not

For all of these, the default values are the same as those that PHP sessions commonly run using.

For security purposes, I normally prefer to move the session data out of the default `/tmp` directory, and put them in a directory that only the individual site will use. The following example also changes the session name:

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'session' => [
            'savePath' => '/path/to/my/dir',
            'sessionId' => 'Session'
        ],
        // Even more other stuff.
    ]
];
```

The save path, in case you're not familiar with it, is where the session data is stored on the server. By default, this is a temporary directory, globally readable and writable. Every site running on the server, if there are many—and shared hosting plans can have dozens on a single server—share this same directory. This means that any site on the server can read any other site's stored session data. For this reason, changing the save path to a directory within your own site can be a security improvement.

An alternative to the default behavior of storing session data in the file system is to store it in a database. Doing so adds a layer of security and better supports distribution of a site across multiple servers. To implement this storage solution in Yii, change the session class used from the default `yii\web\Session` to `yii\web\DbSession`:

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',
            'db' => 'session_db',
        ],
    ],
    // Even more other stuff.
```

You can also perform any other session configuration changes in that code block, too. The `DbSession` class extends `Session`, so it inherits the properties you've already seen.

If you choose this route, create the following table in the session database:

```
CREATE TABLE session (
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

Yii 2 also supports storing session data in a cache, [Redis](#), or [Mongo](#), by similarly changing and configuring the class used for session handling.

When the user logs out, you should formally eradicate the session. To do so, call `Yii::$app->session->destroy()` to get rid of the actual data stored on the server and to obliterate the session itself:

```
# controllers/SiteController.php::actionLogout()
$session = Yii::$app->session;
if ($session->isActive) {
    \Yii::$app->session->close(); // End the session
    \Yii::$app->session->destroy();
}
```

Yii supports a third route for maintaining state, rather ingenious, and not built into PHP out of the box: *flash messages*. Flash messages provide functionality that's commonly needed by websites: an easy way to create and display one-time errors or other messages. Flash messages aren't the same kind of storage mechanism as cookies or sessions, however, as they are automatically cleared once used.

{NEW} In Yii 2, the default behavior is for flash messages to be retained until used, not merely retained for a limited number of URL requests.

Flash messages are most often used to convey the success or error of a recent user action. Flash messages are great conveyances when redirection is also involved, as they provide a way to pass a message to be displayed on another page.

Create a flash message by calling the `setFlash()` method of the `Session` object. Provide to this method two arguments: an identifier and a value. This is normally done in a controller:

```
# controllers/SomeController.php
public function actionSomething() {
    if (true) {
        Yii::$app->session->setFlash('success',
            'The thing you just did worked.');
    } else {
        Yii::$app->session->setFlash('error',
            'The thing you just did DID NOT work.');
    }
    return $this->render('something');
}
```

The identifier is a reference to a specific flash message. Common identifiers are simple labels like "success" and "error", but identifiers can be anything. One recommendation would be to align your flash message labels with your CSS classes, for reasons you'll soon see. As for the message itself, it must be a simple, scalar data type, such as a string, intended to be relayed to the user.

To use a flash message, invoke the `hasFlash()` method to see if a given flash message exists. Then use `getFlash()` to retrieve the actual message. This is most logically done in a view file:

```
<?php if(Yii::$app->session->hasFlash('success')):>
    <div class="info">
        <?php echo Yii::$app->session->setFlash('success'); ?>
    </div>
<?php endif; ?>
```

Once the `getFlash()` method is called to retrieve a flash message, the flash message is removed from the session. You cannot retrieve a flash message twice.

To clear a flash message without using it, invoke `setFlash()`, providing the same identifier but no value:

```
// Whatever code.
Yii::$app->user->setFlash('success', null);
```

Alternatively, you can pass a third argument of `false` to the `setFlash()` method, which tells Yii to automatically delete the flash message after the next request, whether or not the message was used:

```
# controllers/SomeController.php
public function actionSomething() {
    if (true) {
        Yii::$app->session->setFlash('success',
            'The thing you just did worked.', false);
    } else {
        Yii::$app->session->setFlash('error',
            'The thing you just did DID NOT work.', false);
    }
    return $this->render('something');
}
```

This was the default behavior in Yii 1.

If it's possible that there would be more than one flash message, you can use `getAllFlashes()` to return them all and loop through them:

```
foreach (Yii::$app->session->getAllFlashes() as
    $key => $message) {
    echo '<div class="alert-' . $key . '">' .
        $message . '</div>';
}
```

In that particular bit of code, each flash message's identifier is used as part of the CSS class that wraps the message, letting you easily create one message with an "alert-info" class and another with an "alert-success" class (using the Twitter Bootstrap classnames).

Chapter 11

User Authentication and Authorization

Authentication is the process of identifying a user. On websites, basic authentication is most often accomplished through one of two avenues:

- Directly, by providing a username/password combination (or email/password)
- Via a third-party, such as the user's Twitter or Facebook account

More secure authentication is accomplished using a physical input as well, such as a two-factor authentication (2fa) code, thumbprint, or the like.

Un-authenticated site visitors qualify as anonymous users, or guests.

Related to authentication is *authorization*. Authorization is the process of determining whether the current user is allowed to perform a specific task. Users don't necessarily need to be authenticated to be authorized: for example, an un-authenticated guest can view your home page, but even in those situations, the authorization still uses authentication—specifically the lack of authentication—to dictate what the user can do.

The previous chapter explains how to maintain state in Yii: storing and retrieving data that continues to be associated with a user as she travels from page to page. Now it's time to learn how to fully implement user authentication and authorization in Yii (aka, "auth and auth").

Note that so much of this process has changed so dramatically in Yii 2 that the chapter generally doesn't indicate what's new as other chapters do.

Authentication is a matter of using the proper classes defined in the framework. And although the authentication classes are easy enough to use, the authentication process can be a bit confusing due to the number of pieces involved and the role each plays. To hopefully minimize confusion, let's start by looking at the fundamentals of authentication, and the logic flow entailed.

The Yii authentication process is designed to be quite flexible. Authentication can be performed against:

- Static values (e.g., demo/demo and admin/admin)
- Database tables
- Third-parties (e.g., Facebook or Twitter)
- Lightweight Directory Access Protocol (LDAP)

Authentication can also be as simple as checking an access token or as complicated as requiring a username, plus a password, plus two-factor authentication.

The minimum pieces for authentication in Yii are the “user” application component and a user identifying class. The “user” application component provides the tools for easily working with user instances. The identifying class defines what a user instance is. For example, an identifying user class can be a generic object type, or it can extend `ActiveRecord` to correspond to a database table.

Depending upon the particulars of the authentication, you’ll likely also need one or more controller methods, view files, and even perhaps other models. For example, the code created in the basic Yii application has a `LoginForm` model.

The first step you’ll want to take is to configure the application’s “user” component. This is an object of type `yii\web\User`, and dictates how authentication in the application is handled in terms of:

- Period after which the user is considered idle and will be de-authenticated (i.e., logged out)
- Whether or not cookie-based login—“remember me” functionality—is allowed
- The URL to which the user should be redirected upon login
- The name of the identity class
- And more

All “user” configuration is done in the configuration file by assigning values to the corresponding `yii\web\User` attributes:

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true
        ],
        // Even more other stuff.
    ]
]
```

A representation of the current user—the person accessing the current page—is always available through the “user” application component. This is true whether the user is authenticated (logged-in) or not. The `isGuest` attribute stores a Boolean indicating authentication status:

```
if (Yii::$app->user->isGuest) {
    // Do whatever.
} else {
    // Do this.
}
```

The “user” component’s `identityClass` property sets the class for individual user instances: when a user logs in, an object of this type is created. This is a class that defines what a user looks like. In Yii 2, this class must implement `IdentityInterface`. As with any interface in OOP, the class implementing the interface must define certain methods. With `IdentityInterface`, those methods are:

- `findIdentity()`
- `findIdentityByAccessToken()`
- `getId()`
- `getAuthKey()`
- `validateAuthKey()`

If you look at the original `User` model defined in the basic Yii application, you’ll see definitions of each of these. The first two methods look up a user instance by ID or access token (the latter would be used by an API, for example). The `getId()` method returns the user’s ID (for a previously authenticated user). And the last two methods are used for more secure logins with cookies. For more details on the role of each method, and how each must be defined, check out the class reference for [IdentityInterface](#).

Note that no method overtly validates a password. While your identity class will likely have such, it’s not a requirement of the *interface*. More importantly, as the `User` class only needs to implement an interface, it can extend any class you want. You can have your `User` class extend `ActiveRecord` when storing users in the database.

When the user-provided data passes authentication, the `yii\web\User` class’s `login()` method is invoked. It saves the authenticated user’s identity in the “user” component. From there on, different controllers can use the saved user identity to determine *authorization*. Controllers will do so in one of two ways:

- Basic access control (list-like)
- Role-Based Access Control (RBAC)

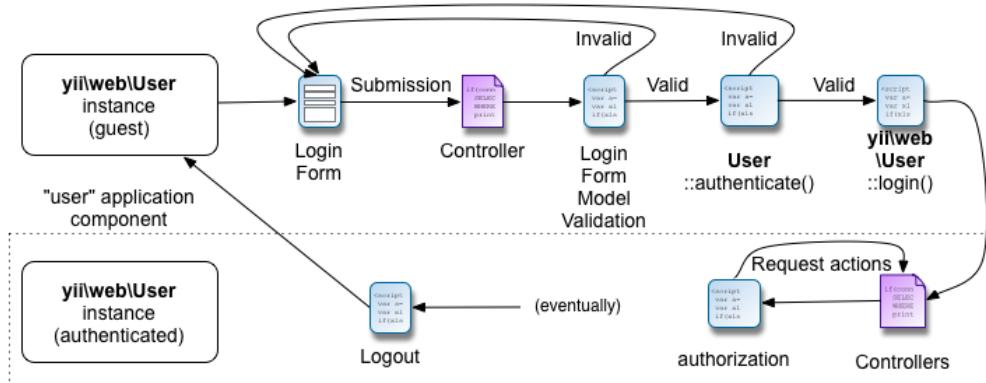


Figure 11.1: The authentication and authorization process.

Finally, the user will log out. Logging out involves calling the `logout()` method of `yii\\web\\User`, thereby removing the user's saved identity.

Of course, thrown into this mix, you also have the login form, which is tied to a model, which entails its own validation. **Figure 11.1** shows the logic flow of this entire process.

With that overview in mind, let's now walk through the authentication process using the code created in the basic Yii application.

The default application created by the Yii framework has built-in authentication using hard-coded values. It does so using these files:

- `models/LoginForm.php`
- `models/User.php`
- `views/site/login.php`

And there's also some code added to `controllers/SiteController.php` that comes into play. The controller file gets the action going, of course. The view file is the login form itself. The `LoginForm` model defines the rules and behaviors for the login data. And the `User` class defines what a site's user is.

Let's follow this logic through its execution, and at the end of this section, an image will demonstrate that workflow, too.

The URL to login will be `example.com/index.php/site/login` (or a variation on that), as the basic application puts login/logout functionality in the “site” controller by default. When the user clicks on a link to go to the login page, she'll go through the “site” controller and call the `actionLogin()` method. That method is defined as (with some comments and Ajax functionality removed):

```
1 # controllers/SiteController.php::actionLogin()
2 public function actionLogin() {
3     if (!\Yii::$app->user->isGuest) {
4         return $this->goHome();
5     }
6
7     $model = new LoginForm();
8     if ($model->load(Yii::$app->request->post()) && $model->login()) {
9         return $this->goBack();
10    } else {
11        return $this->render('login', ['model' => $model]);
12    }
13 }
```

The code first checks if the user is logged in—is not a guest, in which case, the user is redirected to the home page. There's no reason to log in when you're already logged in!

Next, a new object of type `LoginForm` is created (line 7). That class is defined in the `LoginForm.php` model file. The model extends `yii\base\Model`, and has three public attributes: `username`, `password`, and `rememberMe`. There's also a private `_user` attribute, used upon logging in.

Line 8 of the code is the most interesting one. It checks for the form's submission by loading the data and attempts to log the user in, by calling the `LoginForm::login()` method. If the user can be logged in, the user will be redirected to whatever URL got her here in the first place.

If the form has not been submitted, or if the user cannot be logged in, the login form is displayed, and the `LoginForm` object is passed along to it (line 11). There's nothing unusual about that form, so I'll leave it to you to examine that view file if needed.

The call to the `login()` method in the above code means the `LoginForm::login()` method must return a Boolean true. Here is that definition:

```
# models/LoginForm.php
public function login() {
```

```
if ($this->validate()) {
    return Yii::$app->user->login($this->getUser(),
        $this->rememberMe ? 3600*24*30 : 0);
} else {
    return false;
}
```

The `login()` method of the model is responsible for registering the `UserIdentity` object to the “user” component. This allows the entire site to track and recognize an authenticated user.

The first line within the method checks that the form data passes the validation rules established in the `LoginForm` class. I’ll return to that shortly. If so, the “user” component’s `login()` method is invoked, providing it the user instance (via `$this->getUser()`) and a cookie duration value. That value is determined by whether or not the user opted to be remembered.

This constitutes the *end* of the login process. If the user is validated, they are registered with the “user” application component. Validation is accomplished within the `LoginForm` and `User` models: `LoginForm` defines the process for the validation, and `User` defines the values.

Returning to the above code, it invokes `$this->validate()`, which is to say the `LoginForm` instance data must pass the model’s data rules. This is basic model validation as defined by the `rules()` method of that model:

```
# models/LoginForm.php::rules()
return [
    // username and password are both required
    [['username', 'password'], 'required'],
    // rememberMe must be a boolean value
    ['rememberMe', 'boolean'],
    // password is validated by validatePassword()
    ['password', 'validatePassword'],
];
;
```

One little trick here is the `validatePassword()` validation requirement. This is an example of a user-defined filter, explained in Chapter 4, “[Initial Customizations and Code Generations](#)”. Here’s that method’s definition:

```
1 # models/LoginForm.php::validatePassword()
2 public function validatePassword($attribute, $params) {
3     if (!$this->hasErrors()) {
4         $user = $this->getUser();
5         if (!$user || !$user->validatePassword($this->password)) {
6             $this->addError($attribute, 'Incorrect username or
;
```

```
7             password.');
8         }
9     }
10 }
```

That method actually requests the authentication: comparing the submitted values against the required values. To start, it only performs this task if the model does not already have any errors (line 3). There's no point in attempting validation if a username or password was omitted.

Next, the method creates a new `User` object, by invoking the `LoginForm::getUser()` method. I'll get back to that shortly.

The rest of `validatePassword()` defines an error under either of two conditions:

- If no user could be found by that username
- If the provided password does not match the stored password for that user

To do the latter, the `User` class's `validatePassword()` method is called. First, though, here's the `getUser()` method, defined in `LoginForm`:

```
# models/LoginForm.php (original)
public function getUser() {
    if ($this->_user === false) {
        $this->_user = User::findByUsername($this->username);
    }
    return $this->_user;
}
```

The `findByUsername()` method simply fetches and returns a user using the username from the static array of values hardcoded into the class.

The `User` instance found here is returned to `LoginForm::validatePassword()`, which then calls `User::validatePassword()`. This is the `originalUser` created by the basic application, not the database-derived ActiveRecord version.

The `User` class's `validatePassword()` method simply compares the already returned user instance's password against the provided password (upon login):

```
# models/User.php (original)
public function validatePassword($password) {
    return $this->password === $password;
}
```

This is the final authentication step in the process.

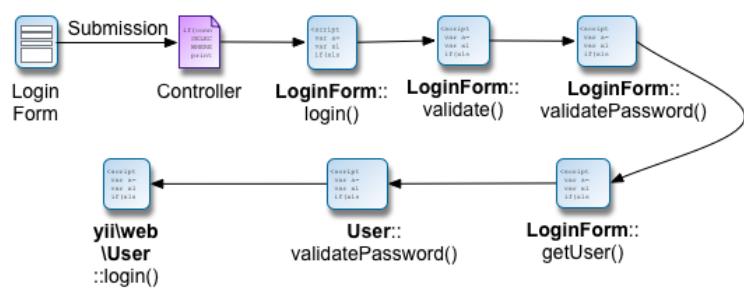


Figure 11.2: How the Yii app gets to the `User::validatePassword()` method.

Figure 11.2 demonstrates the logical flow that got the application to this point.

As a reminder, the returned value from `User::validatePassword()` is used by the `validatePassword()` method of the `LoginForm` model:

```
if (!User || !$user->validatePassword($this->password)) {  
    $this->addError($attribute, 'Incorrect username or password.');//  
}
```

Thus, if the login form values don't authenticate in `User`, a new error is added to the login form model instance. (And that error can be used on the login form view page.)

Finally, the function returns true or false. This returns the logic flow back to the controller:

```
# controllers/SiteController.php::actionLogin()  
if ($model->validate() && $model->login())  
    $this->redirect(Yii::$app->user->returnUrl);
```

The user is redirected if she was able to be logged in. If not, the form will be displayed again.

Whew! This probably seems like a lot of code and logic. And, well, it kind of is. But by separating all the pieces of the authentication process, it's an extremely flexible process. Here's what you have to work with:

- A model (`LoginForm`)
- A view (`views/site/login.php`)
- A controller (`SiteController`)
- Validation (through the model)
- Error reporting (through the view)
- Authentication (through `User`)
- Registration of the authenticated user with the application (through `Yii::$app->user->login`)

Because these are separate components, you can make changes to one without having to even look at the others. You want to authenticate based upon an email address instead of a username? No problem. You want to authenticate against a database? No problem. You want to change what errors are displayed? No problem.

Hopefully you were able to follow this logic, because you'll need to understand the role each part plays in order to customize the authentication process for your own sites. You'll learn how to do that after first learning a couple more things about the “user” component in Yii.

By default, Yii uses sessions to store the user identity. As with any use of sessions, this means that after a relatively short period of inactivity, the user will need to login again. If you'd like users to be recognized by your system for a longer duration, including after closing the browser and later returning, you can tell Yii to use cookies instead of sessions for storing the user identity.

{TIP} As a session's true expiration is partly based upon PHP's garbage collection mechanism, how quickly a session actually expires (once it becomes inactive) depends upon several factors, including how busy your site is.

The first thing you'll need to do is set the `enableAutoLogin` “user” component property to true in your configuration file:

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true
        ],
        // Even more other stuff.
    ]
];
```

The default value for this property is false, but the code created by the basic Yii application configures it to true (i.e., that code is already in the generated configuration file).

That configuration *allows* for cookies to be used. The next thing you need to do is tell Yii to actually use cookies for the user identity. To do that, provide a duration argument to the `login()` method of the `yii\web\User` class (aka, the “user” component of the application). This value should be a number in seconds.

The code generated for you will already do this if the “remember me” checkbox is checked:

```
# models/LoginForm.php::login()
return Yii::$app->user->login($this->getUser(),
    $this->rememberMe ? 3600*24*30 : 0);
```

By default, the cookie will last whatever duration you specified *from the time the cookie is first sent*. If the cookie is set to last for 30 days, that's 30 days from the original login, not from the point of last activity. If you'd like to change it so that the cookie is resent when the user is active, set the `autoRenewCookie` property to true:

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
            'autoRenewCookie'=>true
        ],
        // Even more other stuff.
```

With that configuration, the cookie will automatically be resent with each user request, thereby continuing to push back the cookie's expiration. Yii will continue to use the original duration period for each cookie's expiration value. For that reason, you'd likely want to reduce the duration to a shorter period, such as a few days.

{WARNING} The **autoRenewCookie** feature can adversely affect performance.

If you want to otherwise customize the cookie, configure the **identityCookie** property of the `yii\web\User` object, providing new values using the `yii\web\Cookie` properties (explained in Chapter 10, “[Maintaining State](#)”):

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
            'identityCookie' => [
                'domain' => 'store.example.org',
                'secure' => true
            ]
        ],
        // Even more other stuff.
```

I've just explained *how* you would use cookies instead of sessions to store the user's identity, but a secondary issue is *should you?*. Remember that cookies are less secure than sessions, as they are transmitted back and forth between the client and the server. As always, you must match the level of security to the site.

I would generally recommend that you use only sessions and disable `allowAutoLogin`, unless security is less of a concern for the site in question *and* it would be an unreasonable inconvenience to expect the user to frequently login. Good examples that meet both of these criteria are social media sites such as Facebook. Later in the chapter, though, you'll see how to store other information as part of the user's identity, and you must be careful about doing so when `allowAutoLogin` is enabled (I'll remind you about the security issues then, too, just for safe measure).

{TIP} If you're not allowing auto login (i.e., cookies storage for the user identity), then be certain to remove all references to the "remember me" checkbox from your model and view files.

If you are restricting the site to sessions for the user identity, you can make the system even more secure by restricting the authentication time period to an even smaller duration than the default session duration. To do that, set the `authTimeout` property to a time period, in seconds:

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
            // Next line not needed, as it's the default:
            'allowAutoLogin'=>false,
            'authTimeout' => (60*15) // 15 minutes
        ],
        // Even more other stuff.
    ]
]
```

Finally, you also ought to know how to log out a user. That's accomplished by invoking the `logout()` method of the "user" component (the `yii\web\User` class). This code comes from the "site" controller:

```
public function actionLogout() {
    Yii::$app->user->logout();
    return $this->goHome();
}
```

Now that you have an understanding of authentication in Yii, let's start tweaking the default authentication system. To begin, you can simply change the static authentication. Then you'll see how to perform the more common task of using database authentication.

The first, and by far the easiest, option for authentication in your site is to continue using the static authentication but change the login values. I've worked on a couple of projects built in Yii where only one user ever needed to login: a single administrator. In the rare situations where that's also true for you, then you can open the original ***User.php** file and change this code:

```
private static $users = [
    '100' => [
        'id' => '100',
        'username' => 'admin',
        'password' => 'admin',
        'authKey' => 'test100key',
        'accessToken' => '100-token',
    ],
    '101' => [
        'id' => '101',
        'username' => 'demo',
        'password' => 'demo',
        'authKey' => 'test101key',
        'accessToken' => '101-token',
    ],
];
```

to

```
private static $users = [
    '100' => [
        'id' => '100',
        'username' => 'whatevername',
        'password' => 'whateverpassowrd',
        'authKey' => 'whateverkey',
        'accessToken' => 'whatevertoken',
    ],
];
```

At that's it! (Except this assumes you haven't created a new **User** class as in the CMS example.)

{TIP} Don't forget to remove the hint paragraph from the view, as demo/demo and admin/admin will no longer work.

For security reasons, I prefer administrators to login for each session, though. In these situations, I would also disable the **allowAutoLogin** and remove references to the "remember me" attribute and checkbox.

If you can use static authentication, that's great, but more frequently authentication will be performed against a database table. No matter what the source is for your authentication, you still need to:

- Create a login form that's associated with a model
- Create a `User` class that authenticates the user
- Register the `User` identity class with the “user” component

The process is the same regardless of the source; the actual implementation is mostly a matter of what classes you want to use.

With the authentication changes implemented in Yii 2, switching from static to database-driven authentication is even easier than it was in Yii 1. For this example, let's assume there's a `user` table, and that the user will login by providing a combination of an email address and password. The password is hashed using the Yii `generatePasswordHash()` function, as explained in Chapter 9, “[Working with Forms](#).” This table maps to the `User` class created in the CMS example, and that will be used for the login.

From this point, there are two obvious paths to take. The first is to only use the `User` class, meaning it will act as both the user identity class and as the login model. The second option is to continue using the `LoginForm` class for the login form, while using `User` as the identity. Let's look at both approaches, in that order. But, first, the `User` class needs to be prepared to act in this new capacity (as an identity class).

The class that serves as the user identity needs to implement `IdentityInterface` and five methods:

- `findIdentity()`
- `findIdentityByAccessToken()`
- `getId()`
- `getAuthKey()`
- `validateAuthKey()`

To start, implement the interface:

```
class User extends \yii\db\ActiveRecord
    implements \yii\web\IdentityInterface
```

This class extends `ActiveRecord`, so it can interact with the database, but implements `IdentityInterface`.

Next, those five methods need to be defined in the class. The easiest way to define all of these is to copy the existing definitions from the original `User` class and then update them. For now, three of them aren't going to be used, and can be defined as empty:

```
public static function findIdentityByAccessToken($token,
    $type = null) {
}
public function getAuthKey() {
}
public function validateAuthKey($authKey) {
}
```

The other two methods need to be defined:

```
public function getId() {
    return $this->id;
}
public static function findIdentity($id) {
    return User::findOne($id);
}
```

The `getId()` method returns the ID for the current model instance. The `findIdentity()` method returns a `User` instance provided a user ID. To do so, it simply needs to call the `findOne()` ActiveRecord method, which uses the primary key by default.

Now the `User` class meets the requirements for implementing the interface. There are two more methods to define that will be vital for the login process. The first retrieves a user by email address and is similar to `findByUsername()` defined in the original `User` class:

```
public static function findByEmail($email) {
    return User::find()->where(['email' => $email])->one();
}
```

The second necessary method should validate the submitted login password. The assumption is the class uses `generatePasswordHash()` to encrypt the password during registration:

```
public function beforeSave($insert) {
    if ($insert) {
        $this->pass = Yii::$app->getSecurity()
            ->generatePasswordHash($this->pass);
    }
    return parent::beforeSave($insert);
}
```

Therefore, validation of the password is simply:

```
public function validatePassword($password) {
    return Yii::$app->getSecurity()
        ->validatePassword($password, $this->pass);
}
```

Now the `User` class is usable as a “user” application component identity class. The next decision is whether to use `User` for logging in or to use a separate class for that. Let’s look at both.

It’s quite simple to adopt the original `LoginForm` class to use the new `User`. Start by changing the attributes in `LoginForm` to those you require: `email`, `password`, possibly `rememberMe` (depending upon the site, I’ll remove it), and `_user`:

```
# models/LoginForm.php
class LoginForm extends Model {
    public $email;
    public $password;
    private $_user = false;
    // Et cetera
```

Next, alter the rules accordingly. Instead of username and password being required, `email` and `password` are now required. Also, the `email` should be in a valid email address format. The application of the `validatePassword()` method to validate the password remains:

```
# models/LoginForm.php
public function rules() {
    return [
        // email and password are both required
        [['email', 'password'], 'required'],
        // email must be a syntactically valid email address
        ['email', 'email'],
        // password is validated by validatePassword()
        ['password', 'validatePassword'],
    ];
}
```

Next, remove references to `rememberMe` in the `login()` method:

```
public function login() {
    if ($this->validate()) {
        return Yii::$app->user->login($this->getUser());
    } else {
        return false;
    }
}
```

Finally, the `getUser()` method needs to be updated. The original code calls the `User` class's `findByUsername()` method, which doesn't exist. Update `getUser()` to use `findByEmail()` instead:

```
public function getUser() {
    if ($this->_user === false) {
        $this->_user = User::findByEmail($this->email);
    }
    return $this->_user;
}
```

And that takes care of edits to the `LoginForm` class.

The last remaining change is to the form itself. First, the hint paragraph needs to be removed, as `demo/demo` and `admin/admin` will no longer work. Then the code that displays the “remember me” checkbox should also be excised. Remember me functionality is only good for cookies, so it's useless here. Finally, the form should take an email address, not a username, so those two lines must be changed. The complete form code is now (**Figure 11.3**):

```
<?php $form = ActiveForm::begin([
    'id' => 'login-form',
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => [
        'template' => "{label}\n<div class=\"col-lg-3\">{input}</div>\n<div class=\"col-lg-8\">{error}</div>",
        'labelOptions' => ['class' => 'col-lg-1 control-label'],
    ],
]); ?>

<?= $form->field($model, 'email') ?>

<?= $form->field($model, 'password')->passwordInput() ?>

<div class="form-group">
    <div class="col-lg-offset-1 col-lg-11">
        <?= Html::submitButton('Login', ['class' => 'btn btn-primary',
            'name' => 'login-button']) ?>
    </div>
</div>

<?php ActiveForm::end(); ?>
```

And that's it for the `LoginForm` aspect. You should now be able to login using a previously registered user.

The image shows a login form titled "Login". Below the title, a message reads "Please fill out the following fields to login:". There are two input fields: one labeled "Email" and another labeled "Password". Below these fields is a blue rectangular button labeled "Login".

Figure 11.3: The updated login form.

An alternative to logging in through `LoginForm` is to use the `User` model directly. The argument for this approach is that it reduces the code redundancy. With the `LoginForm` model, you've duplicated two attributes—the email address and password—that are already in `User`. You've also duplicated the logic surrounding those attributes, such as the validation rules and the attribute labels. If you later want to make a simple change, such as changing a label, you'll need to remember to make that change in two places. This isn't a terrible thing, to be sure, but generally redundancies are to be avoided in any software.

Tapping into the `User` model for authentication is not that hard, so long as you remember your *scenarios*. Most of the validation rules apply when a new user is created (i.e., registers) or when an existing user updates her information, but those rules would *not* apply during login. For example, the username would not be required upon login (if you're using the email address to login) and you wouldn't set the user's type then, either.

To make the model work for all situations, I would add new rules for the “login” scenario and exempt every other rule from that scenario. Here's part of that:

```
# models/User.php::rules()
return array(
    // Always required fields:
    [['email', 'pass'], 'required'],
    // Only required when registering:
    [['username'], 'required', 'on' => 'register'],
    // Password must be authenticated when logging in:
    [['pass'], 'validatePassword', 'on' => 'login'],
    // And so on.
```

{NOTE} The `User` model uses `pass` as its password attribute name, not `password` as in `LoginForm`. You'll need to make sure all the code consistently uses the right attribute name.

Next, add to the `User` model the `login()` method as previously explained for `LoginForm`:

```
public function login() {
    if ($this->validate()) {
        return Yii::$app->user->login($this);
    } else {
        return false;
    }
}
```

Because `$this` is a `User` instance, calling `validate()` on it checks the `User` class business rules. If all checks pass, the instance is registered with the “user” component.

Next, create the `actionLogin()` method of the `UserController` class. It needs to create and use an object of type `User` instead of `LoginForm`:

```
# controllers/UserController.php::actionLogin()
if (!\Yii::$app->user->isGuest) {
    return $this->goHome();
}
$model=new User(['scenario' => 'login']);
if ($model->load(Yii::$app->request->post()) && $model->login()) {
    return $this->goBack();
} else {
    return $this->render('login', [
        'model' => $model,
    ]);
}
```

And, finally, create the login form:

```
<?= $form->field($model, 'email')->textInput(['maxlength' => true]) ?>
<?= $form->field($model, 'pass')->passwordInput(['maxlength' => true]) ?>
<div class="form-group">
    <?= Html::submitButton('Login') ?>
</div>
```

And that should do it!

Having seen the code required to use `User` for authentication, you can now appreciate the arguments against this approach. First, you'll end up adding two methods to the class that will only be used during the login process. Second, it makes the logic with the rules a bit more complicated, which could lead to bugs.

That being said, which approach you use—the `LoginForm` or the `User` class—is up to you.

Unlike in Yii 1, Yii 2 has no “state” concept built into the “user” component. This is to say there’s no way to store data specifically on the user instance. Logically, you’ll want to use cookies and sessions for that purpose instead. But data that’s part of the user instance—i.e., defined in the related `User` object—can be accessed through the user identity instance:

```
$identity = Yii::$app->user->identity;  
// Use $identity->email if you want.
```

That code will return null if the user hasn’t been authenticated. To test for that scenario, call the `isGuest()` method:

```
if (!Yii::$app->user->isGuest) {  
    $identity = Yii::$app->user->identity;  
}
```

If you only need to access the user’s ID, you can directly get to it through `Yii::$app->user->id`. This code invokes the `getId()` method of the `User` class.

Now that you (hopefully) have a firm grasp on authentication, it’s time to turn to its sibling, *authorization*. As a reminder, authentication is a matter of verifying who the user is; authorization is a matter of confirming if the user has permission to perform a certain task. Authorization is implemented using access control.

Access control is an integral aspect of any website, dictating what a user can and cannot do based upon factors of your choosing. There are a couple of ways to implement access control in Yii, starting with the Access Control Filter (ACF). This approach is similar to the Access Control Lists (ACL) used by operating systems and is simple to implement in Yii.

{NEW} In Yii 2, Gii does not implement basic access control in generated controllers as it did in Yii 1.

The code generated as part of the basic Yii application uses ACF in the “site” controller. First, it includes the corresponding namespace:

```
use yii\filters\AccessControl;
```

Then the controller attaches ACF to itself via the `behaviors()` method:

```
public function behaviors() {
    return [
        'access' => [
            'class' => AccessControl::className(),
            'only' => ['logout'],
            'rules' => [
                [
                    'actions' => ['logout'],
                    'allow' => true,
                    'roles' => ['@'],
                ],
            ],
            // Other stuff.
        ];
}
```

The code says that the “access” behavior is defined by the `AccessControl` class.

The “only” property dictates to which actions access control should be restricted, if at all. In the above code, access control is only restricted to the “logout” action. You can omit this property to make the rules apply to every action, or use “except” instead, which lists the actions to which no access control applies.

{WARNING} When you add new actions to a controller, be sure to update your access rules!

Next, the “rules” property defines the actual rules: who can do what. You’ll want to define one or more rules for every action to which access control applies. Each rule lists:

- The actions to which the rule applies
- Whether it is an allow or deny rule
- To whom the rule applies

The syntax for defining rules comes from the `yii\filter\AccessRule` class. Once again, look at that class definition to know what attributes exist to which you can assign values.

The “site” controller only has one rule. This rule applies only to the “logout” action and the rule says to allow access to logged-in users. Yii uses two special characters to identify user types:

- ?, anonymous (i.e., not logged-in) users
- @, logged-in users

As a corollary, the same effect—only allowing logged-in users to access the “logout” action—can be had by instead denying access to anonymous users.

These rules are checked in order from top down. Once a rule applies, that’s it: the rules do not continue to be evaluated. If no rules apply, then the action is *not* allowed. In other words, *if access control applies to an action, no access is the default behavior*.

You’ll want to customize the rules to each controller and situation. For example, in a CMS site, anyone (or more specifically, anonymous users) needs to perform create actions with a `User` model and controller, as that constitutes registration. But perhaps only logged-in users can create comments, and only certain users or user types can create posts.

With basic access control, there are two ways to identify or restrict to what users a rule applies:

- By authentication status (only anonymous or only logged-in)
- By IP address

The first possibility has already been covered. Setting rules by IP address is logical if an administrator will only ever be accessing the site from a specific IP address. To do that, set an “ips” value:

```
'rules' => [
  [
    'actions' => ['admin'],
    'allow' => true,
    'roles' => ['@'],
    'ips' => ['127.0.0.1']
  ],
]
```

That code says that the user must be logged-in and coming from the 127.0.0.1 IP address. This, of course, is the IP address for localhost, which means that the administrator is allowed to perform those actions when working on the same computer as the site. To allow remote access, you would need to add the IP address of the administrator’s computer (or network).

IP addresses are a great way to allow for access via localhost, but using IP addresses to identify users over networks is problematic for two reasons. First, a person’s IP address can change frequently, depending upon her Internet access provider. Second, multiple people accessing the same site from the same network (e.g., a company, organization, or school) may all have the same IP address.

There are two more ways you can restrict who can do what. The first is to use the “verbs” property. It takes an array of request types to which the rule applies. For example, if you have a page that both shows some information and displays a form, you might allow everyone to perform a GET request (i.e., see the information) but only logged in users can submit the form (perform a POST request):

```
'rules' => [
    [ // Anyone can "GET" the page
        'actions' => ['someAction'],
        'allow' => true,
        'roles' => ['?'],
        'verbs' => ['GET']
    ],
    [ // Must be logged in to POST
        'actions' => ['someAction'],
        'allow' => true,
        'roles' => ['@'],
        'verbs' => ['POST']
    ]
],
```

Of course, you’d also want to code your view so that it doesn’t show the form to those that can’t submit it, but this rule acts as a security measure.

Or, as another example, you might have a cron or other automated process that regularly performs HEAD requests of an action to confirm that the site is up and running. That kind of request would be restricted to anonymous users.

If you look at the code for the “site” controller, you’ll see there’s also a “verbs” behavior:

```
public function behaviors() {
    return [
        'access' => [ /* Access configuration */ ],
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'logout' => ['post']
            ]
        ]
    ];
}
```

This “verbs” is not the same as the ACF, per se, although it can more bluntly be used for similar effects. The “verbs” filter allows you to define how actions can be

requested: via GET, via POST, etc. Unlike the access control filter, “verbs” does not look at the user when allowing or denying action usage.

The last way in which you can control access to controller actions is to use a matching callback function. Either assign a function name to the `matchCallback` property, or directly assign a PHP function. If a match callback function returns true, access is allowed. If it returns false, access is denied.

A matching function can be used to evaluate more complicated scenarios, but not as complicated as using Role-Based Access Control (RBAC). For example, if the user with an ID of 1 will always be the administrator, you could check for that:

```
'rules' => [
    [
        'actions' => ['someAction'],
        'allow' => true,
        'roles' => ['@'],
        'matchCallback' => function($rule, $action) {
            return (Yii::$app->user->id == 1);
        }
    ],
]
```

As another example, the CMS site has user types: public, author, and admin. You could allow creation to authors and administrators, denying other users:

```
'rules' => [
    [
        'actions' => ['create'],
        'allow' => true,
        'roles' => ['@'],
        'matchCallback' => function($rule, $action) {
            $user = Yii::$app->user->identity;
            return ($user->type == 'author')
                || ($user->type == 'admin');
        }
    ],
]
```

The “roles” parameter requires that the user be logged in. Then the match callback function returns true if the user type is “author” or “admin”.

As another example, you could use a similar rule to restrict deleting of pages to only administrator types.

Callback functions can be very useful, but they also start blurring the lines with RBAC, which is a more sophisticated—and complex—access control tool.

You should also be aware of what happens when Yii denies access to an action. If the user is not logged in and the rule requires that she be logged in, the user will be

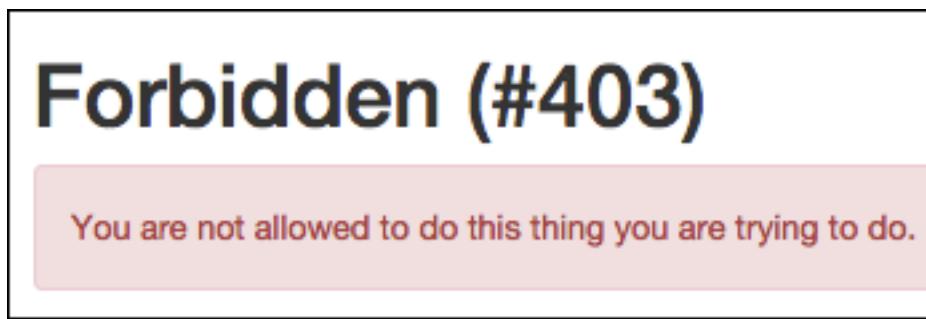


Figure 11.4: A custom denial message.

redirected to the login page by default. After successfully logging in, the user will be redirected back to the page she had been trying to request.

By default, the login route is “site/login”, but you can change this value by configuring the `loginUrl` property of the `yii\web\User` class (which is to say the “user” component).

If the user *is* logged in but does not have permission to perform the task, an HTTP exception is thrown using the error code 403. That value matches the “forbidden” HTTP status code value.

To change an error message associated with an exception, assign a function definition to the “denyCallback” property and throw the exception within it (**Figure 11.4**):

```
'rules' => [
    [
        'actions' => ['someAction'],
        'allow' => true,
        'roles' => ['@'],
        'denyCallback' => function($rule, $action) {
            throw new yii\web\ForbiddenHttpException('You are not
                allowed to do this thing you are trying to do.');
        }
    ],
],
```

When basic access control is too simple for your application, you can move on to the more custom and elaborate way to implement authorization in Yii: using Role-Based Access Control (RBAC). With RBAC, the goal is to identify who is allowed to do what, but RBAC uses a hierarchy that often better correlates to a site’s users. For example, in a CMS site, an administrator has more power than an author who has more power than a public user. Of course, with this hierarchy comes more complexity.

The fundamental unit in RBAC is a *permission*, which is the approval to perform a

specific action. A permission is a single atomic act: edit a page, delete a user, post a comment, etc.

Roles are allotted specific permissions, and then *users* are assigned specific roles. Given an identified user, RBAC can check the user's role, and therefore, whether the user has permission to perform a specific task.

Permissions, roles, and users are the core primitives in RBAC. Although a simple hierarchy descends from users to roles to permissions, permissions can have hierarchical relationships to other permissions, and roles can have hierarchical relationships to both roles and permissions.

Permissions can also be modified using *rules*. Rules contextualize permissions. For example, in a site with users, a user would almost always be able to update their own account information (e.g., change their password), whereas an administrator could update any user record. The administrator would therefore have “update user” permissions while each user would have “update user” permissions with the additional “only their own user record” rule.

In the CMS example, basic permissions for pages might be:

- Add a page
- Edit a page
- Delete a page

A basic role might be an “author”, with that role being able to add a page and edit a page they authored. Another role might be “editor”, with that role inheriting all of the “author” permissions, plus the ability to edit any page. The “administrator” role might inherit all of the “editor” permissions, and also have the ability to delete any page.

To implement RBAC in your site, you should first start by sketching out the various permissions and roles involved. The end goal is to design your hierarchy as efficiently as possible. Understand that you only need to represent permissions that are restricted. In the CMS example, any type of user, including non-authenticated guests, can view a page of content, so that does not need to be represented in the hierarchy.

Figure 11.5 shows a subsection of permissions, roles, and rules for the CMS example. Understand that there's no one right answer here. Any two people could create slightly different hierarchies for the same site (in much the same way that two developers might come up with slightly different database schemes, both of which would work fine).

Once you've sketched out your items and hierarchy, you can begin defining the permissions, roles, and rules in Yii.

RBAC requires the use of an *authorization manager* to function. The authorization manager first defines authorization items, and later performs the act of confirming the user's ability to perform a specific task.

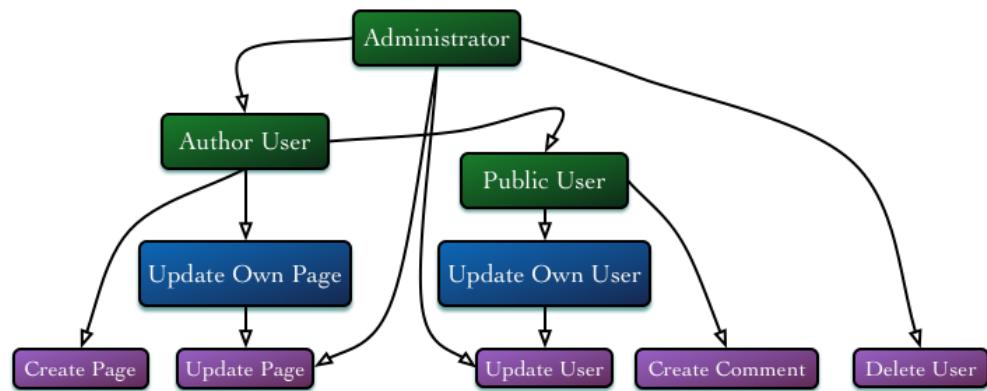


Figure 11.5: Some RBAC items for the CMS example.

```
~/S/yii2-test » ./yii migrate --migrationPath=@yii/rbac/migrations/
Yii Migration Tool (based on Yii v2.0.6)

Total 1 new migration to be applied:
m140506_102106_rbac_init

Apply the above migration? (yes|no) [no]:yes
*** applying m140506_102106_rbac_init
> create table {{%auth_rule}} ... done (time: 0.010s)
> create table {{%auth_item}} ... done (time: 0.009s)
> create index idx-auth_item-type on {{%auth_item}} (type) ... done (time: 0.015s)
> create table {{%auth_item_child}} ... done (time: 0.010s)
> create table {{%auth_assignment}} ... done (time: 0.007s)
*** applied m140506_102106_rbac_init (time: 0.060s)

Migrated up successfully.
```

Figure 11.6: Creating the RBAC database tables.

Yii provides two classes of auth managers for you:

- **PhpManager** uses a PHP script
- **DbManager** uses the database

Normally you'll use the database, which is what this chapter focuses on.

Tell your application which authorization manager to use by configuring the “authManager” application component:

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\DbManager'
        ],
        // Even more other stuff.
```

Next, you'll need to create the necessary database tables (where Yii stores all the RBAC details). To do that, first also add the “authManager” configuration to **console.php**, so the console application is aware of the change. Then run a migration that comes with the Yii framework (**Figure 11.6**);

```
./yii migrate --migrationPath=@yii/rbac/migrations
```

Chapter 18, “[Advanced Database Issues](#),” explains migrations in more detail, but simply running that command—from the command-line—from within the application directory should suffice.

When using `PhpManager`, you'll need to create a writable `rbac` directory within the application folder. The authorization manager will write all the details to files there. But note that you normally shouldn't use the `PhpManager` option. Using it for a complex set of rules will not be efficient, as reading from large text files is never as efficient as using an indexed database.

With the configuration in place, you can begin telling your Yii application what permissions and roles should exist, in that order.

To define a permission, you should first get a reference to the authorization manager:

```
$auth = Yii::$app->authManager;
```

A question you may have is: where do I put that code? You only need to establish the authentication items once for a site, so my recommendation would be to create a specific controller and/or action that performs the authorization initialization. For example, you might create an `actionSetup()` method in the “site” controller that only administrators can execute. The administrator would then execute that action *once*, before the site is live. In fact, you may just create the authorization items once while developing the site, and then recreate them on the live site when you replicate the database.

{NOTE} A console script, see Chapter 16, “Leaving the Browser”, is a great choice for performing a site's setup.

Next, invoke the `createPermission()` method to create each permission. Its lone argument should be a unique name. If you want, set the permission's description next. Finally, add the permission to the authorization manager.

Here are a couple operations based on Figure 11.5:

```
# controllers/SiteController.php::actionSetup()
$auth = Yii::$app->authManager;
$createPage = $auth->createPermission('createPage');
$createPage->description = 'Create a page';
$auth->add($createPage);
$updatePage = $auth->createPermission('updatePage');
$updatePage->description = 'Update a page';
$auth->add($updatePage);
$updateUser = $auth->createPermission('updateUser');
$updateUser->description = 'Update a user';
$auth->add($updateUser);
$deleteUser = $auth->createPermission('deleteUser');
$deleteUser->description = 'Delete a user';
$auth->add($deleteUser);
```

Those four permissions define a decent range of the kinds of things required by the CMS site. With the permissions defined, you'll next define a permission with an accompanying rule.

{TIP} In theory you could reuse variables for creating permissions, but it'll be cleaner if you define each permission—and later, role—as its own variable.

To apply a rule to a permission, you must first define the rule itself. A rule is defined as a class that extends Yii's `Rule` class.

A logical place to define rules is within the application's `rbac` directory (which you'd have to create). Let's call this first rule "OwnerRule". It should extend the `yii\rbac\Rule` class:

```
# rbac/OwnerRule.php
<?php
namespace app\rbac;
use yii\rbac\Rule;

class OwnerRule extends Rule {
    public $name = 'isOwner';
}
```

Rules have a `$name` property, which should be both meaningful and unique.

Next, rules need to define an `execute()` method. The method takes three arguments and returns a Boolean value. If the method returns true, permission will be granted.

For this particular example, the CMS content types—pages, files, and comments—each has a `user_id` property. The rule should see if the `user_id` of the given object type matches the ID of the current user.

```
public function execute($user, $item, $params) {
    if (!isset($params['object'])) return false;
    if (!isset($params['object']->user_id)) return false;
    return $params['object']->user_id == $user;
}
```

This rule is being written very generically such that it can check if *any* object's `user_id` property matches the current user's ID. The method will receive the current user's ID in `$user`. The method will receive an object in `$params['object']`.

From there, the method first returns false if no object was provided or if the object does not have a set `user_id` property. Finally, the method returns a Boolean indicating if the object's `user_id` value matches the `$user` value.

With the rule defined, it can be applied to a permission to create a more nuanced permission. First, add the rule to RBAC. Next, apply it to a permission:

```
# controllers/SiteController.php::actionSetup()
$auth = Yii::$app->authManager;

// Add the rule:
$isOwner = new \app\rbac\OwnerRule;
$auth->add($isOwner);

// Add the rule to a permission:
$updateOwnPage = $auth->createPermission('updateOwnPage');
$updateOwnPage->description = 'Update your own page';
$updateOwnPage->ruleName = $isOwner->name;
$auth->add($updateOwnPage);
```

Because this rule is written generically, it can be applied to other permissions, too:

```
// Add the rule to a permission:
$updateOwnUser = $auth->createPermission('updateOwnUser');
$updateOwnUser->description = 'Update your own user';
$updateOwnUser->ruleName = $isOwner->name;
$auth->add($updateOwnUser);
```

In situations wherein rules are applied to permissions, it may make sense to also identify the hierarchical relationship between that permission with a rule and the one without. This is done using the authorization's `addChild()` method, which takes the parent as the first argument and the child as the second:

```
# controllers/SiteController.php::actionSetup()
$updatePage = $auth->createPermission('updatePage');
$updatePage->description = 'Update a page';
$auth->add($updatePage);
$updateOwnPage = $auth->createPermission('updateOwnPage');
$updateOwnPage->description = 'Update your own page';
$updateOwnPage->ruleName = $isOwner->name;
$auth->add($updateOwnPage);
$auth->addChild($updateOwnPage, $updatePage);
```

With all of the permissions defined, you next create roles and assign permissions to those roles:

```
# controllers/SiteController.php::actionSetup()
$auth = Yii::$app->authManager;
// Create permissions.
$public = $auth->createRole('public');
$auth->add($public);
$auth->addChild($public, $createPage);
$auth->addChild($public, $updateOwnPage);
// And so on.
```

That code creates one role and assigns two permissions to it. Also remember that in the CMS example, “public” is the lowest level of authenticated user, different from an un-authenticated guest. Certain permissions, such as the viewing of a page, the creation of a user record (for registering), or the reading of a user record (for logging in) won’t be restricted at all.

To make the most of the hierarchical structure, you would add not only permissions to some roles but also roles to other roles. An author is just a basic (i.e., public) user with the added ability of creating pages and files and editing pages and files she created. An administrator is an author user with the added abilities of:

- Editing any page, file, or comment
- Deleting any page, file, or comment
- Editing any user
- Deleting any user

Here’s how that might play out:

```
# controllers/SiteController.php::actionSetup()
$auth = Yii::$app->authManager;
// Create permissions.
$public = $auth->createRole('public');
$auth->add($public);
$auth->addChild($public, $updateOwnUser);
$auth->addChild($public, $createComment);
// And so on.
$author = $auth->createRole('author');
$auth->add($author);
$auth->addChild($author, $public);
$auth->addChild($author, $updateOwnPage);
$auth->addChild($author, $updateOwnFile);
// And so on.
$admin = $auth->createRole('admin');
$auth->add($admin);
$auth->addChild($admin, $author);
```

```
$auth->addChild($admin, $updatePage);
$auth->addChild($admin, $updateFile);
$auth->addChild($admin, $updateComment);
```

Note that the code makes reference to permissions not in Figure 11.5 or previously discussed, just to give the code more bulk. You'll also notice that in the design, the "update page" operation is linked to the administrator twice: once directly and once through the "author" user. The "author" user doesn't actually have "update page" functionality, only "update own page", with the extra logic enforced. An administrator could update her own page, if she created it, but also needs to be able to update any page.

{TIP} To make this example a bit simpler, I did not create a "content" management task that is the parent of pages and files, although you certainly could.

Again, taking advantage of the hierarchical structure of users and permissions makes building up complex authorization structures much, much easier. The last step in the definition process is to assign roles to specific site users.

Finally, you need to associate authenticated users with authorization roles. This is done via the `assign()` method. Its first argument is the role being assigned and the second is an identifier of the user to which the role is assigned. If you are using static logging in (against an array of values, as in the default code), you would do this:

```
# controllers/SiteController.php::actionSetup()
$auth = Yii::$app->authManager;
// Create permissions.
// Create roles.
$auth->assign($public, 100);
$auth->assign($admin, 101);
```

Now the "demo" user, with an ID of 100, has been assigned certain permissions and the "admin" user, with an ID of 101, has been assigned those permissions and more. Note that you only have to invoke the `assign()` method once, not each time the user logs in, as that code creates the record in the database.

All of the code to this point creates a slew of records in the database or text file, which will then be used to test authorization.

{TIP} If you need to tweak or re-define your authorization items, first truncate the underlying database tables and then rerun your setup action.

In a few pages, you'll learn how to tie role assignments to database users, but let's go with static users for the time being, while you see how to enforce the authorization items you've declared.

{TIP} Roles can also be assigned using default roles, as explained in the [Guide](#).

All of the code to this point establishes the authorization rules (as a text file or a database). Now you can make use of those rules to allow users to perform tasks.

Enforcing RBAC authorization is quite straightforward: invoke the `can()` method of the application's "user" component, providing the name of the permission to check:

```
# controllers/PageController.php
public function actionCreate() {
    if (!\Yii::$app->user->can('createPage')) {
        throw new \yii\web\ForbiddenHttpException('Nope!');
    }
}
```

That code, within the "create page" action, checks if the current user has permission to create pages. If not, a "forbidden" HTTP exception will be thrown (**Figure 11.7**).



To test a permission that also has a rule, use the second argument to `can()` to pass additional parameters. Looking at the "isOwner" rule, it expects to receive an object—with a `user_id` property, in the "object" parameter. Here's how it would be used:

```
# controllers/PageController.php
public function actionUpdate($id) {
    $model = $this->findModel($id);
    if (!\Yii::$app->user->can('createPage',
        ['object' => $model])) {
        throw new \ForbiddenHttpException('You cannot!');
    }
    // Other stuff.
}
```

The code to this point, and much that you'll find elsewhere, associate roles with static users. That's fine in those rare situations where you're using static users (in which case, you may not even need the complexity of RBAC), but most dynamic sites store users in a database table. How do you associate database users with RBAC roles?

The goal is to invoke the `assign()` method once *for each user*, as that's what the RBAC system will need in order to confirm permission.

The first thing you'll need to do is determine what user identifier counts. In other words: what table column and model attribute differentiates the different roles? Logically, this would be a property such as `user.type` in the CMS example. The goal, then, is to do this:

```
$auth = Yii::$app->authManager;
if ($user->type === 'admin') {
    $role = $auth->getRole('admin');
    $auth->assign($role, $user->id);
} elseif ($user->type === 'author') {
    $role = $auth->getRole('author');
    $auth->assign($role, $user->id);
} elseif ($user->type === 'public') {
    $role = $auth->getRole('public');
    $auth->assign($role, $user->id);
}
```

That code associates the user's ID with a specific RBAC role. As each `$user->type` value directly correlates to a role, that code can be condensed to:

```
$auth = Yii::$app->authManager;
$role = $auth->getRole($user->type);
$auth->assign($role, $user->id);
```

Second, you need to determine *when* it would make sense to invoke `assign()`. A logical time would be after the user registers. To do that, you could create an `afterSave()` method in the model class:

```
# models/User.php
public function afterSave($insert, $changedAttributes) {
    $auth = Yii::$app->authManager;
    if (!in_array($this->type,
        $auth->getRolesByUser($this->id))) {
        $role = $auth->getRole($this->type);
        $auth->assign($role, $this->id);
    }
    return parent::afterSave($insert, $changedAttributes);
}
```

That code will be called after a model record is saved. This could be after a new record is created or after it is updated (like when the user changes her password). Because the second possibility exists, this code first checks that the assignment has not already taken place. If not, then the assignment is performed.

{TIP} If you have a situation where the user's permissions may be changed, you'd need to remove the existing role assignment and add the new one.

Chapter 12

Working with Widgets

Widgets resolve a common problem with the MVC approach: you shouldn't put much programming logic in your view files, and yet, a decent amount of logic is required to render the proper output, particularly with websites. Once you factor in dynamic client-side behavior driven by JavaScript, the complexity of a view becomes even more elaborate. If you take as an example a navigable calendar or a dynamic table of data, you can appreciate how much HTML, JavaScript, and logic is required by one simple component on a page. Widgets wrap up all the HTML, JavaScript, CSS, and logic functionality into one component that can be dropped into a view file cleanly.

The focus in this chapter is on using widgets in general, and using the widgets defined within the Yii framework specifically. To start, you'll see how to add a widget to a view, and how to customize a widget's behavior. Then the chapter walks through the usage and configuration of the most popular widgets.

Widgets are, at their root, just an instance of a class. Specifically, the class must extend `yii\base\Widget`. Yii has dozens of applicable widget classes defined for you, such as:

- `ActiveForm` creates dynamic HTML forms
- `Breadcrumbs` creates breadcrumbs
- `Captcha` creates a CAPTCHA with a form
- `DetailView` displays one record
- `GridView` displays lots of data in a grid format
- `ListView` displays lots of data in a list format
- `Nav` creates navigation menus

Plus there are widgets specifically tied to [Twitter Bootstrap](#) and [jQuery User Interface](#) functionality.

This chapter focuses on these predefined widget classes. In Part 3 of the book, you'll see how to create your own widget class.

Class yii\widgets\DetailView

All Classes | Properties | Methods

Inheritance	yii\widgets\DetailView » yii\base\Widget » yii\base\Component » yii\base\Object
Implements	yii\base\Configurable, yii\base\ViewContextInterface
Available since version	2.0
Source Code	https://github.com/yiisoft/yii2/blob/master/framework/widgets/DetailView.php

DetailView displays the detail of a single data \$model.

DetailView is best used for displaying a model in a regular format (e.g. each model attribute is displayed as a row in a table.) The model can be either an instance of yii\base\Model or an associative array.

Figure 12.1: The package and inheritance details for `DetailView`.

Once you know what class to use, you create a widget in one of two ways. The first route is to invoke the `widget()` method of the widget class. Its lone argument is for customizing that class instance (which is to say the widget). Here's an example from a view file generated by Gii:

```
# views/user/view.php
<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        'id',
        'username',
        'email:email',
        'pass',
        'type',
        'date_entered',
    ],
]) ?>
```

You can find most of the built-in widget classes, such as `DetailView` above, in `vendor/yiisoft/yii2/widgets` (**Figure 12.1**).

When a widget class is defined within the framework, you can just provide the class name to create an instance of that widget, as in the `DetailView` example. For classes not built into the framework, you need to provide a complete reference to the class. For example, you can install the [jQuery UI widget](#) and then create a `DatePicker` instance:

```
<?= yii\jui\DatePicker::widget([/* customization */]) ?>
```

The alternative way to instantiate a widget is to use the `begin()` method. This is to be followed by content that gets captured by the widget. And finally you invoke `end()` to complete the widget creation. This is how the `ActiveForm` widget is used:

```
<?php $form = ActiveForm::begin([/* customization */]); ?>
<?= $form->field($model, 'username')->textInput(
    ['maxlength' => true]) ?>
<?= $form->field($model, 'email')->textInput(
    ['maxlength' => true]) ?>
<!-- And so on. -->
<?php ActiveForm::end(); ?>
```

With that particular case, which is the most frequent use of `begin()` and `end()` that you'll see, those method calls end up creating the opening and closing FORM tags, with the form being written between them.

There are two challenges to using widgets in Yii:

- Knowing what widgets exist
- Customizing the widgets to function as you need them to

The rest of this chapter introduces the most important widgets. You can find others by searching online, searching the [Yii site](#), asking in the [Yii forums](#), or by looking in the [class docs](#) to see what classes extend `Widget` and its children.

To customize the widgets—to know what to provide as an array to the `widget()` or `begin()` method, you'll want to look at the public, writable attributes of the associated class. For example, `ActiveForm` has the following public, writable attributes (plus a couple more):

- `action` dictates the form's “action” attribute
- `enableAjaxValidation` turns Ajax validation on or off
- `enableClientValidation` turns client-side validation on or off
- `errorCssClass` sets the CSS class used for errors
- `method` dictates the form's “method” attribute

Knowing this, you can use those attribute names for your configuration array's indexes. For the values, if it's not obvious what an appropriate value or value type would be, check out the class's documentation for those attributes. A simple customization:

```
<?php $form = ActiveForm::begin([
    'enableAjaxValidation' => true,
    'enableClientValidation' => true,
    'errorCssClass' => 'error'
]); ?>
<!-- And so on. -->
<?php ActiveForm::end(); ?>
```



Figure 12.2: A navigation menu.

The class docs also indicate the default properties, so you know whether customizations are even required.

If you created a site using the basic application template and Gii, you'll already have several widgets in use, and you can check out your view files to see what they do. To start, let's investigate three non-jQuery UI widgets that are part of the Yii framework: `Menu`, `Breadcrumbs`, and `Captcha`.

The `Menu` widget provides a way to display a hierarchical navigation menu using nested HTML lists. In Yii 2, `Menu` is not used in the basic application template, which instead uses a Twitter Bootstrap `Nav` widget. Here is an example of the `Menu` widget:

```
echo Menu::widget([
    'items' => [
        ['label' => 'Home', 'url' => ['site/index']],
        ['label' => 'About', 'url' => ['site/page'], 'view' => 'about'],
        ['label' => 'Contact', 'url' => ['site/contact']],
        ['label' => 'Login', 'url' => ['/site/login'],
         'visible' => Yii::$app->user->isGuest],
        ['label' => 'Logout', 'url' => ['/site/logout'],
         'visible' => !Yii::$app->user->isGuest]
    ]
]);
```

That creates four menu items, as the Login/Logout items will only appear if the user is or is not logged in (**Figure 12.2**).

(The image shows the menu without CSS formatting, which would normally convert the list items into something more appealing.)

Configuring the widget is a matter of assigning values to the writable properties of the `Menu` class. There are only a few, such as `activeCssClass` for indicating the name of the CSS class to be applied to the currently active menu item. There are similar properties for setting the CSS class for the first and last item in the menu (or submenu).

The most important property is `items`. It's used to establish the navigation items,

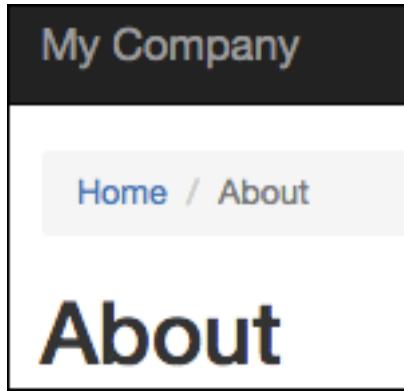


Figure 12.3: A rather short trail of breadcrumbs.

and is declared as an array. Each item is represented by its own array, with any of the following indexes:

- active
- items
- label
- options
- template
- url
- visible

The most important of these are “label” and “url”, as shown in the previous code. The label is displayed to the user. The URL value is used for the link. For it, follow the same rules as for `normalizeUrl()` (covered in Chapter 7, “[Working with Controllers](#)”), with one exception: you must always specify a controller and an action (i.e., you cannot rely upon the default controller or a default controller action).

To create a submenu, provide an “items” subarray to an individual item. In theory. How well this works will depend upon your CSS, HTML, and such. The fact is that `Menu` is better for simpler presentations of menus. More elaborate menus are best left to extensions.

Another widget that helps the user navigate the site is `Breadcrumbs`. It’s used to create the breadcrumbs effect at the top of the page, which users and search engines alike both appreciate (**Figure 12.3**).

The most important property is `links`, which should be an array. If an array element has an index and a value, that will be used for the link label and URL. If just provided with a value, the value will be the label and no link will be created. This structure allows you to link to items higher up the breadcrumb trail but only show, not link, the current page. For example, Figure 12.3 shows the (current) “About” page unlinked, with the “Home” page linked.

To generate that result, the page would have code like:



Figure 12.4: The breadcrumb trail to a specific page being viewed.

```
<?= Breadcrumbs::widget([
    'links' => [
        ['label' => 'About'],
        ['url' => 'site/about']
    ]
]) ?>
```

Being a common element on every page, it makes more sense to put the widget in a layout file. Then each page only needs to set its breadcrumbs. This can be done through the `params` attribute of the view:

```
$this->params['breadcrumbs'] [] = $this->title;
```

The layout file then contains:

```
<?= Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ?
        $this->params['breadcrumbs'] : [],
]) ?>
```

Note that the home page is always assumed and you never have to specify it. Here's another example (**Figure 12.4**):

```
# views/page/view.php
$this->params['breadcrumbs'] [] =
    ['label' => 'Pages', 'url' => ['index']];
$this->params['breadcrumbs'] [] = $this->title;
```

The order the items are listed in the array is the order in which they will be listed (from left to right) in the breadcrumbs.

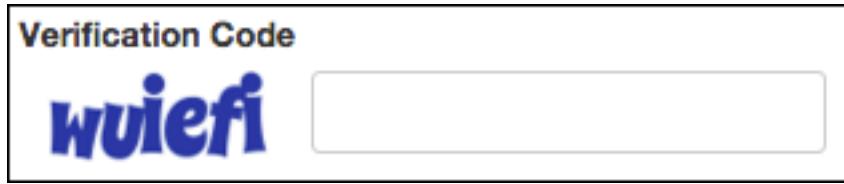


Figure 12.5: The CAPTCHA widget creates the image and the input.

Next, let's take a look at how one implements CAPTCHA: Completely Automated Public Turing test to tell Computers and Humans Apart. Using this widget can be easily explained as the code included in the basic application template implements CAPTCHA on the contact form already.

First, the `views/site/contact.php` page references the namespace for the widget:

```
# views/site/contact.php
use yii\captcha\Captcha;
```

Next, the form creates a CAPTCHA using this code:

```
<?= $form->field($model, 'verifyCode')->widget(Captcha::className(), [
    'template' => '<div class="row"><div class="col-lg-3">
        {image}</div>
        <div class="col-lg-6">{input}</div></div>',
]) ?>
```

This is an extension of code shown in Chapter 9, “[Working with Forms](#).“ The `ActiveForm` class’s-itself a `widget-field()` method returns an object of type `ActiveField`. This object type has a `widget()` method that can be used to render a form field using a widget. The method’s first argument is the class of the widget and the second is for customizing the widget. The code here provides the widget template as the customization.

The `Captcha` class requires the GD extension in order to work. As a safety measure, you can invoke the `checkRequirements()` method to confirm that the minimum requirements are met prior to attempting to use the class. If that method returns true, then you simply create the widget.

The widget itself creates the HTML IMG tag that shows the CAPTCHA image, along with the input for the user to enter the CAPTCHA value (**Figure 12.5**):

```
<label class="control-label" for="contactform-verifycode">Verification
Code</label>
<div class="row">
    <div class="col-lg-3"></div>
```

```
<div class="col-lg-6"><input type="text"
    id="contactform-verifycode" class="form-control"
    name="ContactForm[verifyCode]"></div>
</div>
```

You can configure the CAPTCHA presentation a bit by setting the various `Captcha` class attributes, such as customizing how the user would refresh the CAPTCHA image.

Turning to the model associated with the form, you need to apply the CAPTCHA validation rule to the text input. Again, this can be set to allow for an empty value if the PHP installation does not meet the minimum requirements:

```
# models/ContactForm.php
public function rules() {
    return [
        [['name', 'email', 'subject', 'body'], 'required'],
        ['email', 'email'],
        ['verifyCode', 'captcha'],
    ];
}
```

Unlike the CAPTCHA widget in Yii 1, this is all the code needed to use it. Nothing special has to happen in the controller. So that's all there is to it! If you ever need to use CAPTCHA on one of your forms, just copy and tweak the code already generated for you by the basic Yii application.

The next group of widgets to be covered are all similar to each other in that they present data. Some widgets present multiple records at once, offering great features like sorting, pagination, and filtering, while others present just a single record. Therefore, before getting into the specifics of each widget, you must first learn how you provide data to them. For some of these widgets, you don't just provide an array of objects, but rather a specific kind of data type, one that supports features such as sorting, pagination, and filtering.

The data formats you'll use with some of the Yii widgets are defined as classes that implement the `yii\data\DataProviderInterface` interface. The base class is `yii\data\BaseDataProvider`, which is then extended by child classes. The child classes can all be used in the same way as a data source for a widget. They differ in where they get their data from, and how:

- `yii\data\ActiveDataProvider`, uses an Active Record model
- `yii\data\ArrayDataProvider`, uses an array
- `yii\data\SqlDataProvider`, uses an SQL query
- `yii\sphinx\ActiveDataProvider`, uses a Sphinx query

Put another way, each of these classes is a wrapper that can take a different data source and make it universally usable by the various widgets. Through the data source, you also configure the data's pagination and sorting behavior.

As an example the following creates a data source from a basic array, using a hypothetical students-grades premise:

```
use yii\data\ArrayDataProvider;
$provider = new ArrayDataProvider([
    'allModels' => [
        ['id'=>1, 'first'=>'Jane', 'last'=>'Doe', 'grade'=>85],
        ['id'=>2, 'first'=>'John', 'last'=>'Doe', 'grade'=>67],
        /* More items. */
        ['id'=>25, 'first'=>'Askini', 'last'=>'Mayur', 'grade'=>85],
    ]
]);
```

The code creates a data source by providing a multidimensional array to the `allModels` attribute.

It's not common that you'll have an array as your data source; normally complex data comes from a database. Exceptions would be data read in from files or network channels.

To create a data source using an ActiveRecord model, create an object of type `ActiveDataProvider`. This time provide the data to the `query` property. This can be either a `yii\db\Query` or `yii\db\ActiveQuery` object. The latter is generated by a `find()` call:

```
use yii\data\ActiveDataProvider;
use app\models\Page;
$provider = new ActiveDataProvider([
    'query' => Page::find()
]);
```

That code executes a `Page::find()` query and returns the results as a `ActiveDataProvider` object, usable by a widget.

Here's how you'd accomplish the same thing using a direct query:

```
use yii\data\SqlDataProvider;
$provider = new SqlDataProvider([
    'sql' => Yii::$app->db
        ->createCommand('SELECT * FROM page')->queryAll()
]);
```

That code will run the query on the database and return the results as an `SqlDataProvider` object, usable by a widget. Understand that this particular query is more simple than you'd normally use for `SqlDataProvider`, but it works just the same.

A common configuration option for the various sources is “pagination”. You can use it to dictate how pagination is accomplished with the data set. The most common configuration setting is how many items should be in a page of results. To set that, assign a value to the pagination's “`pageSize`” property:

```
$provider = new ActiveDataProvider([
    'query' => Page::find(),
    'pagination' => ['pageSize' => 10]
]);
```

Pagination works the same when using an SQL command, except that you need to tell the class how many total records exist in that situation:

```
$q = 'SELECT COUNT(id) FROM user WHERE type="author"';
$count = Yii::$app->db->createCommand($q)->queryScalar();
$dp = new SqlDataProvider([
    'query' => Yii::$app->db->createCommand('SELECT * FROM
        user WHERE type="author"')->queryAll(),
    'totalCount' => $count,
    'pagination' => ['pageSize' => 10]
]);
```

{WARNING} If your primary query uses a conditional to limit the results, the count query must use that same condition or else the number of pages won't match the number of records to display.

Adding sorting to the process is a bit more complicated, but entirely customizable. You should indicate the columns used for sorting as one configuration item, “`attributes`”:

```
$provider = new ActiveDataProvider([
    'query' => Page::find()->with('user')->all(),
    'pagination' => ['pageSize' => 10],
    'sort' => [
        'attributes' => [
            'title',
            'name' => ['user.last', 'user.first']
        ],
    ],
]);
```

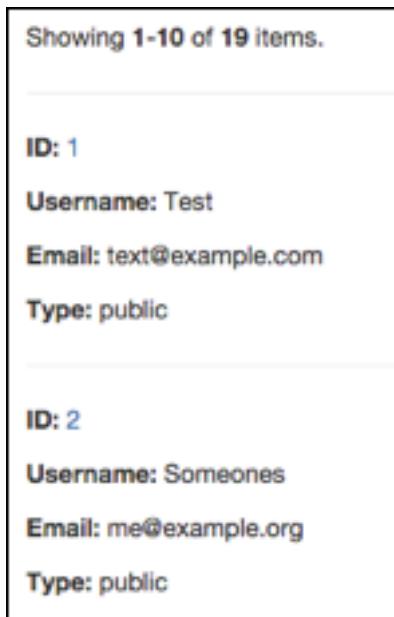


Figure 12.6: A list of users.

That code defines two sortable attributes: the page title and the author’s name, first by last name and then by first (so that “Doe, Jane” is alphabetized before “Doe, John”). By default, ascending sorts are used. Learn more about the `Sort` class to see how to customize this further.

Once you’ve created a data source, most widgets will take that object as the value for its “`dataProvider`” property.

Next the chapter moves into a series of widgets that make the presentation of data much, much easier. The first of these is `ListView`. The `ListView` class presents multiple records of data in a list, as opposed to a table (**Figure 12.6**).

That output is created by the following code:

```
use yii\widgets\ListView;
use yii\data\ActiveDataProvider;
$dataProvider = new ActiveDataProvider([
    'query' => User::find(),
    'pagination' => ['pageSize'=>10]
]);
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_view',
]);
```

You’ll notice the data provider there, which can be any of the class types already explained.

The “itemView” index is used to identify the view file that will present each record being displayed. The above code specifies the individual view file as `_view.php` (in the same `views` subfolder). Here’s a snippet of that code:

```
<hr>
<div class="view">
<p><strong><?php
use yii\helpers\Html;
echo Html::encode($model->getAttributeLabel('id'));
?></strong>
<?php echo Html::a(Html::encode($model->id),
array('view', 'id'=>$model->id)); ?>
</p>
<p><strong>
<?php echo Html::encode($model->getAttributeLabel('username'));
?></strong>
<?php echo Html::encode($model->username); ?>
</p>
// And so on.
```

For each item shown, the individual view file is passed a specific item as the `$model` array. If the data provider object contains ActiveRecord `User` objects, then `$model` in the file will be a `User` instance. For that reason, you can reference the model’s methods and attributes accordingly. To change the presentation of the information in `ListView`, either edit the individual view file or use one of your own creation.

Pagination of the records can automatically be applied, as in the above code.

There are more attributes you can set, too, that will do things like set the text before and after the sorting links, change the CSS classes used, and so forth. These are all explained in the [Yii class docs](#).

The `DetailView` widget is used to display information about a single record. An example of its usage is written into the `view.php` file by Gii ([Figure 12.7](#)).

Unlike `ListView` and `GridView`, `DetailView` expects as its data source either a single model instance or a single associative array. This is assigned to the `DetailView` class’s `model` property.

The second most important property is `attributes`. This is how you dictate which values in the model or array are displayed:

```
# views/page/view.php:
<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        'id',
        'user_id',
```

Aliquam malesuada, ligula sit amet.	
Update Delete	
ID	1
User ID	3
Live	1
Title	Aliquam malesuada, ligula sit amet.
Content	<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ad litora torquent per conubia nostra, per inceptos himenaeos. Nulla lobortis nulla odio porta non. Donec eu metus tincidunt tellus convallis tincidunt a ac quam. Quis porta lacus aliquet. Pellentesque molestie viverra purus et lacinia. Nulla facilisi. Cur mus.</p>

Figure 12.7: The view display for a single Page record.

```

    'live',
    'title',
    'content:ntext',
    'date_updated',
    'date_published',
],
]) ?>

```

The order in which the attributes are listed are the order in which they will be displayed (from top to bottom).

If you don't want to display an attribute, just remove it from that list. If you're using an Active Record model instance and you want to show an attribute from a related model, use *relationName.attribute*. For example, `Page` is related to `User` via the `user_id` column. As the detail view should probably show the associated user's username, not the `user_id`, replace `user_id` in that list with `user.username` (**Figure 12.8**).

The general format for specifying how something is displayed is *Attribute:Format:Label*, with the latter two being optional. The “format” value dictates how the value is formatted, with plain text being the default. Other possible values are:

- “raw” does not change the value
- “text” HTML-encodes the value
- “ntext” HTML-encodes the value and applies `nl2br()`
- “html” purifies and returns the value as HTML

ID	1
Username	Some Author
Live	1
Title	Aliquam malesuada, ligula sit amet.

Figure 12.8: The same page with the author's username now displayed.

- “date” formats the value as a date
- “time” formats the value as a time
- “datetime” formats the value as a date and time
- “boolean” displays the value as a Boolean
- “integer” formats the value as an integer
- “decimal” formats the value as an decimal
- “email” wraps the value in a “mailto” link
- “image” creates the proper IMG tag to show the value
- “url” formats the value as a hyperlink

These options come from the `Formatter` class, which is used to format the presentation of data. If you use data formatting a lot, you’ll want to spend some time reading [its documentation](#).

By default, all values are shown as encoded text. With the page example, you may want to make a few changes (**Figure 12.9**):

```
# views/page/view.php:
<?= DetailView::widget([
    'model'=>$model,
    'attributes'=>[
        'id',
        'user.username',
        'live:boolean',
        'title',
        'content:html',
        'date_updated',
        'date_published',
    ],
]); ?>
```

<h1>Aliquam malesuada, ligula sit amet.</h1>	
Update Delete	
ID	1
Username	Some Author
Live	Yes
Title	Aliquam malesuada, ligula sit amet.
Content	<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ipsum litora torquent per conubia nostra, per inceptos himenaeos. Nulla lobortis nulla et odio porta non. Donec eu metus tincidunt tellus convallis tincidunt a ac quam. Qui porta lacus aliquet. Pellentesque molestie viverra purus et lacinia. Nulla facilisi. Cum mus.</p>

Figure 12.9: The same page again, with a better display.

If you don't specify a label, the model's attribute label value will be used.

Instead of the "Name:Format:Label" structure for each attribute, you can format each attribute as an array. This flexibility allows you to further customize the output. The following code will change the displayed value to be the author's username, linked to that author's view page:

```
# views/page/view.php:
<?= DetailView::widget([
    'model'=>$model,
    'attributes'=>[
        'id',
        [
            'label' => 'Author',
            'value' => Html::a(Html::encode(
                $model->user->username),
                ['user/view','id'=>$model->user_id]),
            'format' => 'html'
        ],
        'live:boolean',
        'title',
        'content:html',
        'date_updated',
        'date_published',
    ],
])
```

Showing 1-19 of 19 items.

#	ID	Username	Email	Password	Type	
1	1	Test	text@example.com	21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6	public	
2	2	Someones	me@example.org	21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6	public	
3	3	Some Author	auth@example.net	21f9b9e3c98236d3efb6c8b47a4137a1c7ad59cc84af20e2e8ba41c946f34ab6	author	

Figure 12.10: The admin grid of users.

```
]); ?>
```

`DetailView` uses a table row to display each item. Naturally, this is also customizable. To use, say paragraphs instead of a table, you would assign a value to the `template` property. Use the placeholders `{label}` and `{value}`:

```
# views/page/view.php:
<?= DetailView::widget([
    'model'=>$model,
    'template' => '<p><b>{label}</b>: {value}</p>',
    'attributes'=>[
        'id',
        [
            'label' => 'Author',
            'value' => Html::a(Html::encode(
                $model->user->username),
                ['user/view','id'=>$model->user_id]),
            'format' => 'html'
        ],
        'live:boolean',
        'title',
        'content:html',
        'date_updated',
        'date_published',
    ]
]); ?>
```

The `GridView` class is the true workhorse of the bunch. You can see it in action on the `index.php` page (**Figure 12.10**).

`GridView` presents a series of records in a table and provides the following functionality:

- Pagination

- Sorting
- Links to view, edit, or delete a record
- Basic searching by field
- Advanced searching
- Live search results via Ajax

If you've spent any time creating similar functionality, normally for the administration of a site, you can appreciate just how much work goes into implementing all that capability.

Here's the code that creates the widget:

```
# views/user/index.php
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        'id',
        'username',
        'email@email',
        'pass',
        'type',
        // 'date_entered',
        ['class' => 'yii\grid\ActionColumn'],
    ],
]); ?>
```

This is a great “bang for your buck” example: just a bit of code delivers tons of functionality. But, how, exactly does it work?

{NOTE} The dynamic display and form submission handling of the advanced search is accomplished via some JavaScript and jQuery, to be explained in Chapter 14, “[JavaScript and jQuery](#).”

Three `GridView` attributes are set in that code: `dataProvider`, `filterModel`, and `columns`. The `dataProvider` should be familiar to you, although its value is derived in a new way. The `filterModel` attribute is also a new one. And `columns` is how you dictate what columns are shown. Let's look at each of these properties in detail. But first, let's look at the controller that renders this view.

{NOTE} I could easily write an entire chapter on just this widget, considering all the permutations and configurations people may want to make to a `CGridView` widget. As one chapter on just this would be impractical (and still not exhaustive), I'll cover the absolute fundamentals and

rely upon you to search online for more and more examples as you need them.

The code generated by Gii creates the following in the controllers:

```
# controllers/UserController.php
public function actionIndex() {
    $searchModel = new UserSearch();
    $dataProvider = $searchModel->search(Yii::$app->request->queryParams);
    return $this->render('index', [
        'searchModel' => $searchModel,
        'dataProvider' => $dataProvider,
    ]);
}
```

As you can see, `$searchModel`, which gets passed to the view, is an instance of the `UserSearch` class. `UserSearch` extends the `User` class, defines its own `search()` method—called in the controller per the above code, and sets its own rules:

```
public function rules(){
    return [
        [['id'], 'integer'],
        [['username', 'email', 'pass', 'type', 'date_entered'], 'safe'],
    ];
}
```

You'll see that the rules here declare most of the properties as safe. This allows searches to be performed using partial or incorrect data: a snippet of an email address, part of a date, etc. This means that “varmit” will be accepted as an email address or an ID value! To know why this is *correct* requires an understanding of how model rules are used.

Model rules only allow values to be assigned to model attributes if the values pass the validation rules. For example, only a syntactically valid email address can be assigned to the `User` model's `email` attribute. This is normally a good thing. But the grid has a search component that allows the user to look up records by attributes. A user, when performing a search, may only provide *part* of an email address for searching: instead of “test@example.com”, the user might search for just email addresses that start with “test” or use the “@example.com” domain. If you don't make the `email` attribute safe without passing the email validation rule, the search functionality will be too limited. On the other hand, if you have model attributes that would never be used for searching, you should remove those from the rule, just to be more secure.

{NEW} Yii 1 used a scenario for searches; Yii 2 uses an extended class.

Returning to the controller, the next line is:

```
$dataProvider = $searchModel->search(Yii::$app->request->queryParams);
```

From this point, everything interesting happens within the `UserSearch::search()` method, discussed in detail next. But to finish the controller, notice that both the data provider and the `UserSearch` model are passed to the view:

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

Both are used in the widget within the view:

```
# views/user/index.php
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    // And so on.
```

The `GridView` class's `filterModel` attribute is optional, but accepts a model instance as its value. If no `filterModel` attribute value is set, then the filtering boxes above the grid would not be shown. You can set the `filterPosition` attribute of the widget to “`FILTER_POS_HEADER`”, “`FILTER_POS_BODY`”, or “`FILTER_POS_FOOTER`” to change where the boxes appear: just above the column headings, just below the column headings (“body”, the default), or below the final record.

For the `dataProvider` attribute of the grid widget, the value is the data provider created in the controller and passed to the view. The data provider is generated by invoking the search model's `search()` method. Here's what that method looks like:

```
# models/UserSearch.php
public function search($params) {
    $query = User::find();

    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);

    $this->load($params);

    if (!$this->validate()) {
```

```
// uncomment the following line if...
// $query->where('0=1');
return $dataProvider;
}

$query->andFilterWhere([
    'id' => $this->id,
    'date_entered' => $this->date_entered,
]);

$query->andFilterWhere(['like', 'username', $this->username])
    ->andFilterWhere(['like', 'email', $this->email])
    ->andFilterWhere(['like', 'pass', $this->pass])
    ->andFilterWhere(['like', 'type', $this->type]);

return $dataProvider;
}
```

At the end of the method, you can see that an `ActiveDataProvider` object is returned. As already explained, this object uses the `query` attribute as the basis for the data retrieval. This query starts off as a simple `User::find()`, but can then be modified in several ways based upon the presence of values in the parameters.

Logically, the records are selected based upon the search criteria provided by the user. The desired result is to create some number of conditions in the WHERE clause that would be added to the SELECT query, equivalent to `SELECT * FROM user WHERE email LIKE '%@example.com%' AND type='public'`, as an example. Those conditions are added to the criteria by the `andFilterWhere()` method, which has not yet been discussed.

The `yii\db\QueryInterface` class's `andFilterWhere()` method adds comparison expressions to a query. In situations where the condition may be built up dynamically, `andFilterWhere()` is a much better solution than trying to create your own complex “condition” value. The `andFilterWhere()` method takes an array of values. For details on possibilities, see the documentation for the `where()` method.

As you can see in the code, the current model instance's values are used for the values of the comparisons. When a user enters “test” in the username box, `$model->username` gets a value of “test” in the controller, meaning it will have that value in the view and in this method when it's called. If a model has an empty value for any attribute, then that condition is *not* added to the query.

The default code results in partial match conditions separated by AND. If the user enters “@example.com” for the email address and “public” for the type, the result will be a query like `SELECT * FROM user WHERE email LIKE '%@example.com%' AND type LIKE '%public%'`. Understanding how this works, there are some edits you'll likely want to make to the `search()` method.

First, remove any column that should not be searchable. Also remove that column from the `UserSearch` rules. You may even want to remove that column from the displayed list in the grid (that's up to you).

Second, see if you can change any comparison from a partial match to a full match. LIKE conditionals are much less efficient to run on the database than equality conditionals. In a situation like an email address or a username, you probably need to allow for partial matches. For numeric columns, however, partial matches often don't make sense. For example, if you were to allow the results to be searched by primary key, you wouldn't want a partial match there, as the primary key 23 should not also bring up 123, 238, and 4231. Similarly, ENUM or SET columns can be set to full matches if you also edit the filtering so that the user can only select from appropriate values (more on that shortly).

Finally, if you want, you can change each use of `andFilterWhere()` to `orFilterWhere()`. If you do so, make sure you change them all to be consistent. And, more importantly, add some text to the view to notify the user that multiple search criteria expands the search results, not limits them.

The data provider and the filter dictate which rows of records are displayed. You can also change what columns are displayed, or how they are displayed. The first way of doing so is to change the values listed for the `GridView` class's `columns` attribute. This is an array of attribute names by default. To start customizing this aspect, remove any attributes you don't need to show, such as a user's password.

{TIP} The order of the column listings dictates the order of the displayed columns in the grid, from left to right.

From there, you can customize the column values the same way you customize the `attributes` property in the `DetailView`. For example, the `Page` model's `live` attribute is a 1 or a 0. It would make more sense to display that as "Live" or "Draft" (or something like that). Just change the value displayed accordingly:

```
# views/page/index.php
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        'id',
        'user.username',
        [
            'header' => 'Live?',
            'value'=>function($data) {
                return ($data->live == 1) ? "Live" : "Draft";
            }
    ]
])
```

Showing 1-20 of 22 items.

#	ID	Username	Live?	Title	Snippet	
1	1	Some Author	Live	Aliquam malesuada, ligula sit amet.	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ipsum tincidunt lac	
2	2	Another Author	Live	Ten years ago a...	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ipsum tincidunt lac	
3	3	Moe	Draft	Phasellus dapibus dolor et mauris.	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ipsum tincidunt lac	

Figure 12.11: The updated page grid view.

```
],
'title',
[
    'header' => 'Snippet',
    'value'=>function($data) {
        return substr($data->content, 0, 100);
    },
    'format' => 'html'
],
'content:html',
// 'date_updated',
// 'date_published',
['class' => 'yii\grid\ActionColumn'],
],
)); ?>
```

For the `user_id` value, you probably instead want to change the displayed value from the author's ID to the actual author name. Just use `relationName.attribute` as explained for `DetailView`. The above code already does this (**Figure 12.11**).

As the above code also demonstrates in two places, you can use a custom function to return a dynamic value. This function should be written to accept one argument—a model instance—and it should return the value to display.

{NOTE} Changing the displayed values will probably cause problems with the filtering/search functionality as in Figure 12.11. I'll explain why, and the fix, shortly.

With the default code, in the far right column of the grid, three buttons are displayed: view, update, and delete. If you want to change these, you need to configure the `yii\grid\ActionColumn` class. The two most important properties are `buttons` and `template`. The `template` property lays out the buttons. You can use the `{view}`, `{update}`, and `{delete}` placeholders to reference default buttons.

This code removes the delete option and swaps the order of the other two:

```
# views/page/index.php
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'searchModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        // Other columns.
        [
            'class' => 'yii\grid\ActionColumn',
            'template'=>'{update} {view}'
        ],
    ],
)); ?>
```

The searching and filtering built into the grid is a wonderful start, but can be improved. For example, the available user types are only “public”, “author”, and “admin”. There’s no point in allowing the user to search by *any* user type value, and it would be far more accurate to pre-set those values. To accomplish that, you’d need to change the filter box for the user type column from a text input to a drop down menu. That’s done by setting the “filter” property of a column:

```
# views/user/index.php
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'searchModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        'id',
        'username',
        'email:email',
        [
            'attribute' => 'type',
            'filter' => ['public' => 'Public', 'author' => 'Author',
                'admin' => 'Admin'],
        ],
        ['class' => 'yii\grid\ActionColumn'],
    ],
)); ?>
```

And that will do that. If you read the documentation for the `filter` property of `yii\grid\DataColumn`, which is what each column in a `GridView` is, you’ll see that providing the `filter` argument with an array creates a drop down list. That code alone makes the grid filterable by type (**Figure 12.12**).

Because the values returned by the drop down exactly match those used in the database, no further customization is required. As you can see in the figure and

Email	Type	
text@example.com	public	
me@example.org	public	
jane@example.net	public	
john@example.org	public	
curly@example.net	public	
larry@example.com	public	

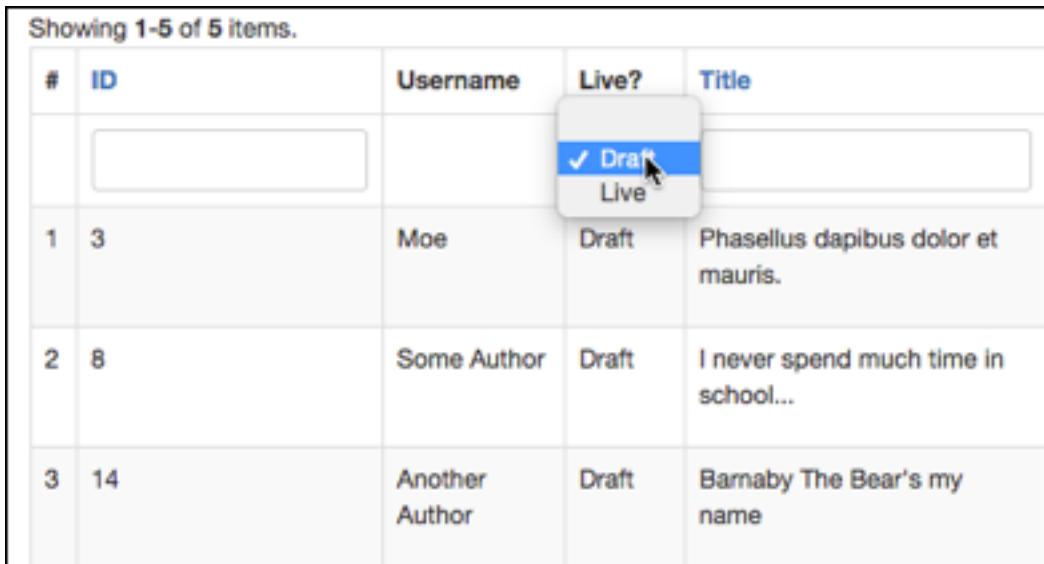
Figure 12.12: Users can now be filtered by specific type values.

code, I've also changed the displayed value to capitalize the type, but that only impacts the display.

Another good example would be to use a drop down list to filter pages by those that are live or not. The grid may display the words “Live” and “Draft” for those values, but the drop down list should use the corresponding database values:

```
# views/page/index.php
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        'id',
        'user.username',
        [
            'attribute' => 'live',
            'header' => 'Live?',
            'value'=>function($data) {
                return ($data->live == 1) ? "Live" : "Draft";
            },
            'filter' => ['0' => 'Draft', '1' => 'Live'],
        ],
        'title',
        [
            'header' => 'Snippet',
        ],
    ],
]);
```



#	ID	Username	Live?	Title
1	3	Moe	Draft	Phasellus dapibus dolor et mauris.
2	8	Some Author	Draft	I never spend much time in school...
3	14	Another Author	Draft	Barnaby The Bear's my name

Figure 12.13: The drop down filter for the page’s status.

```

        'value'=>function($data) {
            return substr($data->content, 0, 100);
        },
        'format'=>'html'
    ],
    // 'date_updated',
    // 'date_published',

    ['class' => 'yii\grid\ActionColumn'],
],
]); ?>

```

You can see the results in **Figure 12.13**.

When you’re working with a single model, you can customize the filtering pretty easily. When you have related models, it becomes a bit trickier. Take, for example, a grid for pages that shows the username of each page author (see the above code and Figure 12.13). That value comes from the `user` table. As it stands, the above code will display the username, but won’t do proper filtering by username as the underlying `Page` attribute is `user_id`. As you can see in Figure 12.13, no filter box is provided anyway. But even if a text input *were* present, the person using the grid could enter “test”, but that will never match a `user_id` value.

Two steps are required to solve this riddle. First, a text input must be displayed for the column. The filters are based upon the model, and as the model has no `user.username` attribute, no text input shows. The fix for that is to name the column `user_id` for the filters, but still use `pageUser.username` as the value of the

column:

```
# view/page/admin.php
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        'id',
        [
            'attribute'=>'user_id',
            'header'=>'Author',
            'value'=>'user.username',
        ],
        // And so on.
```

Now the grid shows an input for the column (test it for yourself to see).

Next, the `search()` method must be changed so that the query uses the supplied `user_id` value to compare against the `username` column in the `user` table.

The default `PageSearch::search()` method looks like this:

```
# models/PageSearch.php
public function search($params) {
    $query = Page::find();

    $dataProvider = new ActiveDataProvider([
        'query' => $query,
    ]);

    $this->load($params);

    if (!$this->validate()) {
        // uncomment the following line if...
        // $query->where('0=1');
        return $dataProvider;
    }

    $query->andFilterWhere([
        'id' => $this->id,
        'user_id' => $this->user_id,
        'live' => $this->live,
        'date_updated' => $this->date_updated,
        'date_published' => $this->date_published,
```

```
]);  
  
    $query->andFilterWhere(['like', 'title', $this->title])  
        ->andFilterWhere(['like', 'content', $this->content]);  
  
    return $dataProvider;  
}
```

First, you'll need to change the model's rules, removing the restriction that `user_id` must be an integer:

```
# models/PageSearch.php  
public function rules()  
{  
    return [  
        ['id', 'live'], 'integer'],  
        ['title', 'content', 'user_id', 'date_updated',  
            'date_published'], 'safe'],  
    ];  
}
```

Next, within the `search()` method, add a `joinWith()` clause to change from lazy loading of the user records to eager loading and to always force use of a JOIN. Then, change the comparison to use the `user.username` field instead of `user_id`:

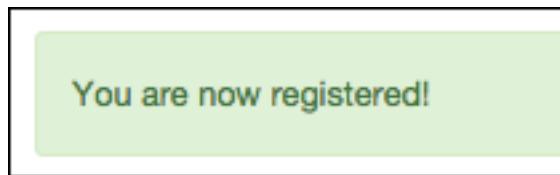
```
# models/PageSearch.php  
public function search($params) {  
    $query = Page::find();  
    $query->joinWith('user');  
    // Other stuff.  
    $query->andFilterWhere([  
        'id' => $this->id,  
        'user.username' => $this->user_id,  
        'live' => $this->live,  
        'date_updated' => $this->date_updated,  
        'date_published' => $this->date_published,  
    ]);  
    // More other stuff.  
    return $dataProvider;  
}
```

And now the grid of pages can be filtered by author name (**Figure 12.14**).

New in Yii 2 is a collection of widgets tied to [Twitter Bootstrap]. This is a logical addition, as Bootstrap is used by default for the basic application. (There was a Bootstrap extension for Yii 1, created by a third-party.)

Showing 1-6 of 6 items.

#	ID	Author	Live?	Title
		Moe	▲▼	
1	3	Moe	Draft	Phasellus dapibus dolor et mauris.
2	6	Moe	Live	Test Title
3	11	Moe	Live	There's a voice that keeps on calling me.

Figure 12.14: Only pages by “Moe” are now shown.**Figure 12.15:** A Bootstrap alert component.

Just some of the widgets are:

- ActiveForm
- Alert
- Button
- Carousel
- Nav
- NavBar
- Progress
- Tabs

The ActiveForm, Nav, and NavBar widgets are used in the basic application by default. The rest are fairly straightforward, and just reading the [code documentation](#) should suffice. For example, here’s how you’d create an alert (**Figure 12.15**).

```
echo \yii\bootstrap\Alert::widget([
    'body' => 'You are now registered!',
    'options' => [
        'class' => 'alert-success'
    ]
]);
```

The `options` property maps to HTML attributes and values set in Bootstrap.

One of the features of Yii that I always appreciated is that it has support for jQuery built-in (the Zend Framework, by comparison, took years to add a jQuery component). Chapter 14 goes into JavaScript and jQuery in Yii in more detail, but while talking about widgets, I'll go ahead and mention the [jQuery User Interface](#) (jQuery UI) widgets now.

The jQuery User Interface is a package of useful components built on top of jQuery. jQuery UI includes:

- Functionality such as dragging, dropping, sorting, and resizing
- Widgets, like date pickers
- Effects such as hiding, showing, color animation, and so on
- A couple of utilities

The jQuery UI widgets include much of the functionality common in today's Web sites:

- Accordion
- Autocomplete
- Datepicker
- Dialog
- Menu
- Slider
- Spinner
- Tabs
- Tooltips

As jQuery is built-into Yii, it was only natural to have parts of jQuery UI ported into Yii as well. About a dozen jQuery UI components have been recreated in Yii as widgets, found within the `yii2-jui` package. Most of these are pretty easy to use just by looking up the corresponding Yii [class documentation](#).

The next few pages demonstrate a couple widgets that are the easiest to use, most necessary, and do not require additional JavaScript (such as an Ajax component). Chapter 14 discusses a couple more.

Note that you must install the `yii2-jui` package in order to use any of these widgets. You can install it by adding it to your Composer requirements:

```
"yiisoft/yii2-jui": "~2.0.0"
```

As a first example, the `yii\jui\Accordion` creates an accordion display of content (**Figure 12.16**).

That output is created by this Yii code:

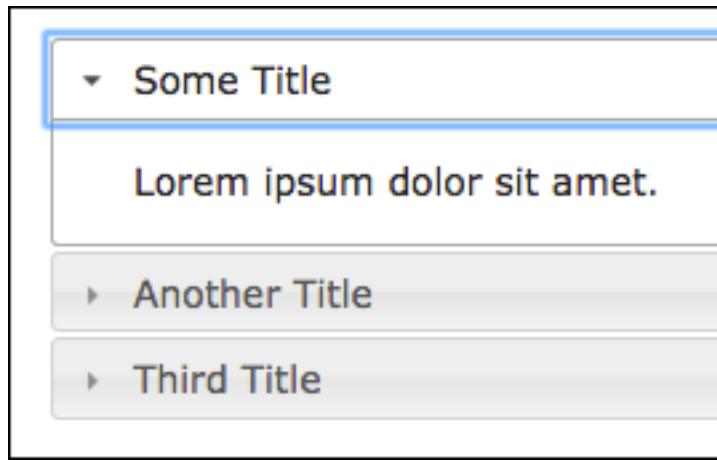


Figure 12.16: A *jQuery UI accordion*.

```
echo yii\jui\Accordion::widget([
    'items'=>[
        ['header' => 'Some Title',
         'content' => 'Lorem ipsum dolor sit amet.'],
        ['header' => 'Another Title',
         'content' => 'Mauris pharetra viverra lacinia.'],
        ['header' => 'Third Title',
         'content' => 'Morbi iaculis fermentum lorem eu.']
    ]
]);
```

And that will do it! Of course, it's not truly reasonable to hardcode the different content into the widget. Normally the content will be dynamically generated. When that's the case, there are a couple of ways you can approach the issue. As an example of this, let's say the controller is passing the accordion view page an array of information:

```
# controllers/SomeController.php::someAction()
$item = [
    ['header' => 'Some Title',
     'content' => 'Lorem ipsum dolor sit amet.'],
    ['header' => 'Another Title',
     'content' => 'Mauris pharetra viverra lacinia.'],
    ['header' => 'Third Title',
     'content' => 'Morbi iaculis fermentum lorem eu.']
];
return $this->render('accordionView', ['items' => $item]);
```

Then, in the **accordionView.php** page, you would use the received data:

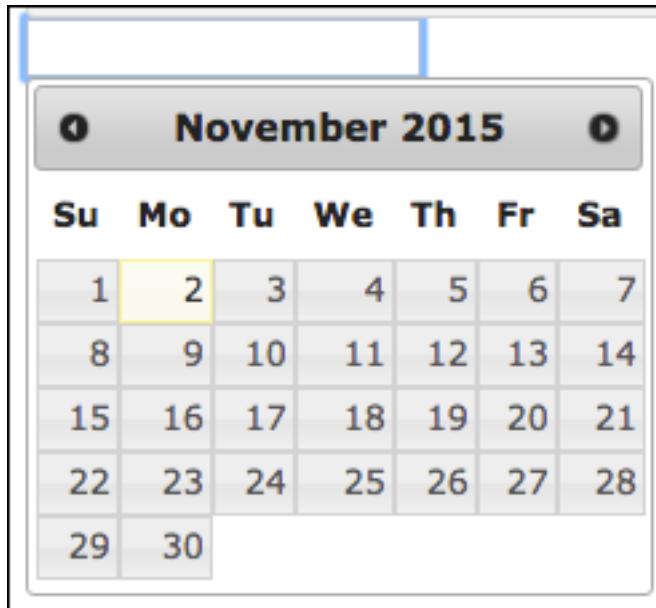


Figure 12.17: A *jQuery UI datepicker*.

```
echo yii\jui\Accordion::widget([
    'items' => $items
]);
```

Alternatively, you could choose to render a partial view for tab content.

The `yii\jui\Tabs` widget works exactly the same way as “Accordion”, but lays out the content in tabs:

```
echo yii\jui\Tabs::widget([
    'items'=>[
        ['header' => 'Some Title',
         'content' => 'Lorem ipsum dolor sit amet.'],
        ['header' => 'Another Title',
         'content' => 'Mauris pharetra viverra lacinia.'],
        ['header' => 'Third Title',
         'content' => 'Morbi iaculis fermentum lorem eu.']
    ]
]);
```

Alternatively, you could choose to render a partial view for tab content.

Another great and useful widget is the Datepicker (**Figure 12.17**).

That's created by this code:

```
echo yii\jui\DatePicker::widget([
    'model' => $model,
    'attribute' => 'date_published'
]);
```

Because this is a form element, you can associate it with a model, as the code shows.

Although the Datepicker is easy to use, there is a catch when it's associated with a model and an underlying database that expects dates to be in a particular format. I'll explain that in the next section.

As with any widget, the available properties of the associated class can be used to customize its behavior. The most important property for the jQuery UI widgets is `clientOptions`. Through the `clientOptions` property you can configure the widget's behavior. For the `clientOptions` indexes and values, turn to the corresponding [jQuery UI documentation](#). For example, the jQuery UI accordion has the “animate”, “collapsible”, “heightSize” and other properties. To set the accordion as collapsable, which means that every section can be closed at the same time, set that property to true:

```
echo yii\jui\Accordion::widget([
    'items' => $items,
    'clientOptions' => [
        'collapsible' => true
    ]
]);
```

The tabs widget has properties for dictating how tabs are hidden and shown (among others);

```
echo yii\jui\tabs::widget([
    'items' => $items,
    'clientOptions' => [
        'hide' => 'fade',
        'show' => 'highlight'
    ]
]);
```

The Datepicker has a [ton of options](#). The “dateFormat” is used to set the format for the selected and displayed dates. You can set the latest date that can be selected via “maxDate” and the earliest via “minDate”.

```
echo yii\jui\DatePicker::widget([
    'model' => $model,
    'attribute' => 'date_published',
    'options' => [
        'maxDate' => '2012-12-31',
        'minDate' => '2012-01-01',
        'dateFormat' => 'yy-mm-dd'
    ]
]);
```

```
'clientOptions' => [
    'dateFormat'=>'yy-mm-dd',
    'maxDate'=>'+1m', // One month ahead
    'minDate'=>'new Date()', // Today
]
]);
```

To find what options are available, and what an appropriate value would be, check the jQuery UI documentation.

As for the trick mentioned earlier that may be required when using Datepickers with models, if the model attribute correlates to a database column, updates and inserts will only work properly if the submitted date is in a format that MySQL accepts. The default format for the date picker is “mm/dd/yy”, which will fail when used to update or insert a record in the database. One solution is to customize the widget to use a better format, as in the code above.

Some jQuery UI widgets, such as DatePicker, are used in forms, like the CAPTCHA widget. When using ActiveForm, simply invoke the `widget()` method on a field to apply a widget to it. Provide the method with the class name of the desired widget:

```
<?= $form->field($model, 'date_published')
->widget(yii\jui\DatePicker::className(), [
    'attribute' => 'date_published',
    'clientOptions' => [
        'dateFormat'=>'yy-mm-dd',
        'maxDate'=>'+1m', // One month ahead
        'minDate'=>'new Date()', // Today
    ]
]) ?>
```

The model attribute will already be associated with the widget through the form field. Additional widget customizations can be provided as an array.

Chapter 13

Using Extensions

Extensions are quite literally that: extensions of the core Yii framework. For most of the functionality the framework itself provides, there may be an extension that builds upon that functionality or offers a variation on the default behavior.

This chapter starts by explaining the concept of extensions in Yii. Basic recommendations as to how to select an extension are provided, as are general installation instructions.

The bulk of the chapter highlights a few notable extensions from the many available. To generate this list, I looked at the extensions that:

- I've personally used
- Are the most downloaded
- Are the highest rated
- Are newer (at the time of this writing)
- Are the easiest to get started with

Obviously, this chapter cannot be exhaustive in terms of the extensions covered, or the coverage of individual extensions. But by the end of the chapter, you should better understand the functionality range that extensions offer, and how you go about using them.

{NOTE} In Chapter 19, “Extending Yii,” you’ll learn how to write your own extensions of the framework.

The term “extension” refers to third-party software specific to the Yii framework. Third-party software that will work with Yii *or any framework* is simply a “library”.

Extensions can serve many different roles. They can act as application components, add new behaviors, be used as widgets, create filters and validators, run as stand-alone modules, and more.

In fact, this book has already used several extensions. First, there's the Gii extension, which is an application component. Second, there's Bootstrap, which is an extension that defines several widgets, among performing other roles. Third, there's Debug, an awesomely useful debugging tool that runs in the browser. All of these extensions are automatically installed as part of the basic application template.

To find available extensions, head to the [Yii 2.0 extensions](#) page. At the time of this writing, there are over 200 extensions available. As Yii 1 has over 2,000, more Yii 2 extensions are certain to come.

From the options, you'll need to choose what extensions to use for your project. Obviously the primary criteria will be your application's needs, such as:

- A WYSIWYG editor
- Easy and powerful authorization management
- Charts widget
- Cache management
- The ability to send HTML email

Whatever the need, there's a chance there's already an extension that will do the job. (If not, head over to Chapter 19 to learn how to write your own!)

Once you've identified your criteria, and have used the extensions page to find options that *may* do the job, make a specific decision based upon (in this order):

- **What version of Yii it requires**

Obviously it'll need to run on Yii 2.

- **What version the extension is currently in**

You probably don't want to use a beta version of an extension on a production site. On the other hand, if the extension looks to be perfect for your needs, using the beta version while you develop the site, and helping the developer find and fix bugs, could be a symbiotic relationship. Moreover, some extension developers mark their releases as betas just to cover themselves when the extension is solidly usable for all practical purposes.

- **How well maintained it is**

To me, one of the most important criteria is how well maintained an extension (or any software/code you use) is. It's best not to rely upon an extension that will become too outdated to be useful. Look at the extension's version number to tell how well maintained an extension is. Also note how recently updated the extension is. And check out how active the developer is in replying to comments and bug fixes.

- **How well documented it is**

There's no point in attempting to use an extension that you won't be able to figure out how to use. And, as a writer, I particularly value good documentation.

- **How popular it is (in terms of both downloads and rating)**

Popularity isn't always a good thing, and it's certainly not the most important criteria, but can be useful when making a final decision.

{TIP} You can also learn a lot about an extension—how useful it is, how well maintained it is, etc.—by searching for the extension in the Yii forums.

Once you've identified the extension to use, you install it either manually or using Composer. The latter route is preferred, although not all extensions are Composer-ready. But if an extension is:

1. Add the extension requirement to your **composer.json** file.
2. Run **composer install** or **composer update**.

These two steps install the extension files in your application, placing them within the **vendor** directory. The list of installed extensions also gets written to the **vendor/yiisoft/extensions.php** file. This file indicates the versions installed, and sets aliases for each.

Alternatively, you can install an extension by directly invoking **composer require extension-id**. This command installs the specific extension and adds it to your **composer.json** file. When using this approach, running **composer update** has the side benefit of updating all your extensions.

Because Composer is easy and reliable, you should default to Composer extensions, checking [Packagist](#) for them.

If an extension isn't Composer-ready, you'll need to manually install it:

1. Download the code (from its extension page).
2. Expand the downloaded code (from a **.zip** or other file type to a folder).
3. Move or copy the resulting folder to the **vendor** directory.
4. Connect the extension through an autoloader.

These are generic instructions. Some extensions will require that you rename the resulting folder (from, say, “yii-bootstrap-2.0.3.r329” to just “bootstrap”). Just read

and follow the installation instructions that the extension provides. If it doesn't have installation instructions, then you don't want that extension. The biggest catch is how to tie in the autoloader. See Chapter 20, “[Working With Third-Party Libraries](#),” for more details on that subject.

That being said, I strongly recommend only using Composer-supported extensions unless absolutely necessary.

How you configure and use an extension also differs from one extension and type to the next. Some are configured as application components, others just need to be referenced where you're using it (e.g., a widget).

Before moving on, I have two specific recommendations. First, if your permissions are not properly set (if the web server cannot read everything within the **vendor** directory), you'll get errors and unusual results when you go to use any extension. I found when using Mac OS X, that any time I had unusual results when first using an extension, I would have to fix the permissions on the applicable directories to get the problem sorted.

Second, trying to use any extension for the first time can be quite frustrating. In originally writing this chapter, I ran into many hurdles with extensions that I would have liked to cover (not insurmountable hurdles, necessarily, but too many hurdles to reasonably still use the extension in a book). As this will likely be the case for you as well, I would recommend first installing and testing new extensions on a practice project. By using a demo site, you won't run the risk of cluttering up, or worse yet, breaking, your actual project. Or, of course, use version control to test and easily roll back the changes you've made.

With that general introduction in place, let's look at some specific extensions. Note that this list is entirely different for the second edition of this book than for the first. For example, Bootstrap is used by the basic application by default, and is further covered in the previous chapter. Two topics—Elasticsearch and SwiftMailer—were originally covered as third-party libraries in the first edition of this book, but are now directly supported through extensions.

Probably the first set of third-party extensions you should look at are the [Krajee Extensions](#). Krajee offers a great selection of Yii 2 extensions, including:

- Additional application templates
- Helpers
- Modules
- Widgets

Some of the specific functionality offered includes:

- Masked text inputs (e.g., to suggest the format for phone numbers or dates)
- Markdown support
- Social media plugins

- HTML navigation interfaces
- Data exporting
- PDF generation
- Inline HTML editing

The Krajee extensions are well enough documented and have been updated as recently as July 2016 (at the time of this writing).

To use the Krajee extensions, you must first include the [Krajee base extension](#), which provides core functionality for the entire Krajee library:

```
composer require kartik-v/yii2-krajee-base
```

With the extension base installed, I'll quickly mention three specific Krajee extensions. As a quick heads-up, extensions that are widgets are added to Active Record forms via the `widget()` method on the form element. See Chapter 12, “[Working with Widgets](#),” for more details.

The Krajee [yii2-password](#) extension provides password strength validation tools. One ought to be judicious in using such tools but if done properly, they can help users make better decisions.

Install the extension using this command:

```
composer require kartik-v/yii2-password
```

To enforce certain types of password strength regulations, add the following to your model, presumably a user or login model:

```
use kartik\password\StrengthValidator;
```

Next, apply the Krajee “StrengthValidator” to the password attribute:

```
public function rules() {
    return [
        // Other rules.
        [['password'], StrengthValidator::className(),
            'preset'=>'medium', 'userAttribute'=>'username']
    ];
}
```

The “userAttribute” parameter points the extension to the `username` attribute of the model so it can check that the password does not include the user's `username`. You can also set the minimum length, maximum length, and the minimum number



Figure 13.1: The extension's password input.

of lowercase letters, uppercase letters, digits, and special characters the password should include.

Common combinations are defined in the extension as presets. For example, the “medium” present requires the password to:

- Be at least 10 characters long
- Include at least 1 uppercase letter
- Include at least 1 lowercase letter
- Include at least 2 digits
- Include at least 1 special character
- Not include the username
- Not include an email address

When using this extension, the `password` attribute must still be set as required. The extension's rules only apply when a value is provided. But the extension does perform both client-side and server-side validation.

You can use the rules in conjunction with the extension's password input. The input is a widget that displays a password input along with a strength indicator (**Figure 13.1**).

The corresponding code is:

```
# views/site/login.php
use kartik\password\PasswordInput;
// Other stuff.
<?= $form->field($model, 'password')->passwordInput()
    ->widget(PasswordInput::classname(), [
        'pluginOptions' => [
            'showMeter' => true,
            'toggleMask' => false
        ]
    ]); ?>
```

See the [extension's documentation](#) for additional information.

The Krajee Extensions provide additional tools for “masking” form inputs. *Masking* adds a syntactic interface to an element to instruct the user on how to format their

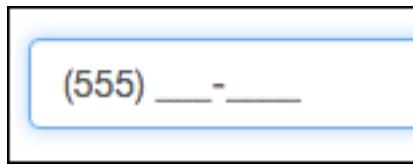


Figure 13.2: A masked phone input.

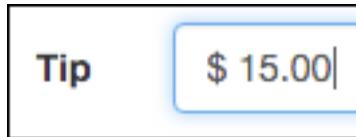


Figure 13.3: A masked input for money.

entry or to more meaningfully display a user's input. Yii2 has a masking widget built-in (**Figure 13.2**).

If you like this effect, you may also like the [Yii2 Money](#) extension.

First, include the "kartik-v/yii2-money" extension in your application:

```
composer require kartik-v/yii2-money
```

Next, within the view file, include the proper namespace:

```
use kartik\money\MaskMoney;
```

Then, on a form that uses Active Record, invoke the `widget()` method, passing along `MaskMoney` as the class, and configuring it accordingly:

```
<?= $form->field($model, 'tip')->widget(MaskMoney::classname(), [
    'pluginOptions' => [
        'prefix' => '$ ',
        'allowZero' => true
    ]
]); ?>
```

That code prefixes the amount entered with a dollar sign but does allow a zero value (**Figure 13.3**).

Otherwise, the widget uses the default settings:

- A comma as the thousands separator
- The decimal point as the decimal separator
- Precision to two decimal points
- Does not allow negative values

All of these are adjustable, either when instantiating a specific widget or by setting global “MaskMoney” defaults. See the extension’s documentation for more.

The third Krajee extension I’ll highlight is the Yii2 GridView. This widget provides a super advanced alternative to Yii’s built-in grid. Some of its features include:

- Ability to export data
- Internationalization and translation
- Finely tuned table styling
- [Pjax](#) support
- Page summaries
- Toolbars
- Flexible headers and footers

Install the extension using:

```
composer require kartik-v/yii2-grid
```

To use the extension, add it as a module to your application’s configuration:

```
# config/web.php
// Other stuff.
$config = [
    // More other stuff.
    'modules' => [
        'gridview' => [
            'class' => '\kartik\grid\Module'
        ]
    ],
// Even more other stuff.
```

You’ll learn more about modules in Chapter 19. For now, note that the module is enabled by configuring the “modules” element of the main configuration array. This particular module must be named “gridview”. Default behavior can be configured after the class, should you so choose.

Creating an instance of the widget is essentially the same as using the built-in “GridView”. Start by replacing the user of the Yii widget with the new one:

```
# views/someviewfile.php
// use yii\grid\GridView; # OLD
use kartik\grid\GridView; # NEW
```

Next, in the widget creation code, provide the widget with a data provider and a filter model:

```
1. GridView::widget([
    'dataProvider'=>$dataProvider,
    'filterModel'=>$searchModel,
2. => false
]);
```

3. also want to disable exporting up front, unless you've also installed all of the secondary extensions required by that functionality.
4. that basic start, the remaining configuration requires using the [Krajee documentation](#). For example, to disable the default table striping, set `striped` to `false`. To have the table retain the user's resizing of the columns (using local storage), set `persistResize` to `true`.
5. encourage you to look around at the available options, and check out the [online demo](#).

The [2amigos](#) (or dosamigos) group has also created a number of Yii2 extensions. Although they aren't well documented, they tend to fit good niches. Let's look at two.

The first 2amigos extension to highlight is a nice [file input widget](#). The widget could be better documented, but if you look at the [source code](#) in GitHub, you can find what you need.

The “FileInput” widget adds attractive styling to a file input, in keeping with Twitter Bootstrap. Most importantly, it has the handy feature of showing a preview of a previously uploaded file (**Figure 13.4**).

Install the extension using this command:

```
composer require 2amigos/yii2-file-input-widget
```

Then include the namespace in your view file and create an instance of the widget:

```
use dosamigos\fileinput\FileInput;
<?=FileInput::widget([
    'model' => $model,
    'attribute' => 'image',
    'thumbnail' => $model->getImage(),
    'style' => FileInput::STYLE_IMAGE
]);?>
```

The “thumbnail” option needs to be provided with the IMG code for the thumbnail image. Logically, this might be provided by a model method:

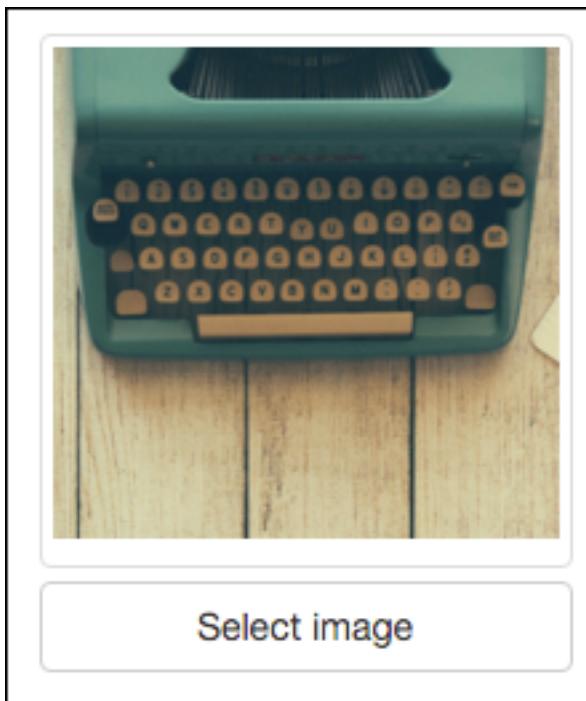


Figure 13.4: A styled file input with the existing image preview.

```
public function getImage() {
    // Assumes the "preview" attribute stores the path:
    return Html::img($this->preview);
}
```

To use this widget within an Active Record form, invoke the `widget()` method of the field object, passing along this class name and configuration.

Often, a site like the CMS example will require an administrative area to dynamically manage the site's content. Much of the content can contain some HTML, including media (images, videos, etc.), typography, lists, and so forth. So that non-technical people can create nice-looking HTML, I normally turn to a web-based WYSIWYG editor like [CKEditor](#) or [TinyMCE](#). Getting either to work within the Yii environment isn't too hard, once you know what to do. However, the process can be greatly simplified thanks to the right extension.

If you want to use CKEditor, the [CKEditor Widget for Yii2](#) extension by 2amigos is easy to use. It creates a widget that you can drop into your view files wherever you need an instance of the CKEditor.

Install the extension using this command:



Figure 13.5: The CKEditor instance used to create and edit a Page record.

```
composer require 2amigos/yii2-ckeditor-widget
```

To use the CKEditor in a form, you'll need to edit the `_form.php` file for the particular view, such as for “page” in the CMS example. By default the form will have a standard textarea for every TEXT type:

```
# views/page/_form.php
<?= $form->field($model, 'content')->textarea(['rows' => 6]) ?>
```

To use CKEditor instead of the textarea, invoke the widget by replacing that code with this:

```
use dosamigos\ckeditor\CKEditor;
# views/page/_form.php
// Other stuff
<?= $form->field($model, 'text')->widget(CKEditor::className(), [
    'options' => ['rows' => 6],
    'preset' => 'basic'
]) ?>
```

If you load page/create in your browser, you should now see a lovely WYSIWYG editor in lieu of the text area (**Figure 13.5**).

There are further configurations explained in the [extension's source code](#) (again, there's not much standalone documentation with 2amigos extensions). For example, you can easily switch among the basic, full, and standard presets (**Figure 13.6**):

```
# views/page/_form.php
<?= $form->field($model, 'text')->widget(CKEditor::className(), [
    'options' => ['rows' => 6],
    'preset' => 'standard'
]) ?>
```

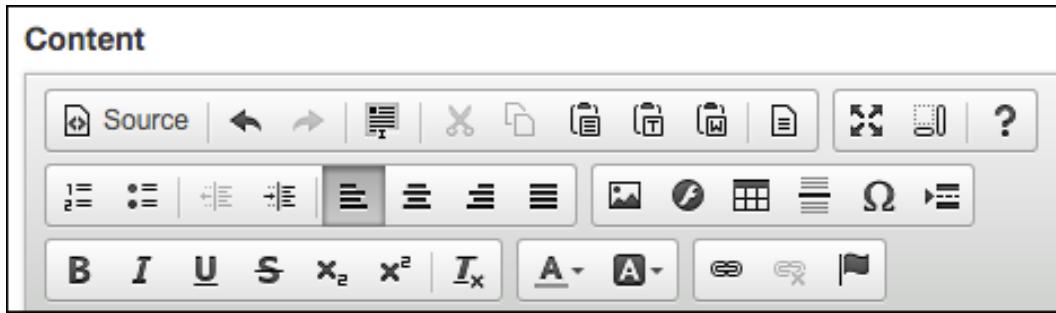


Figure 13.6: The standard CKEditor toolbar.

Those steps are pretty easy to understand and will get you a working WYSIWYG editor in no time. But you'll likely need to tweak how the CKEditor behaves, too, which is much more complicated.

Finally, if you want to allow the admin to upload files to the server, like images or videos, you'll need to enable the file manager. This gets complicated, as CKEditor does not come with this functionality built-in. You'll need to either buy the commercial [CKFinder](#), or find a third-party alternative.

Once you've done that, you'd configure CKEditor through the widget's `clientOptions` parameter.

As most sites have some sort of user registration and login component, there's a strong argument for using a third-party extension or defining your own to provide this functionality. A number of user extensions already exist, and of them, Amnah's [Yii2 User](#) extension is worth consideration. It's up-to-date, decently documented, and includes many common features:

- Can be username- or email- based
- Can require email confirmation
- Supports authentication via third-party sites (e.g., social media)
- Defines an account management page
- Built-in password recovery
- Easily extendable

Install the extension using this command:

```
composer require amnah/yii2-user
```

Next, you need to configure your application to use the extension, which requires two settings. You must both enable the module and tell Yii that the extension defines the user component class:

```
'modules' => [
    'user' => [
        'class' => 'amnah\yii2\user\Module',
    ],
],
'components' => [
    'user' => [
        'class' => 'amnah\yii2\user\components\User',
    ],
],
// Other components.
```

In the module section, you can configure the [settings](#), such as:

- If an email address or username is required or optional
- What parameter—email address or username—is used to log in
- If email confirmation is required
- How long cookies and sessions should last

Next, perform a migration to create the necessary database tables:

```
php yii migrate --migrationPath=@vendor/amnah/yii2-user/migrations
```

In order for the extension to run, it requires that the database has no existing `user` table.

At this point, you have a working user extension. If you load the “user” route, you’ll see an intro page with a list of new links ([Figure 13.7](#)).

Log in using `neo/neo`. Then head to “user/admin” to change the admin username and password to something unique and secure ([Figure 13.8](#)).

From there, you can use the user identity class as you would any other, including checking authorization through the `can()` method (see Chapter 11, “[User Authentication and Authorization](#)”).

[Swift Mailer](#) is a library that provides an object-oriented interface for reliably sending out email using PHP. As you may know, sending plain-text email in PHP is blazingly simple, assuming the server is properly configured. If it’s not, then...it’ll be much harder. Sending HTML email, on the other hand, pretty much always requires the use of a third-party library in order to be done reliably.

Swift Mailer has many great features:

- Ability to send email via sendmail, postfix, SMTP, or other mechanisms
- Support for authentication
- Prevents header injection attacks to reduce the chance of spam being sent

<h1>Yii 2 User Module - 4.0.0</h1>	
<h2>Actions in this module</h2>	
<p>Note: Some actions may be unavailable depending on if you are logged in/out,</p>	
URL	Description
/user	This 'actions' list. A
/user/admin	Admin CRUD
/user/login	Login page

Figure 13.7: The user extension module home page.

Account

Current Password

Email Changing your email requires email confirmation

Username

New Password

Update

Figure 13.8: The account management page.

- Support for attachments and inline images

There are Yii extensions for Swift Mailer, including an [official Yiisoft one](#).

As this is a Yiisoft extension, it may already be included in your application. If not, install the extension using this command:

```
composer require yiisoft/yii2-swiftmailer
```

Then configure the “mailer” component:

```
# config/web.php
'components' => [
    'mailer' => [
        'class' => 'yii\swiftmailer\Mailer',
    ],
]
```

Using SwiftMailer is straightforward. Because it’s configured as an application component, you can access it through `Yii::$app->mailer`.

This code, taken from the [guide](#) shows a simple email sent in both plain text and HTML formats:

```
Yii::$app->mailer->compose()
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Message subject')
    ->setTextBody('Plain text content')
    ->setHtmlBody('<b>HTML content</b>')
    ->send();
```

See the guide for details on generating HTML, embedding images, attaching files, and so forth.

This code does assume that the local PHP installation is configured to send email. If you need to go through an external mail server, configure the “transport” property of the component:

```
# config/web.php
'components' => [
    'mailer' => [
        'class' => 'yii\swiftmailer\Mailer',
        'transport' => [
            'class' => 'Swift_SmtpTransport',
            'host' => 'smtp.example.com',
        ],
    ],
]
```

```
'username' => 'username',
'password' => 'password',
'port' => '587',
'encryption' => 'tls',
],
],
```

For the last extension in this chapter, I want to use the [official Yiisoft](#) extension for [Elasticsearch](#). Elasticsearch, in case you're not familiar with it, is, well, the latest and greatest search engine around.

Elasticsearch is a real-time, distributed search engine and more. Elasticsearch supports full-text searching, structured searching, and analytics. Elasticsearch, like Solr, is built on top of Apache Lucene, a Java-based full-text search engine. Lucene, though, is not quite so easy to use (I've played with it and Solr extensively and...it wasn't fun). Elasticsearch uses a RESTful API and JSON for its interactions, making common activities—indexing content, searching content, etc.—as simple as performing any other API call.

Elasticsearch is already in use by Wikipedia, StackOverflow, GitHub, and many other extremely popular and active sites. Still, Elasticsearch is remarkably easy to begin using. I'll provide a basic introduction to Elasticsearch here, and demonstrate how you'd use it with Yii to create a search engine for your site.

To start, you need to install Elasticsearch, which is quite simple.

1. Head to <https://www.elastic.co/>.
2. Navigate to the free version of **Elasticsearch**.
3. Click **Download**.
4. Download the ZIP version (or whichever you prefer).
5. Expand the downloaded file.

The download will be a file named something like **elasticsearch-2.1.1.zip**. Expand this file to get a folder named something like **elasticsearch-2.1.1**. That's it! You've now installed Elasticsearch!

{NOTE} Elasticsearch does require that you have the Java Development Kit installed on your machine.

Like the MySQL database, Elasticsearch must be running in order to be used. It's started via a command-line interface.

1. Access your computer via a command-line interface.
2. Navigate to the Elasticsearch's **bin** directory.

```
~/D/e/bin » cd ~/Downloads/elasticsearch-2.1.1/bin
[~/D/e/bin » ./elasticsearch
[2016-01-12 10:06:47,498] [INFO ] [node           ] [Callisto] version[2.1.1],
[2016-01-12 10:06:47,499] [INFO ] [node           ] [Callisto] initializing ...
[2016-01-12 10:06:47,546] [INFO ] [plugins       ] [Callisto] loaded [], sites
[2016-01-12 10:06:47,561] [INFO ] [env            ] [Callisto] using [1] data p
[2016-01-12 10:06:48,565] [INFO ] [node           ] [Callisto] initialized
```

Figure 13.9: Starting Elasticsearch.

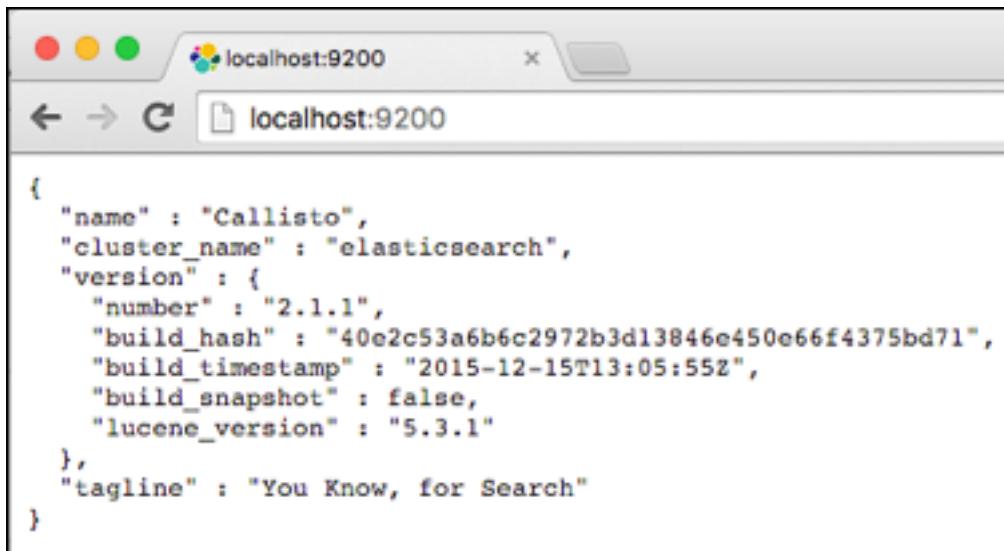


Figure 13.10: Elasticsearch is now running.

```
cd /path/to/elasticsearch-2.1.1/bin
```

3. Execute this command: `./elasticsearch` (**Figure 13.9**).

And that should be it! You should see that Elasticsearch is running. You can confirm this by loading <http://localhost:9200> in your browser (**Figure 13.10**).

When it's time to stop Elasticsearch, head back to the same terminal or DOS prompt window where you started it, and press Control+C.

Elasticsearch uses a RESTful API, which means that accessing different URLs has different effects. For example, searches are performed through http://localhost:9200/_search. But before you can search, you must have an index of data to search through.

Indexes are created by sending a PUT request to: <http://localhost:9200/index/type/id> (with “index”, “type”, and “id” replaced by meaningful values). Elasticsearch stores its data in “indexes”, which are the equivalent to databases. Within a single index (or database), you can have multiple “types”, which are equivalent to tables. Each record of data stored in a type can be associated with an ID value. In Elasticsearch

parlance, a record that is indexed is a “document”, containing both metadata and a body.

With all of this in mind, this means that you can create an index for a page of content in a CMS example by performing a PUT request to `http://localhost:9200/cms/page/1`. I’ll return to the syntax of that request later.

Note that unlike a database, you don’t have to take any steps to create an index or type in advance. The simple act of making the PUT request creates everything necessary.

Existing indexes are searched using a POST request to—

`http://localhost:9200/index/type/_search`

—with both the index and type being optional. This means that a request to `http://localhost:9200/cms/_search` will perform a search within the “cms” index and a request to `http://localhost:9200/cms/page/_search` will perform a search within the “page” type of that index.

When it comes time to perform a search, Elasticsearch has its own query Domain Specific Language (DSL). This is roughly equivalent to SQL. At the most basic level, you can pass a “q” value to perform a search:

`http://localhost:9200/cms/page/_search?q=yii`

Basic queries use the Lucene query parser syntax. By default, all indexed fields will be searched. As you’ll soon see, you can indicate fields when indexing content, such as author, date, and content for a page in a CMS site.

For the rest of the chapter, most of the interactions with Elasticsearch will be done through the Elasticsearch Yii extension. However, it helps to first communicate with Elasticsearch directly in order to understand the basics of the engine. This will be much the same as learning how to first execute SQL queries straight against the database before doing so from a PHP script.

Communications are done through the URL, `http://localhost:9200`. You can access this through your browser or using cURL in the command-line. When it comes to interacting with Elasticsearch, I highly recommend the excellent Sense extension for the Chrome browser, available in the Chrome store (it’s free). It’s still in beta, but more than sufficient for basic purposes.

After installing Sense, you’ll see a pane on the left for editing and a pane on the right for showing Elasticsearch’s response. Sense supports autocompletion (using Enter/Tab), code indentation, code folding, and a history of executed commands. Click the green triangle (pointing right, like a Play button) to execute the edited command.

To start, let’s create an index, type, and ID to store a bit of data for later searching. To do that, you’ll need to make a PUT request. You can execute the following using Sense:



The screenshot shows a browser window with the title "Server localhost:9200". On the left, a code editor displays a PUT request:

```

1 PUT /cms/page/1
2 {
3     "author": "Larry",
4     "post_date": "2014-04-01",
5     "title": "The Lorem Example",
6     "content": "Lorem ipsum dolor..."
7 }
8

```

On the right, the response is shown in JSON format:

```

1 {
2     "_index": "cms",
3     "_type": "page",
4     "_id": "1",
5     "_version": 1,
6     "_shards": {
7         "total": 2,
8         "successful": 1,
9         "failed": 0
10    },
11    "created": true
12 }

```

Figure 13.11: A document has been indexed.

```

PUT /cms/page/1
{
    "author": "Larry",
    "post_date": "2014-04-01",
    "title": "The Lorem Example",
    "content": "Lorem ipsum dolor..."
}

```

After clicking the green arrow, you should see a positive response on the right (**Figure 13.11**).

{TIP} If you make another PUT request to an existing index/type/ID combination, the result will be an update of that stored content.

To search through that indexed document, use this syntax:

```

GET /cms/page/_search
{
    "query": {
        "match": {
            "author": "Larry"
        }
    }
}

```

The screenshot shows a web interface for a Elasticsearch search. The URL is `localhost:9200`. The search query is:

```

1 GET /cms/page/_search
2 {
3     "query": {
4         "match": {
5             "author": "Larry"
6         }
7     }
8 }

```

The results are:

```

1 {
2     "took": 39,
3     "timed_out": false,
4     "_shards": {
5         "total": 5,
6         "successful": 5,
7         "failed": 0
8     },
9     "hits": {
10        "total": 1,
11        "max_score": 0.30685282,
12        "hits": [
13            {
14                "_index": "cms",
15                "_type": "page",

```

Figure 13.12: A basic search, with one hit.

That query looks for an author match, which should return one record (**Figure 13.12**).

This is the most basic search one can do. The query results begin with the general information: how long it took, if the request timed out, how many shards were searched, etc. Then you'll see how many hits were found and a maximum relevancy score. This will be followed by the records that count as hits.

The same query can be executed by accessing this URL directly in your browser or via cURL, not using Sense:

`http://localhost:9200/cms/page/_search?q=author:Larry`

If you wanted to search through every field, you'd use “`_all`” instead of the field name.

Again, “`match`” provides a very simple search, but it does work as a full-text search on multiple words:

```

GET /cms/page/_search
{
    "query": {
        "match": {
            "content": "dolor sit"
        }
    }
}

```

That search would return any record that contains “`dolor`” or “`sit`”, with records

```
"highlight": {
    "content": [
        "Lorem ipsum <em>dolor</em>..."
    ]
}
```

Figure 13.13: Search results with highlighted terms.

that contain both being ranked higher. To turn this into a phrase match—only return records with “dolor sit” exactly, you’d use “match_phrase” instead of “match”.

As a cool feature, Elasticsearch has a built-in highlighter, wrapping found words or phrases in HTML EM tags. To use that, change the JSON request to:

```
GET /cms/page/_search
{
  "query": {
    "match": {
      "content": "dolor sit"
    }
  },
  "highlight": {
    "fields": {
      "content": {}
    }
  }
}
```

The query results will be the same, but extra information will be returned in a “highlight” response field. It will contain the portion of the text that contains the highlight (**Figure 13.13**).

Next, if you change the search term to “Lore”, and click the green arrow, you’ll notice that no hits are returned, despite the first word in the content being “Lorem”. This is due to the default Elasticsearch indexing, which I will address in more detail shortly.

If you want, you can run any of the above searches on the entire index (all of “cms”) by using the URL http://localhost:9200/cms/_search instead.

Results can also be changed by providing:

- `size`, the number of results to return
- `from`, an offset

- `fields`, the specific fields to return in the hit results
- `sort`, an alternative sort order

{TIP} By default, Elasticsearch returns the top 10 hits for any search query.

The real power comes from the query DSL. You can use configurations to indicate what must be present, what must not be, ranges of numbers or dates, even geolocation! You can also tap into filters to apply limitations to the query results. And analytics can be used to aggregate data. Elasticsearch is an extremely powerful search engine!

Before moving back to Yii, I'll recommend that you read Joel Abrahamsson's [Elasticsearch 101](#) tutorial. It's easy to digest and does a great job of explaining query DSL in particular.

With an understanding of Elasticsearch in place, it's time to turn to the Elasticsearch Yii extension, developed by the Yii core team. Install the extension using this command:

```
composer require yiisoft/yii2-elasticsearch
```

Next, enable and configure the extension:

```
# config/web.php
'components' => [
    'elasticsearch' => [
        'class' => 'yii\elasticsearch\Connection',
        'nodes' => [
            ['http_address' => '127.0.0.1:9200'],
        ],
    ],
],
```

The component name is “elasticsearch”, and the minimum configuration is presented there. As a component, it's accessible via `Yii::$app->elasticsearch`. You'll want to make a reference to this in your controller:

```
$es = Yii::$app->elasticsearch;
```

Subsequent tasks—indexing content, performing searches, etc.—will be performed through that object.

While getting up to speed with Elasticsearch, you'll likely want to make use of the Elasticsearch debug panel, which rides on top of the “debug” module. Add it to your configuration:

Time	Url / Query
3.1 ms	GET myindex/users/_search {"fields":["id, name"], "size":10, "query":{"match_all":{}}} <i>/Users/larry/Sites/yii2-scratch/controllers/SearchController.php(33)</i>
7.6 ms	no success

Figure 13.14: The Elasticsearch debug panel.

```
# config/web.php
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = [
        'class' => 'yii\debug\Module',
        'panels' => [
            'elasticsearch' => [
                'class' => 'yii\\elasticsearch\\DebugPanel',
            ],
        ],
    ];
}
```

You'll then see an additional item in the browser debug window, named “Elasticsearch”. On that panel you can see the Elasticsearch queries that ran and even rerun them to see the results for yourself (**Figure 13.14**).

For the rest of the chapter, I'm going to assume you'd create a “SearchController”, with these methods:

- `actionCreate()`, for creating the index
- `actionIndex()`, for indexing content
- `actionSearch()`, for search the index

The last method will be the default, and you'll probably want to add access control to the other two in a real-life site, limiting the ability to update or create indexes to just an administrator. Better yet, those two methods could be converted into command-line versions (see Chapter 16, “[Leaving the Browser](#)”).

For the specific example, I'll use something similar to CMS, this time indexing some book content. The fields will be: title, author, and content. You can download

the SQL required to create the database table and records from the [book download page](#).

When you store data in Elasticsearch, the engine indexes the content. With text, this means *tokening* it: breaking it into chunks and filtering them. By default, Elasticsearch will break words using spaces and punctuation, filter them as all lowercase—both “Lorem” and “lorem” will be represented as “lorem”, and then filter out any stopwords (such as “the” in English). The indexing engine relies upon an *analyzer* to perform these steps.

When you do a search, another analyzer is applied. It may or may not take the same steps as the index analyzer, depending upon the configuration.

In order to be able to search through stored data making matches on parts of a word, you’ll need to configure how the indexing analyzer works. Unlike in the previous, browser-based example, where an index was created by indexing content, you can also create and configure an index as a first, separate step. I’ll go through an example, but you may want to read the Elasticsearch documentation for all the gritty details, or [this tutorial](#) for a quick overview.

To allow for partial word matches in the example I’m using, the index analyzer needs to be told to tokenize the words not just on spaces and punctuation, but also on a range of lengths. Reasonably, the range could go from 3 characters to 10. The result would be that the word “Lorem” would be indexed as all of the following:

- lor
- lore
- lorem
- ore
- orem
- rem

(All in lowercase because of the filter.)

In a few pages, I’ll explain the exact code needed to make this change.

Another way you can improve the quality of the indexing and searching is by applying *mapping*. By default, Elasticsearch is schema-less, meaning you don’t need to tell it what kind of data you’re storing. Elasticsearch will dynamically determine the proper data type and index it according to some standard rules.

That’s often fine, and certainly okay for basic examples and practicing, but there are advantages to defining your schema by mapping the data you’ll be indexing to certain types:

- string
- date
- long

- double
- byte
- short
- integer
- float
- boolean
- object
- ip (as in IP address)

Along with those types, there are a couple of geolocation-related types.

With this in mind, it's time to create the index in Yii.

As just mentioned, the `actionCreate()` method of the “Search” controller will create the index. This is going to be a bit complicated, and took some trial-and-error on my part to figure out, but I'll explain the code as best as I can.

To start, create an array of parameters:

```
$params = [];
```

The next goal is to set the index to allow for partial matches. This is done by configuring the “analysis” body setting, which means you'll be assigning an array to `$params['body']['settings']['analysis']`. The term “body” is used there, because it applies to the body of the documents being indexed.

This main array should have two subarrays: “filter” and “analyzer”. The filter should identify the filter that parses out strings of 3 to 10 characters in length. The analyzer is then told to use this filter. Here's that code:

```
$params['body']['settings']['analysis'] = array(
    'filter' => array(
        'my_filter' => array(
            'type' => 'ngram',
            'min_gram' => 3,
            'max_gram' => 10
        )
    ),
    'analyzer' => array(
        'my_analyzer' => array(
            'type' => 'custom',
            'tokenizer' => 'standard',
            'filter' => array('lowercase', 'my_filter')
        )
    )
);
```

“ngrams” is the type of filter that can be used to break a string into substrings. It’s provided minimum and maximum values, and the whole configuration is assigned the name “my_filter”.

The analyzer is of type “custom”, and given the name “my_analyzer”. This analyzer uses the standard tokenizer, and applies two filters. The first is the normal lowercase filter; the second is the custom filter.

Next, the mappings can be established. Specifically, the mappings for a “book” type. First the configuration will say that the source should be included. Then each property of a “book” is identified by name and type. For the title, author, and content, the custom analyzer is assigned.

```
$params['body']['mappings']['book'] = array(
    '_source' => array(
        'enabled' => true
    ),
    'properties' => array(
        'id' => array(
            'type' => 'integer',
        ),
        'title' => array(
            'type' => 'string',
            'analyzer' => 'my_analyzer'
        ),
        'author' => array(
            'type' => 'string',
            'analyzer' => 'my_analyzer'
        ),
        'content' => array(
            'type' => 'string',
            'analyzer' => 'my_analyzer'
        )
    )
);
```

Hopefully there’s nothing too confusing there, but you can read more online about creating analyzers if you’re curious.

With all of the parameters defined, you can create an index. To do so, invoke the `createCommand()` method of the Elasticsearch component to get a `Command` object. This is similar to a database command object that could be used to execute queries directly on the database. Then invoke the `createIndex()` method on the command object, passing the index name and parameters as its two arguments:

```
$es = Yii::$app->elasticsearch;
$cmd = $es->createCommand();
$cmd->createIndex('books', $params);
```

And that should do it! If a problem occurs, an exception will be thrown. Obviously you can render a view file that says something (although a method like this would only be invoked once and could be invoked from the command-line).

Going to this URL in your browser will show the existing indexes:

```
http://localhost:9200/_cat/indices?v
```

If, while playing around with this, you find you made a mistake in creating the index, you can delete it using this code:

```
$cmd->deleteIndex('books');
```

With the index created and configured, you can start throwing documents into it. As already mentioned, the content could be derived by crawling a website or come direct from the database. In this case, I'm going to do the latter. There's no need for ActiveRecord here, you can just run a query on the database, fetch the results, and index them.

This code would go in the `actionIndex()` method:

```
$q = 'SELECT * FROM books';
$cmd = Yii::$app->db->createCommand($q);
$result = $cmd->queryAll();
foreach ($result as $row) {
    // Index $row.
}
```

To index the content, you'd again get an Elasticsearch command object:

```
$es = Yii::$app->elasticsearch;
$es_cmd = $es->createCommand();
```

Note that the Elasticsearch command object needs a unique name so as not to conflict with the database command object.

This would be before the `foreach` loop.

Next, within the `foreach` loop, invoke the `insert()` method on the Elasticsearch command object, providing: the index name, the type, and the data. The index name should be “books”, and the type “book”. The data is an array of names and values of the document being indexed. In this case, that's:

- title
- author
- content

So within the `foreach` loop, you'd have code like this:

```
$data = [
    'author'=>$row['author'],
    'title'=>$row['title'],
    'content'=>$row['content']
];
```

Note that you're not referencing the book ID value here, because that value isn't being indexed. The ID value will instead be the value of the index itself (assigned next).

That code needs to be followed by the ID, assigned to `$id`:

```
$id = $row['id'];
```

And now you can index that content:

```
$es_cmd->insert('books', 'book', $data, $id);
```

Here's the mostly complete method body, then:

```
$es = Yii::$app->elasticsearch;
$es_cmd = $es->createCommand();
$q = 'SELECT * FROM books';
$cmd = Yii::$app->db->createCommand($q);
$result = $cmd->queryAll();
foreach ($result as $row) {
    $data = [
        'author'=>$row['author'],
        'title'=>$row['title'],
        'content'=>$row['content']
    ];
    $id = $row['id'];
    $es_cmd->insert('books', 'book', $data, $id);
}
```

Again, you'll want to create a view file to show the results, or use this from the command-line.

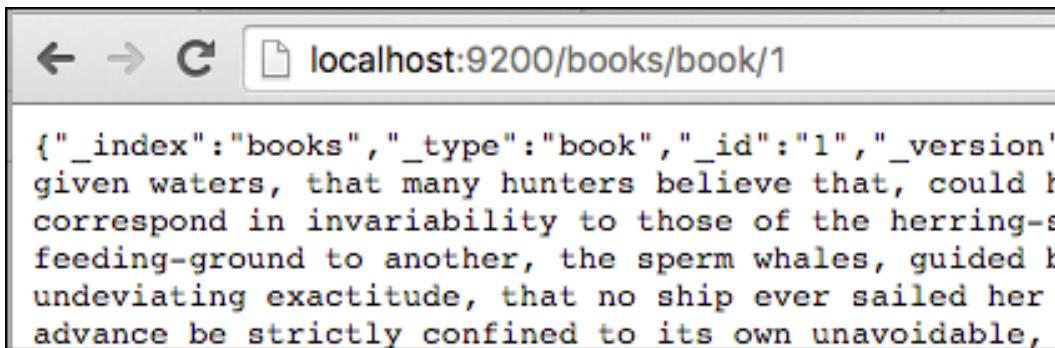


Figure 13.15: The first indexed book.

To confirm this worked, you could now perform a GET request (in the browser) of `http://localhost:9200/books/book/1` (**Figure 13.15**).

Last, and most importantly, it's time to search the index. My assumption is that you'd create a static HTML form with a text input named "terms". This form would use the GET method, and point to the `actionSearch()` method of the "Search" controller. That would be set as the default method of the controller:

```
class SearchController extends Controller {
    public $defaultAction = 'search';
```

Thanks to that line, just the URL `/search?terms=blah` will work (as opposed to having to use `/search/search?terms=blah`).

The search form would be one of the very rare forms in Yii that wouldn't need an underlying model. (Although the Elasticsearch extension does support interacting with Elasticsearch via ActiveRecord.)

The `actionSearch()` method should perform a search if terms were provided. Unlike running a search on a database, you don't need to worry about SQL injection attacks here. The search results should be passed as an array to the view file for display.

You can perform a basic search using an Elasticsearch Query object. It functions much like a database query builder. Let's look at the functional code, and then explain it.

```
use yii\elasticsearch\Query;
public function actionSearch($terms = null) {
    $results = [];
    if (isset($terms)) {
        $query = new Query;
        $query->fields(['author', 'title', 'content'])
            ->from('books', 'book')
```

```
        ->query(['match' => ['_all' => $terms]]);
    $results = $query->search();
}
return $this->render('search', ['results' => $results]);
}
```

First, the action method takes the search terms as an optional argument. This allows the page to show the results, if a term was provided, or show the search page otherwise.

If terms were provided, a new `Query` object is created. The fields to be returned are listed, as an array. You have to do this, or else no fields will be returned as part of the search results. The `from()` method is used to indicate the index and type being searched.

Next, the `query()` method sets the specific query. Using syntax already identified, the hope is to match all fields against the provided search terms.

Finally, the query is executed. The results of executing that code will be the same as running a match query within the browser, as shown earlier. This means that you can loop through the results and find every returned hit.

To be more specific, `$results` will be an array, corresponding to the main array shown in the results in Sense. This means that you'll have these array elements:

- `took`
- `timed_out`
- `_shards`
- `hits`

The `$results['hits']` element will itself have these elements:

- `total`
- `max_score`
- `hits`

So in your PHP code, `$results['hits']['total']` is the number of hits found by Elasticsearch.

`$results['hits']['hits']` is going to be an array with one element for each hit. That subarray will have these elements:

- `_index`
- `_type`
- `_id`
- `_score`

- **fields**

Within `fields`, you'll find the document's properties. In this case that's author, title, and content. Each field is also an array.

To reiterate all this, `$results['hits']['total']` is the number of hits found by Elasticsearch. And `$results['hits']['hits'][0]` is the first found hit (the highest ranking one, by default). And `$results['hits']['hits'][0]['fields']['title'][0]` would be the title value for the first found hit. Whew!

Again, just look at the results in Sense if you're ever lost with all of this. If you're ever in doubt as to what's returned by a search, take the same debugging steps you would with MySQL. Start by using the Elasticsearch debugging panel to view the query being run and execute it directly. With Elasticsearch, you can also run the same query using the browser or cURL to see the results without the additional Yii layer.

So now, after running a simple search, you have search results ranked by relevancy. And you can display the particulars—author, title, and content—with ease. Further, since the index ID values correlate to the database ID values in this example, it would be easy to link the search results to the page where they could be found (assuming you fleshed out this example).

The only problematic aspect of this minimal approach is that it wouldn't be easy to display the specific content that matched the search term. The example data I used had a few paragraphs for each book. If I indexed an entire book, it'd be ridiculous to show the entire content in the search results.

As previously mentioned, the “highlight” feature of Elasticsearch does a wonderful job in this area, so it'd be great to use that.

To add found terms highlighting to the output, add a `highlight()` method call to the Elasticsearch query. Here's the code, which I'll explain subsequently:

```
$query = new Query;
$query->fields(['author', 'title'])
    ->from('books', 'book')
    ->highlight(["require_field_match" => false,
        "fields"=>["*"=>["fragment_size" => 150]]])
    ->query(['match' => ['_all' => $terms]]);
```

First, the `highlight()` method takes an array, used to customize the “highlight” property in the JSON request sent to Elasticsearch. Within that array, the “`require_field_match`” property is set to false, which is something you'll want to do when finding matches and returning highlights within multiple possible fields.

Next, the “`fields`” property is set, passed as array, to customize what fields are highlighted and how. The initial asterisk says to highlight the found term in every

1 Record(s) Found Searching for "moby"

Moby Dick by Herman Melville

... sperm whales. So that though *Moby* Dick had in a former year been seen, for example, on

Figure 13.16: Search results with highlighted terms in the browser.

field. The “fragment_size” changes the returned segment of highlighted code from the default size of 100 characters to 150.

This code now returns both the hits and the highlighted section of the hits. The highlighted snippets will be found in:

```
$results['hits'][0]['highlight']['field_name']
```

(In this case, the “field_name” would be “content” or “title” or “author”, as that’s where the search is finding the terms.)

My goal is to show these results, with the highlights, in a display as that shown in **Figure 13.16**.

To accomplish that, use the above code to customize `actionSearch()`. Next, fetch the number of results:

```
$total = $results['hits']['total'];
```

Finally, loop through the hits, assigning the data wanted into a new array:

```
foreach ($results['hits'] as $hit) {
    $id = $hit['_id'];
    $hits[$id]['title'] = $hit['fields']['title'][0];
    $hits[$id]['author'] = $hit['fields']['author'][0];
    $hits[$id]['highlight'] = $hit['highlight']['content'][0];
}
```

Passing all this to the view file, the method concludes with:

```
return $this->render('search', [
    'hits' => $hits,
    'total' => $total,
    'terms' => $terms
]);
```

And the view file does this:

```
<?php echo '<h2>' . $total . ' Record(s) Found Searching for "' .
    $terms . '"</h2>';
foreach ($hits as $hit) {
    echo '<div><h3>' . $hit['title'] . ' by ' . $hit['author'] . '</h3>';
    echo '<p>...' . $hit['highlight'] . '...</p>';
    echo '</div>';
}
?>
```

But that's not all...

There are two more ways the results can be improved. Instead of limiting the returned fragment to 150 characters, it should include all of the title and author but only 150 characters of the content:

```
$query = new Query;
$query->fields(['author', 'title'])
    ->from('books', 'book')
    ->highlight(["require_field_match" => false, "fields"=>[
        "content" => ["fragment_size" => 150],
        "title" => ["number_of_fragments" => 0],
        "author" => ["number_of_fragments" =>0]
    ])
    ->query(['match' => ['_all' => $terms]]);
```

By setting the “number_of_fragments” to 0, Elasticsearch won’t return a fragment; instead it returns the whole value.

The second issue is the previous code assumed the highlight would be found in the content, but it could be in the title or author, too. This does mean that the PHP code needs to be updated, as it can no longer assume that there will only be one highlight, or that it’ll be under “content”. Here’s that updated code, that goes within the `foreach`:

```
foreach ($hit['highlight'] as $field => $h) {
    $hits[$id]['highlight'][$field] = $h[0];
}
```

The loop goes through each highlight, finding the field and the highlight text. These are then assigned to a `$hits[$id]['highlight']` array.

To be clear, this is a replacement of:

```
$hits[$id]['highlight'] = $hit['highlight']['content'][0];
```

Now the view has to be updated to acknowledge an array of highlights:

1 Record(s) Found Searching for "moby"

Moby Dick by Herman Melville

Title: *Moby Dick*

Content: sperm whales. So that though *Moby Dick* had in a former year been seen, for example,

Figure 13.17: Search results with multiple highlighted fields.

```
foreach ($hit['highlight'] as $field => $h) {  
    echo '<p><strong>' . ucfirst($field) . '</strong>: ' . $h . '</p>';  
}
```

You can see the results in (Figure 13.17).

Chapter 14

JavaScript and jQuery

As Yii is used to create dynamic websites, being able to apply JavaScript to a Yii-based site is a critical skill. That is exactly the goal of this chapter, covering three core concepts:

- Adding raw JavaScript to a page (as opposed to jQuery)
- Client-side validation
- Implementing Ajax

The chapter concludes walkthroughs on several common needs.

Note that this chapter does assume comfort with JavaScript and jQuery. It's just impossible to try to teach either the JavaScript language or the jQuery JavaScript framework in this book, let alone in this chapter. If you aren't already comfortable with JavaScript, might I (selfishly) suggest you read my "[Modern JavaScript: Develop and Design](#)" book, which teaches JavaScript for beginners, and introduces the jQuery library.

Yii2 dramatically changed how JavaScript and jQuery are used within view files. Instead of tightly integrating JavaScript into views and widgets, Yii2 decouples JavaScript from them for better compartmentalization of application components. Consequently, using JavaScript and jQuery in Yii2 is much closer to how you use both in a non-Yii site.

In Chapter 19, “Extending Yii,” you’ll learn how to control whether or not jQuery—or other assets—are included in your applications.

Despite the fact that you need to know JavaScript and jQuery in order to make the most of this chapter, history would suggest some readers will continue through this chapter regardless. Thus, as a precaution, the chapter begins with a few fundamentals of JavaScript, jQuery, and Yii that everyone needs to understand.

First, *jQuery is JavaScript*. This should be obvious, but some developers don’t appreciate the significance of this fact. When you’re programming in jQuery, you’re

actually programming in JavaScript (just as when you’re using Yii, you’re programming in PHP). jQuery is extremely reliable and easy to use, which has a negative consequence: many people implement jQuery without actually knowing JavaScript. That is a problem. Before attempting to use jQuery, learn JavaScript, because jQuery is JavaScript!

Second, *you will inevitably have issues due to how browsers load the Document Object Model (DOM)*. This is common. The DOM provides a way for browsers to represent and interact with elements in a web page. But a browser does not have access to *any* page element until it has loaded *every* page element. That is a bit of an oversimplification, but programming as if that last sentence is exactly the case is most foolproof. This trips up JavaScript programmers that attempt to make immediate reference to DOM elements. The solution is to only reference DOM elements when the window’s contents have been loaded, or when jQuery’s “ready” event has occurred.

Third, *identify, install, and familiarize yourself with some good JavaScript debugging tools*. As JavaScript runs in the browser, you’ll need to use your browser debugging tools to identify and fix any problems. If you know JavaScript, you know this already.

The first thing to know for using JavaScript and jQuery in Yii is how to add JavaScript to a web page. As with any standard web page, there are two primary options:

- Link to an external file that contains the JavaScript code
- Place the JavaScript code directly in the page using SCRIPT tags

Just as I assume you’re already comfortable with JavaScript, I’ll also assume you know the arguments for and against both approaches. (Technically, there’s a third option: place the JavaScript inline within an HTML tag. This is not a recommended approach in modern websites, however, and isn’t demonstrated here.)

External JavaScript files are linked to a page using the SCRIPT tag:

```
<script src="/path/to/file.js"></script>
```

The contemporary approach is to link external files at the end of the HTML BODY, although some JavaScript libraries must be included in the HEAD.

If you need to include a JavaScript file on every page of your site, an option is to just add the reference to your layout file:

```
<script src="php echo Yii::$app-&gt;request-&gt;baseUrl;
?&gt;/path/to/file.js"&gt;&lt;/script&gt;</pre
```

Do be certain to use an *absolute reference* to the file, for reasons explained in Chapter 6, “Working with Views.”

Alternatively, you can use Yii’s `Html::jsFile()` method to create the entire HTML tag:

```
<?= Html::jsFile('/path/to/main.js') ?>
```

The end result is the same.

As a shorthand, you can use the special value `@web` to refer to the web root directory:

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

Sometimes, you’ll have external JavaScript files that should only be included on specific pages. In theory, you could just add the appropriate SCRIPT tag to the corresponding view files, but that’s less than ideal for a couple of reasons. For one, the final page will end up with SCRIPT tags in the middle of the page BODY, which is sloppy. Another reason why you don’t want to take this approach is that it gives you no vehicle for putting the script in the HTML HEAD, should that be necessary.

The better way to add external files to a page from within the view file is to use Yii’s `registerJsFile()` method, part of the view:

```
# views/foo/bar.php
<?php $this->registerJsFile('/path/to/file.js'); ?>
```

The `registerJsFile()` method works like `jsFile()` except that it can be called anywhere in a view file (or a controller), Yii will automatically include a link to the named JavaScript file in the complete rendered HTML. Further, if, for whatever reason, you register the same JavaScript file more than once, Yii will still only create a single SCRIPT tag for that file.

By default, Yii will link the registered script in the HTML HEAD. To change the destination, add a second argument to `registerJsFile()`. This argument should be an array, with a “position” element whose value is a constant that indicates the proper position in the HTML page for the JavaScript file reference:

- `View::POS_HEAD`, in the HEAD before the TITLE (the default)
- `View::POS_BEGIN`, at the beginning of the BODY
- `View::POS_END`, at the end of the BODY

To have the SCRIPT tag added at the end of the body, you would do this:

```
# views/foo/bar.php
<?php $this->registerJsFile('/path/to/file.js',
    ['position' => View::POS_END]); ?>
```

If your JavaScript file requires that another JavaScript file be included first, use the “depends” configuration option, passing along the class name of the JavaScript asset that must be loaded first:

```
# views/foo/bar.php
<?php $this->registerJsFile('/path/to/file.js',
    ['position' => View::POS_END,
     ['depends' => [\yii\web\JqueryAsset::className()]]); ?>
```

Note that “depends” requires an array as its value. The above example states that **file.js** depends upon jQuery, so Yii will only include the former after it includes the latter.

When you have short snippets of JavaScript code, or when the code only pertains to a single file, it’s common to write that code directly between the HTML SCRIPT tags. Again, you *could* do this in your view files:

```
<script>
/* Actual JavaScript code. */
</script>
```

You could also use the Yii `Html::script()` method to create the SCRIPT tag for you:

```
<?php echo Html::script('/* Actual JavaScript code. */'); ?>
```

You *could* add SCRIPT tags in either of those ways, but there’s a better approach: the view’s `registerJs()` method. This is the companion to `registerJsFile()`, but instead of linking to an external JavaScript file, it’s used to add JavaScript code directly to the page.

The method’s first argument is the JavaScript code itself. This code should not include the SCRIPT tags themselves.

```
<?php $this->registerJs("alert('Testing');");
?>
```

That code will have Yii insert the following into the page:

```
<script>
  alert('Testing');
</script>
```

As you can see in that example, the combination of PHP and JavaScript can easily lead to syntax errors. You should use one set of quotation types to *encapsulate* the JavaScript (passed to `registerJs()`) and another type *within* the JavaScript. Also be certain to terminate JavaScript commands with semicolons, and terminate the PHP command, too. If the JavaScript you write doesn't work, start by confirming that the resulting JavaScript code (in the browser's source) is syntactically correct.

As with `registerJsFile()`, `registerJs()` takes another argument to indicate where, in the HTML page, the JavaScript should be added:

- `View::POS_HEAD`, in the HEAD before the TITLE
- `View::POS_BEGIN`, at the beginning of the BODY
- `View::POS_END`, at the end of the BODY
- `View::POS_LOAD`, within a `window.onload` event handler
- `View::POS_READY`, within a jQuery "ready" event handler (the default)

The two additional options are necessary because of the way the browser loads the DOM. If you are using jQuery and you want to execute some JavaScript when the document is ready, use `View::POS_READY`, the default. It's slightly faster than the standard JavaScript `window.onload` option. If you're not using jQuery, then use `View::POS_LOAD`.

Unlike `registerJsFile()`, the position is provided as a second argument by itself, not as an array with a "position" element:

```
# views/foo/bar.php
<?php $this->registerJs('/* Actual JavaScript code. */',
  View::POS_LOAD); ?>
```

The `registerJs()` method takes a third argument: is a unique identifier you can give to the code snippet. One of the benefits of providing a unique identifier is that the component will manage the code bits so that even if the same code (by identifier) is registered multiple times, it will still only be placed on the page once. If you do not provide an identifier, the entire JavaScript code block is used as the identifier.

```
# views/foo/bar.php
<?php $this->registerJs('/* Actual JavaScript code. */',
  View::POS_LOAD, 'unique-identifier'); ?>
```

One of the absolutely most critical uses of JavaScript in today's websites is for form validation. This is no less true when using Yii, although Yii can do much of the work

for you, as is the case with so many things. Let's quickly look at how JavaScript is used with forms in Yii, specifically when using the `ActiveForm` widget.

When you create a new `ActiveForm` widget instance, you can configure how it behaves, as is the case with most widgets. Configuration is performed by passing an array of name=>value pairs as an argument to the `widget()` or `begin()` method (see Chapter 12, “[Working with Widgets](#)”).

To enable client-side JavaScript form validation, set the “enableClientValidation” property to true:

```
# views/page/_form.php
<?php $form = ActiveForm::begin(
    ['enableClientValidation' => true]
); ?>
```

By setting this property to true, Yii will add the appropriate JavaScript to the page to perform client-side validation. (This is the default behavior, so you don't have to manually set it; use false for the value to disable it.)

The validation will use the same rules as defined in the associated model, assuming that the validator is supported on the JavaScript side. Most of the core validators are, including:

- `boolean`
- `captcha`
- `compare`
- `email`
- `double`
- `int`
- `required`
- `string`
- `url`

To be perfectly clear, this means that if you have an `email` attribute in a model that's associated with an “email” form input and that has an “email” validator in the model's rules, that validation can be performed client-side, too. On the other hand, attributes that have the “default”, “date”, “exist”, and other validators not listed above applied to them cannot be validated in the client.

{TIP} You may not be able to see, or appreciate, the effects of client-side validation until you configure the validation options as well. I'll explain those in just a couple of pages.

Not only will your existing model validation rules be used when you enable client-side validation, but so will the existing error messages for reporting problems. If

you customize an error message in a validation rule, the client-side validation will use that when the data does not pass.

If the user does not have JavaScript enabled, then client-side validation cannot occur (of course). In those cases, the server-side validation will still be used in the controller that handles the form submission. In fact, for security purposes, your controllers should always be written to perform server-side validation. Client-side validation is a convenience to the user; not a security technique.

{WARNING} Always use server-side validation!

Another way to validate a form using JavaScript is via Ajax. Ajax validation makes an actual request of the server to validate the form data. For this reason, Ajax validation can be used to validate form elements that cannot be validated via client-side JavaScript alone, such as:

- The availability of a username
- Confirming that a value exists in a related table
- If a value is unique in the database

A limit to Ajax validation is that it cannot be used to validate uploaded files. This is a restriction on Ajax in general, not in Yii.

To enable Ajax validation, set “enableAjaxValidation” to true when configuring the ActiveForm widget:

```
# views/page/_form.php
<?php $form = ActiveForm::begin(
    ['enableAjaxValidation' => true]
); ?>
```

Unlike “enableClientValidation”, Ajax validation is not enabled by default because there are two sides to Ajax, of course: the client-side JavaScript and the server-side code that handles the JavaScript request. For the Ajax validation to work, you must create the appropriate server-side code, too. That is easily done, though, using this code:

```
# controllers/PageController.php
use yii\web\Response;
use yii\widgets\ActiveForm;
public function actionCreate() {
    $model=new Page;
    if (Yii::$app->request->isAjax &&
        $model->load(Yii::$app->request->post()))
        Yii::$app->response->format = Response::FORMAT_JSON;
```

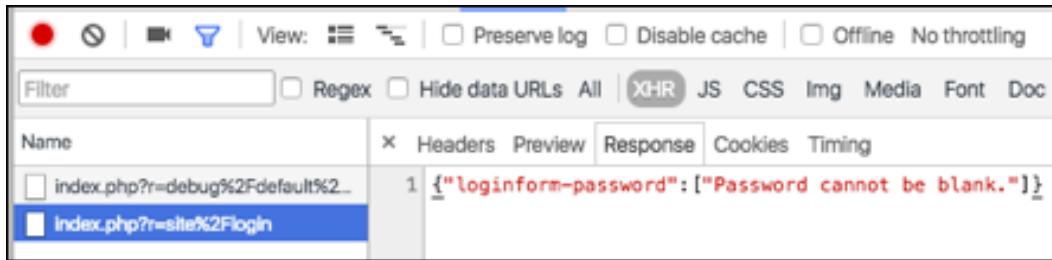


Figure 14.1: The JSON reporting for the validation of a user.

```

        return ActiveForm::validate($model);
    }
    // Rest of the action.
}

```

The code first checks if this is an Ajax request and if the model data can be loaded. If so, the controller sets the response format to JSON (as opposed to HTML) and prints out the result returned by calling the `ActiveForm::validate()` method. The `validate()` method returns the results as JSON data (**Figure 14.1**), so that's what the JavaScript in the browser will receive.

In most situations, the entire form does not need to be validated via Ajax, just specific form elements. In those cases, enable Ajax validation on the applicable form element:

```
echo $form->field($model, 'username', ['enableAjaxValidation' => true]);
```

Assuming, in this example, the `username` attribute has a “unique” validation on it, the same code used above will work for the server-side validation.

With Ajax validation enabled, the browser will first perform basic client-side validation and, if that passes, then perform the Ajax request.

The `ActiveForm` widget is very powerful and easy to use. But there are a few more configuration options which you ought to be familiar.

For example, you can identify a different URL to use for Ajax validation purposes by assigning a value to “validationUrl” (by default, the validation URL is the same as the form’s “action” attribute). Or you can change when validation is performed. By default, validation is performed when any form element’s value changes, but you can set the validation to occur upon submission instead:

```
# views/page/_form.php
<?php $form = ActiveForm::begin([
    'validationUrl' => ['page', 'create'],
    'validateOnSubmit' => true,
```

```
'validateOnChange' => false,  
]); ?>
```

As another example, you can change the CSS classes associated with validation by changing the corresponding property:

- `errorSummaryCssClass`, which styles the container in which the error occurred (defaults to “error-summary”)
- `errorCssClass`, which styles the error message itself (defaults to “has-error”)
- `requiredCssClass` (defaults to “required”)
- `successCssClass`, which styles the container to indicate success (defaults to “has-success”)

For all the possibilities, see the [ActiveForm](#) documentation in the Yii API. Note that these settings can impact both types of client-side validation: only JavaScript or also Ajax.

Ajax is one of the reasons why JavaScript is so critical to today’s websites. Ajax has been around for more than a decade now, and the features Ajax can add to a website are pretty much expected by most users anymore (whether they know it or not). I’ve already mentioned Ajax once in this chapter—for validation purposes, and assume you do know the fundamentals of this vital technology. But let’s quickly look at a couple more ways to implement Ajax in a Yii-based site.

Ajax blends the two sides of web development: the client-side (aka, the browser) and the server-side. When developing Ajax processes, I like to start on the server-side of things so that I know what to do and expect on the client-side.

Server-side Ajax resources in Yii are represented as controller actions, just like regular web pages. But there are a few major differences between an Ajax action and a standard one. The first key difference is that an Ajax response is almost always made up of the most minimal amount of data:

- Short, plain text
- A snippet of HTML
- More complex data as JSON
- More complex data as XML

For this reason, Ajax controller actions almost never use the `render()` method to create the output. Instead, there are two common approaches:

1. Directly print the desired output from the controller
2. Use `renderPartial()` to have a view represent the output (but without the primary layout file)

{NOTE} In Chapter 16, “[Leaving the Browser](#),” I present a situation in which you would use `render()`: You could output XML, with the primary layout file representing the beginning and end of the XML document.

Ajax actions are also different in that they’re not meant to be accessed directly by users in the browser. It’s not a big deal, normally, but at the very least, the user will have an unappealing, if not confusing, experience if she ends up directly requesting an Ajax resource. There are a couple of ways in Yii that you can limit access to an action to an Ajax request. One option is to set a filter. However, there is no built-in “Ajax-only” filter in Yii 2 (yet), so you’d have to write your own.

Alternatively, you can check for an Ajax request within an action method via `Yii::$app->request->isAjax`.

Some actions are written to be accessed via Ajax and non-Ajax alike, reacting slightly differently in each case. In such situations, you can test if an Ajax request is being made via the “request” application component:

```
# controllers/SomeController.php
public function actionSomething() {
    // React different based upon Ajax request status:
    if (Yii::$app->request->isAjax) {
        // Do things this way.
    } else {
        // Do things this other way.
    }
}
```

In situations where the same action may be used by Ajax and non-Ajax requests, for the Ajax portion, you’ll also need to have the method invoke `return` in order to terminate the output immediately. This prevents the non-Ajax output from being added to the result:

```
# controllers/PageController.php
public function actionCreate() {
    $model=new Page;
    if (Yii::$app->request->isAjax &&
        $model->load(Yii::$app->request->post())) {
        Yii::$app->response->format = Response::FORMAT_JSON;
        return ActiveForm::validate($model);
    }
    // Rest of the action.
}
```

The last thing to keep in mind with Ajax processes is to set the proper controller permissions. It’s not obvious to many developers, but when an Ajax request is

performed, it's as if the user requested the resource directly. In other words, an Ajax request made from a user's browser is still being made by that user. This means that Yii's permissions apply to Ajax requests just the same as they do to other requests. Keep this in mind when creating actions and setting permissions.

As a rule of thumb, if the "foo" page makes an Ajax request of the "bar" action, then, logically, both "foo" and "bar" need to have the same permissions in the controller. Still, very rarely will an Ajax action need protection at all, so you can normally make them publicly accessible. Do so if you'd rather not run the risk of having Ajax request failures due to permission issues.

Another approach for the Ajax permissions is to create a controller explicitly for all Ajax requests. That controller would have open permissions, like the "site" controller does. Let's look at that idea in more detail.

In order to test Ajax processes, at least within the confines of this book, it may help to have a couple of test Ajax processes for experimentation. To be clear, I'm talking about making sample PHP resources that client-side JavaScript can request. Let's create a new controller for this purpose:

```
# controllers/AjaxController.php
<?php
class AjaxController extends Controller {
}
```

Within that controller, define three actions:

- One that returns (or prints) a simple string
- One that returns some HTML
- One that returns dynamic HTML (in theory)

In just a few pages, another action that returns data in JSON format will be added. Note that all of these actions as defined will be rather static, but they are all easy enough to update to being truly dynamic in a real-world site. Also, there are no filters in this controller, such as the access control filter, so every action will be executable by any user. No check will be made for Ajax requests either, so you can test these directly in your browser.

The first action only returns a simple text message:

```
# controllers/AjaxController.php
public function actionSimple() {
    echo 'true';
}
```

This simple action might be used to verify that an username is available or that an email address has not yet been registered. In a real-world site, the action would

perform the necessary logic, and then print “true” or “false” accordingly. Note that the Ajax action must print *strings*, not Booleans.

Next, there’s an action that returns a bit of HTML. The premise is the same, but the text is actually HTML:

```
# controllers/AjaxController.php
public function actionHtml() {
    echo '<p>Lorem ipsum <em>dolor</em>...</p>';
}
```

Finally, the third action returns more dynamic HTML, using a variable and a view file:

```
# controllers/AjaxController.php
public function actionDynamicHtml() {
    // Dynamic data:
    $data = array(
        'title'=>'Dynamic!',
        'content'=>'<p>Lorem ipsum <em>dolor</em>...</p>'
    );
    // Render the page:
    return $this->renderPartial('dynamicHtml', array('data'=>$data));
}
```

Obviously, in the real-world, the data itself might be pulled from the database.

The view file for the dynamic action looks like this:

```
# view/ajax/dynamic.php
<?php
/* @var $this yii\web\View */
/* @var $data array */
?>
<article>
    <h3><?php echo $data['title']; ?></h3>
    <div><?php echo $data['content']; ?></div>
</article>
```

The received `$data` array’s pieces are placed within a context of HTML (**Figure 14.2**). Changing the data values in the controller therefore changes the output.

Now that three sample Ajax processes have been defined, you can test them in your browser by going to:

- `ajax/simple`

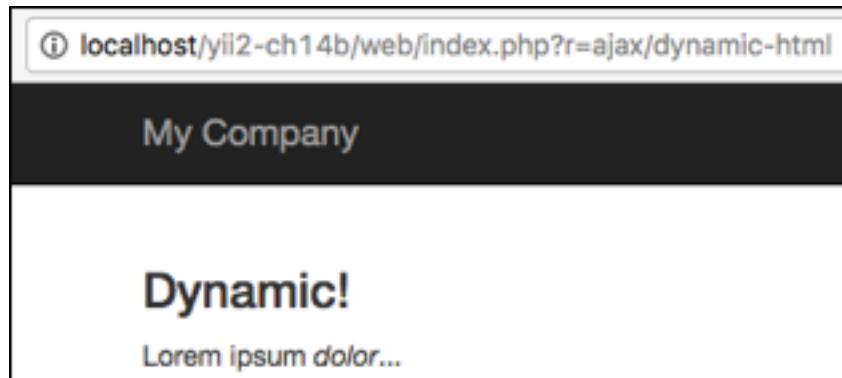


Figure 14.2: The “dynamic” HTML response.

- ajax/html
- ajax/dynamic-html

{TIP} I always recommend testing server-side Ajax resources directly first, to confirm they are working, before connecting them to the JavaScript.

Once you have the Ajax server-side actions working as you would hope, it’s time to turn to the JavaScript. There are four ways you can perform an Ajax request using Yii:

- Via an immediate Ajax request
- Via a link
- Via a button
- By tying an Ajax request to another DOM element

Unlike in Yii 1, where JavaScript was more directly tied to widgets and HTML elements, in Yii 2 you add JavaScript as you would if not using a framework. Assuming you know JavaScript, it’s really straightforward.

To start, let’s look at how to make a direct and immediate Ajax call. This is normally accomplished in Yii applications via the jQuery `ajax()` method. It takes two arguments: the URL to send the request to and an array of options. As soon as this method is invoked, the Ajax request is begun.

{NOTE} You could, of course, use raw JavaScript to perform the Ajax request, but if your site is already using jQuery, it makes sense to invoke the jQuery method.

The syntax is:

```
// views/foo/bar.php
jQuery.ajax('/index.php/ajax/simple', {
    'dataType':'text',
    'method':'get',
    'success':function(result) { alert(result); },
    'cache':false,
    'data':jQuery(this).parents("form").serialize()
});
```

That method call creates the JavaScript required to perform an Ajax request. The JavaScript itself uses the jQuery `ajax()` method. For the options, you can start with the possible settings outlined for the jQuery `ajax()` method in the [jQuery documentation](#).

The most important of the configuration options are:

- “`data`”, which is data to be sent as part of the request
- “`dataType`”, the type of data expected in return (“`text`”, “`html`”, “`json`”, etc.)
- “`method`”, the request, or method, type (i.e., “`get`” or “`post`”)
- “`success`”, the JavaScript function to call upon a successful request being made

There are a couple of things to notice there. First, for the URL, use Yii to create a proper URL (e.g., using `Url::to()`; don’t hardcode it as in the previous example). Not using an accurate URL is a common cause of problems. Second, the “`success`” item takes a JavaScript function that will be invoked when the request is successfully completed. This can be the name of an existing JavaScript function, or an anonymous function as in the above. Per how jQuery’s `ajax()` method works, this function can be written to take up to three arguments, the first being the actual response.

With that code, when the page is loaded, the Ajax request will be made and the response alerted. Merely change the URL being requested to get different responses (**Figure 14.3**).

Of course, you don’t want to just alert the Ajax response. Normally, you’ll update the DOM in some way, perhaps based upon what the response was:

```
// views/foo/bar.php
jQuery.ajax('/index.php/ajax/simple', {
    'dataType':'text',
    'method':'get',
    'success':function(result) {
        if (result === "true") {
            $("#response").text("The username is available.");
        } else {
```

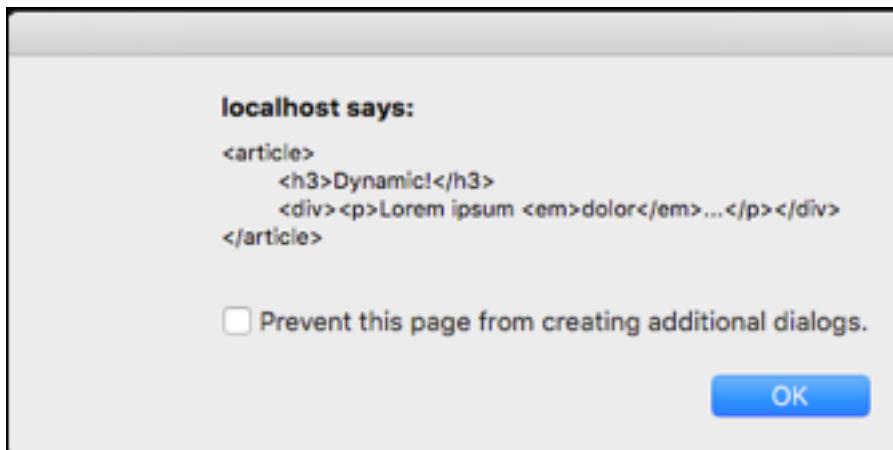


Figure 14.3: The response from the HTML action.

```
        $("#response").text("The username has been taken.");
    },
    'cache':false,
    'data':jQuery(this).parents("form").serialize()
});
```

(A quick reminder: I assume you're comfortable with JavaScript and jQuery. If not, learn them now!)

Or, you may add the response itself to the page:

```
// views/foo/bar.php
jQuery.ajax('/index.php/ajax/simple', {
    'dataType':'text',
    'method':'get',
    'success':function(result) {
        $("#destination").html(result)
    },
    'cache':false,
    'data':jQuery(this).parents("form").serialize()
});
```

Moving beyond this simple approach, you could create HTML elements (using Yii or raw HTML) and then code jQuery event handlers on them. Simply be certain to give your elements ID value, and take it from there as if you weren't using a framework. You'll see an example of this in the next section.

The three Ajax controller actions defined offer a range of possibilities, but there's one more example to implement. When you need to return more complex data from

the server to the client, plain-text and HTML formats are insufficient. Originally, eXtensible Markup Language (XML) was used as the data format (“Ajax” either is or is not an acronym for “Asynchronous JavaScript and XML”, depending upon whom you ask). These days, JSON (JavaScript Object Notation) is the norm. The JSON format is compact, resulting in faster response times, and readily usable by JavaScript in the client.

The downside to JSON is that its syntax is particular and can be difficult to get right. Fortunately, Yii can output JSON directly and easily by setting the response format, as you’ve already seen. You can return almost any data type and Yii will output proper JSON. Let’s add another demo action to the “ajax” controller:

```
# controllers/AjaxController.php
public function actionJson() {
    $data = [
        'title'=>'Dynamic!',
        'content'=>'<p>Lorem ipsum <em>dolor</em>...</p>'
    ];
    Yii::$app->response->format = Response::FORMAT_JSON;
    return $data;
}
```

And here’s how that might be used in the view file:

```
<h3 id="updateTitle"></h3>
<div id="updateContent"></div>
<button id="click-btn">Click Me!</button>
<?php
use yii\web\View;
$this->registerJs("$('#click-btn').click(function() {
    jQuery.ajax('/index.php/ajax/json', {
        'dataType':'json',
        'method':'get',
        'success':function(result) {
            $('#updateTitle').html(result.title);
            $('#updateContent').html(result.content);
        },
        'cache':false,
    });
});", \yii\web\View::POS_READY);
?>
```

Figures 14.4 and 14.59 show this in action.

This chapter concludes with a few common needs and points of confusion when it comes to JavaScript and jQuery in Yii. Again, because Yii2 has mostly decou-

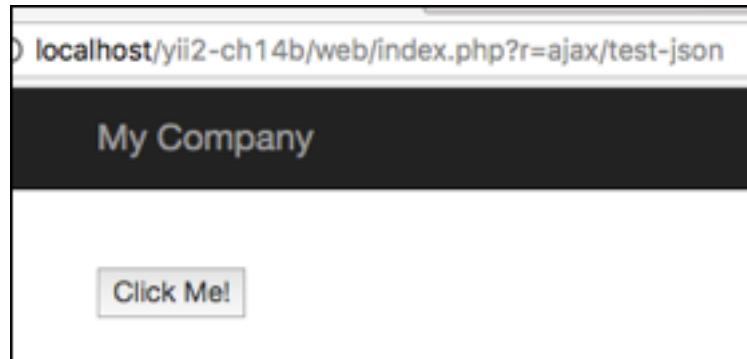


Figure 14.4: The page when the user first sees it.

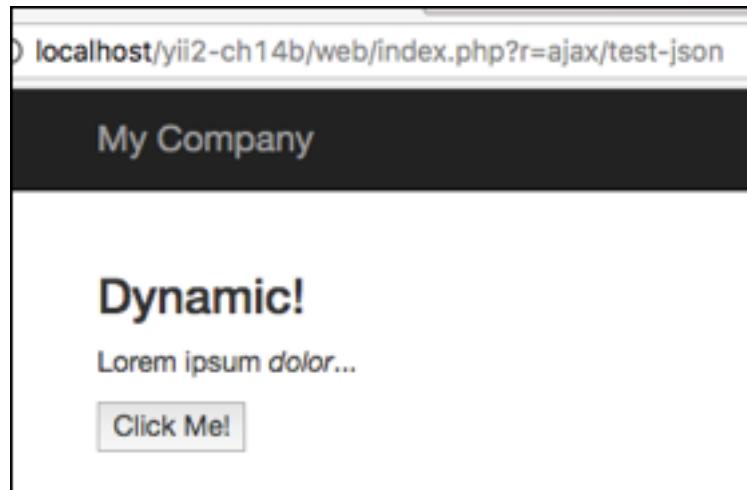


Figure 14.5: The same page after the user has clicked the button.

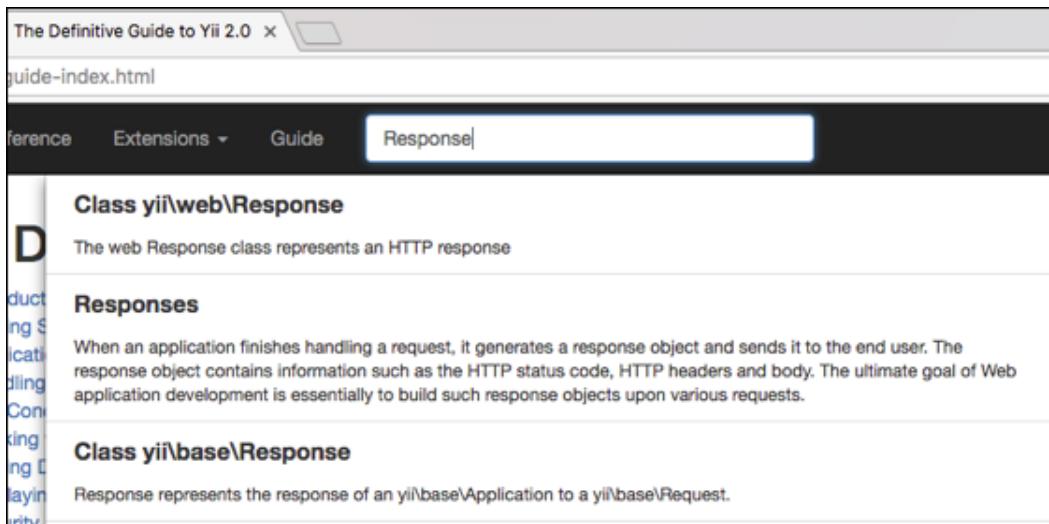


Figure 14.6: Autocomplete functionality in the Yii class reference.

pled JavaScript logic from view components, it's now mostly a matter of just using JavaScript as you'd otherwise use JavaScript.

If you want to set the focus on a particular form element (e.g., have the user's cursor begin in that element), there are a couple of options. The first is available if you're using HTML5: set the "autofocus" property on the element. Here's how that would look in straight-up HTML:

```
<input type="email" name="email" autofocus>
```

When you're using Yii to create form elements, just add this as an additional HTML attribute:

```
<?= $form->field($model, 'attribute')  
    ->textInput(['autofocus' => 'autofocus']) ?>
```

The HTML5 "autofocus" property is supported by most modern browsers, but not in Internet Explorer until version 10.

A common use of JavaScript and Ajax is *autocomplete* functionality. First popularized as Google's Suggest tool, autocomplete is now a web standard, including in the [Yii class reference](#) (**Figure 14.6**).

Thanks to the jQuery UI autocomplete widget, and the Yii `yii\jui\AutoComplete` class, it's pretty easy to implement autocomplete on your website. As an example of this, let's create the ability to autocomplete books by title (**Figure 14.7**).

First, you need to install the jQuery UI extension:

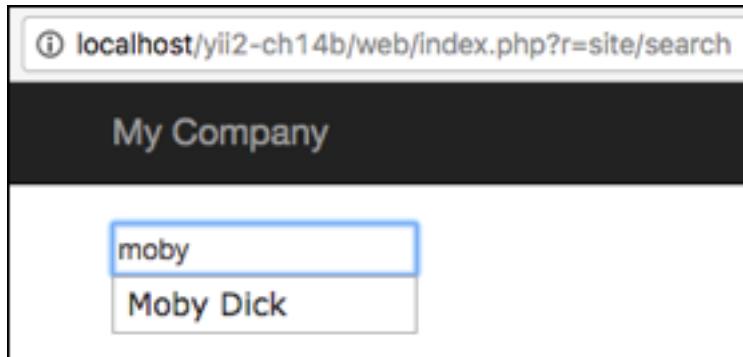


Figure 14.7: Autocompletion of page titles.

```
composer require --prefer-dist yiisoft/yii2-jui
```

That done, in the view file, create an instance of the `JuiAutoComplete` widget:

```
1 <?php
2 use yii\helpers\Url;
3 use yii\jui\AutoComplete;
4 echo AutoComplete::widget([
5     'model' => $model,
6     'attribute' => 'title',
7     'clientOptions' => [
8         'minLength'=>'2',
9         'type'=>'get',
10        'source'=>Url::to(['ajax/get-page-titles']),
11        'select'=>'function(event, ui) {
12            $("#selectedTitle").text(ui.item.value);
13        }'
14    ],
15 ]);
16 ?>
17 <span id="selectedTitle"></span>
```

This widget will, by default, create a text input with a “name” value based upon the model and attribute, in this case “`Books[title]`”.

The “clientOptions” section is where you configure the jQuery UI options, found in the [jQuery UI documentation for the autocomplete widget](#). In those options, I’ve set the “`minLength`” to 2, so that no results are returned until at least 2 characters are entered. When using dynamic data returned by an Ajax request, the “`source`” value needs to point to the controller action that will return the results (line 10).

I’ve also created a function that will be called when a selection is made from the list of options. The anonymous function takes two arguments: an event and an object.

This object is conventionally, in jQuery UI, called “ui”, and its `item` property will represent the selected item. To make it obvious which value was selected, a SPAN is updated upon selection.

With the widget in place in the view, it’s time to create the controller action that provides the source data for the widget. Per the widget configuration, the source URL is “`ajax/get-page-titles`”, which means that there needs to be a “`getPageTitles`” action in the “`ajax`” controller. This action should use the submitted term—what the user typed—and return an array of values in JSON format:

```
public function actionGetPageTitles($term) {
    $data = [];
    if (isset($term)) {
        $data = Yii::$app->db->createCommand('SELECT id,
            title AS value FROM books
            WHERE title LIKE :terms')
            ->bindValue(':terms', '%' . $term . '%')
            ->queryAll();
    }
    Yii::$app->response->format = Response::FORMAT_JSON;
    return $data;
}
```

The specific query is supposed to fetch the book ID and title for every book whose title is similar to the provided input. To accomplish that, I’m using Data Access Objects (DAO), explained in Chapter 8, “[Working with Databases](#).“ I’ve chosen to make the `LIKE` condition extremely flexible (i.e., `LIKE %term%`), but you could change it to just `LIKE term%` to be less so. The jQuery autocomplete widget will provide what the user typed as “`term`”, so that’s available in `$_GET['term']` or in the function parameter `$terms`.

Finally, notice that I’ve chosen to select the book title aliased (in the query) as “`value`”. This makes it easy to use in the generated drop-down list of autocomplete matches. This is also why the “`select`” JavaScript function in the widget refers to `ui.item.value`. If the book titles were selected as “`title`”, you would also have to configure how the matches are rendered by jQuery UI.

And that’s enough to implement autocomplete in a Yii-based site. To properly use the selected value, just change the contents of the “`select`” function to suit your needs.

If you have any problems in implementing this, begin by confirming the results of your Ajax request, as that’s the most likely cause of problems. Also familiarize yourself with the jQuery UI autocomplete widget, as the `JuiAutoComplete` class is just a wrapper to it.

Part III

Advanced Topics

Chapter 15

Internationalization

Increased adoption of *internationalization*, commonly abbreviated *i18n*, is an acknowledgement that the World Wide Web is indeed global. A website is available to anyone anywhere in the world, as long as they have a browser connected to the Internet, including on a mobile device. To support as wide of an audience as possible, your site should embrace internationalization.

In this chapter, you'll learn what i18n is if you're not already familiar with it, and how you implement i18n in a Yii-based site.

{TIP} The abbreviation “i18n” simply stands for the initial “i”, and terminating “n”, plus the 18 letters that come between them.

Before getting into the details, know that Yii uses the [PHP intl extension](#) to do much of the work. The intl extension in turn uses the [ICU library](#). For best results, make sure you have the PHP intl extension installed and version 49 or greater of the ICU library.

Internationalization is the act of writing software so that it can adapt to the different languages and customs of the software's various users. The most obvious example of customization for the user is the language used by the site. By applying internationalization, a site could present all of its navigation and interface items in English for some users, in French for others, in Russian for others, and so on. Behind the scenes, the core functionality would be the same, but the user experience is greatly enhanced.

The differences between any two languages can vary in many ways:

- The characters used (Latin-based languages share many common characters, Cyrillic languages use entirely different ones)
- The characters used to represent numbers specifically
- The direction in which words and sentences are written (left-to-right, right-to-left, or even top-down)

- How words are capitalized
- How words are sorted (as in an alphabetical list)

{TIP} In order to support any possible language, be certain that the web page and the database use UTF-8 encoding.

Along with the primary issue of the language used, two other topics are commonly associated with internationalization:

- Cultural habits
- Writing conventions

Cultural differences range from the very simple to the highly complex (and sensitive). Simple differences include the:

- Formatting of telephone numbers
- Formatting of addresses and postal codes
- Currency used
- Weights and measures used

If you really want to focus on cultural differences, you'll get into issues such as the significance given to certain colors, images, words, and so forth. For example, in the Western world, brides wear the color white, but in China and parts of Africa, white is a mourning color. If you're creating a truly international site, those are the kinds of things you'll want to get right. That being said, the high end of cultural issues are well beyond the scope of this chapter.

Next, there's the issue of formatting conventions, which is different than the language issue. The most obvious examples are how one writes dates, times, and numbers.

With dates, the issue is most problematic when using two-digit representations: 08/02/06. In the United States, that would be read as August 2nd, 2006. In Europe, that would be read as February 8th, 2006. It could also be interpreted as February 6th, 2008. Until everyone agrees on the proper, standardized way of representing dates—which will never happen, you may want to have your site adjust accordingly.

With numbers, the differences are mostly a matter of what character is used for separating thousands and for separating decimals. Should it be **1,000.78** or **1.000,78**?

Internationalization starts with the biggest, global differences, such as the character set and language used. This is further customized via *localization* (l10n), which takes into account *locale* issues. A user's locale consists of her preferred language, region, and sometimes other cultural preferences. When you install a new operating system for the first time, and it asks you about your physical location, preferred language (e.g., English US, English UK, etc.), time zone, and so forth; those answers all go into your *locale*.

With an understanding of i18n in place, the next issue is: should you use internationalization? As with almost everything, the answer is “it depends”. If you’re creating a site for a local restaurant, you probably don’t need to worry about internationalization at all. An exception would be if there’s a large population in the area that speaks another language or represents a significantly different culture. On the other hand, if you’re creating a site that you intend to be a global resource or e-commerce marketplace, embracing some internationalization would certainly help.

To implement internationalization in a Yii-based site, you’ll need to use a combination of the framework and some of your own logic and effort. Yii itself provides:

- Locale data
- A tool for displaying text in different languages
- Locale-specific date, time, and number formatting

Let’s look at these features of Yii in more detail.

{NOTE} You’ll need to add your own logic to your site to display dates and times in the user’s time zone.

To use internationalization, you have to establish the locales involved. Specifically, you must set two:

- The application’s (i.e., the language in which you designed and wrote the application)
- The user’s

The Yii documentation refers to these as the *source* and *target* languages, accordingly.

For a simple example, say you create a site, and your locale is English (US). For someone in the United Kingdom, you may want to present the site customized to her locale, English (UK).

{NOTE} If the application and the user share a single locale, no internationalization is necessary, as no changes are required.

Locales can be set by assigning values to two properties of the “application” object:

- `sourceLanguage` is the locale of the application (i.e., the “from” language)
- `language` is the user’s locale (i.e., the “to” or destination language)

As these are public, writable properties of the “application object, you can assign values to them in your configuration file. Locale values are represented as *LanguageID_RegionID*, with the language ID being two lowercase letters and the region ID being two uppercase letters. For example, the default locale is *en_US*, which stands for English in the United States.

{NEW} In Yii 2, locales are formatted using two uppercase letters for the region ID. In Yii 1, the entire locale is lowercase.

```
# config/main.php
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'sourceLanguage' => 'en_US',
    'language' => 'en_UK',
    // Other stuff.
];
```

English (US) is the default, so you only need to provide a value if you’ll be using a locale other than the default. Second, it’s not often that you’d set the target locale in the primary configuration file (i.e., identify two separate locales on a site-wide basis). Normally you’ll want to set the user’s locale dynamically:

```
\Yii::$app->language = 'en_UK';
```

How you determine the user’s locale is covered next.

{TIP} The locale represents more than just the user’s preferred language, but the term is commonly used to refer to the user’s language.

To identify the user’s preferred locale, you’ll need to get that information from the user. This can be done either overtly or secretly. The overt option is to present the user with an interface element through which he can set his locale. This might be a drop-down menu of languages or a series of flag icons. When the user makes a selection in the drop-down menu, or clicks on a flag icon, the site would then set the locale to that selection and update the page to that locale.

{TIP} When using the drop-down menu route, be sure the options are in the native languages. Showing an English user the word “English” written in Chinese characters will do little good.

A second option is to get the user’s preferred locale from the browser itself. This value is available in the `preferredLanguage` property of the `yii\web\Request` object:

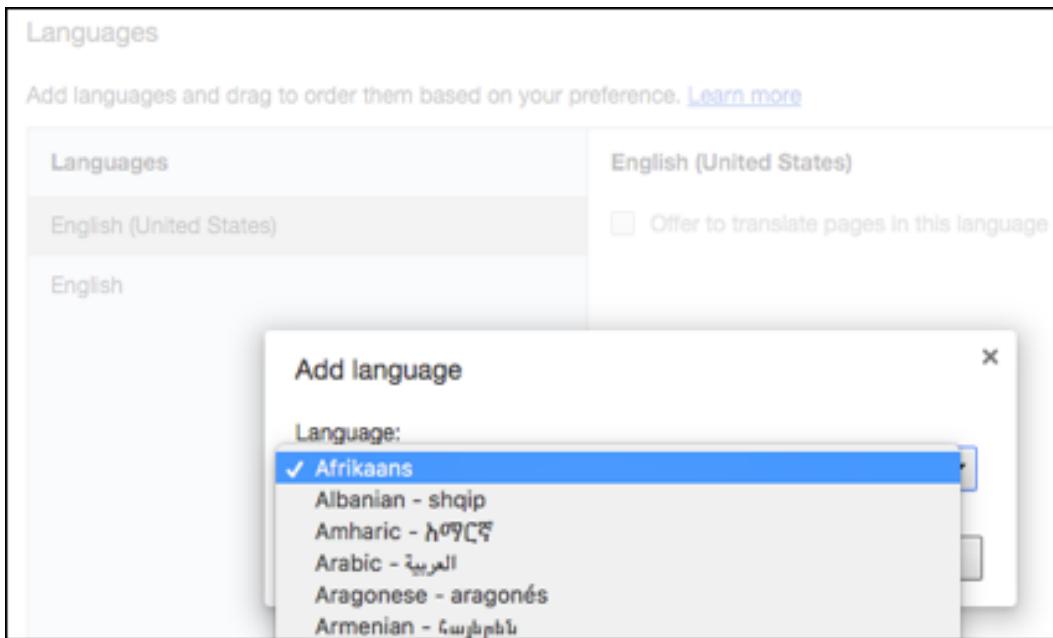


Figure 15.1: Chrome’s interface for managing languages.

```
if (!empty(\Yii::$app->request->preferredLanguage)) {
    \Yii::$app->language = \Yii::$app->request->preferredLanguage;
}
```

The browser has this value by either getting it from the computer itself, or by having it be set within the browser’s preferences (**Figure 15.1**). If set, the browser may provide this to sites as part of the request (in an “Accept-Language” header), which is why the value would be found in the `yii\web\Request` object in Yii.

{NOTE} The Yii framework includes locale data for almost every language and region, thanks to the [Common Locale Data Repository](#) (CLDR).

The problem with using the user’s provided locale (i.e., that provided by the browser) is that it assumes your site is setup to handle any possible language value. Of course, that will never be the case. I would recommend that you setup your system so that it checks the user’s preferred language, and uses that if possible. If the preferred language is not supported, you would then prompt the user to select her preferred language from a list of supported ones.

Moreover, there is a strong argument to supporting both methods of user locale selection regardless: both overt and implied. For example, I might be on vacation in Greece and stop in at an Internet café. The locale on those computers could logically be `el_GR`, which is what the browser would provide to a site I visit. In



Figure 15.2: The Spanish Google interface.

order for me to use the site, however, I'd need a way to manually set the locale to my preferred *en_US*.

With the target (user) locale set, you can now perform internationalization: customize the experience to the user.

The biggest role i18n plays is providing language-appropriate text. For example, standard Google has a “Google Search” button, but Spanish Google has “Buscar con Google” (**Figure 15.2**).

The Yii Guide refers to the ability to changing text based upon the language in use as “translation”. However, I tend to think of translating as an active, thoughtful act, whereas what you’ll do in Yii is really string replacement. That being said, certainly “translation” is a pithier label, and I’ll continue to use it in this chapter to be consistent with the Yii documentation.

Translating is a two-step process:

1. Define in your application the translations for various words and strings.
2. Invoke a specific method to have Yii retrieve the string in the desired language.

I'll explain these steps over the next several pages.

The reason I don't care for the term “translation” is that Yii is not actively translating anything. What *is* happening is that you define the strings you'll need to use in the languages you want to support. Then, when the time comes to present that string, instead of printing the string itself, you invoke a function that retrieves the correct version of the saved string in the destination language.

The first step, then, is defining the “translations”. The translations are stored in one of three places:

- A PHP file
- The database
- [GNU gettext](#) files

These are known as *message sources*. Behind-the-scenes, the `yii\i18n\MessageSource` class defines the necessary storage and retrieval functionality. The three listed storage options all use classes that extend `MessageSource`. You can also extend it yourself to create your own message source. For simplicity sake here, let's use a PHP file.

The name and location of the PHP file depends upon two factors:

1. The locale ID
2. A category name

The locale ID is the Yii format version, such as `en_US` or `el_GR`.

By default, PHP message files go in `@app/messages/LocaleID`. If your application is going to support three languages, besides the source/application language, then you'll need to have three subdirectories within `@app/messages`.

{NOTE} You do not need to provide a message source for the application's primary language, as no translations are required when the primary and target languages are the same.

Within each locale directory, you'll have one PHP file for each message category. Categories are simply an organizational scheme that makes creating, maintaining, and using translations easier and faster. Instead of having, say, 200 strings defined in one file, those 200 can be broken up into 5 or 10 categories.

The categories are of your own creation, organized as you see fit. As a simple starting point, you might want a category for each model in your application, plus a category for the application as a whole (the most universal category). In a CMS example, that would mean you'd create these files:

- `app.php`
- `comment.php`
- `file.php`
- `page.php`
- `user.php`

Understand that you'll need to create each of these files for each locale you'll be supporting.

Each file needs to return an array. The array should use the message in your source language as its key and the same message in the target (aka user's) language as its

value. For example, here's what part of the "app" category would look like with French translations:

```
<?php
# messages/fr_FR/app.php
return [
    'Home' => 'Accueil',
    'Register' => 'S\'enregistrer',
    'Login' => 'Se connecter',
    'Logout' => 'Déconnexion',
    'Subject' => 'Sujet',
    'Body' => 'Contenu',
    'Submit' => 'Soumettez'
];
```

The success of the translation system is dependent upon the index there. This is case sensitive and must exactly match the string in the source language.

Here's how the same words would be represented in Norwegian:

```
<?php
# messages/nb_NO/app.php
return [
    'Home' => 'Hjem',
    'Register' => 'Registrer deg',
    'Login' => 'Logg inn',
    'Logout' => 'Logg ut',
    'Subject' => 'Emne',
    'Body' => 'Melding',
    'Submit' => 'Send'
];
```

For now, I'm going to work with those simple translations. In just a few pages, you'll learn how to make translations more flexible.

{TIP} When using translations, for optimal performance, you'll want to set a `cachingDuration` in the application's configuration which tells Yii to cache the messages. You'll learn more about caching in Chapter 17, "Improving Performance".

Once you've defined the translations, using them is remarkably easy. The static `t()` method of the `BaseYii` class serves this purpose. Its first argument is the category (which, therefore, identifies the message source) and its second is the string to be translated. This needs to match the indexes used in the PHP array, and should be in the application's native language.

Because `t()` is a static method, it's invoked without an object instance:

```
echo \Yii::t('app', 'Home');
```

And that's all there is to it! If the target (user) locale is `fr_FR`, then "Accueil" will be printed by that line. If the target locale is `nb_NO`, "Hjem" will be printed instead. If the target locale is the same as the application's, then no translation occurs at all and the provided string is printed.

{TIP} If an extension, such as a module or a widget, supports i18n, you can access those translation strings using `extName.category`.

Websites have plenty of strings that can be translated directly, without any dynamic functionality at all. For example, navigation elements, such as "Home", "Search", and so forth, can all be treated literally without modification. Other times, the translations need to be more dynamic. For example, a message that indicates an email address was not found in the system may want to display the actual email address, too. For these situations, you can use placeholders and parameters.

Placeholders go in the string: both the index/original language version and in the translated version. Placeholders are words wrapped in curly brackets: `{placeholder}`. Note that these are not variables!

Here is an example definition:

```
<?php
# messages/es_CO/app.php
return [
    // Other stuff.
    'The username {username} is not available.' => 'El
    nombre de usuario {username} no está disponible.',
    // More other stuff.
];
```

Again, note that the placeholder appears in both the index—the application language version—and the translation.

To use placeholders in a translated string, provide it as a third argument to the `t()` method:

```
echo \Yii::t('app', 'The username {username} is not
available.', ['username' => $username]);
```

Now the translated text will dynamically insert the value of the `$username` variable into the target language's version of the string (**Figure 15.3**). (Note that you don't



Figure 15.3: The localized version of the message also displays the value of the `username` variable.

use the braces around the placeholder when passing a value: it's `username`, not `{username}` in the above.)

Thanks to some helpful people on Twitter, here are localized versions of that same message in other languages if you want to practice with this:

- Il nome utente `{username}` non è disponibile. (it_IT)
- De gebruikersnaam `{username}` is niet beschikbaar. (nl_NL)
- Nazwa użytkownika `{username}` jest już zajęta. (pl_PL)
- Der Benutzername `{username}` ist nicht verfügbar. (de_DE)
- O usuário `{username}` não está disponível. (pt_BR)
- O utilizador `{username}` não está disponível. (pt_PT)
- Lietotājs `{username}` nav pieejams. (lv_LV)

{TIP} You can use multiple placeholders in a string, too.

Use of placeholders can be customized by applying formatting rules. For example, a placeholder might represent a number that can be formatted as a locale-specific currency or a date that can be formatted appropriately. To use this functionality, expand the placeholder definition. The complete syntax for a placeholder is either `{name, type}` or `{name, type, style}`.

A few allowed types are:

- date
- duration (as in amount of time)
- number
- ordinal
- plural
- time

You can find more in the [ICU documentation]. Naturally, the available styles depend upon the type:

- currency, integer, percent (number)



Figure 15.4: Spanish formatting of a currency and date.

- short, medium, long, full (date)
- short, medium, long, full (time)

Here are currency and date example definitions:

```
<?php
# messages/es_ES/app.php
return [
    // Other stuff.
    'Total: {total,number,currency}' => 'Total: {total,number,currency}',
    'Event Date: {event_date,date,short}' => 'Event Date: {event_date,date,short}',
    // More other stuff.
];
```

To use parameter formatting, provide the named value to the `t()` method as you normally would. The only catch is that you must use the same index value, including the full parameter name, as you would before:

```
$total = 1234.56;
$event_date = time(); // Dummy value!
echo \Yii::t('app', 'Total: {total,number,currency}',
    ['total' => $total]);
echo \Yii::t('app', 'Event Date: {event_date,date,short}',
    ['event_date' => $event_date]);
```

When you use parameter formatting, Yii automatically inserts the provided value into the translated string and formats it in accordance with the style and the user's locale (**Figure 15.4**).

There are other ways strings may need to be localized, such as by gender, holiday (e.g., country-specific celebrations), or season (e.g., summer vs. winter). You *could* create separate translation strings for each possibility, but Yii is prepared for this scenario and lets you create more flexible translations. This concept is called *choice format* and uses the “select” type to gracefully handle known options. The most common need for this would be to provide gender-based translations:

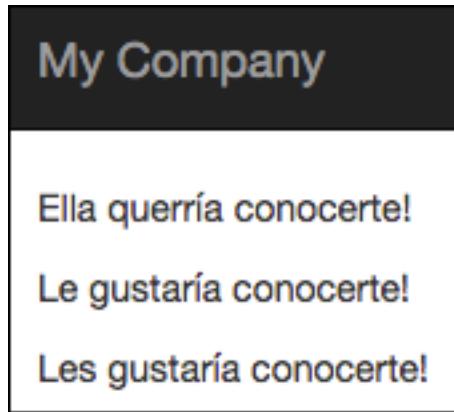


Figure 15.5: Spanish translations of three different gendered messages.

```
# messages/es_ES/app.php
<?php
return [
    // Other stuff.
    '{gender,select,female{She} male{He} other{They}} would like to
    meet you!' => '{gender,select,female{Ella querría} male{Le gustaría}
    other{Les gustaría}} conocerte!',
    // More other stuff.
];
```

This translation only requires that a gender be provided. If no gender is provided, the “other” value will be used (**Figure 15.5**):

```
echo \Yii::t('app', '{gender,select,female{She} male{He} other{They}}'
    would like to meet you!', ['gender' => 'female']);
echo \Yii::t('app', '{gender,select,female{She} male{He} other{They}}'
    would like to meet you!', ['gender' => 'male']);
echo \Yii::t('app', '{gender,select,female{She} male{He} other{They}}'
    would like to meet you!');
```

Similar to selections based upon gender or whatever, some strings refer to a noun that may be singular or plural, such as:

- The item has been added to your cart.
- The items have been added to your cart.

This situation can be handled using a variation on the choice format, known as *plural forms*. Use the “plural” type, defining the message to print for various quantities:

- 0

Spanish (Spain) es_ES			
Based on PHP intl data: ICU 4.8.1.1. Data 4.8.1.			
	General	Message Formatting	Number Formatting
Language	es	Spanish	español
Region	ES	Spain	España
Script	none	none	none
Default Currency	EUR (€)	Euro	euro

Figure 15.6: Details for the es_ES locale.

- 1
- few
- many
- other

When “few”, “many”, and “other” applies differ by language, but you’re generally safe specifying 0, 1, and “other”. Within each message, the number sign is used as a placeholder for the provided number.

The trickiest aspect of using this type is every quantity you specify has to be supported by the destination language. Using a quantity unsupported by the destination language means no translation will occur. Fortunately you can look up supported values at Alex Makarov’s excellent—and Yii-based—<http://intl.rmcreative.ru/> (Figure 15.6):

This probably sounds more confusing than it is, so let’s look at an example. Here is how you would define a translation stating how many items are in a shopping cart:

```
<?php
# messages/es_ES/app.php
return [
    // Other stuff.
    'You have {n,plural,=0{no items} =1{one item} other{# items} in
    your cart.' => 'TRANSLATION',
    // More other stuff.
];
```

The translation is used by merely providing a number when calling `t()` (Figure 15.7):

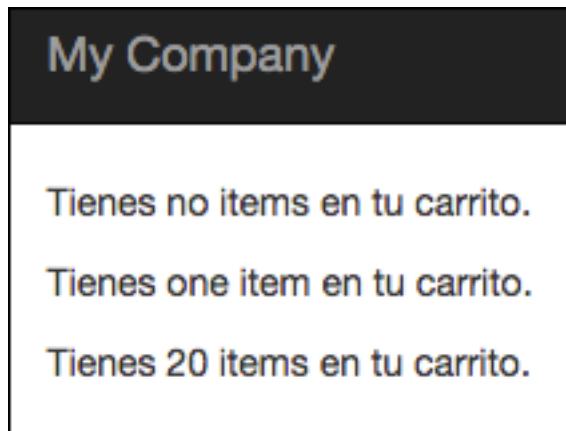


Figure 15.7: Spanish translations of three different quantities.

```
echo \Yii::t('app', 'You have {n,plural,one{one item} other{# items}} in your cart.', ['n' => 0]);
echo \Yii::t('app', 'You have {n,plural,one{one item} other{# items}} in your cart.', ['n' => 1]);
echo \Yii::t('app', 'You have {n,plural,one{one item} other{# items}} in your cart.', ['n' => 20]);
```

If you want to use the actual number in your message, use the placeholder #, as in the above code.

You can add in other placeholders, too, so long as the “n” value is passed first:

```
echo
\Yii::t('app', 'The {product} has been added to your cart.| The {n} {product} have been added to your cart.',
array($num, '{product}' => $product));
```

To be clear, the actual rules for plural forms comes from the CLDR database, and vary from one language to the next. Russian, for example, has more complex plural form rules than simply the singular or plural distinction that exists in English.

The `t()` method in Yii 2 has been greatly expanded such that it can be used to handle both translations and locale formatting in one step. In some situations, however, it may make more sense to separate simple formatting from any translating. When that is the case turn to Yii’s built-in formatters, defined within the `yii\i18n\Formatter` class.

To format numbers and currency in Yii, use the `yii\i18n\Formatter` class. An instance of it, set to the target locale, is available through `\Yii::$app->formatter`:

```
$formatter = \Yii::$app->formatter;
```

Once you have that class instance, there are several methods you'll use:

- `asCurrency()`
- `asDecimal()`
- `asInteger()`
- `asPercent()`

To output a formatted currency, provide that method with the value to be formatted and the currency code:

```
echo $formatter->asCurrency($value, 'USD');
```

The currency code is a three-letter code defined by [ISO 4217](#).

The `asDecimal()` method just takes the value to be formatted:

```
echo $formatter->asDecimal($value);
```

The `asInteger()` method also just takes the value to be formatted:

```
echo $formatter->asInteger($value);
```

To output a formatted percentage, you would just provide the method with the value to be formatted:

```
echo $formatter->asPercent($value);
```

Here are some samples, with the resulting output shown in [Figure 15.8](#):

```
<?php
// Set the number:
$number = 23049.59;

// Start in English (US):
\Yii::$app->language = 'en_US';
$formatter1 = \Yii::$app->formatter;
echo '<p>English (decimal): ' . $formatter1->asDecimal($number)
. '</p>';
echo '<p>English (currency): ' . $formatter1->asCurrency($number, 'USD')
. '</p>';
```

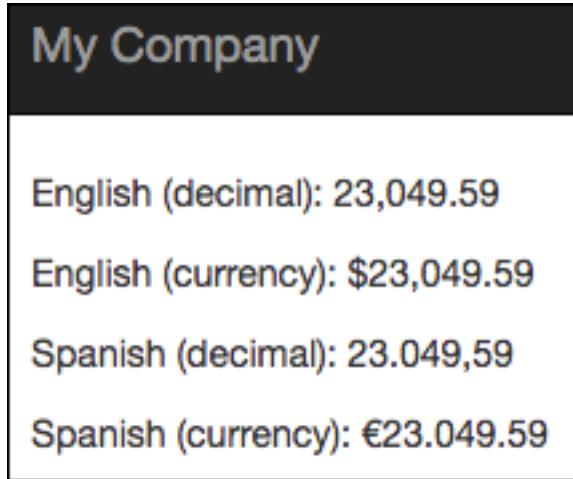


Figure 15.8: Numbers and currencies formatted in different ways, including Spanish standards.

```
// Change to Spanish:  
\Yii::$app->language = 'en_GB';  
$formatter2 = \Yii::$app->formatter;  
echo '<p>Spanish (decimal): ' . $formatter2->asDecimal($number)  
    . '</p>';  
echo '<p>Spanish (currency): ' . $formatter2->asCurrency($number, 'EUR')  
    . '</p>';
```

Next up, you'll want to know how to format dates and times in a manner that's consistent with the user's locale. Formatting of dates in times in a locale-sensitive way is also done via the `yii\i18n\Formatter` class:

```
$formatter = \Yii::$app->formatter;
```

Once you have that instance, you'll call one of its methods:

- `asDate()`
- `asDateTime()`
- `asTime()`
- `asTimestamp()`

The most important of these are the first three. Provide each with the date or date and time and the format to use. The date or date and time can be provided as: a Unix timestamp, a DateTime Object, or a string formatted in a way that the PHP `strtotime()` function understands.

For the format, you'll normally use combinations of "y", "M", "d", "h", and "m" (note all the capitalizations). These come from [Unicode's date format patterns](#). Some examples (using 1:15PM on April 1st, 2017):

- *YYYY-MM-dd* would be 2017-04-01
- *MM/dd/YY HH:mm* would be 04/01/17 13:15
- *MMM d, YYYY* would be Apr 1, 2017
- *MMMM d, YYYY h:mm a* would be April 1, 2017 1:15 PM

Instead of providing the format to each individual call, you can globally set the formats using the `dateFormat`, `timeFormat`, and `datetimeFormat` properties:

```
$formatter->dateFormat = 'YYYY-MM-dd';
```

However, as formatting of dates and times is another regional difference, instead of hard-coding the format, it's better to tell Yii what rough format to use and then let the application handle it. The above methods accept "short", "medium", "long", and "full" as shortcuts to locale-aware formats.

Here are some samples, with the resulting output shown in **Figure 15.9**:

```
<?php

$time = time();
$formatter = \Yii::$app->formatter;
echo '<p>English (long): ' .
    $formatter->asDatetime($time, 'long') . '</p>';
echo '<p>English (medium): ' .
    $formatter->asDatetime($time, 'medium') . '</p>';
echo '<p>English (short): ' .
    $formatter->asDatetime($time, 'short') . '</p>';
echo '<p>English (short, date only): ' .
    $formatter->asDate($time, 'short') . '</p>';

// Change the target language:
$formatter->locale = 'es_ES';

// Show it in Spanish:
echo '<p>Spanish (long): ' .
    $formatter->asDatetime('now', 'long') . '</p>';
echo '<p>Spanish (short, date only): ' .
    $formatter->asDate('now', 'short') . '</p>';
```

{NEW} Yii 2 supports setting the locale specifically on the formatter.

```
English (long): December 2, 2017 at 8:19:49 PM GMT+1  
English (medium): Dec 2, 2017, 8:19:49 PM  
English (short): 12/2/17, 8:19 PM  
English (short, date only): 12/2/17  
Spanish (long): 2 de diciembre de 2017, 20:19:49 CET  
Spanish (short, date only): 2/12/17
```

Figure 15.9: The date and time formatted in different ways, including Spanish.

Two related functions are `asRelativeTime()` and `asDuration()`. The former returns a relative version of the current date and time compared to another date and time (e.g., as “1 hour ago”). The latter returns an interval in a human-readable form (e.g., “1 day, 2 hours”). See the documentation for details on using either.

Another way you might want to customize the look and behavior for the user is to take into account the user’s time zone. This is especially true on sites where time-sensitive events occur. For example, if your site sells tickets to an event, and the tickets do not go on sale until 10am on a certain day, whose 10am is that?

New in Yii 2 is the ability to automatically convert dates to a specific timezone. Simply set the formatter’s `timeZone` property to the user’s timezone:

```
<?php  
  
$formatter = \Yii::$app->formatter;  
$formatter->timeZone = 'America/New_York';
```

Yii assumes the default time zone is UTC, unless configured otherwise. Thus the above code will take a date and time (or timestamp) and adjust the outputted values by 5 hours, depending upon Daylight Savings. Be sure to always store dates in your database in UTC format. (This is just good practice.)

Most formatting needs fall under numbers and dates, but the `yii\i18n\Formatter` class supports several other formats, too, through these methods:

- `asBoolean()`
- `asEmail()`

- `asHtml()`
- `asText()`
- `asUrl()`

You'll want to use these to create appropriate output of data, such as:

- “Yes” for a true Boolean value
- A properly formatted “mailto” link
- HTML-encoded data (which is what the `asText()` method does)

There's a fair amount of complexity to internationalization, just within the language area alone, but this chapter ought to mention a couple more items that you may want to investigate further.

In the same way that Yii can create different versions of a string in different languages, it can also render different language-specific view files, too. To use this, create locale-specific directories within each of your `views/ControllerID` directories. Then render the view file as you normally would.

For example, say the “user” controller's “create” view file should be localized to Spanish. To do that, make the `create.php` file, using all the Spanish you want. Store this file in `views/user/es_ES`. Then your controller simply has to do:

```
# controllers/UserController.php
return $this->render('create');
```

If the target (user's) locale is “es_ES”, the Spanish version of the view file will be rendered instead of the default one.

If no translation file is available for the view, the default, untranslated version will be used instead.

If you use translations a lot, or extensively on one site, it may be worth your while to investigate the available `i18n` extensions. Specifically, you may want to consider an extension that allows you to localize your URLs.

For example, instead of showing a Spanish user the URL `http://example.com/site/contact`, you could chose them `http://example.com/sitio/contacto`. The [Yii2 Language URL Manager](#) serves this role.

Relatedly, it's fairly common to use a dedicated subdomain (e.g., `es.example.com`) or top-level folder (e.g., `example.com/es/`) to clearly indicate that a translated version of the site is being presented. By changing the base URL accordingly, and using redirects when the user's language is detected or selected, you should be able to support either approach.

As much of a site depends upon its models, this chapter concludes by revisiting the topic of models with respect to internationalization. For example, if your site

has an international audience and a “user” model, it would make sense for model references to be localized. In other words, have the words “username”, “password”, and so forth be displayed in the user’s preferred language.

This is easily accomplished by changing the values returned by the model’s `attributeLabels()` method. Instead of having it return hard-coded strings in one language, have it return localized strings via the `t()` method:

```
# models/AnyModel.php
public function attributeLabels() {
    return [
        'username' => \Yii::t('user','Username'),
        'email' => \Yii::t('user','Email Address'),
        'password' => \Yii::t('user','Password')
    ];
}
```

For each language the site would support, create a `messages/LocaleId/user.php` file that returns an array of those values with the proper translations. (Or use a database that does the same.)

Similarly, you may want to automatically display dates formatted for the user’s locale and time zone. One way of doing so is to pass model values to a date formatter method. Assuming that the `$model` is an instance of the `User` class and has a `date_entered` attribute, you could display the date that the user registered, formatted to the user’s locale and time zone:

```
<?php
$formatter = \Yii::$app->formatter;
$formatter->timeZone = 'America/New_York'; // Or whatever is the user's.
echo '<p>Member since: ' .
    $formatter->asDateTime($model->date_entered, 'medium') . '</p>';
```

(Obviously you’d want to translate the “Member since” part, too.)

An alternative approach would be to embed the date and time formatting within the model itself:

```
# protected/models/User.php
public function getDateRegistered() {
    $formatter = \Yii::$app->formatter;
    $formatter->timeZone = 'America/New_York'; // Or whatever is the user's.
    return $formatter->asDateTime($this->date_entered);
}
```

Now a view file can use `$model->getDateRegistered()` to get the `date_entered` value formatted to the user’s locale. The primary downside to this approach is that the formatting style—long, medium, or short—is embedded into the method.

To explain a slightly more advanced option, you can also use `$model->dateRegistered` in your view files to get that same value. In other words, `$model->dateRegistered` and `$model->getDateRegistered()` are the same. Chapter 19, “Extending Yii,” explains this in more detail, but the simple reason is that models extend the `Component` class, and one of the features that `Component` creates is the ability to get and set attributes. This was mentioned in Chapter 5, “[Working with Models](#).”

Chapter 16

Leaving the Browser

Somewhat ironically, not everything that pertains to a website makes use of a browser. Although what is normally considered to be a “website” is the combination of HTML, JavaScript, CSS, and media that the user sees, today’s sites often other resources that are never intended to be accessed by a browser.

This chapter explains how to use the Yii framework in ways that *don’t* make use of the browser in three specific ways:

- Proxy scripts
- RESTful web services
- Console applications

Now, in truth, the first two of these *can* be accessed via the browser in that they’ll be available over HTTP. But the distinction being made is that all three uses are not intended to be directly accessed by end users with their browsers. Put another way, this chapter explains situations in which you’d use the Yii framework but generally *not* use the primary layout file to render a complete HTML page.

Proxy scripts, in case you’re not familiar with them, are agents in a website, standing in for other resources. Proxy scripts are commonly used to:

- Provide access to restricted resources
- Hide direct access to resources
- Track usage of materials, such as the number of times a file has been downloaded

For example, if you have files uploaded outside of the web directory, a proxy script would be required to provide those files to the browser. Or if only logged-in users could view a PDF, a proxy script could enforce that restriction.

{TIP} In the website I created for [selling this book](#), a proxy script prevents un-paid users from downloading copies. The same proxy script also tracks the number of downloads.

The most important distinction between a proxy script and a standard PHP page is that proxy scripts generally don't output *any* HTML. When it comes to using the Yii framework, this means that your controllers use `renderPartial()` instead of `render()`, so as to omit the layout file:

```
# controllers/SomeController.php
public function someAction() {
    // Do whatever.
    return $this->renderPartial('thing');
}
```

Of course, what, exactly, the `thing.php` view file does depends entirely upon the specific situation. The chapter comes back to this aspect shortly.

Proxy scripts run through the Yii framework can easily take advantage of the same access control that any other page can. If part of a proxy script's role is to limit access, just apply the information covered in Chapter 11, “[User Authentication and Authorization](#).”

In certain situations, it's useful to hide direct access to a resource. Maybe a page or file is directly available, but you'd rather not make the URL obvious in the browser. If it's a standard HTML page, you can use `render()` but render a page from another controller or view:

```
# controllers/SomeController.php
public function someAction() {
    // Do whatever.
    $this->render('//secret/thing');
}
```

Other times, the resource being “hidden” is a file. In standard PHP, you'd use several `header()` function calls, plus `readfile()` to send the file to the browser. Yii has simplified that process with the `sendFile()` method of the `CHttpRequest` class. An instance of that class is available through the application's `getRequest()` object:

```
Yii::$app->response->sendFile($path_to_file, $filename);
```

The first argument needs to be the path to the file on the server. The second is the file's name. The name is optional; if not provided Yii will use the name of the file as it is on the server. Provided with these two pieces of information, this method will automatically:

- Determine the proper MIME type
- Send the proper headers
- Send the content itself (i.e., the file)
- Terminate the application

The simplest way to use this method is like so:

```
# controllers/SomeController.php
public function someAction() {
    // Do whatever.
    $filename = 'file.ext';
    $file = "/path/to/$filename";
    return Yii::$app->response->sendFile($file, $filename);
}
```

That approach is fairly good, but could be improved for security purposes. First, arguably one should rename uploaded files for security purposes. They might be renamed as a simple integer or a hash. In such situations, you'd want the browser to be provided with the original file name (so that the original file name would be the default file name when the user downloads the file). A second security concern is you'd want to make sure the script couldn't be used to download any file on the server.

For a hypothetical example that solves both of these issues, let's assume that the files are represented by the `File` class. The class stores the file's original name in the `name` property. When the file was uploaded, it was stored under the model's ID (e.g., the file "page.pdf" is stored as just "345"). The controller simply needs to load the `File` instance, which also restricts what file can be download, and then serves that file, providing the original name in the process:

```
# controllers/FileController.php
public function downloadFile($id) {
    $f = File::findOne($id);
    if ($f === null) {
        throw new \yii\web\HttpException(404,
            'The file could not be found.');
    }
    $file = "/path/to/" + $f->id;
    return Yii::$app->response->sendFile($file, $file->name);
}
```

Another possible issue with the `sendFile()` method is that it always sends files using the "attachment" Content-Disposition header. The actual impact will depend upon the browser and the file, but this normally results in the file being downloaded. When serving images, that may not be the desired intent (i.e., you'd want to use

the “inline” Content-Disposition). In that case, provide an optional third argument to `sendFile()`, which is an array of options, setting `inline` to true:

```
return Yii::$app->response->sendFile($file, $file->name,
    ['inline' => true]);
```

A “web service”, in case you’re unfamiliar with the term, is a defined method of communication that occurs over the web. A web service can be so simple that it merely returns the current time, or so complex that it expects multiple parameters (e.g., a list of stock abbreviations, a format designator, and so forth) and returns an array of objects in a class understood by both parties.

In terms of technologies involved, web services encompasses a wide variety, and quite a lot of acronyms. In whatever format, though, tapping into web services makes for very powerful sites.

Web services can be implemented in many ways. This chapter touches upon RESTful (Representational State Transfer) services. Support for creating potent RESTful services is greatly expanded in Yii 2, while support for the older SOAP standard has been removed. After getting started with RESTful services, read the [Yii guide](#) to delve further into what’s possible, which includes rate limiting, versioning, and more sophisticated data management.

Once defined, web services can also be used in two generic ways:

- Directly in the browser, via Ajax
- Directly from the server, via cURL or the like

For example, you might create a service on your site that’s intended to be invoked via Ajax from a page within your own site. Chapter 14, “[JavaScript and jQuery](#),” already demonstrated that. Or, within a controller on your site, you might invoke a service on another site in order to obtain information to be passed to the view file (e.g., a stock quote).

For a working example of this, let’s take the CMS project discussed throughout the book and create a web service that returns a page’s information when provided with a page ID.

RESTful services should be defined as a formal API, grouped together in a single controller. In Yii 2, this controller extends `yii\rest\ActiveController`, instead of `yii\Web\Controller` (assuming the service builds on top of Active Record).

```
# controllers ApiController.php
<?php

namespace app\controllers;
```

```
use yii\rest\ActiveController;

class ApiController extends ActiveController {
    public $modelClass = 'app\models\Page';
}
```

The `$modelClass` attribute ties the controller to an existing Active Record model.

By extending `yii\rest\ActiveController`, this controller inherits commonly needed actions:

- index
- view
- create
- update
- delete
- options

The controller also inherits user authorization.

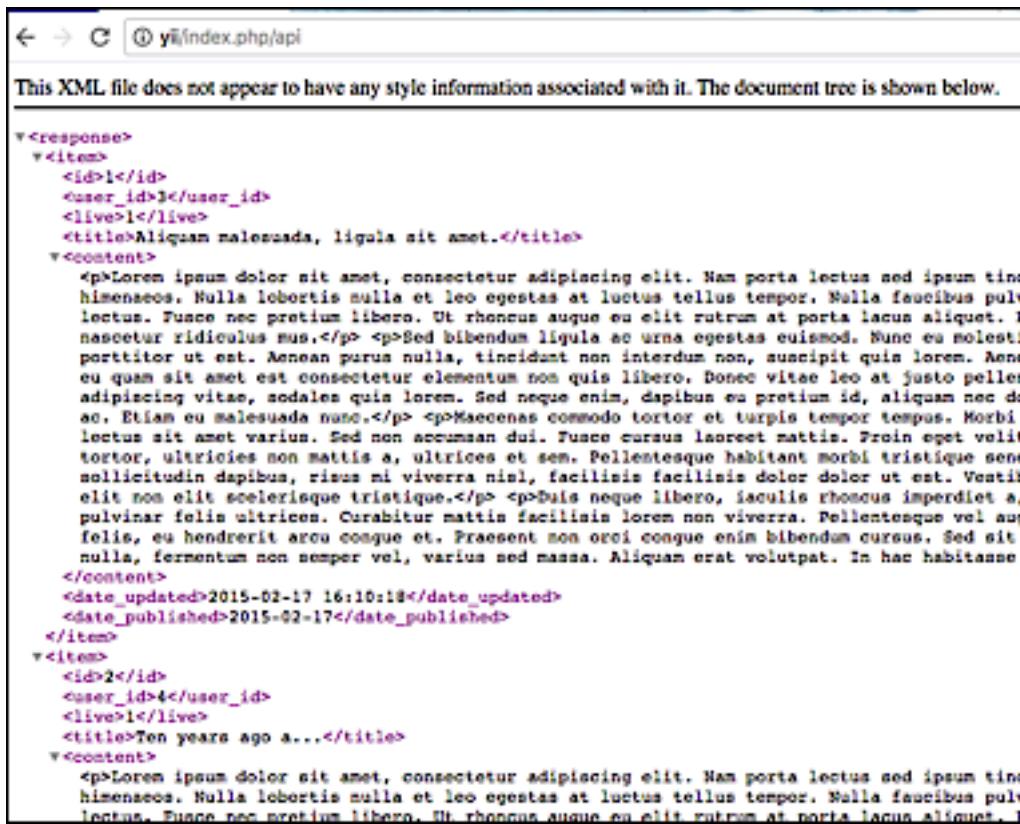
To create additional actions, define methods as you would with a standard controller, with one major difference: the method should return raw data instead of using `render()`:

```
# controllers ApiController.php
public function actionList() {
    // Do the work.
    return $data;
}
```

That's it for the controller for now.

Unlike normal controllers, Yii does not automatically support RESTful routes just based upon the REST controller. You have to formally define the routes in your configuration file. Add the new controller as a rule:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'api',
         'pluralize' => false],
    ],
],
```



The screenshot shows a browser window with the URL `http://www.example.com/index.php/api`. The page content is an XML document. The XML structure is as follows:

```
<?xml version="1.0"?>
<response>
  <item>
    <id>1</id>
    <user_id>3</user_id>
    <live>1</live>
    <title>Aliquam malesuada, ligula sit amet.</title>
    <content>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ipsum tincidunt. Nulla lobortis nulla et leo egestas at luctus tellus tempor. Nulla faucibus pulvinar. Fusce nec pretium libero. Ut rhoncus augue eu elit rutrum at porta lacus aliquet. In hac habitasse porttitor ut est. Aenean purus nulla, tincidunt non interdum non, suscipit quis lorem. Aenean eu quam sit amet est consectetur elementum non quis libero. Donec vitae leo at justo pellentesque. Sed neque enim, dapibus eu pretium id, aliquam nec dico. Etiam eu malesuada nunc.</p> <p>Maecenas commodo tortor et turpis tempor tempus. Morbi lectus sit amet varius. Sed non accumsan dui. Fusce cursus laoreet mattis. Proin eget velit tortor, ultricies non mattis a, ultrices et sem. Pellentesque habitant morbi tristique sed sollicitudin dapibus, risus mi viverra nisl, facilisis facilisis dolor dolor ut est. Vestibulum elit non elit scelerisque tristique.</p> <p>Duis neque libero, iaculis rhoncus imperdiet a, pulvinar felis ultrices. Curabitur mattis facilisis lorem non viverra. Pellentesque vel augue felis, eu hendrerit arcu congue et. Praesent non orci congue enim bibendum cursus. Sed sit nulla, fermentum non semper vel, varius sed massa. Aliquam erat volutpat. In hac habitasse
```

The XML includes two items. The first item has an ID of 1, a user ID of 3, and is live. Its title is "Aliquam malesuada, ligula sit amet." and its content is a long paragraph of Lorem ipsum text. The second item has an ID of 2, a user ID of 4, and is live. Its title is "Ten years ago a..." and its content is another long paragraph of Lorem ipsum text.

Figure 16.1: A view all request of the API.

By default, Yii automatically pluralizes the controller name when it creates a RESTful service. For example, if the controller was called “PageController” and “page” was entered in the above, the route would be **pages**. As this example uses “api” for the route, “apis” doesn’t make sense, so the pluralization is disabled.

That’s it! You’ve created a RESTful service! This service is available like any other controller action, which you can test by going to this URL in your browser:

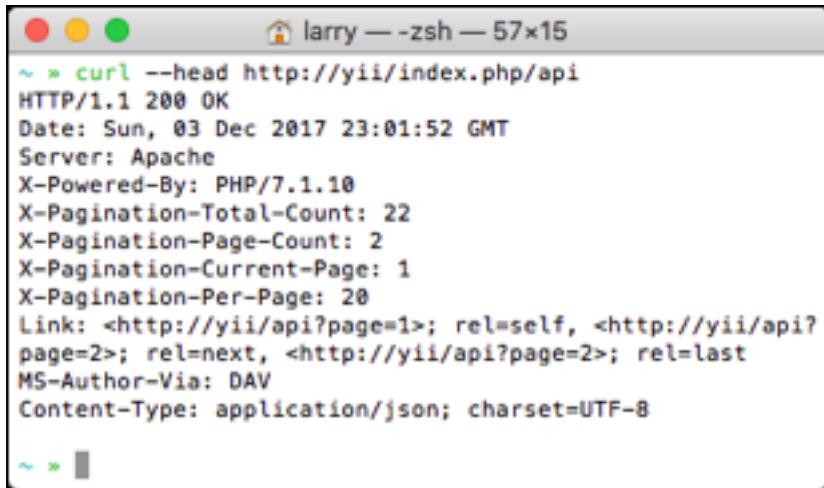
`http://www.example.com/index.php/api`

You should see a list of pages displayed as XML (**Figure 16.1**):

Load **`http://www.example.com/index.php/api/1`** to see the XML for just one page (with an ID of 1).

It’s easy enough to test basic GET requests in a browser, but RESTful services normally make use of other request types for other purposes. GET requests fetch every record or just one, if an ID was provided. For other request types, cURL provides an easy and common interface.

A HEAD request fetches an overview of information for every record or just one, if an ID was provided (**Figure 16.2**):



```
curl --head http://yii/index.php/api
HTTP/1.1 200 OK
Date: Sun, 03 Dec 2017 23:01:52 GMT
Server: Apache
X-Powered-By: PHP/7.1.10
X-Pagination-Total-Count: 22
X-Pagination-Page-Count: 2
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://yii/api?page=1>; rel=self, <http://yii/api?page=2>; rel=next, <http://yii/api?page=2>; rel=last
MS-Author-Via: DAV
Content-Type: application/json; charset=UTF-8
```

Figure 16.2: A HEAD request of a single record.

```
curl --head http://www.example.com/index.php/api
```

A DELETE request removes a single record:

```
curl -X DELETE http://www.example.com/index.php/api/42
```

An OPTIONS request returns the verbs (i.e., actions) available at an endpoint:

```
curl -i -X http://www.example.com/index.php/api
```

A POST request creates a new record:

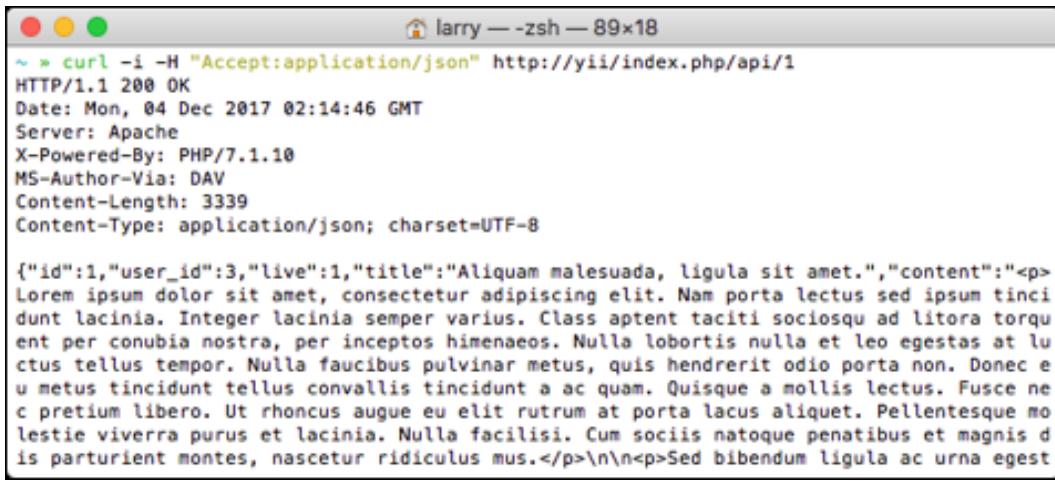
```
curl -X POST -d "user_id=1&live=0&title=This..." \
http://www.example.com/index.php/api
```

However, creating pages via a cURL command is tedious, as is updating a record via a PATCH request:

```
curl -X PATCH -d "user_id=1&live=0&title=This..." \
http://www.example.com/index.php/api/42
```

As you can see, without much coding at all, you now have a fairly complete RESTful service.

As Figure 16.1 shows, by default the RESTful service returns responses in XML format. Out-of-the box, it can also return responses in JSON format, when requested (**Figure 16.3**):



```
larry -- zsh -- 89x18
~ » curl -i -H "Accept:application/json" http://yii/index.php/api/1
HTTP/1.1 200 OK
Date: Mon, 04 Dec 2017 02:14:46 GMT
Server: Apache
X-Powered-By: PHP/7.1.10
MS-Author-Via: DAV
Content-Length: 3339
Content-Type: application/json; charset=UTF-8

{"id":1,"user_id":3,"live":1,"title":"Aliquam malesuada, ligula sit amet.", "content": "<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam porta lectus sed ipsum tinci  
dunt lacinia. Integer lacinia semper varius. Class aptent taciti sociosqu ad litora torqu  
ent per conubia nostra, per inceptos himenaeos. Nulla lobortis nulla et leo egestas at lu  
ctus tellus tempor. Nulla faucibus pulvinar metus, quis hendrerit odio porta non. Donec e  
u metus tincidunt tellus convallis tincidunt a ac quam. Quisque a mollis lectus. Fusce ne  
c pretium libero. Ut rhoncus augue eu elit rutrum at porta lacus aliquet. Pellentesque mo  
lestie viverra purus et lacinia. Nulla facilisi. Cum sociis natoque penatibus et magnis d  
is parturient montes, nascetur ridiculus mus.</p>\n\n<p>Sed bibendum ligula ac urna egest
```

Figure 16.3: One record returned by the API, as JSON.

```
curl -i -H "Accept:application/json" http://www.example.com/index.php/api/1
```

By default, the service accepts data (e.g., for creation and update requests) in application/x-www-form-urlencoded and multipart/form-data formats. You can also configure it to support JSON for incoming data. Configure the “request” component to use `yii\web\JsonParser`:

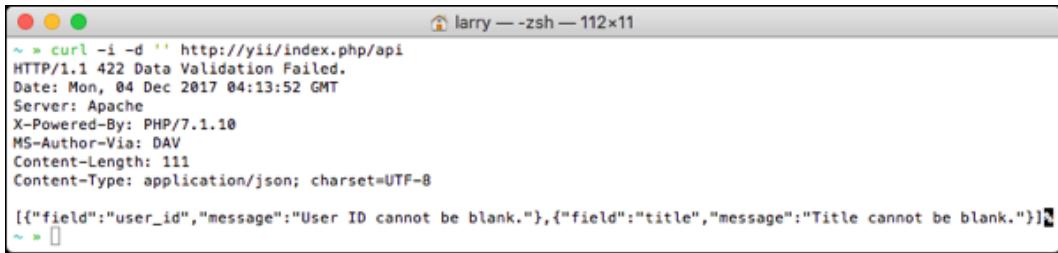
```
'request' => [
    'parsers' => [
        'application/json' => 'yii\web\JsonParser',
    ]
]
```

Now requests to create new records or update existing ones can be made using the more natural JSON format.

To make this service more polished, it ought to include an approach for indicating problems. As written, the service returns an individual page record when provided with a corresponding page ID, but what if the provided page ID does not correlate to an existing record? The service could, in that case, return an empty data set, but that approach won’t differentiate among:

- The page ID not matching a valid record
- An invalid request being made of the service
- The service generally not working
- And other possibilities

The solution to this problem uses a communication device that already exists: since the RESTful request is being made over HTTP, HTTP status codes are perfect for indicating the success of the request.



```
larry -- zsh -- 112x11
~ > curl -i -d '' http://yii/index.php/api
HTTP/1.1 422 Data Validation Failed.
Date: Mon, 04 Dec 2017 04:13:52 GMT
Server: Apache
X-Powered-By: PHP/7.1.10
MS-Author-Via: DAV
Content-Length: 111
Content-Type: application/json; charset=UTF-8
[{"field": "user_id", "message": "User ID cannot be blank."}, {"field": "title", "message": "Title cannot be blank."}]
~ > [
```

Figure 16.4: Failure to provide proper arguments when creating a record results in a detailed error.

Some of the most common status codes, and their meaning, are listed in the following table.

Code	Meaning
200	OK
201	Created
204	No Content
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
418	I'm a teapot
500	Internal Service Error

These are logically and directly mappable to the service responses, except perhaps for the “I’m a teapot” status, which is actually assigned to 418 (you can look it up).

The service as written already returns error codes for inappropriate requests. Moreover, it returns error messages(**Figure 16.4**)!

The error messages come from the underlying model. Apply the information covered in Chapter 5 to customize these.

The last subject to discuss in this chapter are *console applications*. Console applications, or console scripts, are intended to be run not in a browser (i.e., accessed over HTTP) but within a console or command-prompt environment.

For beginning developers, the need for, let alone actually using, console applications can often be unclear. Console applications are great in situations where:

- The resulting output will be minimal or immaterial
- The execution will take excessive time (more than even a complex web page)
- The execution should be run on a schedule

Common uses of console applications include:

- Generation of resources, such as images, PDFs, and even code
- Maintenance of the file system, database, search indexes, etc.
- Other services, like sending emails

When you have tasks like these that need to be performed in conjunction with a Yii-based website, it makes sense to create a Yii-based console application.

Just as web pages are run through the **web/index.php** script, console applications go through their own, again auto-generated as part of the base Yii application. In the root application directory, you'll find **index.php**, containing:

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 */

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

require __DIR__ . '/vendor/autoload.php';
require __DIR__ . '/vendor/yiisoft/yii2/Yii.php';

$config = require __DIR__ . '/config/console.php';

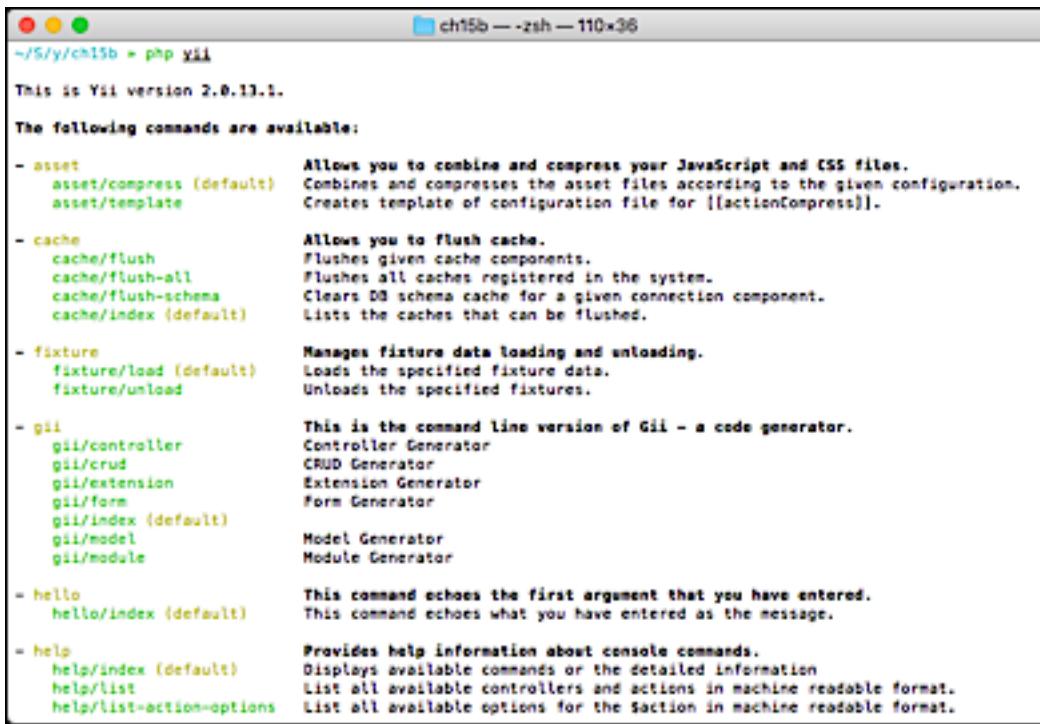
$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

As you can see, the script:

- Defines environment variables
- Requires the framework files
- Pulls in the **console.php** configuration file
- Creates and runs a new application

Whereas the web entry script creates a **yii\web\Application** instance, this script creates a **yii\console\Application** using the **console.php** file. So this is similar to what the bootstrap **index.php** script does, except for the type of application being created: web or console. Also, because it's in the root directory, the console **index.php** script should not be accessible via a browser.

{TIP} The **yiic.bat** file, also found in the root directory, is a Windows tool for running console applications.



```
ch15b -- zsh -- 110x36
~/S/y/ch15b > php yii
This is Yii version 2.0.13.1.

The following commands are available:

- asset           Allows you to combine and compress your JavaScript and CSS files.
  asset/compress (default) Combines and compresses the asset files according to the given configuration.
  asset/template   Creates template of configuration file for [[actionCompress]].

- cache           Allows you to flush cache.
  cache/flush      Flushes given cache components.
  cache/flush-all  Flushes all caches registered in the system.
  cache/flush-schema Clears DB schema cache for a given connection component.
  cache/index     (default) Lists the caches that can be flushed.

- fixture         Manages fixture data loading and unloading.
  fixture/load    (default) Loads the specified fixture data.
  fixture/unload  Unloads the specified fixtures.

- gii             This is the command line version of Gii - a code generator.
  gii/controller  Controller Generator
  gii/crud        CRUD Generator
  gii/extension   Extension Generator
  gii/form        Form Generator
  gii/index     (default) Model Generator
  gii/model       Module Generator

- hello            This command echoes the first argument that you have entered.
  hello/index    (default) This command echoes what you have entered as the message.

- help             Provides help information about console commands.
  help/index     (default) Displays available commands or the detailed information
  help/list       List all available controllers and actions in machine readable format.
  help/list=action=options List all available options for the action in machine readable format.
```

Figure 16.5: A few of the built-in console application commands.

The structure of **console.php** is the same as the other two main configuration files: **web.php** and **test.php**. But due to the narrower scope of console applications, the console-specific configuration file will never need as much customization.

The configuration file pulls in the same parameters and database information—stored in **params.php** and **db.php**, accordingly—as **web.php**. For this reason, you probably won’t do much console configuration unless you depend upon console scripts a lot.

As always, you can find the full list of **yii\console\Application** properties in the [Yii class docs](#).

Just like the RESTful service, the console application comes with several built-in commands. To see those, run this from within the application’s root directory (**Figure 16.5**):

```
php yii
```

If there’s no PHP executable in your path, you’ll need to call it explicitly:

```
C:\xampp\php\php.exe yii
```

Depending upon your environment, you might also be able to use just:

```
./yii
```

You'll need to play around with these variations, and pay attention to any error messages you might see, until you find the syntax that works for your situation. The rest of this chapter just uses `php yii` to be consistent; change it to suit your proper solution.

Looking at the response of successfully executing that command, you'll see that out-of-the-box the console application can:

- Manage JavaScript and CSS assets
- Manipulate the cache
- Load and unload fixtures (for testing)
- Generate files with Gii
- Assist with the creation of translation files
- Perform migrations
- Run PHP's built-in web server

To execute a specific command, use this syntax:

```
php yii <command-name>/<action-name>
```

This mimics the basic routing for web pages: `/`.

With the built-in “help/index” command, that syntax would be:

```
php yii help/index
```

The output is the same as in Figure 16.5. This is because “help” is the default command and “index” is the default action. You can change this behavior by setting the `defaultRoute` property in the configuration file.

{NOTE} You do not need to quote the command name, the action name, or the combined route.

If an action takes an argument, provide that after the route (**Figure 16.6**):

```
php yii hello/index 'Testing this command.'
```

Now that you know how to execute the built-in commands, it's time to learn how to define your own.



Figure 16.6: The `hello/index` command echoes its provided argument.

Having configured and tested your console application, the next step is to create your custom commands. A single console application can have any number of commands to be executed.

Each command is defined as its own controller that must extend the `yii\console\Controller` class. The class should be in the `app\commands` namespace, although this is configurable. The class name takes the format “`CommandNameController`”:

```
# commands/PurgeController.php
<?php
namespace app\commands;
use yii\console\Controller;
class PurgeController extends Controller {
    // Do the work!
}
```

The class must be defined in a file whose name matches the class name (as always), and be stored in the `commands` directory. You’ll find the generated `HelloController.php` in this directory already.

The class can have (or not have) attributes and methods like any other class. Within the class, you can create multiple actions by defining a series of methods, just as you would in a web controller. For example, the `DatabaseController` class might define methods for optimizing tables, backing up the database, etc.

To implement this, create one or more methods with the name of “action” followed by the command name: `actionOptimize()`, `actionBackup()`, etc.

```
<?php
# commands/DatabaseController.php
namespace app\commands;
use yii\console\Controller;
class DatabaseCommand extends Controller {
    public function actionOptimize() {
        echo 'Optimizing...' . PHP_EOL;
        // Do whatever!
    }
    public function actionBackup() {
        echo 'Backing up...' . PHP_EOL;
        // Do whatever!
    }
}
```

```
    }
}
```

Within each method you can do whatever needs to be done, including making use of the Yii application's defined models. As with a web-based scripts, the application instance is available through `\Yii::$app`, although here that instance is of type `yii\web\Application`, not `yii\console\Application`.

Keep in mind that the console application method would not render any views, as the command is being executed in the command-line environment. You can print simple messages, if you want, although if you plan on executing the script through a cron, no one would ever see those messages (but they can be useful for debugging purposes while you're developing the code).

You may commonly write console applications that do different things based upon different arguments. For example, maybe the maintenance would be performed on a specific table or list of tables, to be indicated when the command is run.

To enable this functionality, just have the controller method accept arguments as you would with a web-based controller:

```
# commands/DatabaseController.php
public function actionBackup($table) {
    echo 'Backing up...' . PHP_EOL;
    // Do whatever!
}
```

To pass arguments to that method, use this syntax:

```
php yii database/backup ---table=some_table
```

You don't need to quote any strings unless they contains spaces or other potentially problematic characters. (That being said, there's no harm in quoting the parameter values, either.)

As a hypothetical, let's say that there's a "purge" command with a "clear" action. That action may take as arguments: the type of thing to purge, an expiration unit (e.g., hour or day), and an expiration quantity. It might then be called like so:

```
php yii purge/clear --type=cache --interval=hour --number=2
```

And the method would be defined as:

```
# commands/PurgeController.php
<?php
public function actionClear($type, $interval, $number) {
    // Do the work!
}
```

You don't have to provide the arguments in order, however. This would also work:

```
php yii purge/clear --number=2 --interval=hour --type=cache
```

The final thing you should think about when it comes to creating command-line scripts are exit codes. Exit codes aren't commonly used by PHP programmers, as the server itself normally handles the proper codes to indicate the success or failure of an operation. But in the command-line world, including in languages such as C and Java, having a code be returned to indicate the success or failure of an operation is the norm.

In the command-line interface, returning a code from an action or command is as simple as having the associated method use a `return` statement. The standard is for an integer to be returned, with 0 being returned to indicate success:

```
<?php
# commands/DatabaseController.php
namespace app\commands;
use yii\console\Controller;
class DatabaseCommand extends Controller {
    public function actionOptimize() {
        echo 'Optimizing...' . PHP_EOL;
        // Do whatever!
        return 0;
    }
    public function actionBackup() {
        echo 'Backing up...' . PHP_EOL;
        // Do whatever!
        return 0;
    }
}
```

This always struck me as a little backwards (after all, 0 equates to false in many situations), but you can think of it as "0 errors occurred".

At a minimum, returning 1 indicates that an error occurred. You can also use escalating integers (up to 254) to indicate a level of error. I would caution not to go overboard with that, though, as the returned number would not have inherent meaning outside of the command-line script itself.

Alternatively, you can make use of the `yii\console\ExitCode` constants, such as `OK` (0), `UNSPECIFIED_ERROR` (1), `USAGE` (64), and `NOINPUT` (66).

Chapter 17

Improving Performance

Using a framework such as Yii provides an exponentially faster development time, but it does so at a cost: how well the site performs. Tapping into frameworks can be a faster way to create software, and the end result may even be more secure and feature-rich, but in all likelihood, the generated software performs more poorly than it would have if written from scratch (assuming general competency on the programmer's part with both approaches).

However, there is a counter argument as to why this performance hit is acceptable. First, there are limited ways to speed up your development time when *not* using a framework, and it's impossible to get wasted time back. Second, you *can* improve the performance of your framework-based project. Third, a framework-based site might be able to scale better than a non-framework site, as much of the code has already been abstracted.

This chapter explains the myriad ways you can improve your website's performance. Some approaches make use of smart features the Yii framework has, while others are mostly be a matter of just making smart decisions as you develop the site.

The chapter also covers how you test a site's performance, as you can't know whether your performance improvements are making a difference or not without concrete numbers. The chapter concludes with a suggested, cohesive plan for how you should develop, and then speed up, your Yii applications.

Due to the number of possible factors, there are very few absolutes when it comes to improving a site's performance. An approach that has a huge impact in one situation may have no benefit, or even a cost, in another. Thus it's imperative that you know how to test performance so that you can make the decisions that best benefit your site running on your server.

When it comes to a website, performance can be lumped into two broad categories:

- How quickly the resulting page is outputted *by the server*
- How quickly the resulting page is downloaded and rendered *by the browser*

On the server side of things, you want to write good code, make the database as efficient as possible, and throw in some caching. On the browser side, you want to limit how much data is transmitted, restrict the number of HTTP requests that have to be made, and watch how external resources such as JavaScript files are loaded.

The chapter covers these topics in detail, but just as there are different approaches for server performance and browser performance, there are different testing methodologies as well.

A somewhat technical but highly useful way of testing your site's performance is to use [ApacheBench](#). ApacheBench comes with the Apache web server and is used to benchmark a web server's performance. The most important information this tool returns is the number of requests per second (RPS) the server can handle.

Requests per second reflects many aspects of a server, from its processor speed and available memory, to how efficient the underlying code is. Simply put, more requests roughly equates to better performance. If your site does not perform well, the server won't be able to handle that many requests for your site, which means:

- Your users see long delays in loading pages
- Your site won't scale well
- Your site won't be able to handle traffic spikes well

The ApacheBench tool is run from a command-line interface. Assuming you have ApacheBench installed on your computer (which generally means you have Apache installed), you can use it with the command (note that you're testing some other server from your computer):

```
ab -n # -c # http://www.example.com/
```

{NOTE} You need to end your URL with a slash (unless it ends with a filename).

The `-n` flag indicates the number of requests to make. The `-c` flag dictates the number of *concurrent* requests to make. This is great because no matter how fast you are with your browser, you can't single-handedly make multiple simultaneous requests of a server.

If you look at the ApacheBench documentation (linked above or available through the `--help` command), you can also indicate a time limit, send POST data along with each request, and so forth.

As a basic test, you can see the performance of a site for 100 requests in groups of 5 (**Figure 17.1**):

The screenshot shows a terminal window titled "The Yii Book" running on Mac OS X. The command entered is "ab -n 100 -c 5 https://larry.pub". The output displays performance metrics for the website "larry.pub".

```

~ * ab -n 100 -c 5 https://larry.pub/
This is ApacheBench, Version 2.3 <Revision: 1887734 >
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking larry.pub (be patient).....done

Server Software:      Apache/2.4.7
Server Hostname:     larry.pub
Server Port:        443
SSL/TLS Protocol:   TLSv1.2,ECDHE-RSA-AES256-GCM-SHA384,2048,256
TLS Server Name:    larry.pub

Document Path:       /
Document Length:   18340 bytes

Concurrency Level:  5
Time taken for tests: 3.583 seconds
Complete requests: 100
Failed requests:  0
Total transferred: 1872680 bytes
HTML transferred: 1834080 bytes
Requests per second: 27.91 [#/sec] (mean)
Time per request: 179.150 [ms] (mean)
Time per request: 35.830 [ms] (mean, across all concurrent requests)
Transfer rate: 292.34 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:    115 137 58.1 124 373
Processing: 29 40 11.9 37 56
Waiting:    28 37 11.6 35 95
Total:     148 177 68.6 162 469

Percentage of the requests served within a certain time (ms)
  50% 162
  65% 167
  75% 171
  80% 174
  90% 185
  95% 333
  99% 459
 100% 469 (longest request)
~ * 

```

Figure 17.1: The ApacheBench test results for my website.

```
ab -n 100 -c 5 https://larry.pub
```

You'll get a bunch of information back, after a few seconds or minutes. You'll see the number of complete and failed requests, which is nice, but the requests per second (RPS) is the most common benchmark for a site's performance. In other words, this server can handle X number of requests for this URL per second. The more RPS, the better: it's that simple.

With this in mind, and as the plan at the end of the chapter explains, your basic performance testing sequence is:

1. Run `ab` to benchmark your initial RPS.
2. Make a tweak to improve the performance.
3. Re-benchmark the site to find the new RPS.
4. Repeat.



Figure 17.2: The Yii debugger.

The debugger extension, added in version 2 of the Yii framework, is a wonderful tool for basic profiling of your application. The debugger is enabled by default while in development mode. Out of the box, the debugger details:

- Log messages
- Total time it took the server to render and send the complete page
- Total memory required to render and send the complete page
- The number of database queries executed
- The total time it took the server to execute those database queries

You can access this all at the bottom of the browser window (**Figure 17.2**). Click any section of the debugger to see more details.

For basic performance testing, focus on the database, profiling, and timeline subpanels. This chapter returns to the first two; the timelines subpanel is simply a nice, visual graph of the page’s rendering.

The Yii framework has built-in profiling functionality whose results can be outputted to the debugger’s profiling subpanel. If you’ve peeked at the debugger, you’ll see this functionality is built-in and enabled when in development (**Figure 17.3**).

To delve further into a query’s execution, click the “Explain” link. The debugger will then show the results from running an EXPLAIN query on the database (**Figure 17.4**).

The EXPLAIN command tells MySQL to report how it would execute the query, including what indexes would be used, how many rows would be returned, and so forth. EXPLAIN is a good way to confirm that you’ve got the right indexes in place. You can also use it to experiment with queries to see if there’s not a better solution than the one you’re currently using.

Without using Yii, there are tools you can use to profile your database interactions, although these are a bit more technical and may require administrative control over your database. The most obvious is to use your database application’s “slow query” log. The slow query log records every SQL command that took longer than a specified amount of time to execute. For MySQL, the slow query log needs to be enabled and configured. For information on MySQL’s slow query log, see [this page of the MySQL manual](#).

Additional tools for database profiling differ by database application.

Another way to profile your web page is to see how quickly it loads in the browser. This can be further broken down into two areas:

Database Queries			
Time ↓	Duration	Type	Query
00:19:59.455	0.3 ms	SHOW	SHOW FULL COLUMNS FROM 'page' /Users/larry/Sites/yilcms/controllers/PageController.php:121 /Users/larry/Sites/yilcms/controllers/PageController.php:57
00:19:59.455	0.2 ms	SELECT	<pre> kcu.constraint_name, kcu.column_name, kcu.referenced_table_name, kcu.referenced_column_name FROM information_schema.referential_constraints AS rc JOIN information_schema.key_column_usage AS kcu ON (kcu.constraint_catalog = rc.constraint_catalog OR (kcu.constraint_catalog IS NULL AND rc.constraint_catalog IS NULL)) AND kcu.constraint_schema = rc.constraint_schema AND kcu.constraint_name = rc.constraint_name WHERE rc.constraint_schema = database() AND kcu.table_schema = database() AND rc.table_name = 'page' AND kcu.table_name = 'page' /Users/larry/Sites/yilcms/controllers/PageController.php:121 /Users/larry/Sites/yilcms/controllers/PageController.php:57 </pre> [+] Explain
00:19:59.457	0.1 ms	SELECT	<pre> SELECT * FROM 'page' WHERE 'id'=1 /Users/larry/Sites/yilcms/controllers/PageController.php:121 /Users/larry/Sites/yilcms/controllers/PageController.php:57 </pre> [+] Explain [+] Explain all

Figure 17.3: The time it took to execute the page's queries is reflected in the debugger.

id	select_type	table	partitions	type	possible_keys	key	key_len			ref	rows	filtered	Extra
							CONSTRAINT_SC	HEMA_TABLE_NA	ME				
1	SIMPLE	rc	NULL	ALL	NULL	CONSTRAINT_SC	NULL	NUL	NULL	UL	L		Using where; Open_full_table; Scanned 0 databases
1	SIMPLE	kcu	NULL	ALL	NULL	TABLE_SCHEMA,TABLE_NAME	NULL	NUL	NULL	UL	L		Using where; Open_full_table; Scanned 0 databases; Using join buffer (Block Nested Loop)

Figure 17.4: MySQL's explanation for how the query was executed.

- How quickly the data is received by the browser from the server
- How quickly the browser renders the whole page

The former will be impacted by a host of factors, including:

- Your server's performance
- The amount of data being transferred
- The applicable networks
- The use of a Content Delivery Network (CDN)
- Browser caching

You can test how quickly a live page loads on your server using any number of tools found online (via a quick Google search) or already in your browser (e.g., Opera Dragonfly or Chrome's Developer Tools).

How quickly the browser renders the page is impacted by:

- The browser in use
- The amount of data being transferred
- The number of HTTP request being made
- The complexity of the Document Object Model (DOM)
- Your use of JavaScript, CSS, and various media

Again, there are lots of resources online for profiling how quickly a page loads in the browser. I've always liked Yahoo!'s [YSlow](#), which provides both analytics and recommendations.

Now that you know how to test your site's performance, you can start learning the different ways you can improve it.

To begin, there are a couple of changes you can easily make that will impact your site's performance. First, you should make edits to the **index.php** bootstrap file. You should do two things there:

- Disable debugging
- Change the environment

The default **index.php** as included in the default application includes these two lines:

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

You would want to change that to just:

```
<?php  
defined('YII_ENV') or define('YII_ENV', 'prod');
```

The removal of the first line disables debugging. By default YII_DEBUG is set to false in a Yii application; the original line just overrides that. The change in the second line changes the environment from development to production. Both changes should improve the performance of the application and make it safer to deploy.

As a final reminder, these changes make sense when you're taking your site live and moving it to production. During the development of the site, making such performance-based changes is unwarranted and impedes development (e.g., by not reporting errors).

{WARNING} As with everything related to performance, the actual results of any particular change can vary from server to server, environment to environment. You should always benchmark and profile all changes to see the actual results for yourself.

Turning to improving performance in the configuration file, first look at the routing rules you've defined. Strive to employ as few route rules as possible. The application has to evaluate each rule for each request until a match is found, which can be expensive. Towards that end, you could also make sure the rules are ordered such that matches can be found as quickly as possible.

Second, look at the application components. By default, application components in Yii are created on demand. This means an application component may not be created at all if it is not accessed during a user request. As a result, the overall performance may not be degraded even if an application is configured with many components. Thus you don't need to go around removing components from your configuration file in the interest of improving performance.

The only exception to this behavior are *bootstrapped* components, such as "log":

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

As application component listed under "bootstrap" is loaded and exist through the entire request lifecycle. In a basic Yii application, the "log" component is bootstrapped by default, which makes sense. In Yii 2, the "debug" and "gii" components are additionally bootstrapped in the dev environment. That absolutely makes sense, but be judicious about what other application components are bootstrapped in the application.

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

Before looking at how to make the Yii application itself more performant, understand that you can improve your site's performance by upgrading aspects of the server itself. For starters, you should also run the latest, stable version of PHP, as great strides have been made in the language's performance over the years.

Second, enable bytecode caching with [OPcache](#). OPcache stores precompiled versions of scripts in memory, reducing the need for PHP to load and parse scripts per request.

Third, and similar to the first step, keep your other major software components up to date, starting with your:

- Web server application
- Database application
- Operating system

You could go far down this path, and may quickly eclipse your skill level and access, so decide what's most appropriate for your situation.

Perhaps the most potent tool available for improving the performance of your website is *caching*. Caching is a way of saving output for future use. For example, browsers automatically cache web page resources, such as images, CSS, and JavaScript. By caching those resources, when the visitor views another page that uses those same resources, the browser retrieves them from storage and doesn't need to download them again. Hopefully this is a concept with which you are already familiar.

The chapter returns to the topic of browser caching (and browser performance in general) later, but first focuses on server-side caching. There are several different server-side pieces that can be cached:

- Discrete pieces of data (e.g., variables)
- Fragments of pages
- Entire web pages
- Database schemas
- Session data

The next few pages explain how to cache each type, but first you need to know how to choose and enable a *caching engine*.

To make use of caching, one has to first enable it in the application. To do that, you must choose which caching engine to use. The Yii framework supports all the popular PHP caching tools:

- Alternative PHP Cache (APC)
- Memcached
- Redis
- WinCache
- Zend Data Cache

Note that these are all tools that must be installed on your server. Some, like APC, may be built into your PHP installation (run a `phpinfo()` script to confirm support for it). Others, such as Memcached, require a separate Memcached installation (and it needs to be running). Rather than discuss the pros and cons of the various caching tools in any detail, my recommendation is to start with what you have. For example, although APC may not be the best option for all situations, it's the most likely to already be installed.

{TIP} Most of the caching tools store data in memory. The more memory you have available for caching, the better the performance, in general.

There are also Yii classes for using a database or the file system as a caching mechanism. And because every one of these classes extends the `Cache` class, it's easy to use any tool you want for caching (such as technologies to later come out, or those of your own creation).

Once you've selected a caching mechanism to use, you need to enable and configure it in your primary configuration file. Here's how you would enable APC:

```
# web.php
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
]
```

The specific configuration properties that are available depend upon the caching mechanism used, and therefore the associated class. The `MemCache` class, for example, has a `servers` attribute that's used to point to the Memcached server instances:

```
# web.php
'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
    'servers'=> [
        [
            'host'=>'server1',
            'port'=>11211,
            'weight'=>60,
```

```
],
[
    'host'=>'server2',
    'port'=>11211,
    'weight'=>40,
],
],
],
```

How you configure any particular caching mechanism depends upon the mechanism in use, your server, the application, and so forth. This chapter gets you started down this path; subsequently, you should really learn about and master the caching tool of your choice.

The most atomic type of caching you can utilize is data caching. This is just the storing of a variable's value—or any piece of data—in the cache. To do this, first get a reference to Yii's cache:

```
$c = Yii::$app->cache;
```

Regardless of what caching mechanism you're using, the above line references Yii's cache.

Next, call the `set()` method to store some data in the cache. Its first argument is a unique identifier and its second is the value being stored:

```
$c = Yii::$app->cache;
$c->set('foo', 'bar');
```

That line stores the data in the cache without an expiration (although it could be removed if the caching mechanism itself clears it). To cache the data for a certain period, add a third argument, a time in seconds to persist:

```
$c = Yii::$app->cache;
$c->set('foo', 'bar', 60*60); // One hour
```

If you'd only like to store the data in the cache if it is not already cached, use `add()`. It takes the same three initial arguments as `set()`:

```
$c = Yii::$app->cache;
$c->add('foo', 'bar', 60*60); // One hour
```

To retrieve cached data, use the `get()` method, providing the same unique identifier:

```
$c = Yii::$app->cache;
$data = $c->get('foo');
```

If the data was no longer available, the value `false` is returned:

```
$c = Yii::$app->cache;
$data = $c->get('foo');
if ($data !== false) {
    // Use $data
} else {
    // Use non-cached version and re-cache.
}
```

As of Yii 2.0.11, the `getOrSet()` method can be used to retrieve cached data or, if none is present, create and cache the data. To use `getOrSet()`, have a function provided as the second argument determine and return the data to be cached:

```
$c = Yii::$app->cache;
$data = $c->getOrSet('foo', function () {
    // Dynamically determine what should be cached.
    // Return that cached data:
    return $cached_data;
});
```

If you'd prefer, you can use array syntax to set and get cached data:

```
$c = Yii::$app->cache;
$c['foo'] = 'bar';
echo $c['foo'];
```

You cannot set an expiration when using this approach, however.

To fetch multiple pieces of cached data, use `multiGet()`. It takes an array of identifiers to fetch and returns an array of key=>value pairs:

```
$c = Yii::$app->cache;
$c->set('x', 1);
$c->set('y', 2);
$c->set('z', 3);
$data = $c->multiGet(['x', 'z']);
echo $data['z']; // 3
```

For some caching mechanisms (e.g., Memcached and APC), using `multiGet()` can also result in better performance.

{NEW} The `mget()` method from Yii 1 has been renamed as `multiGet()`.

To remove data from the cache, invoke `delete()`, providing it with the identifier:

```
$c = Yii::$app->cache;  
$c->delete('foo');
```

To fine-tune the caching behavior, you can add *dependencies*. Dependencies are different types of criteria you can associate with cached data that determine when cached data is used as opposed to uncached data.

To add a dependency to a data cache, provide a fourth argument to `set()` or `add()`. This argument needs to be an object of a type of `yii\caching\Dependency`:

- `yii\caching\ChainedDependency`, dependent upon a list of other dependencies
- `yii\caching\DbDependency`, dependent upon the results of the provided SQL statement changing
- `yii\caching\DbQueryDependency`, dependent upon the results of the provided `yii\db\QueryInterface` changing
- `yii\caching\ExpressionDependency`, dependent upon the results of a PHP expression
- `yii\caching\FileDependency`, dependent upon the provided file having been changed
- `yii\caching\TagDependency`, dependent upon a provided tag

Three of these dependencies rely upon database or file sources. Three of these dependencies can be used to make more complex dependency scenarios.

As an example, you might cache an array that represents the files found in a specific directory. You would want this array to be recreated (and re-cached) if that directory changes. To pull that off, add the `yii\caching\FileDependency` to the caching of the variable:

```
$c = Yii::$app->cache;  
$dir = 'somedir';  
$pics = glob($dir . '/*.png');  
// Keep the array cached for a day  
// Or until the directory changes:  
$c->set('pics', $pics, 60*60*24,  
    new yii\caching\FileDependency($dir)  
)
```

If the contents of that directory changed since the array was stored in the cache, then the `get()` method will return false and you'd want to re-cache the data (using the code just demonstrated).

Later the chapter also shows some examples of the database caching, which can be very helpful in improving the performance of your site.

A broader type of caching than simple data caching is fragment caching. This is used to cache parts of a page. Fragment caching is performed differently than data caching. In fact, the approach is similar to using nested layouts as explained in Chapter 6, “[Working with Views](#)”.

Fragment caching is a breeze to implement. Within a view file, wrap the content fragment to be cached within `beginCache()` and `endCache()` calls. The former should be provided with a unique identifier:

```
<?php if ($this->beginCache('newsSidebar')) : ?>
<div id="sidebar">
<?php NewsWidget::widget(); ?>
</div><!-- sidebar -->
<?php $this->endCache(); ?>
<?php endif; ?>
```

In that hypothetical example, the “news” widget output is cached. By wrapping the `beginCache()` call in a conditional, the fragment is automatically retrieved from the cache if it exists, or generated and cached if it does not.

The `beginCache()` method takes an optional second parameter used to configure the caching behavior. This should be an array. The possible values are any public, writable attributes of the `yii\widgets\FragmentCache` class. The most important of these is `duration`, used to set the caching duration in seconds.

```
<?php if ($this->beginCache('newsSidebar',
    ['duration' => 60*60]) : ?>
<div id="sidebar">
<?php NewsWidget::widget(); ?>
</div><!-- sidebar -->
<?php $this->endCache(); ?>
<?php endif; ?>
```

As with data caching, you can add dependencies to fragment caching, although the syntax differs. Add a “dependency” index item to the customization array, passing key values to configure that dependency. This example makes use of the global variable dependency:

```
<?php if ($this->beginCache('promo',
    [
        'duration' => 60*60,
        'dependency' => [
            'class' => 'yii\caching\ExpressionDependency',
            'expression' => '$var == true'
        ]
    ])
) : ?>
<div id="sidebar">
<?php PromoWidget::widget(); ?>
</div><!-- sidebar -->
<?php $this->endCache(); ?>
<?php endif; ?>
```

With that code, the fragment cache is used until the duration passes or the hypothetical `$var` variable is no longer true.

Similarly, you can toggle the caching of a fragment more directly using the “enabled” property:

```
<?php if ($this->beginCache('promo',
    ['enabled' => true, ])) : ?>
```

Obviously you’d want to set the Boolean value based upon a logical condition, such as only caching GET requests:

```
<?php if ($this->beginCache('promo',
    ['enabled' => Yii::$app->request->isGet])) : ?>
```

Note that none of this code dictates whether content *is shown*, only whether the content might be cached.

To cache different states of content, set the “variations” property:

```
<?php if ($this->beginCache('promo',
    ['variations' => [Yii::$app->language]])) : ?>
```

That code creates a separate cached version of the fragment for each application language. Note that the value is always passed as an array.

You could also use “variations” and “enabled” to change when caching applies in other ways:

- When certain parameters are present in the URL

- Based upon the route
- By session (i.e., each unique session would have its own fragment caching)
- Using a custom expression of your choosing

All of these possibilities, and the corresponding methods, are detailed in the Yii manual.

A more expansive type of caching than fragment caching is page caching. This is, as you would imagine, the caching of an entire rendered page. To cache an entire page, you need to add the `yii\filters\PageCache` class as a behavior to the controller:

```
# controllers/SomeController.php
public function behaviors() {
    return [
        [
            'class' => 'yii\filters\PageCache',
            'duration' => 60 * 60,
            'variations' => [
                \Yii::$app->language,
            ],
        ],
    ];
}
```

{TIP} To confirm that page caching is working, check out your profile summary report to see if database queries are still being run (assuming that page normally runs queries).

As with any filter, you likely want to limit the filter to only certain actions (i.e., pages). You'd likely want to cache a “view” or “index” action, but not “add” or “update”:

```
# controllers/SomeController.php
public function behaviors() {
    return [
        [
            'class' => 'yii\filters\PageCache',
            'only' => ['index', 'view'],
            'duration' => 60 * 60,
            'variations' => [
                \Yii::$app->language,
            ],
        ],
    ];
}
```

The above performs server-side caching. The Yii framework only re-executes the applicable controller actions once the cache has expired. An alternative is to make use of browser-side caching. This is simply a matter of indicating caching recommendations to the user's browser, giving it the information it needs to do what it'd do anyway.

Browser caching is controlled (or to be more precise, informed) by a couple of things. The two most important are:

- The Last Modified header
- Entity Tags (aka, ETags)

The Last Modified header is a simple indicator of when the content last changed. The browser can use this to know whether the cached version should be displayed (if it exists) or a version should be pulled from the server.

ETags are server-provided unique identifiers for a resource. By associating an ETag with a page, an update to the page results in a new ETag. The browser can then see if the ETag for the version in cache matches the ETag for the resource being requested. If so, the cached version is used. Otherwise, the updated version is be pulled from the server and the ETag updated.

You can manually manage both the Last Modified header and the ETags for any page, but once again Yii provides a nice tool for doing this automatically. The `yii\filters\HttpCache` class can be assigned as a filter to your controller and sends the Last Modified and ETag headers for you.

```
# controllers/SomeController.php
public function behaviors() {
    return [
        'class' => 'yii\filters\HttpCache',
        'only' => ['view'],
        'lastModified' => $timestamp,
    ];
}
```

The `lastModified` value can be a Unix timestamp or a human-readable date. To dynamically determine this value, you might query your database (depending upon the underlying table structure):

```
# controllers/SomeController.php::behaviors()
return [
    'class' => 'yii\filters\HttpCache',
    'only' => ['view'],
    'lastModified' => function ($action, $params) {
        $q = new \yii\db\Query();
```

```
        return $q->from('tableName')->max('date_updated');
    },
];
```

To also set an ETag, provide a value to the `etagSeed` property. This value can be any string or array that will properly identify the state of the resource. Returning a page's title and content are logical choices:

```
# controllers/SomeController.php::behaviors()
return [
    'class' => 'yii\filters\HttpCache',
    'only' => ['view'],
    'etagSeed' => function ($action, $params) {
        $page = $this->findModel(\Yii::$app->request->get('id'));
        return serialize([$page->title, $page->content]);
    },
];
```

When you're using page caching (and less often, fragment caching), you'll sometimes have parts of the page that shouldn't be cached. For example, maybe your whole home page should be cached except for:

- The snippet that greets a logged-in user by name
- The widget that displays the Twitter feed
- Any other element that's highly variable and/or time sensitive

To prevent Yii from caching some content, flag the content as dynamic. Doing so requires changing your views a bit. To start, among some content that is otherwise being cached, insert some dynamic content using the `renderDynamic()` method:

```
<!-- HTML page being cached. -->
<div class="span-5 last">
<div id="sidebar">
<?php echo $this->renderDynamic('$this->getNews()'); ?>
<!-- sidebar --></div>
<!-- Rest of the HTML page being cached. -->
```

The `renderDynamic()` method takes a single argument: PHP code to be executed whose returned value is the dynamic content. The easiest way to provide this value is as a method in the current controller. Thus, using the above code, the current controller should have a method named `getNews` that returns the dynamic content:

```
# controllers/AnyController.php
public function getNews() {
    // return dynamic content
}
```

Another type of caching you can add to your site to improve its performance is session caching. Session caching isn't quite the same kind of caching as data, fragment, or page, but rather an easy way to move the storing of session data from an inefficient methodology to a more efficient one.

By default, PHP stores session data in plain text files in a common, world-writable directory on the server. The file system interactions required—all of those reads and writes—are slow. Moreover, on a shared server, session data can be compromised due to its publicly-available status in a common directory. A solution to both problems is to use session caching in Yii. With this enabled, instead of storing the data in the file system, it's stored in the caching mechanism. This can have a nice performance benefit.

To enable session caching, configure the “session” component in your primary configuration file:

```
# config/web.php
[
    // Other stuff.
    'components'=>[
        'session'=>[
            'class'=>'yii\web\CacheSession',
            'cache' => 'sessionCache',
        ],
        'sessionCache' => [
            'class'=>'yii\caching\RedisCache',
            'hostname'=>'localhost',
            'port'=>6379,
            'database'=>0,
        ],
    ],
    // More other stuff.
]
```

First, the “session” component is configured to use the `yii\web\CacheSession` class, instead of the default class. This configuration is also assigned an identifier of “`sessionCache`”. In the next bit of code, this identifier is used to tell Yii to use Redis caching for that specific cache type.

{TIP} Redis is the recommended caching engine for session data.

If you have an even larger Yii application that runs on multiple servers, you can use Memcached for your sessions caching. Assuming that Memcached is running on one common server, by storing all your session data there, you also end up resolving the issue of supporting user sessions across multiple web servers.

As a rule of thumb, file system interactions tend to be the biggest bottlenecks in any site's performance. And a database, of course, is essentially a managed file system. Finding ways to improve the performance of your database interactions go a long way towards improving the overall performance of your site.

Improving the performance of your database interactions comes down to:

- Preventing the execution of any query that you can avoid
- Improving the execution of any queries that you must run

The first thing to do is use profiling and logging to see what queries are being run on a page and how long each is taking to execute. That provides a baseline to begin with (see Figures 17.3 and 17.4).

Next, make sure that you're only running the queries you absolutely must. It's unlikely that you have unnecessary queries, but sometimes you slipped in your logic and added a process that wasn't necessary.

{TIP} Caching pages or fragments will also reduce the number of queries being executed.

The third step is to examine how you might make a query more efficient. For non-Active Record queries, this can be a matter of changing what columns are selected, how many rows are returned, how JOINs are performed, and so forth.

For Active Record queries, it is possible to limit what columns are selected, but doing so undermines much of the point of using AR in the first place. You may want to consider switching from AR to Query Builder or Direct Access Objects (DAO) for straight-up SQL. You'll see the biggest benefit to dropping AR when you have complex, demanding queries. The pro's and con's of AR vs. Query Builder vs. DAO were explained in Chapter 8, “[Working with Databases](#)”.

Those are some general recommendations for improving your database performance. Let's look at some database design thoughts, and then move to specific Yii features that help here.

{TIP} If you have administrative control over your MySQL installation, tweaking its configuration can also greatly improve performance.

A lot of application performance comes down to just employing best practices:

- Don't create variables you don't need.

- Watch how many files you include.
- Simplify logic and execution flow.
- And so forth.

This is especially true with your database. Proper database design goes a long way towards improving your site's performance. Hopefully you've made the right decisions well before this point, but as a reminder, here are some best practices.

First, try to use numbers whenever possible. Databases (and, well, pretty much all computer programs) work with numbers faster than they do strings. So when you have a choice, go with a number (e.g., a zip code).

Second, make good use of indexes. To start, you should index columns that are any of the following:

- The primary key
- Frequently used in WHERE clauses
- Frequently used in ORDER BY clauses
- Frequently used as the basis for a JOIN

You should *not* index columns that:

- Allow NULL values
- Have a very limited range of values (such as Y/N or 1/0)

Again, watch the slow query log and use EXPLAIN to identify and rectify problematic queries.

Third, higher-end and more active applications tend to move away from a strictly normalized structure. Normalization is great for data integrity but bad for performance (JOINS are especially costly). A performance solution is to start breaking normalization and creating tables that perform better.

For example, you might have a `user` table that stores everything about a user. Each time a user logs in, or any time any query references that table, all extraneous information stored in the table—the user's address, gender, whether the user likes watermelon or not, whatever—impacts the performance of that query, even when that information is not referenced by the query itself. The fix in such cases is to put the bare minimum required information in the primary `user` table (e.g., username, email address, password, user ID, and registration date), and move everything else to another table. Again, multiple queries are required to retrieve all the information, but multiple smaller queries can execute faster than single larger queries. You'll also find that all queries run better when large text and binary columns are moved to their own separate tables.

If you do this right, the first table could have SELECT queries performed using any column (e.g., the email address or the username), but the secondary tables would

always use `SELECT` queries off of foreign keys: the FK related to the PK from the original table.

Of course, the downside to this is that it'll wreak havoc with your AR models.

Similarly, use database view tables to represent complex, relational data sets. Again, you won't be able to use AR with views, but you can use DAO effectively with these.

Active Record is a lovely design but isn't so kind when it comes to performance. For the framework to do anything with a particular model, it must first run two queries:

- `SHOW FULL COLUMNS FROM tablename`
- `SHOW CREATE TABLE tablename`

Again, to be clear, any use of, say, an `employee` table requires these two queries be run first. If you have the related `Employee` and `Department` models, a `JOIN` across them requires four queries before any data is retrieved at all!

This is the downside of Active Record: it does a lot of the work for you, but at a cost of performance. You can quickly and easily improve your Yii application's performance by limiting how often these basic queries are run. To do so, you'll need to cache the database schema (i.e., tell Yii to remember what columns exist in the tables).

Caching the database schema is done in the `db.php` configuration file:

```
<?php
return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
    'enableSchemaCache' => true,
    'schemaCacheDuration' => 604800, // One week,
    'schemaCache' => 'cache',
];
```

Three configuration lines are added, although they may be present (but disabled) in the default application. Assign to `schemaCachingDuration` a logical value, such as an hour or a week (the value is in seconds).

That configuration line change enables caching of the database schema but only if Yii has a caching component registered. Ensure that's also being done within the main configuration file:

Database Queries

Total 1 item.

Time ↓	Duration	Type	Duplicated	Query
23:35:25.692	0.2 ms	SELECT	1	SELECT * FROM `page` WHERE `id`='1' /Users/larry/Sites/yii/cms2/controllers/PageController.php:121 /Users/larry/Sites/yii/cms2/controllers/PageController.php:56 [+] Explain

Figure 17.5: Thanks to schema caching, two queries are no longer necessary.

```
'cache' => [
    'class' => 'yii\caching\FileCache',
],
```

To confirm schema caching is working, look at your web profiling log for any page that uses a model (**Figure 17.5**). The two queries mentioned earlier—and shown in Figure 17.3—should not be run (after you’ve reloaded the page to cache the results once, that is). This is very important: if you follow these steps and don’t see the SHOW commands disappear from the web profiling log, the caching isn’t working.

After you’ve had Yii cache the database schema, you may want to consider having Yii cache the results of specific database queries. This concept is much the same as the data caching explained already.

The first thing you’ll need to do is ensure that caching is enabled in your configuration. Then, call the `cache()` method when executing any query:

```
$result = $db->createCommand($q)->cache(60*60)->queryAll();
```

Alternatively, you can provide to the `cache()` method a function that executes a query:

```
$result = $db->cache(function($db) {
    $q = 'SELECT * FROM table_name';
    return $db->createCommand($q)->queryAll();
});
```

This is similar in construct to the `getOrSet()` method. You define a function that returns data. If there are cached results, those are returned instead. If no cached results exist for that SQL command, Yii actually executes the query to get the results, and then caches those.

When used like so, the `cache()` method takes an optional second argument, which is the duration:

```
$result = $db->cache(function($db) {
    $q = 'SELECT * FROM table_name';
    return $db->createCommand($q)->queryAll();
}, 60*60);
```

That example uses DAO, but you can cache AR queries, too. Again call `cache()`, passing it a function definition.

```
$page = Page::getDb()->cache(function($db, $id) {
    return Page::find()->with('pageUser', 'pageComments')
        ->where(['id' => $id])->all();
});
```

This query fetches a page, along with the page's author, and all the comments posted on that page, along with the authors of those comments (this is from Chapter 8).

There are two more things you ought to know about query caching. First, you can use durations and dependencies with query caching, just as you can with data caching. For example, if a query retrieves the latest pages in a CMS example, you may want to cache those results for a certain amount of time or until a new page is added (whichever comes first):

```
$dep = new \yii\caching\DbDependency('SELECT
    MAX(date_published) FROM pages');
$page = Page::getDb()->cache(function($db, $id) {
    return Page::find()->all();
}, 60*60, $dep);
```

The second thing to know is that, by default, the cache applies to every query executed within the function. If that's not appropriate, you can use `noCache()` and another anonymous function to stop caching of specific queries. See the Yii documentation for details.

Since you can implement caching in so many different ways, a common question is whether you can use different types of caching mechanisms for different purposes. For example, you might want to use Memcached for your sessions, data, schema, and query caching, but APC or file caching for your fragment and page caches. This is definitely possible and easily done in Yii 2!

The trick here is the cache ID. By default, Yii uses a cache ID of "cache" for all caching mechanisms. This means that anything that uses a cache—data, queries, pages, etc.—has, by default, a cache ID of "cache". This cache is configured like so:

```
# web.php
'components' => [
```

```
'cache' => [
    'class' => 'yii\caching\ApcCache',
],
```

To use a different caching mechanism for anything, simply assign it a different caching ID. For example, you might store the database schema in Memcached, but have APC be the default cache. Here's how that configuration file would look:

```
# web.php
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
    'schemaCache' => [
        'class' => 'yii\caching\MemCache',
    ],
]
```

If you want to use a different caching mechanism for the query cache, you'd assign a unique identifier to the `schemaCache` property of the "db" component.

```
# db.php
<?php
return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yiobook2_cms',
    'username' => 'root',
    'password' => 'password',
    'charset' => 'utf8',

    // Schema cache options (for production environment)
    'enableSchemaCache' => true,
    'schemaCacheDuration' => 60,
    'schemaCache' => 'schemaCache',
];

```

The focus thus far in the chapter has been on improving the performance on the server-side of things. But the server is only one side of the equation, all sites should optimize the browser experience as well. Doing so when using Yii involves largely the same techniques as you'd use on any site, although once again there are a couple of Yii-specific tools you can utilize. Naturally the chapter focuses on the Yii-specifics, but some general browser improvement tips are covered first.

With respect to the browser, the general goal is to reduce the amount of data being transmitted to the browser and the number of HTTP requests required. The former is accomplished by:

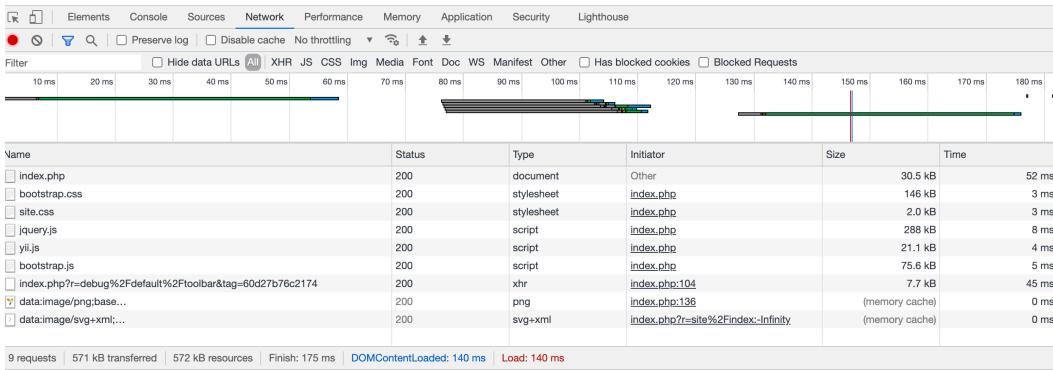


Figure 17.6: Viewing the number of required requests using the browser tools.

- Optimizing your images
- Reducing the amount of HTML
- Minifying your HTML, JavaScript, and CSS

For lots of concrete recommendations, check out the excellent resource “[How to lose weight \(in the browser\)](#)”. That resource also explains the best places to put your JavaScript and CSS references within the HTML document (it matters).

The second task—reducing the number of requests required—can be accomplished by:

- Using CSS sprites
- Combining multiple CSS files into one
- Combining multiple JavaScript files into one

With these last two concepts, Yii can help.

Minimizing the number of HTTP requests that a browser has to make—how many resources it must download—can have a significant impact on how quickly the page renders. You can minimize the number of requests by using CSS sprites to combine lots of your images. On a more advanced level, you can also minimize the number of *initial* requests by having some resources be downloaded after the page has been rendered. But for the JavaScript and CSS resources that are needed by the page from the outset, the only solution is to use but a single file for each type (i.e., one JavaScript file and one CSS file).

As an example, **Figure 17.6** shows the number of requests required to view a page using the standard Yii template. There are 8 requests for that page alone, without any media whatsoever!

Other pages might have extra JavaScript files to be included.

To start minimizing these requests, you might take your base JavaScript files—

- **menus.js**

- **forms.js**
- **llamas.js**

—and combine them into one called **all.js**. (You would not normally combine your custom JavaScript files into your jQuery or other libraries.)

And you'd do the same with your CSS files, combining these into **all.css**:

- **screen.css**
- **print.css**
- **main.css**
- **forms.css**
- **styles.css**

Now you'll do this in your layout:

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

By taking these steps, you've reduce six or eight or X HTTP requests down to two. That's great! But...

Unfortunately, you're using a widget in your sidebar that requires its own CSS file, **widget.css**. And some pages use an extension that requires its own **ext.js** and **ext.css** files. Now you're back up to 5 requests again! Fortunately, there's an easy solution.

New in Yii 2 is the concept of an **asset bundle**. An asset bundle provides a mechanism for managing—and efficiently serving—assets like CSS and JavaScript.

The default, basic Yii application defines one asset bundle for you:

```
# assets/AppAsset.php
<?php
namespace app\assets;
use yii\web\AssetBundle;
class AppAsset extends AssetBundle {
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.css',
    ];
    public $js = [];
    public $depends = [
        'yii\web\YiiAsset',
```

```
    'yii\bootstrap\BootstrapAsset',
];
}
```

This tool is then registered in the a layout file:

```
# views/layouts/main.php
use app\assets\AppAsset;
AppAsset::register($this);
```

That's all that's required! Any required CSS or JavaScript will automatically be inserted into the HTML output **in an efficient manner**.

To make use of this in your site, add your CSS and JavaScript files to the `AppAsset` configuration:

```
# assets/AppAsset.php
public $css = [
    'css/site.css',
    'css/forms.css',
    'css/print.css',
];
public $js = [
    'js/menus.js',
    'js/forms.js',
    'js/llamas.js'
];
```

{NOTE} You can use absolute URLs for the JavaScript files, such as those hosted on Google APIs. But by doing that you undercut whatever caching the browser may already be doing of these common JavaScript libraries.

Now when a page is created using the main layout, the assets are bundled and published as a single file:

```
<link href="/assets/4d97dc6/css/styles.css" rel="stylesheet">
<script src="/assets/78c792d7/yii.js"></script>
```

The asset bundle is a great addition to Yii 2, and it'll be discussed more in Chapter 19, “Extending Yii”.

Now that you've been presented with a whole host of possible performance improvements, how should you proceed? One recommendation before getting into specifics

is that you avoid *premature optimization*. The goal while developing a site should be to get it working while also employing best practices. In fact, many best practices have performance benefits anyway. Once the functionality is in place, and the site is almost ready to go live, then you can implement some of these other performance approaches, such as caching.

Here's a step-by-step approach:

1. Develop the site using best practices, completing all necessary functionality and appearance.
2. Test the site extensively, revise and tweak as needed, and get approval from the client (if applicable).
3. Reduce your JavaScript code into a reasonably small number of minified files.
4. Reduce your CSS code into a reasonably small number of minified files.
5. Register your CSS and JavaScript files with the asset bundler.
6. Profile the site, in detail, on its destination server, or an extremely comparable development server.
7. Identify the caching mechanisms available.
8. Enable and configure the caching mechanisms, if applicable.
9. Change the bootstrap file, disabling debugging and setting the environment to production.
10. Enable logging and profile logging in your configuration file.
11. Identify, profile, and optimize your database queries. Repeat as needed.
12. Re-profile the site, adjusting as necessary.
13. Test, test, test!
14. Deploy!

Note that you'll profile your site in many ways, depending upon what, exactly, you're examining. Sometimes you'll use ApacheBench, other times a browser-based tool, and still other times Yii's built-in profiling tools.

Chapter 18

Advanced Database Issues

Although the book has covered and explained lots of database-related issues when it comes to the Yii framework, there are still some remaining. This chapter discusses a smattering of topics that are either more advanced than most, or just more esoteric. Because these are more advanced and unusual conditions, they cannot all be explained in absolute, concrete detail. But the chapter does provide as much information as possible in terms of the problem and solution.

Whether it's a matter of bug fixes, or adding features, sites are rarely absolutely done, and there's always code that could be updated. Adding code changes can be done with relative ease in Yii, particularly when also making use of version control, unit testing, and other techniques that help ensure smooth transitions.

Some site changes, however, require changes to the underlying database, too. Normally these are *additive* changes:

- Adding one or more tables
- Adding one or more columns to existing tables
- Adding one or more indexes to existing tables
- Adding records to existing tables (this is less common)

Obviously you can make these changes merely by executing the proper SQL commands. But that approach has a couple of downsides. First, it's a manual step, disassociated from the application, meaning that you'd have to remember to do this when going back and forth from development to production servers. Second, with straight SQL commands, there are no inherent ways to revoke changes, or to only implement only part of the proposed changes.

The solution to these problems is to use *database migration*. Database migration is a system for managing database changes: defining changes, applying them, revoking them, and viewing them.

In Yii, database migration is handled through the command-line `yii` script. Before continuing, make sure you know how to execute this script:

```
~/S/y/cms » ./yii migrate/create create_downloads_table
Yii Migration Tool (based on Yii v2.0.13.1)

Create new migration '/Users/larry/Sites/yii/cms/migrations/m210623_140707_create_downloads_table.php'? (yes|no) [no]:yes
New migration created successfully.
~/S/y/cms »
```

Figure 18.1: Creating a new migration.

```
cd /path/to/app
yii <command>
```

Depending upon your operating system, setup, and so forth, you may need to use a variation on that specific command, such as:

```
cd /path/to/app
./yii <command>
```

Second, you may need to create a **migrations** folder, if one does not exist already. That folder also needs to be writable by the web server:

```
cd /path/to/app
mkdir migrations
chmod 0777 migrations
```

If the permissions were properly set when the application was created, manually creating this folder may not be necessary. If you want, you can try the steps on the next page and come back to this page if something doesn't work.

In Yii 1, a third step would be to tweak the **config/console.php** file for your database, but as of Yii 2, the console configuration file includes the **db.php** configuration already.

New migrations—database changes—are created using the syntax:

```
yii migrate/create <name>
```

The name should be clearly meaningful, but can only contain letters, numbers, and the underscore (for reasons to be explained shortly).

As an example, let's say that you've later decided to add a **downloads** table to the CMS project to track the number of times a file has been downloaded (**Figure 18.1**):

```
yii migrate/create create_downloads_table
```

Then answer “yes” at the prompt. You should see that the new migration was created successfully.

This command creates a new file within the **migrations** directory. The file will be named **mTIMESTAMP_NAME**, with the timestamp in the format *yyymmdd_hhmmss*, and the name coming from the value provided to the `create` command. In the `downloads` migration example, this file might be **m210623_140707_create_downloads_table.php**.

The file defines a class that has the same name as the file itself, in keeping with Yii’s conventions.

The class contains two methods: `up()` and `down()`. These methods define the actions to be taken when invoking the migration and revoking it, respectively.

If your database application supports transactions—the ability to have a number of SQL commands all work or all be reverted—the class uses the names `safeUp()` and `safeDown()` for these two methods. The next several pages go into the details of these methods, whether you have `up()` or `safeUp()`, `down()` or `safeDown()`.

The `up()` method is where the migration changes are executed: creating tables, modifying them, possibly adding records, and so forth. This is not done using the traditional, Yii approaches—Active Record or Query Builder, but instead using the `yii\db\Migration` class.

Looking at a generated migration class file, you’ll see that it extends `Migration`. This class defines a couple dozen methods for interacting with the database. For example, if the migration needs to add an index, you’d call the `createIndex()` method (**Figure 18.2**):

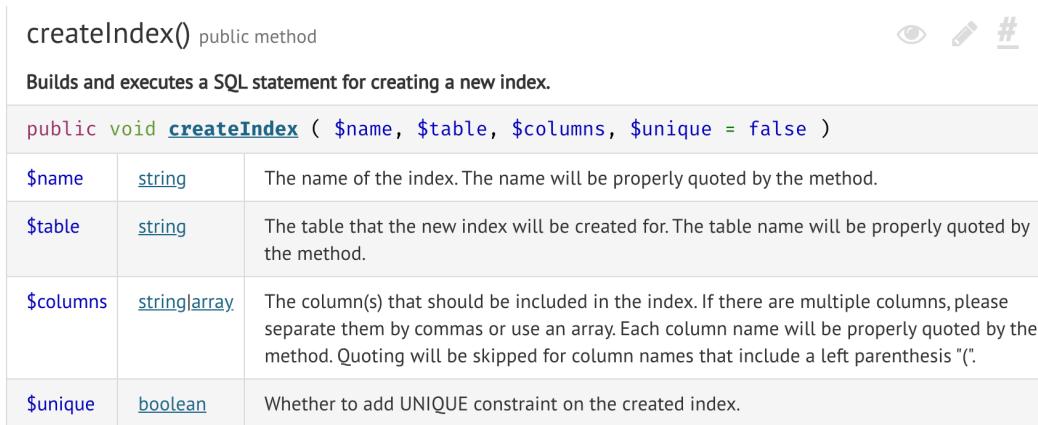
```
# migrations/something.php
public function up() {
    $this->createIndex('login', 'user', 'email,pass');
}
```

That’s all there is to it! When the migration is run, the `up()` method is executed, and that line will create an index named “login” on the `email` and `pass` columns of the `user` table.

The corresponding `down()` method would look like this:

```
# migrations/something.php
public function down() {
    $this->dropIndex('login', 'user');
}
```

To add a column, use the `addColumn()` method:



createIndex() public method

Builds and executes a SQL statement for creating a new index.

public void <code>createIndex</code>	(\$name, \$table, \$columns, \$unique = false)	
\$name	<code>string</code>	The name of the index. The name will be properly quoted by the method.
\$table	<code>string</code>	The table that the new index will be created for. The table name will be properly quoted by the method.
\$columns	<code>string array</code>	The column(s) that should be included in the index. If there are multiple columns, please separate them by commas or use an array. Each column name will be properly quoted by the method. Quoting will be skipped for column names that include a left parenthesis "(".
\$unique	<code>boolean</code>	Whether to add UNIQUE constraint on the created index.

Figure 18.2: The method definition for creating indexes.

```
$this->addColumn('someTable', 'someCol', 'someType');
```

All of the necessary methods can be found in the [Migration class documentation](#) or the Yii guide's discussion of [Working with Database Schema](#).

One trick to remember is Yii uses constants to represent common types, which makes the migration less database-dependent. These constants are defined in `yii\db\Schema`. For example, to add a text column, use:

```
$this->addColumn('someTable', 'someCol', Schema::TYPE_STRING);
```

That line actually creates a `VARCHAR(255)`. (Note that this assumes you're using MySQL; with other database applications, the results may differ.)

If you want to add additional characteristics, you can through concatenation :

```
$this->addColumn('someTable', 'someNum', Schema::TYPE_INTEGER .
    ' UNSIGNED NOT NULL';
$this->addColumn('someTable', 'someTS', Schema::TYPE_TIMESTAMP .
    ' NOT NULL DEFAULT CURRENT_TIMESTAMP');
```

Since this is simple concatenation of strings, be certain to use an initial space in the additional characteristics string.

The `up()` method has a corresponding `down()` method that reverts the changes made by a migration (when possible). For an add column example, the corresponding `down()` method would contain:

```
$this->deleteColumn('someTable', 'someCol');
```

The `createTable()` method is used to make a new table. Its first argument is the table name; its second is an array of column names and types; and, its third is for any additional SQL, such as setting the storage engine or adding indexes:

```
$this->createTable('downloads', array(
    'id' => Schema::TYPE_PK,
    'file_id' => Schema::TYPE_INTEGER . ' UNSIGNED NOT NULL',
    'count' => Schema::TYPE_INTEGER . ' UNSIGNED NOT NULL',
    'last_downloaded' => Schema::TYPE_TIMESTAMP . ' DEFAULT CURRENT_TIMESTAMP',
));
```

If you wanted, you could then add a foreign key constraint (using the `addForeignKey()` method). There are also methods for renaming columns, inserting records, updating records, deleting records, truncating tables, and more. See the documentation for the `Migration` class for details.

An alternative to using the contents with concatenation is to use `SchemaBuilderTrait` methods. This code has the same effect as the above:

```
$this->createTable('downloads', array(
    'id' => $this->primaryKey(),
    'file_id' => $this->integer()->unsigned()->notNull(),
    'count' => $this->integer()->unsigned()->notNull(),
    'last_downloaded' => $this->timestamp()->defaultValue('DEFAULT CURRENT_TIMESTAMP'),
));
```

Whether you use the constants or the methods, both provide a database-agnostic way of handling the migration seamlessly.

{TIP} As of Yii 2.0.7, the `migrate/create` command is capable of taking several arguments that generate your migration actions for you. See the [Generating Migrations](#) documentation for more.

Once you've defined a migration, you implement it—execute its `up()` method—using the `yii migrate` command (**Figure 18.3**):

```
yii migrate
```

That command shows a list of new migrations to apply (it creates, and uses, a `migration` table in the database to track this information). Just type “yes” and press Return/Enter to apply them.

```
[~/S/y/ch17d » ./yii migrate
Yii Migration Tool (based on Yii v2.0.42.1)

Creating migration history table "migration"...Done.
Total 1 new migration to be applied:
    m210623_141328_create_downloads_table

Apply the above migration? (yes|no) [no]:yes
*** applying m210623_141328_create_downloads_table
    > create table {{%downloads}} ... done (time: 0.035s)
*** applied m210623_141328_create_downloads_table (time: 0.039s)

1 migration was applied.

Migrated up successfully.
```

Figure 18.3: Applying migrations.

The migrations are executed in timestamp order, from oldest to youngest.

If you have a lot of migrations queued up and only want to implement a subset of them, you can limit what migrations are invoked in a couple of ways.

One option is to use the `migrate` command, followed by a number, to indicate how many of the available migrations (from oldest to newest) should be applied:

```
yii migrate 1
```

Or you could tell Yii to migrate up until (and including) a specific timestamp:

```
yii migrate/to 130813_152544
```

To view the list of new migrations available, execute:

```
yii migrate/new
```

To view the list of migrations that have already been executed, use:

```
yii migrate/history
```

Both the `history` and `new` commands take numeric limit arguments restricting how many implemented or upcoming migrations are listed.

If you use migrations a lot, you may want to look at the Yii Guides instructions for [customizing the migration command](#).

If it was a mistake to implement a migration, you *may* be able to revert it. The actual revocation would be defined in the migration's `down()` method. That method could drop a table, column, or index that was created by the corresponding `up()` method.

To execute the `down()` method, use the `down` command:

```
yii migrate/down
```

That command revokes the previous migration. If you pass a numeric argument after `down`, that many previous migrations will be revoked. The undoing of migrations is in reverse chronological order: from most recent (i.e., newest) to the oldest.

Understand that not all migrations can be undone, although that's really up to you. By definition, if the associated migration class's `down()` method returns false, as in the default generated code, the migration is considered to be un-revokable. A common example would be a migration that has subtractive results: removes data by deleting records, removes columns, or drops tables. Although you can recreate columns and tables, there's no way to repopulate removed data.

If something didn't quite go right with a migration, you can edit the migration class file and then redo the migration:

```
yii migrate/redo 1
```

That line first revokes the most recent migration and then re-implements it.

some queries auto-commit

Note that usually when you perform multiple DB operations in `safeUp()`, you should reverse their execution order in `safeDown()`. In the above example we first create the table and then insert a row in `safeUp()`; while in `safeDown()` we first delete the row and then drop the table.

A topic not previously covered but worth a couple of pages are *stored procedures*. There are huge benefits to using stored procedures, including improved security and performance. On the other hand, not everyone is comfortable using stored procedures, and many cheaper hosting companies won't let you use them anyway. But if you are comfortable with and capable of using stored procedures, rest assured that you can use them in your Yii-based site, too.

To call a stored procedure in MySQL from a regular PHP script, you execute a query along the lines of `CALL procedure_name(arg1, arg2)`. This assumes that the stored procedure `procedure_name` has already been created in the database, of course.

Therefore, to call a stored procedure from Yii, you'd execute that same query. To execute queries directly, you use Direct Access Objects:

```
$q = 'CALL procedure_name(arg1, arg2)';
$cmd = Yii::$app->db->createCommand($q);
$cmd->execute();
```

Naturally, you'll want to toss in bound parameters to pass values to the call:

```
$q = 'CALL procedure_name(:arg1, :arg2)';
$cmd = Yii::$app->db->createCommand($q)
$cmd->bindParam(':arg1', $some_var, PDO::PARAM_STR);
$cmd->bindParam(':arg2', $other_var, PDO::PARAM_INT);
$cmd->execute();
```

Those are examples of *inbound parameters*. If you're using *outbound parameters* in your stored procedure, there's an extra step required. In this next example, the stored procedure (theoretically) stores some information in the `@val` SQL variable. The PHP script then just needs to select this variable in order to access the value:

```
$q = 'CALL procedure_name(:arg1, :arg2, @val)';
$cmd = Yii::$app()->db->createCommand($q);
$cmd->bindParam(':arg1', $some_var, PDO::PARAM_STR);
$cmd->bindParam(':arg2', $other_var, PDO::PARAM_INT);
$cmd->execute();
$result = Yii::$app()->db->createCommand('SELECT @val')
->queryScalar();
// Use $result
```

And that's all there is to that!

Again, stored procedures are just a somewhat unique type of query that you'd execute using DAO. Other than that, most of this syntax and behavior comes from PHP's PDO, so see those documentation pages if you have any questions. And also see the MySQL documentation if you're unfamiliar with stored procedures in general.

For most database applications, related tables have a simple relationship between one primary key column in Table A and one foreign key column in Table B. Once you've identified the relationship in your models, Yii can easily pull together related data as needed.

Sometimes tables have more complex relationships, normally because the primary or foreign keys are comprised of more than one column. These are called either *composite* or *compound* keys, depending upon the nature of the underlying keys. Yii is capable of supporting these situations, you just need to tell Yii more about the relationships.

Chapter 5, “[Working with Models](#),” explains how to identify standard relationships:

```
# 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']);
}
```

That chapter also explains how to use intermediary tables:

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

That code applies in situations where you have a many-to-many relationship between Table A and Table B (such as `page` and `file` in the above). The intermediary Table C (e.g., `page_has_file`) has a compound primary key but Table A and Table B have single column foreign keys.

In short, although there's a bit of complexity in that database design, it's relatively not that complex.

Let's now look at how you handle situations where you either have complex foreign keys or complex primary keys, but aren't using an intermediary table.

{TIP} Some people distinguish between compound and composite keys, others don't. In theory, a compound key contains two or more columns, all of which relate to other tables. A composite key contains two or more columns, but at least one column is not related to another table.

As an example of a composite primary key, let's say there's an application that tracks when a site's users have accessed site content (e.g., pages), and perhaps even what rating the user gave that page:

```
CREATE TABLE reading (
    user_id INT(10) UNSIGNED NOT NULL,
    page_id INT(10) UNSIGNED NOT NULL,
    PRIMARY KEY (user_id, page_id)
);
```

The `reading` table has compound primary key: the combination of the user ID and the page ID. Note that this design assumes that only a visited/not visited state is being recorded, not the number of times a user has visited a specific page.

The `Reading` model needs to override the `primaryKey()` method to reflect its compound primary key:

```
# models/Reading.php
public function primaryKey() {
    return ['user_id', 'page_id'];
}
```

This overrides the default `primaryKey()` method, which returns `id` as the primary key column. But this is all you need to do to identify a complex primary key in a model.

The relations in all the models would still be defined in the standard ways, as a single column in the `user` and `page` tables relates to a single column in the `reading` table.

Complex foreign keys are trickier than complex primary keys. Complex foreign keys occur in situations wherein more than one column in Table B refers to more than one column in Table A.

For example, take a database about cars. There's the `carmake` table that stores "Audi", "Ford", "Honda", and so forth:

```
CREATE TABLE carmake (
    id INT(10) UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    make VARCHAR(30) NOT NULL,
    UNIQUE (make)
);
INSERT INTO carmake (make) VALUES ('Audi'), ('Ford'), ('Honda');
```

Then there's the `carmodel` table:

```
CREATE TABLE carmodel (
    make_id INT(10) UNSIGNED NOT NULL,
    model VARCHAR(30) NOT NULL,
    PRIMARY KEY (make_id, model)
);
INSERT INTO carmodel VALUES (1, 'TTS Roadster'), (2, 'Fusion'),
(3, 'Fit'), (2, 'Focus'), (3, 'Civic');
```

This table relates to `carmake` through the `make_id`. The `model` values would be like "TTS Roadster", "Fusion", "Fit", etc. The `carmodel` table's primary key is a composite of the `make_id` and the `model`.

{NOTE} You would also add a foreign constraint to the `carmodel` table, naturally.

Crud Generator

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

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

Model Class *

Car

Table 'car' has a composite primary key which is not supported by crud generator.

Figure 18.4: Gii cannot scaffold tables with complex keys.

As a database design rule, there's an argument against having complex primary keys. Instead, a surrogate primary key can be created, which leaves a simple integer as a unique reference point. In other words, in the above design, you could create an `id` field in `carmodel` that acts as a unique identifier when other tables reference it.

But if you prefer—or have—to use a complex primary key, then the next table, for representing individual cars or years of release has a complex foreign key to `carmodel`:

```
CREATE TABLE car (
    make_id INT(10) UNSIGNED NOT NULL,
    model VARCHAR(30) NOT NULL,
    release_year YEAR,
    PRIMARY KEY (make_id, model, release_year)
);
```

In fact, not only is the combination of `make_id` and `model` a foreign key reference to `carmodel`, but this table has a composite primary key comprising `make_id`, `model`, and `release_year`.

Although you can work with complex keys in Yii, the Gii CRUD generator cannot handle complex keys itself (**Figure 18.4**); you'll need to set these relations manually in your models.

To acknowledge this complex relationship, use all applicable columns in the relations definition methods:

```
# models/Car.php
public function getModels() {
    return $this->hasMany(Carmodels::className(),
        ['make_id', 'model' => 'make_id', 'model']);
}
```

That's all you need to do, although be certain to list the columns in the same order in which they appear in the tables.

An added complication appears when you go to add validation with complex keys. For example, you should not be able to add a record to the `car` table unless the `make_id` and `model` combination exists in `carmodel`. Or, in the hypothetical `reading` table, the combination of the `user_id` and `page_id` would have to be unique.

In some situations, you can customize the “unique” and “exists” validators to suit your needs. Otherwise, you might be better off using one of the available complex key validation extensions you can find online.

Next, the chapter moves from complex keys to complex relationships. There are two more advanced relationship topics to cover. The first is using “through” relationships. The second addresses how one handles inheritance.

Sometimes you’ll have two models (and database tables) that have an indirect, but meaningful relationship to each other. For example, in the CMS example, users post comments and pages have comments, but users and pages are not directly related. So how would you find what pages a user has commented on?

In such situations, one option is be to go through the existing relationships. For example, you could do a `find()` on `User`, and toss in a `with('comments')` and then perform lazy loading from the comments results to get the pages. That works, but it’s a lot of extra work and overhead.

The alternative is to set up a “via” relationship. Via relationships create a relationship between two not-directly-related models, through an existing relation. Here’s how that would be defined for the `User` example:

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

With that defined in `User`, you can now perform the query using:

```
// Where $user already represents the current user.
$pages = $user->commentedPages;
```

You can use “via” whenever a `HAS_MANY` or `HAS_ONE` relationship exists between the two models being connected. Also keep in mind that this is only meaningful when using Active Record. You can also always get the same results using Query Builder or Direct Access Objects.

The examples in the book to this point could all be described as flat, or single-layer databases. That’s to say that all of the tables are related to each other (if at all) on the same level. No tables reflect an inheritance relationship.

On the extreme edge of Yii usage people sometimes ask: how do you handle database inheritance in Yii? You've seen class inheritance multiple times over by now, but how do you translate database inheritance into models? Most relational databases don't support inheritance, but your models do. The models, though, are also already inherited from Active Record. So how do you proceed?

As an example, take an application that deals with pets in some regard. From a code perspective, there's a `Pet` model that has the generic properties and methods that apply to all animal types: `name`, `age`, `eat()`, `sleep()`, etc. Then there are individual animal classes that extend `Pet`: `Dog`, `Cat`, etc.

This logical approach results in what's called an [object relational impedance mismatch](#). What this means is that models exist for which there is no underlying database table. In particular, the `Pet` model doesn't correlate to a `pet` table (because you wouldn't store records of a generic pet type).

The best solution here is to implement *single table inheritance*: put all the children in a single table, but use a column to differentiate among them. This is an appropriate solution when the parent table or class would not have any records or be instantiated on its own, as in the pets example.

In that example, create a `pets` table:

```
CREATE TABLE pet (
    id INT(10) UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    type ENUM('dog', 'cat'),
    /* Other columns */
);
```

Then model it:

```
class Pet extends \yii\db\ActiveRecord {
```

The other models extend `Pet`:

```
class Dog extends Pet{}
```

This provides the basic inheritance outlined in the models. But it gets a bit trickier since Active Record uses the table and class names extensively.

By default Active Record assumes the table name is the same as the class name. This means that the code `Dog::findAll()` attempts to use the `dog` table, which does not exist. The first thing you need to do then is to tell Yii to always use the `pet` table:

```
class Pet extends \yii\db\ActiveRecord {
    public static function tableName() {
        return '{{%pet%}}';
    }
}
```

Because `Dog`, `Cat`, and the others extend `Pet`, they'll all now have this same method. (If your `Dog`, `Cat`, and other models have their own `tableName()` methods, remove those.)

Next, you need to also tell Yii what class is being used for `find()` and the other methods that use model instances. This is done by overriding the `instantiate()` method in `Pet`. The method should now return a different class name based upon the `type` attribute of the provided model:

```
class Pet extends \yii\db\ActiveRecord {
    public static function instantiate($attributes){
        switch($attributes['type']){
            case 'dog':
                $class='Dog';
                break;
            case 'cat':
                $class='Cat';
                break;
            default:
                $class=get_class($this);
                break;
        }
        $model=new $class(null);
        return $model;
    }
}
```

Now, whenever you use the code `Dog::find()` or `Cat::find()` or whatever, the proper, specific model used, not the generic `Pet`.

That's all you should need to do go get single table inheritance working on your site. This is a relatively clean and easy solution.

Unfortunately, clean and easy is not always possible. In particular, in situations with multiple levels of inheritance, a single table design won't work. It is technically possible to implement multiple table inheritance in Yii, although it's not easy considering how Active Record works.

Chapter 19

Extending Yii

The Yii framework is fairly complete in its own right, and through the extensions made available by others, you might be able to go a long time before hitting a development wall. But should that day come, Yii is easily extendible by anyone with decent Object-Oriented Programming skills and comfort with the framework. Extensions in Yii are simply that: the creation of additional functionality not already built into the framework.

The term “extension” can be used in a couple of ways. Most strictly, an extension in Yii extends the core framework functionality but is *intended to be shared with third parties*. In other words, instead of writing a chunk of code that helps only you out on one or two projects, a Yii extension is also a way to share the extremely useful code that you’ve developed.

This chapter explains everything you need to know to create, use, and deploy (for the use of others) your own extensions.

Before getting into the particulars of creating an extension, there are several fundamental concepts when it comes to the subject:

- Guidelines
- Structure
- Utilizing Composer
- Publishing assets
- Types of extensions

The purpose of an extension is not just to add needed functionality (including changing existing functionality), but to do so in a manner that’s highly and easily reusable. Moreover, a complete extension is easily reusable not just by you, but by other Yii developers as well. Towards that end, there are many guidelines in place for writing extensions. Note that these are not absolute rules, just good and logical recommendations:

- Come up with a good, unique vendor name you'll use for all of your projects
- Give your extension a unique but meaningful name
- Prefix your extension with "yii2-" to create a "project name"
- Write the extension to be self-contained, managing its own external dependencies (i.e., the user should not have to download additional packages to use your extension, as a general rule, although Composer can help here)
- All extension files should go within an extension directory that uses the same name as the extension itself (but in all lowercase letters)
- Extension classes should be namespaces within `vendorName\extensionName` (where "extensionName" does not have the "yii2-" prefix)
- Include good, reliable installation and configuration instructions
- Provide good documentation for usage, FAQ, and so forth
- Use meaningful version numbers and provide a CHangelog file when you update the extension
- Include a license
- Do your best to maintain the extension

{NOTE} The Yii community is wonderfully generous. Even if you worry that your work might be amateurish, or too limited in scope, I would highly recommend you consider sharing your extensions with others.

New to Yii 2 is an extension generator, built into Gii. It prompts for the basic information and even creates starter code—including a README—for you (**Figure 19.1**).

And, of course, your extension should be written under the best practices, with respect to the programming in general and how code is written in Yii.

{TIP} For help in picking the license that's most appropriate for you and the extension itself, see an online resource such as [Choose a License](#).

Per the extension guidelines, all extension files should go within an extension directory that uses the same name as the extension itself, but in lowercase letters. So if you're creating the *ArgyleBargle* extension, all of its files would go in the `argylebargle` directory.

Place your main class in the root folder of the extension directory. In this hypothetical example, that might be the file **ArgyleBargle.php**.

If **ArgyleBargle** is a fairly common term (e.g., you're making a jQuery or database-related extension), you'd probably want to prefix the extension directory and class to distinguish it. I might use **luargylebargle** and **LUArgyleBargle.php**.

Some extensions require only a single class file, meaning that the extension directory contains only that. Other extension types have their pieces appropriately divided into multiple subdirectories:

Extension Generator

This generator helps you to generate the files needed by a Yii extension.

Please read the [Extension Guidelines](#) before creating an extension.

Vendor Name

larryullman

Package Name

yii2-stripecheckout

Namespace

widget\StripeCheckout\

Type

yii2-extension

Keywords

yii2,extension,stripe

License

GPL-3.0+

Figure 19.1: *Gii's extension generator.*

- `extDir/assets`
- `extDir/models`
- `extDir/views`

This is often the case when creating modules, which have an application-like structure, or widgets, which have view files.

Regardless of the complexity of the extension, you should also be in the habit of including:

- A `README` file
- A `LICENSE` file
- A `CHANGELOG` file

These files should be in plain text or Markdown format.

In Yii 1 it was often customary to place extensions within the Yii application's **protected/extensions** directory. Yii 2 no longer has an **extensions** directory, as the expectation is that most extensions are installed via Composer into the **vendors** directory. For simplicity, the rest of the chapter uses **extDir** as a convention.

When developing your extension as part of an application, go ahead and put it in a logical place, whether that's the **controllers** directory or **widgets** or even in your own defined **extensions** directory. The Gii extension generator suggests the **runtime/tmp-extensions** directory. Then, when you publish your extension using Composer, you can repackage the files appropriately.

Ideally, extensions should be developed as a [Composer package](#). This provides the easiest and most foolproof way of installing third-party libraries. And since Yii itself uses Composer, it's reasonable to expect your extension's users to already have Composer installed, and to be familiar with its usage.

To build an extension as a Composer package, first develop the extension as you would, ensuring that it works well with the Yii framework. Then begin repackaging the files knowing Composer will install them into the end user's **vendors/packageName** directory.

Next, create a **composer.json** file in the root project directory. The JSON file contains metadata about the package, which Composer uses to properly install it.

A basic JSON file is:

```
{  
    "name": "vendorName/yii2-extensionName",  
    "type": "yii2-extension",  
    "description": "Short description of the extension",  
    "license": "BSD-3-Clause",  
    "require": {
```

```
        "yiisoft/yii2": "~2.0.0",
    },
    "autoload": {
        "psr-4": {
            "vendorName\\extensionName\\": ""
        }
    }
}
```

Most of this is straightforward, but three values need to be called out:

- * Use “yii2-extension” as the “type” value, so the extension is installed properly in a new Yii application
- * Always require the Yii framework as your extension depends on it
- * Set your root namespaced class to autoload, so the Yii application knows about your extension

For details on what all can go into the JSON file, see the [Composer documentation](#).

With the extension developed and the **composer.json** file included, the second step is to publish the code to a public repository, such as on GitHub.

The third and final step is to register your extension with [Packagist](#), the default source for Composer packages.

If you bootstrap your extension using Gii, it’ll explain these steps to you after generating the code (**Figure 19.2**).

More complex extensions, particularly widgets and modules, often make use of resources that need to be in a public directory: CSS, JavaScript, images, and videos. But extensions generally go in non-public directories. So a logical question then is: how do you make a shareable extension that also contains resources to be placed in a web-accessible directory?

The answer is to use the Yii framework’s [AssetBundle](#) class. Among other features, this tool handles the copying of resources from a non-public directory to the public **assets** directory. The Yii framework defines a few asset bundles already, including jQuery, jQuery UI, Twitter Bootstrap, and one for Yii functionality.

To use assets in your extension, start by creating the necessary files as you normally would. This includes breaking CSS and JavaScript into logical parts. Next, define a class that extends `yii\web\AssetBundle`:

```
namespace app\assets;

class MyExtAsset extends yii\web\AssetBundle {
```

Within the class, set the source path for the assets: where they can be found. Then identify the assets, grouped by type:

The extension has been generated successfully.

To enable it in your application, you need to create a git repository and require it via composer.

```
cd /Users/larry/Sites/yii/ch19/runtime/tmp-extensions/yii2-stripecheckout  
git init  
git add -A  
git commit  
git remote add origin https://path.to/your/repo  
git push -u origin master
```

The next step is just for *initial development*, skip it if you directly publish the extension on packagist.org

Add the newly created repo to your composer.json.

```
"repositories": [  
    {  
        "type": "git",  
        "url": "https://path.to/your/repo"  
    }  
]
```

Note: You may use the url `file:///Users/larry/Sites/yii/ch19/runtime/tmp-extensions/yii2-stripecheckout` for testing.

Require the package with composer

```
composer.phar require larryullman/yii2-stripecheckout:dev-master
```

And use it in your application.

```
\Widget\StripeCheckout\AutoloadExample::widget();
```

When you have finished development register your extension at [packagist.org](#).

Figure 19.2: Gii's recommended next steps to convert an extension to a Composer package.

```
public $sourcePath = 'myExtDir/assets';
public $css = [
    'css/main.css',
];
public $js = [
    'js/main.js',
];
```

With this class definition, when the asset bundle is registered in a view, Yii copies the named assets from **myExtDir/assets** and publishes them to the web directory. Here's the view code that triggers that:

```
# views/controllerID/someView.php
use app\assets\MyExtAsset;
MyExtAsset::register($this);
```

By default, all published assets go within an **assets** subdirectory, with a simple hashed name like *8e98dfc4*. This name is based upon the basename of the file(s) being copied. The destination directory can be changed, if needed.

No additional steps are required here: the assets are published to the web directory and pulled into the layout file automatically.

{TIP} Yii supports the ability to convert assets across types, for example from LESS to CSS or from TypeScript to JavaScript. See the Yii documentation for details.

The Asset Bundle is both easier to use and more capable in Yii 2 than the alternative in Yii 1. As you learn to use it, read more about its features and configurations, as explained in the [Yii documentation](#).

The particulars of any extension are dependent upon the *type* of extension being created. Fundamentally, it's a question of what role the extension plays in a Yii application, and therefore, from what class the extension's primary class is derived (if any).

Save for modules, the chapter next walks through the basic types of extensions, and outlines how they're written, before delving into some examples of specific types. Due to the more complex nature of modules, those are discussed and demonstrated later in the chapter.

Going even a step further, you can create your own application template, similar to the basic and advanced templates defined by the framework itself. If you create a lot of Yii-based sites and find yourself often making the same changes to the basic template, create your own! See the [Yii guide](#) for the simple instructions.

An application component extension serves the same purpose as the application components built into the Yii framework, such as the URL manager, the asset manager, the session component, and so forth. This is where you implement specific functionality that a website needs that's not particular to any model. Application components are particularly good for making code or data available to every aspect of an application; controllers, models, and views can all readily access application components.

Application components should extend the `yii\base\Component` class:

```
class MyAppComponent extends \yii\base\Component {  
}
```

Within the class, define any attribute or method you want:

```
class MyAppComponent extends \yii\base\Component {  
    public $var = false;  
    private $_thing = 42;  
    public function getThing() {  
        return $this->_thing  
    }  
}
```

Once defined, register the component with the application. This is done in the primary configuration file:

```
# config/main.php  
// Other stuff.  
'components' => [  
    'myappcomponent' => [  
        'class' => 'app\components\MyAppComponent',  
    ],  
],
```

That code tells the application that the `myappcomponent` component is an instance of the `MyAppComponent` class. (And the code assumes that the `MyAppComponent.php` file is within the `app/components` directory.)

With the component registered, you can access an instance of that class using `\Yii::$app->myappcomponent`:

```
\Yii::$app->myappcomponent->var = true;  
\Yii::$app->myappcomponent->getThing();
```

Application components are often configurable, affecting how they behave. For example, the user component can be told to allow for auto login (i.e., “remember me” functionality):

```
# config/main.php
// Other stuff.
'user' => [
    'identityClass' => 'app\models\User',
    'enableAutoLogin' => true,
],
// More stuff.
```

The configuration names and values just correspond to public properties of the underlying class (or “setter” methods, if you’ve gone that route). To make your application configurable, add public variables such as `$var` in the trivial example above. Then assign a value to that variable in your configuration:

```
# config/main.php
// Other stuff.
'components' => [
    'myappcomponent' => [
        'class' => 'app\components\MyAppComponent',
        'var' => true,
    ],
]
```

Do be certain to give your public attributes default values, however, so that a failure to configure the value does not result in errors.

A final consideration for application components is whether there’s any initial setup or other work to be performed before the component can be used. For example, does a database connection need to be made? If there is something to be done, define an `init()` method that performs any initialization work. Your component’s `init()` method should also call `parent::init()` to make sure the inherited initialization work is also performed.

Widgets are a specific, and common, type of extension used to add complex functionality to view files without embedding a lot of code and logic. Yii comes with a number of widgets built-in, as described in Chapter 12, “[Working with Widgets](#).” The built-in widgets:

- Present loads of information in grid format
- Represent jQuery UI components
- Present individual records in a nice format
- And more

A widget must extend the `yii\base\Widget` class (or an existing child class of `yii\base\Widget`). For example, to create your own variation on the Yii Captcha

widget, just extend that class and override its methods, or change its default behaviors, to suit your needs.

When creating a brand-new widget, extend the `yii\base\Widget` class and define at least the `init()` and `run()` methods. The `init()` method is called when an instance of the widget is created. The `run()` method renders the widget for the view file:

```
# widgets/mywidget/MyWidget.php
class MyWidget extends \yii\base\Widget {
    public function init() {
        // Do whatever to start.
    }
    public function run() {
        // Do whatever to end.
    }
}
```

If your widget doesn't need to capture any input, it can be created in a view file using just `widget()`:

```
# views/controllerID/view.php
<?php
use app\widgets\MyWidget;
?>
<?= MyWidget::widget(); ?>
```

In that case, the `init()` and `run()` methods are still invoked in that order.

Some widgets, such as `ActiveForm` expect you to create data—content—that the widget uses. This is done by using the `begin()` and `end()` method calls, with the data between them (in the case of `ActiveForm`, the data created are form elements). The data placed between those method calls is captured by the widget by starting output buffering within the `init()` method and ending it in `run()`:

```
# widgets/mywidget/MyWidget.php
class MyWidget extends \yii\base\Widget {
    public function init() {
        // Do whatever to start.
        parent::init();
        ob_start();
    }
    public function run() {
        // Do whatever to end.
        $data = ob_get_clean();
    }
}
```

```
// Use $data.  
}  
}  
  
# views/controllerID/view.php  
<?php  
use app\widgets\MyWidget;  
?>  
<?= MyWidget::begin(); ?>  
<p>This data is being captured.</p>  
<?php MyWidget::end(); ?>
```

Note that the output buffering captures the data, it does not output it. You'll need to return the `$data` variable to do that.

You can configure how a widget functions when it's created (in the view file) by passing additional arguments to the `begin()` or `widget()` method:

```
# views/controllerID/view.php  
<?php  
use app\widgets\MyWidget;  
?>  
<?= MyWidget::widget(['name' => 'value']); ?>
```

The names of the array elements must match the public properties defined in the widget class:

```
# widgets/mywidget/MyWidget.php  
class MyWidget extends \yii\base\Widget {  
    public $var1;  
    public $var2;  
    public function init() {  
        // Do whatever to start.  
    }  
    public function run() {  
        // Do whatever to end.  
    }  
}
```

```
# views/controllerID/view.php  
<?php  
use app\widgets\MyWidget;  
?>  
<?= MyWidget::widget(['var1' => 42, 'var2' => false]); ?>
```

In that example, the public `$var1` property in the class is assigned the value 42, `$var2` is assigned false.

You can also reference the public attributes in the `run()` method, of course, including performing validation there, if you'd like. If a property value must be assigned to use the widget, you would throw an exception in the `init()` method.

Most widgets output HTML. For anything but the very simplest output, create your own view file (or files) for the widget. The view files would go within the `views` subfolder of your extension's directory, and be given the name `viewfile.php`, as is the case with any controller's view file. You'd then use the `render()` method within the widget's `run()` method, as you would within a controller.

To put this all together, as a stupid, but comprehensible, example of this, say your *MyWidget* outputs some text within a DIV or a paragraph (which is not a good use of a widget). The view files would be:

```
# widgets/mywidget/views/div.php
<div><?= Html::encode($output); ?></div>

# widgets/mywidget/views/p.php
<p><?= Html::encode($output); ?></p>
```

Use of the widget (in a view file) would be:

```
# views/controllerID/view.php
<?php
use app\widgets\MyWidget;
?>
<?= MyWidget::begin(['tag' => 'p']); ?>
<p>This data is the output.</p>
<?php MyWidget::end(); ?>
```

Or:

```
# views/controllerID/view.php
<?php
use app\widgets\MyWidget;
?>
<?= MyWidget::begin(['tag' => 'div']); ?>
<p>This data is the output.</p>
<?php MyWidget::end(); ?>
```

And the widget itself would be defined as:

```
# widgets/mywidget/MyWidget.php
class MyWidget extends \yii\base\Widget {
    public $tag;
    public function init() {
        parent::init();
        ob_start();
    }
    public function run() {
        $output = ob_get_clean();
        if ($this->tag == 'p') {
            return $this->render($this->tag,
                ['output' => $output]);
        } else {
            return $this->render('div',
                ['output' => $output]);
        }
    }
}
```

Actions are controller methods invoked as part of a request. These are methods defined within the controller, whose names start with *action*. For example, the `actionIndex()` method is normally the default method called for every controller (i.e., when no other action is specified).

It's standard for controller actions to be defined within the controller itself, but they can also be defined independently, thereby creating a *standalone action*. As with any extension, the benefit of a standalone action is reusability: if you have an action whose logic could be used by multiple controllers without change, you could define that as an extension and then tell each controller about it.

Actions must extend the `yii\base\Action` class, or an existing child class of `yii\base\Action`. Actions must define a `run()` method that does the bulk of the work:

```
# extensions/myaction/MyAction.php
class MyAction extends yii\base\Action {
    public function run() {
        // Do the work.
    }
}
```

Configure a controller to be able to use that action. This next bit of code tells the `MyController` class to use the `MyAction` class for the `myaction` request:

```
# controllers/MyController.php
class MyController extends Controller {
    public function actions() {
        return [
            'myaction' => 'extensions/myaction/MyAction'
        ];
    }
    // Other controller code.
}
```

If the action needs parameters—values passed to the action when it's executed, you can do that by creating arguments in the `run()` method:

```
# extensions/myaction/MyAction.php
class MyAction extends yii\base\Action {
    public function run($thing) {
        // Do the work.
    }
}
```

When the `myaction` action is executed (associated with a controller), the value of `$_GET['thing']` is passed to the `run()` method.

Behavior extensions provide the ability to add additional functionality to common MVC pieces through an external source. Put another way, behaviors allow you to emulate multiple inheritance by making methods not directly inherited by a class available for use within that class. In OOP, this concept is called a “mixing”.

For example, a behavior might be defined that adds the ability to smoothly handle checkboxes in any model. As another example, the `TimestampBehavior` automatically sets a model's `create_at` attribute to the current moment when the model is first created. It will also do the same for the `update_at` attribute when the model instance is updated.

The behavior extension must implement the `yii\behaviors\Behavior` class. Behaviors intended for specific contexts, such as models in general or Active Record in particular, can extend the more specific `AttributeBehavior` or other class instead:

```
# extensions/mybehavior/MyBehavior.php
class MyBehavior extends AttributeBehavior {
```

To attach a behavior to a model, use the model's `behaviors()` method. This method returns an array of behaviors to attach. For each item, provide a behavior name and class as an array:

```
# models/SomeModel.php
public function behaviors() {
    return [
        'behaviorName' => MyBehavior::class(),
    ];
}
```

The result is that the model now has access to the methods defined in the behavior as if the methods were defined in the model itself. The main difference being that these methods—the behavior—can be added to *any* model.

As you'll see in a later example, behaviors tied to models often use `beforeSave()` and other event-driven methods to perform steps in conjunction with the life of a model. This is how the `TimestampBehavior` works.

Like actions and behaviors, filters are also used by controllers. In fact, a filter is just a special type of behavior. Filters perform a task before and/or after an action is executed. For example, the “access control” filter is used to restrict access to the execution of controller actions.

To create a filter extension, you'll need to extend the `yii\base\ActionFilter` class (or extend a child of that class). The class should have a `beforeAction()` method, which is executed before the controller action. The class should also have a `afterAction()` method, which is executed after the controller action. The `beforeAction()` method needs to return a Boolean indicating whether the controller action should be allowed to execute. The `afterAction()` method does not need to return anything. Both functions should take one argument, the action being taken.

Because a filter may be one in a sequence to be executed, to call the `beforeAction()` and `afterAction()` methods of the parent class in your filter extension. You can use the returned value as the value to be returned by your methods. Here, then, is the shell of a filter extension:

```
# extensions/myfilter/MyFilter.php
class MyFilter extends yii\base\ActionFilter {
    public function beforeAction($action) {
        // Do whatever.
        return parent::beforeAction($action);
    }
    public function afterAction($action) {
        // Do whatever.
        return parent::afterAction($action);
    }
}
```

The `behaviors()` method of a controller is used to apply the filter:

```
# controllers/MyController.php
class MyController extends Controller {
    public function behaviors() {
        return [
            [
                'class' => extension/myfilter/MyFilter'
                'only' => ['index', 'view'],
            ],
        ];
    }
    // Other controller code.
}
```

Console commands are like controller actions meant to be run from a command-line interface. Similarly, you can create a console command as an extension that's distributable and usable in any application.

To create a console command extension, you need to extend the `yii\console\Controller` class. This class is used just like a `yii\web\Controller` class. Start by defining your actions as methods:

```
class MyCommand extends yii\console\Controller {
    public function actionEcho() {
        // Do whatever.
    }
}
```

The command can now be run using `yii`, providing it with a route as you would a web page:

```
yii mycommand/echo
```

To accept values, write the action method to take arguments, as you would any other function:

```
class MyCommand extends yii\console\Controller {
    public function actionEcho($message) {
        echo $message . "\n";
    }
}
```

And here's its usage:

```
yii mycommand/echo hello
```

Conventionally, command-line applications always return an exit code, normally 0 to indicate no problems and a number greater than 0 otherwise. Alternatively you can use the common return values defined as constants in the `ExitCode` class:

```
class MyCommand extends yii\console\Controller {
    public function actionEcho($message) {
        echo $message . "\n";
        return ExitCode::OK;
    }
}
```

To make the command extension available to your application, simply place the completed PHP script in the `commands` directory and ensure that the class is within the `app\commands` namespace. The `console.php` configuration file is already configured to use those classes.

With the fundamental concepts of exceptions explained, and a couple of simple examples in place, it's time to look at some more specific and real-world examples. The next several pages:

- Looks at a built-in behavior
- Outlines a new behavior
- Develops a widget

These three examples reinforce the core concepts when it comes to developing your own extensions.

This chapter already mentioned the timestamp behavior as a useful concept. This behavior, found in the Zii library and written by Jonah Turnquist in Yii 1, is now part of the framework itself as of Yii 2. The behavior can automatically set the value of model attributes to the current timestamp. The extension is mostly defined like so, with comments removed for brevity and clarity:

```
<?php
namespace yii\behaviors;

use yii\base\InvalidCallException;
use yii\db\BaseActiveRecord;

class TimestampBehavior extends AttributeBehavior {
    public $createdAtAttribute = 'created_at';
    public $updatedAtAttribute = 'updated_at';
```

```
public $value;

public function init() {
    parent::init();
    if (empty($this->attributes)) {
        $this->attributes = [
            BaseActiveRecord::EVENT_BEFORE_INSERT =>
                [$this->createdAtAttribute, $this->updatedAtAttribute],
            BaseActiveRecord::EVENT_BEFORE_UPDATE =>
                [$this->updatedAtAttribute,
            ];
    }
}

protected function getValue($event) {
    if ($this->value === null) {
        return time();
    }
    return parent::getValue($event);
}
}
```

As you can see, the class extends `yii\behaviors\AttributeBehaviors`, as the behavior is meant to be applied to a model's attributes. The class has three public attributes, making these configurable options when using the behavior:

```
public function behaviors() {
    return [
        [
            'class' => TimestampBehavior::className(),
            'createdAtAttribute' => 'create_time',
            'updatedAtAttribute' => 'update_time',
            'value' => new Expression('NOW()'),
        ],
    ];
}
```

Being a behavior, the most important code is defined within the `init()` method. Within that method code assigns values during two events: an insertion of a new record or the updating of an existing record. The default value in both cases can be set during the behavior's configuration, or can come from the `getValue()` method. For more on how the underlying functionality, see the definition of the `yii\behaviors\AttributeBehavior` class.

As another example, Chapter 9, “[Working with Forms](#),” discussed how to work with model attributes associated with checkboxes. In particular, you might have an

attribute whose value stored in the database is Y/N, which will need to be properly represented by a form checkbox. The catch being when a checkbox is not checked in the form, the resulting value will be 0 (by default). To address this problem, you could make a behavior extension.

```
<?php
namespace yii\behaviors;

use yii\base\InvalidCallException;
use yii\db\BaseActiveRecord;

class CheckboxBehavior extends AttributeBehavior {
    public $trueValue = 'Y';
    public $falseValue = 'N';
    public $attr = null;
```

The class extends `AttributeBehavior`, of course. Within the class are three public properties, making it configurable. Two properties set the “true” and “false” values as they’re stored in the database. The defaults are Y and N. The third property is the attribute to which the behavior should be applied. The behavior would be used like so:

```
public function behaviors() {
    return [
        [
            'class' => CheckoutBehavior::className(),
            'trueValue' => 'Yes',
            'falseValue' => 'No',
            'attr' => 'receiveEmails'
        ]
    ];
}
```

Next, the behavior has a `init()` method. Its role is to identify the events and attributes to be handled. It can be defined quite similarly to the `TimestampBehavior`:

```
public function init() {
    parent::init();

    if (!empty($this->attributes)) {
        $this->attributes = [
            Base ActiveRecord::EVENT_BEFORE_INSERT => [$this->attr],
            Base ActiveRecord::EVENT_BEFORE_UPDATE => $this->attr,
        ];
    }
}
```

```
    }
}
```

Finally, the `getValue()` method returns the appropriate value for the true or false state:

```
public function getValue($event) {
    if ($this->value == 1) {
        return $this->trueValue;
    } else {
        return $this->falseValue;
    }
}
```

When the registered event occurs—an insertion or an updating of a model—the model uses this behavior to set the value of the configured “attr” attribute. To do so, the model invokes this behavior’s `getValue()` method, thanks to code in `yii\behaviors\AttributeBehavior`. All this method has to do is look at the incoming value and convert it to the proper configured value.

The behavior to this point handles the conversion of an HTML checkbox—returning 1 or 0—to a Yes or No. However, the code should probably also convert the values back upon retrieval of a record. To do that, start by updating the `init()` method to handle that event:

```
public function init() {
    parent::init();

    if (empty($this->attributes)) {
        $this->attributes = [
            BaseActiveRecord::EVENT_BEFORE_INSERT => $this->attr,
            BaseActiveRecord::EVENT_BEFORE_UPDATE => $this->attr,
            BaseActiveRecord::EVENT_AFTER_FIND => $this->attr,
        ];
    }
}
```

And then update the `getValue()` method to inspect the even type and react accordingly:

```
public function getValue($event) {
    if ($event-> == EVENT_AFTER_FIND) {
        if ($this->value == $this->trueValue) {
            return 1;
```

```
    } else {
        return 0;
    }
} else {
    if ($this->value == 1) {
        return $this->trueValue;
    } else {
        return $this->>falseValue;
    }
}
```

And there you have a functional behavior extension that's easy to use and quite helpful. Once applied to a model, you'll never have to think about converting between database and checkbox form values again!

{NOTE} If you're copying this code, you also need a closing bracket to complete the class.

The next example creates a widget, one of the most common extension types around. From a development perspective, creating a widget requires a decent amount of knowledge and effort, although not so much as a full-blown module. Widgets are used to provide more complex logic, normally including HTML, as a component that's separate from the view files.

The specific widget being developed is a reasonably realized, practical use case that also thoroughly demonstrates the process of creating a widget. It plays off the CMS concept, creating a simple way to show comments anywhere (**Figure 19.3**).

To create the widget, begin by identifying the desired end result. For a comment widget, that's a list of comment text, author, and timestamp.

The next decisions to make are:

- What information must the widget user provide?
- What information should be optionally configurable?

For a comments widget, it'd should be able to show comments for the current page or any page. In other words, if placed on a page for a specific post, it only shows the comments for that post, but if placed on the home page, it shows the most recent comments across all posts.

The widget should also default to showing only the 5 most recent comments, but that number should be overridable.

The combination of the required and optional information become the widget's properties, configurable on a widget-by-widget basis.

Comments

This is my comment.

Posted by: Someones on 2015-01-21 16:48:28

This is the page owner's reply.

Posted by: Test on 2015-01-21 16:33:46

Figure 19.3: *The widget outputs these comments*

Next, create the widget class. It must extend `yii\base\Widget`:

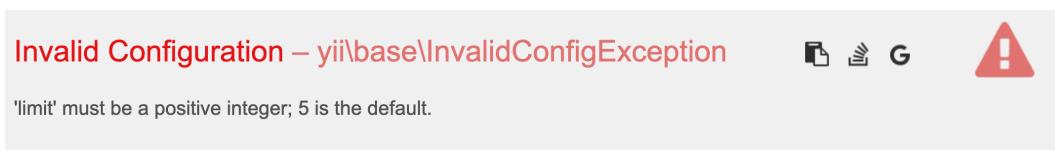
```
# widgets/CommentsWidget/CommentsWidget.php
namespace app\widgets;

use yii\base\InvalidConfigException;
use app\models\Comment;

class CommentsWidget extends \yii\base\Widget {
    public $page_id;
    public $limit = 5;
    public function init() {
        // Do whatever to start.
    }
    public function run() {
        // Do whatever to end.
    }
}
```

The class goes within the `app\widgets` namespace and then identifies two additional namespaces being used. The class has two public properties, one with a default value.

In a real-world (i.e., non-book) class, there would be plenty of documentation in-



The screenshot shows a Yii error message. At the top, it says "Invalid Configuration – yii\base\InvalidConfigException". To the right is a red exclamation mark icon. Below the title, the message "‘limit’ must be a positive integer; 5 is the default." is displayed. At the bottom left, there is a code snippet from "CommentsWidget.php" at line 12. The code is as follows:

```

3
4     use yii\base\InvalidConfigException;
5
6
7     class CommentsWidget extends \yii\base\Widget {
8         public $page_id;
9         public $limit = 5;
10        public function init() {
11            if (!is_int($this->limit) || ($this->limit < 0)) {
12                throw new InvalidConfigException("‘limit’ must be a positive integer; 5 is the default.");
13            }
14        }

```

at line 12

Figure 19.4: A thrown exception.

cluded, starting with phpDoc syntax for the class itself and the properties. The documentation would also include an example of the widget’s usage.

Next, the widget must have two methods: `init()` and `run()`. Here’s the first of those. For any property that’s required, the `init()` method should validate the property’s value, throwing an exception when no valid value is provided:

```

use yii\base\InvalidConfigException;

public function init() {
    if (!is_int($this->limit) || ($this->limit < 0)) {
        throw new InvalidConfigException("‘limit’ must be a positive
                                         integer; 5 is the default.");
    }
}

```

In this widget, the role of the `init()` method is to perform the necessary validation. To start, the method confirms that a valid limit value was provided. If not, an exception is thrown (**Figure 19.4**).

The method could also validate the `$page_id` value, but that is handled differently in the `run()` method.

The other method, `run()`, does the most important work. This method fetches the applicable comments and passes them to the rendered view file. The specific query changes based upon the presence of a valid `$page_id` value:

```

public function run() {
    if (is_int($this->page_id) && ($this->page_id > 0)) {
        $comments = Comment::find()

```

```
        ->where(['page_id' => $this->page_id])
        ->orderBy(['date_entered' => SORT_DESC])
        ->limit($this->limit)
        ->all();
    } else {
        $comments = Comment::find()
            ->orderBy(['date_entered' => SORT_DESC])
            ->limit($this->limit)
            ->all();
    }
    return $this->render('comments', ['comments' => $comments]);
}
```

This is pretty basic use of Active Record, finding all the applicable records, applying “order by” and “limit” clauses, and throwing in a “where” conditional if a page ID was provided.

Now it’s time to define that view file.

The view file, named **comments.php**, should be placed within the **views** directory of the widget extension’s directory. A simple definition of it is:

```
<hr />
<?php
foreach ($comments as $comment) {
    echo "<div>
        <p>{$comment->comment}</p>
        <p>Posted by: {$comment->user->username} on {$comment->date_entered}</p>
    </div><hr />";
}
```

The view receives an array of **Comment** objects, which it iterates over. Thanks to a **getUser()** method in **Comment**, the user’s name is available through **\$comment->user->username**.

Certainly the view could do a lot more:

- Report a message if there are no comments to display
- Format the date and time in a nicer manner
- Use the **Html** helper to output the data securely
- Paginate lots of records

But, for a proof of concept, this is fine!

Finally, there’s the use of the widget. Assuming you’ve provided good documentation, using the widget should be very easy. This would go in one of the site’s view files:

```
<?php
use app\widgets\CommentsWidget;
?>
<?= CommentsWidget::widget(); ?>
```

That invocation of the widget might be used on a home page to see the most recent comments across the site.

Any public property can be set in the configuration:

```
<?php
use app\widgets\CommentsWidget;
?>
<?= CommentsWidget::widget(['limit' => 10, 'page_id' => $this->id]); ?>
```

That invocation of the widget could be used on the page (or post) view file to show only the comments for that page or post.

In a way, modules are the easiest extension type to understand. A module is an entire web application, used as a component of a larger application. A module has the same core pieces as any application—models, views, and controllers, but they are particular to one aspect of a site. For example, your site may have:

- A main area
- Support forums
- A shop
- A blog
- An administrative area

In that example, the main area of your site is the primary Yii application. The other three areas could each be their own module. The result is a fairly complex and sophisticated site that's still easy to use. Moreover, because you created modules for three of these areas, those modules could be turned into reusable extensions for other projects you do.

The only thing you can't do with a module is deploy it on its own. This is really the primary distinction between a module and an application: modules are always subservient to an application.

To best explain how to create and use modules, let's first walk through the basics and then create a specific example as a module.

Although a module is just a mini-application, there are a few things to know before creating one for yourself.

One of the nice things about creating modules in Yii is that the Gii tool can generate the shell of the module for you. To use it:

1. Enable Gii, if it's not already.
2. Access Gii for your site in your browser.
3. Click Module Generator.
4. On the resulting page, enter the full, namespaces class of the module to be generated, in the form `app\modules\yourModule`.
5. Enter the module ID. This should be a simple, logical word choice, like "admin" or "support" or the like.
6. Click "Preview".
7. On the resulting page, confirm the files being created, and click generate.

This is all pretty straightforward. The most common complication could be the lack of an existing **modules** directory, which can result in odd errors.

{WARNING} Gii needs to be able to write to the **modules** folder. The proper permissions should already allow for this, but if you run into problems, manually make this directory writable.

Give serious thought to your module name prior to following those steps. Modules need unique names, which are their identifier. To avoid bugs, it's best to name the module in all lowercase letters, and do not use any numbers, punctuation, or other symbols.

Gii creates a new directory, **modules/modulename**. Within it, you'll find:

- **controllers**
- **controllers/DefaultController.php**
- **views**
- **controllers/default**
- **controllers/default/index.php**
- **modulename.php**

This last item file is the main module class, which extends `yii\base\Module`. In many ways, this class is a corollary to the application's **Application** class.

Once you've defined a module, or added an existing module extension to your site, you just need to tell your site about the module in order to use it. This is done in the "modules" section of the primary configuration file. In fact, you've already seen an example of this with Gii:

```
# config/web
$config['modules']['gii'] = [
    'class' => 'yii\gii\Module',
    // uncomment the following to add your IP if you are not connecting from localhost.
    //'allowedIPs' => ['127.0.0.1', '::1'],
];
```

admin/default/index

This is the view content for action "index". The action belongs to the controller "app\modules\controllers\DefaultController" in the "admin" module.

You may customize this page by editing the following file:

/Users/larry/Sites/yii/ch19/modules/views/default/index.php

Figure 19.5: A default module layout.

To use a different module, just add your module's name to the primary section of the configuration file:

```
# config/main.php
'modules' => [
    'yourModuleID' => [
        'class' => 'app\modules\yourModuleClass',
    ],
],
```

The value used for “yourModuleID” needs to match the ID value of the module itself (which also corresponds to the name of the module's directory). Note that this is also a top-level configuration: it doesn't go under components or anything else other than `$config`.

With the module enabled via the configuration file, it's now usable by your site. You can test this by heading to `http://example.com/index.php/modulename` in your browser.

As with the primary application, if no controller or action IDs are provided, the module runs the default action of the default controller. With a module, that's the “index” action of `DefaultController`. As generated, this action merely renders the `modulename/views/default/index.php` view file (**Figure 19.5**).

```
# modules/modulename/controllers/DefaultController.php
<?php
namespace app\modules\modulename\controllers;
class DefaultController extends Controller {
    public function actionIndex() {
        $this->render('index');
    }
}
```

This controller extends the `Controller` class, so it can be used and behaves like any other controller in your application. You can now also create other controllers, other actions, and other view files. If your module needs models, you can create and use those, too. You'll see more of this in a forthcoming example.

Do note that the controller ought to be under the module's namespace, as in the code above.

{TIP} You can nest one modules within another by adding the submodule to the configuration of the parent module, and placing the submodule in the **modules** directory of the parent module.

When using modules, an additional dimension is added to your URLs. The full syntax of a URL in Yii is:

```
<Domain>/index.php/<Module>/<Controller>/<Action>/<Params>
```

If no action ID is provided, the default action of the requested controller is used. If no controller ID is provided, the default controller of the application or module is used. If no module ID is provided, the default application is used.

When it comes to creating URLs with modules, you still use the methods of the `yii\helpers\Url` class. As is the case when not using modules, if you specify the action and nothing else, the route assumes the same controller, and, in the case of a module, module:

```
# modulename/controllers/DefaultController.php
public function actionDummy() {
    // modulename/default/index:
    $url = Url::to('index');
```

If you specify the controller, the URL is to that controller and action, but still within the same module:

```
# modulename/controllers/DefaultController.php
public function actionDummy() {
    // modulename/other/index:
    $url = Url::to('other/index');
```

If you specify the module name, you can create a URL to a different module within the application:

```
# modulename/controllers/DefaultController.php
public function actionDummy() {
    // testmodule/other/index:
    $url = Url::to('/test/other/index');
```

Note that in this case, you need to start with a slash, to indicate the base of the URL. The same goes if you want to create a URL to the primary application from within a module:

```
# modulename/controllers/DefaultController.php
public function actionDummy() {
    // /other/index:
    $url = Url::to('/other/index');
```

Modules, like any application component or the application itself, can be pre-configured in the primary configuration file. To do that, assign a value to any public property of the main module class:

```
<?php
class TestModule extends \yii\base\Module {
    public $var;

# config/web.php
'modules' => [
    'test' => [
        'class' => 'app\modules\TestModule',
        'var' => 'value',
    ],
],
```

{TIP} If you'd rather, you can create your own configuration file for the module, and then have the main configuration file include it.

Because the base class extends `\yii\base\Module`, you can configure any public property of that class, too:

```
# config/web.php
'modules' => [
    'test' => [
        'class' => 'app\modules\TestModule',
        'var' => 'value',
        'defaultRoute' => 'something',
        'layout' => 'home',
    ],
],
```

Within the module or the application, you can also access any public property through the module itself. First, access the module, which returns a class instance. Then reference the module's properties:

```
$m = MyModuleClass::getInstance();
echo $m->var;
```

Within a module itself, you can always refer to the current module using `\Yii::$app->controller->module`:

```
$m = \Yii::$app->controller->module;
echo $m->var;
```

Further, Yii creates a root alias for each module. Having created the “test” module, in path names in your application, “test” equates to **modules/test**.

To wrap up this discussion of writing extensions, the chapter walks through developing a complete module. The specific example is a reasonably realized, practical use case: a module-based implementation of a [Stripe checkout](#) process.

In case you’re not familiar with [Stripe](#) or how its system works, you use a form on your website for taking the customer’s payment information (i.e., credit card details). This data is sent to Stripe via JavaScript, so that it never touches your server. Not having the credit card information on your server relieves almost all of the PCI compliance burden.

In 2021, the preferred product for processing payments in Stripe is the Payment Intent. Using the Stripe PHP library, you create the Payment Intent from the server. Then a reference to the Payment Intent is passed to the client. Via JavaScript, the customer’s payment details are attached to the Payment Intent to complete the payment request entirely on the server.

{NOTE} Stripe is not currently live in every country, but you can test Stripe with any email address, or without creating an account at all.

For the purposes of full disclosure, I worked for Stripe from 2013 until 2022. But I worked at Stripe because I’m such a huge fan of their product, not the other way around.

Before writing any code it’s best to visualize the desired end result. In terms of what the user sees:

1. The initial page takes the user’s payment details.
2. The next page shows the success of the payment attempt.

So that makes two views and just a single controller action. Of these views, the most important is the payment details page, which contains a form like so:

```
<form id="payment-form" data-secret="<?= $intent->client_secret ?>">
    <div id="card-element">
        <!-- Elements will create input elements here --&gt;
    &lt;/div&gt;

    &lt;!-- We'll put the error messages in this element --&gt;
    &lt;div id="card-errors" role="alert"&gt;&lt;/div&gt;

    &lt;button id="card-button"&gt;Submit Payment&lt;/button&gt;
&lt;/form&gt;</pre>
```

That comes straight from the Stripe documentation. Note the `$intent->client_secret` that must be generated by a PHP request to Stripe and then passed to the HTML form. There's also going to be two Stripe keys—one publishable and one secret—that need to be configurable. The publishable one also gets passed to the view and used in JavaScript.

Along with those views and the action, the modules should:

- Have a **Payment** class...
- With an associated **payment** database table
- Use the Stripe PHP library
- Support a dynamically determined price for the payment

The combination of the required and optional information become the module's configurable properties.

Performing a payment request requires the Stripe PHP library, so the extension needs to have that. Thankfully, the Stripe PHP library is installable via Composer:

```
composer require stripe/stripe-php
```

This also means that if this extension were to be properly completed and made installable via Composer, the Composer configuration for this extension would indicate the Stripe PHP library is a requirement. By doing that, the Stripe PHP library would automatically be installed at the same time as the extension itself (if it wasn't already).

The next step is to enable Gii, and follow the instructions outlined earlier to create the shell of the module. This module is named “pay”(**Figure 19.6**).

The result is the **modules/pay** directory. Within it, Gii creates:

- **controllers**
- **controllers/DefaultController.php**

Module Generator

This generator helps you to generate the skeleton code needed by a Yii module.

Module Class

```
app\modules\pay\Module
```

Module ID

```
pay
```

Code Template

```
default (/Users/larry/Sites/yii/ch19/vendor/yiisoft/yii2-gii/src/generators/module/def...
```

Preview

Figure 19.6: Creating a new module.

- **views**
- **controllers/default**
- **controllers/default/index.php**
- **Module.php**

Now it's time to edit, and add to, those files and directories.

{TIP} You may find the rest of this chapter easier to follow if you also download the code from [GitHub](#).

The next step is to think about how the module may need to be configured when added to a site. With this particular module, there are two pieces of information that must be provided: the user's public and private Stripe keys. These can be provided during the configuration so long as they're public attributes of the main module class:

```
class Module extends \yii\base\Module
    public $publishable_key;
    public $secret_key;
```

Because these values are required (and no default value could work), the class's `init()` method throws exceptions if they're not provided (**Figure 19.7**):

Invalid Configuration – yii\base\InvalidConfigException

Your Stripe 'public_key' must be set.

Figure 19.7: Stripe keys are required.

```
public function init(){
    parent::init();

    if ($this->publishable_key === null) {
        throw new InvalidConfigException("Your Stripe 'publishable_key' must be set.");
    }
    if ($this-> secret_key === null) {
        throw new InvalidConfigException("Your Stripe 'secret_key' must be set.");
    }
}
```

That's all the changes required for the module class itself. Finally, enable the module in the site:

```
# config/web.php
'modules' => [
    'pay' => [
        'class' => 'app\modules\pay\Module',
        'publishable_key' => 'pk_test_APr32Tly9WH6K9XfZpJeEKCH',
        'secret_key' => 'sk_test_CaOU4KkAPr32TZpJeEKCH'
    ],
],
```

You can find your own public and private keys from the [Stripe Dashboard](#). There's no cost and you don't have to activate your account.

The Stripe extension module needs one model class. The class is used to create the form, add validation, and store the data in the database. For that reason, a model based upon Active Record makes sense. Gii can create the model, but the underlying database table needs to exist first. The database table could be defined like so:

```
CREATE TABLE payment (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    payment_id VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL,
    amount INT UNSIGNED NOT NULL,
```

```
~/S/y/ch19 > ./yii migrate/create create_payment_table
./yii migrate/create create_payment_table
Yii Migration Tool (based on Yii v2.0.42.1)

Create new migration '/Users/larry/Sites/yii/ch19/migrations/m210705_183220_create_payment_table.php'? (yes|no) [no]:yes
New migration created successfully.
~/S/y/ch19 >
```

Figure 19.8: Creating a migration.

```
        created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
        UNIQUE (payment_id),
        INDEX (email)
) ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
COLLATE = utf8_general_ci
```

{NOTE} You may notice that the table is not storing any credit card details. When using Stripe, absolutely no credit card information ever touches your server.

You *could* define that SQL command in a text file that you distribute with the extension. But since reusability is the goal, and to do things the Yii way, it's better to create a *migration* for it. See Chapter 18, “[Advanced Database Issues](#)”, for more on migrations.

From the command-line, within the application directory, execute (**Figure 19.8**):

```
yii migrate/create create_payment_table
```

{NOTE} You may need to change the command used to execute **yii** to suit your environment.

This creates a file named something like *m140122_205418_create_payment_table*, stored in the **migrations** directory.

Since this migration isn't particular to the application—it's part of the module, create a **migrations** directory within your module folder, and then move this new file there. Then open the file for editing.

The migration class has two methods: `up()`, for implementing the migration, and `down()`, for undoing its impact (or `safeUp()` and `safeDown()`). In this case, the `up()` method should create the `payment` table and `down` should delete it:

```
public function safeUp() {
    $this->createTable('{{%payment}}', [
        'id' => $this->primaryKey(),
        'payment_id' => $this->string(100)->notNull(),
```

```
[~/S/y/ch19 » ./yii migrate --migrationPath=@app/modules/pay/migrations
Yii Migration Tool (based on Yii v2.0.42.1)

Total 1 new migration to be applied:
  m210705_183220_create_payment_table

Apply the above migration? (yes|no) [no]:yes
*** applying m210705_183220_create_payment_table
  > create table {#payment} ... done (time: 0.063s)
  > create unique index payment_id on payment (payment_id) ... done (time: 0.063s)
  > create index email on payment (email) ... done (time: 0.027s)
*** applied m210705_183220_create_payment_table (time: 0.155s)

1 migration was applied.

Migrated up successfully.
~/S/y/ch19 » █
```

Figure 19.9: Implementing the migration.

```
'email' => $this->string(100)->notNull(),
'amount' => $this->integer()->unsigned()->notNull(),
'created_at' => $this->timestamp()->notNull(),
]);
$this->createIndex('payment_id', 'payment', 'payment_id', true);
$this->createIndex('email', 'payment', 'email');
}

public function safeDown() {
    $this->dropTable('{#payment}');
    $this->dropIndex('payment_id', 'payment');
    $this->dropIndex('email', 'payment');
}
```

With the migration actions defined, execute the migration:

```
yii migrate --migrationPath=@app/modules/pay/migrations --interactive=0
```

By default, this command looks for migrations to be executed found within the app's **migrations** folder. To change that, provide the path to the module's **migrations** directory (**Figure 19.9**):

```
yii migrate --migrationPath=@app/modules/pay/migrations
```

And now the database table should be created! Next, you can generate and edit the corresponding model. More importantly, you've just made your extension that much easier for others to use!

The module uses one model, an Active Record tied to the **payment** database table. For the model, start by having Gii generate it based upon the database table definition:

The code has been generated successfully.

```
Generating code using template "/Users/larry/Sites/yii/ch19/vendor/yiisoft/yii2-gii/src/generators/model/default"
generated models/Payment.php
done!
```

Figure 19.10: Creating a new model.

1. Log into Gii.
2. For the table name, use “payment”.
3. For the model class, use “Payment” (the default).
4. Keep the rest of the defaults.
5. Click Preview.
6. Assuming everything is okay, click Generate (**Figure 19.10**).

By default, Gii creates the file in the application’s **models** directory, so the resulting file is **models/Payment.php**. Move this to the module by creating a **models** directory in the module’s folder (i.e., create **modules/pay/models**), and move this file there. Open the file to edit it.

First, change the model to be in the module’s namespace:

```
namespace app\modules\pay\models;
// Originally app\models
```

None of the credit card information—number, CVC, or expiration date—touch the server, so there’s no point in representing those in the model.

Next, tweak the validation rules slightly:

```
public function rules() {
    return [
        [['payment_id', 'email', 'amount', 'token'], 'required'],
        [['amount'], 'integerOnly' => true, 'min' => 50],
        [['email'], 'email'],
        [['created_at'], 'safe'],
        [['payment_id', 'email'], 'string', 'max' => 100],
        [['payment_id'], 'unique'],
    ];
}
```

There’s nothing too surprising here, but see Chapter 5 for more on validation rules, if need be. The token is marked as required; you’ll see how that plays out over the next several pages. The amount must be an integer at least 50 (cents) or larger, which corresponds to the minimum amount that can be processed through Stripe. Note that Stripe expects all amounts to be in the lowest base unit for the given currency (e.g., cents for USD).

For an added touch, the labels can be updated slightly:

```
public function attributeLabels() {
    return [
        'id' => 'ID',
        'payment_id' => 'Payment ID',
        'email' => 'Email',
        'amount' => 'Amount (in cents)',
        'created_at' => 'Payment Date',
    ];
}
```

{TIP} You may also want to set custom error messages.

And that should take care of the primary model the module uses!

With the model created and edited, turn to the controller (and then the view). By default, modules use the “index” action of the `DefaultController`. It’d be reasonable to change those names to something more meaningful, but this chapter won’t do so in the interest of time.

As previously discussed, the workflow for the module is simple:

1. A Payment Intent is created through Stripe’s API.
2. The Payment Intent is passed to the view file, along with some additional information.
3. The rest of the payment processing occurs in the client via JavaScript (i.e., in the view file).
4. If the payment attempt succeeds, the user is shown a “thanks” page.

The bulk of the work happens within the JavaScript in the view file. Most of that code comes from Stripe’s own documentation.

{NOTE} Stripe allows test transactions to be made over HTTP (in part because only dummy payment details work in test mode).

The controller has just one action, “index”. It needs to:

- Create the `Payment` model instance.
- Upon first use, generate a Payment Intent and pass required information to the view file.
- Upon the form’s submission, save the `Payment` to the database and render a “thanks” page.

Much of this action's code is similar to that you'd find in any controller's "create" action. Note that to create an instance of a `Payment` model, the controller must use that namespace:

```
# modules/pay/controllers/DefaultController.php
<?php
namespace app\modules\pay\controllers;
use yii\web\Controller;
use app\modules\pay\models\Payment;
class DefaultController extends Controller {
```

Then, within the action, create the model instance and set the amount to be charged:

```
public function actionIndex() {
    $model = new Payment;
    $model->amount = 12575;
```

On a real, live site, this amount would most likely be determined dynamically, and would be provided to the extension in some manner:

- Saved in the session
- Calculated by another controller
- Retrieved from the database

For this test purposes the amount is just being hardcoded. Remember that it has to be an integer, representing the amount in cents!

Next, the action checks if the form has been submitted and the model instance can be saved:

```
if ($model->load(\Yii::$app->request->post()) && $model->save()) {
    return $this->render('thanks', ['model' => $model]);
```

The success of the payment all happens in JavaScript in the view file, so there's not much that could go wrong at this point. But if you wanted to be extra careful, you could remove the `save()` invocation, then separately `validate()` and process the model.

If the form hasn't been posted, start by setting the secret Stripe API key:

```
} else {
    $module = \Yii::$app->controller->module;
    \Stripe\Stripe::setApiKey($module->secret_key);
```

The secret key is a configured module property, so to access it, first get a reference to the module.

Next, use the Stripe API to create a Payment Intent:

```
$intent = \Stripe\PaymentIntent::create([
    'amount' => $model->amount,
    'currency' => 'usd',
]);
```

Finally, store the Payment Intent ID in the model and then render the view:

```
$model->payment_id = $intent->id;
return $this->render('index', ['model' => $model, 'intent' => $intent, 'publishable_key'
```

The HTML and JavaScript in the view needs three pieces of information:

- Active Record model instance
- Payment Intent
- Publishable Stripe API key

That's all that's necessary in the controller. In a fully realized module, there may also be an action allowing the administrator to view the list of payments. Neither "update" nor "delete" actions would ever be warranted.

{TIP} Stripe supports the creation of customers and recurring billing via subscriptions. Ability to do both would be a great addition to this extension.

With the controller logic in place, all that remains is the view file. Most of the following code is JavaScript taken from the Stripe documentation.

The HTML form begins as a standard Active Record-based HTML form, like those created by Gii:

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
?>
<div class="pay-default-index">
<?php $form = ActiveForm::begin([
    'id' => 'payment-form',
    'options' => ['data-secret' => $intent->client_secret],
]); ?>
```

Per the desired end result already identified, the form needs to have a “data-secret” attribute with a value of the Payment Intent’s client secret. Other than that, this is a typical Active Record form.

Add the HTML for Stripe Elements:

```
<div id="card-element">
    <!-- Elements will create input elements here -->
</div>
<!-- We'll put the error messages in this element -->
<div id="card-errors" role="alert"></div>
```

Stripe’s Elements product is used to accept the payment details. All it needs is a single, empty DIV, which is later referenced by JavaScript. The second DIV is for displaying Stripe’s validation messages, such as an invalid expiration date. Neither of these elements are associated with a model.

Complete the form:

```
<div id="email">
    <?= $form->field($model, 'email')->textInput() ?>
</div>
<?= $form->field($model, 'payment_id')
->hiddenInput(['payment_id' => $model->payment_id])
->label(false) ?>
    <button id="card-button">Submit Payment</button>
<?php ActiveForm::end(); ?>
</div>
```

The rest of the HTML and PHP completes the Active Record form. There’s an input for the email address, and then the `payment_id` value is stored in a hidden input.

Now it’s a matter of adding the JavaScript. First, the page needs to include the `Stripe.js` file:

```
<?php $this->registerJsFile('https://js.stripe.com/v3/'); ?>
```

`Stripe.js` is Stripe’s primary JavaScript library. It defines the Elements functionality.

Next, the page needs to do a bunch of stuff in JavaScript:

- Set the Stripe public key
- Create the Stripe Element
- Watch for and handle input changes
- Upon the form’s submission, attempt the payment

A screenshot of a web form for entering payment details. It includes fields for a card number, expiration date (MM/YY), and CVC.

Figure 19.11: The payment details inputs

- Report any errors
- Upon the payment succeeding, actually submit the form to the server

Arguably, most of this could go in an external JavaScript library that is then published to the `assets` folder. For simplicity, it's being added to this view directly. All of the JavaScript goes within this:

```
<?php $this->registerJs(" // All JavaScript here. "); ?>
```

Note that the remaining JavaScript goes between these two lines. Because all of the JavaScript is placed between double quotes, the JavaScript code can only single quotes, or escaped double quotes.

Begin by setting the publishable key:

```
var stripe = Stripe('{$publishable_key}');"); ?>
```

Create the card element:

```
var elements = stripe.elements();  
var card = elements.create('card');  
card.mount('#card-element');
```

From these three lines of JavaScript, the Stripe library creates the necessary inputs to accept all of the payment details (**Figure 19.11**).

Watch for input changes on the card element:

```
var displayError = document.getElementById('card-errors');  
card.on('change', function(event) {  
    if (event.error) {  
        displayError.textContent = event.error.message;  
    } else {  
        displayError.textContent = '';  
    }  
});
```



Figure 19.12: An example error message

The Stripe Elements library automatically validates the input as it changes. The page just needs to add JavaScript that watches for and sets any error messages (**Figure 19.12**).

Handle the form submission:

```
var form = document.getElementById('payment-form');
document.getElementById('card-button').addEventListener('click', function(ev) {
    ev.preventDefault();
```

This is fairly standard JavaScript. Upon the form's submission—or, technically, the clicking of the submit button—the form's submission should be prevented, giving the JavaScript time to send the request to Stripe.

Complete the payment request with Stripe:

```
stripe.confirmCardPayment(form.dataset.secret, {
    payment_method: {
        card: card,
    }
}).then(function(result) {
```

This code also comes from the Stripe documentation. It calls the `confirmCardPayment()` method, passing along the Payment Intent's client secret and the card object.

Handle any errors and complete the JavaScript:

```
if (result.error) {
    displayError.textContent = result.error.message;
} else {
    // The payment has been processed!
    if (result.paymentIntent.status === 'succeeded') {
        form.submit();
    }
});
```

An error at this point would be for something discovered on Stripe's end, such as insufficient funds. Any such error is reported immediately. If the payment succeeds, the form is submitted to send the data back to Yii.

Whew! Complicated JavaScript, but it works! Now on to the PHP code that actually processes the payment with Stripe. And that completes the module. It is ready to be used. Once you polish it up, provide adequate documentation, and so forth...

To test the Stripe payment integration (after configuring the module), see [Stripe's documentation](#) for sample credit card numbers to use.

There are many ways this module could be written differently or changed. A couple have already been mentioned, but the most important step in making this a distributable extension is to provide good documentation as to how the extension is used. This is particularly important as the extension's user might want to customize the layout of the form, and you'd want to set the right rules for doing so.

Towards that end, one improvement would be to change the form creation from a static view file to a use of form builder. This would further separate the requirements from the display, making it much easier for the extension user to change the look of the form.

The only requirement that's not well handled by the extension is how the amount gets to it. Truly, an exception ought to be thrown when no amount value exists, as it is required. But the amount could come from many places, so a sophisticated solution would be to have this as a configurable option. The extension user could indicate the source and attribute or index where the amount can be found: the X attribute of the Y model; the X index of the session; etc. Then the module could retrieve it and throw an exception on error.

Other improvements include:

- Allowing for the extension user to change the currency
- Using better names for the controller and action
- Using a filter to require HTTPS for non-test uses

Chapter 20

Working with Third-Party Libraries

The Yii framework in itself is a powerful tool, but it's made even more powerful by its natural support for third-party libraries. The Yii creators never intended Yii to be all things to all people. Instead, they designed Yii to readily work with existing tools that might be better suited for individual tasks.

This is even more true in Yii 2, which relies upon [Composer](#) for installation of the framework itself. I've explained and used Composer since Chapter 2, "Starting a New Application." Since Yii uses Composer and many third-party libraries are also Composer-based, it's straightforward to integrate most third-party libraries into your Yii application.

Chapter 13, "Using Extensions," demonstrated Composer examples with [Swift Mailer](#) and [Elasticsearch](#). Chapter 19, "Extending Yii," showed another with the [Stripe](#) PHP library. In this chapter I'll add another example: using the [Symfony](#) framework.

Before that, I go through more generic ways to use third-party libraries with your Yii application. That content contextualizes how Yii works with other code, which becomes more relevant for any non-Composer library you may end up needing.

To be as obvious as possible when it comes to using third-party libraries with Yii, understand that you must always download and install the third-party libraries to your server yourself. That is not something Yii does for you. (Composer does this, but only after you configure and invoke Composer.)

It's recommend that you put all third-party libraries within the **vendor** directory, with each vendor having its own subdirectory. If you're using multiple third-party libraries, this might mean you'd end up with a structure like:

- **vendor/zend**
- **vendor/stripe**

- **vendor/elasticsearch**

This organization is not required by Yii, but simply makes the most sense. (You'll also sometimes see it as a **vendors** directory, plural, although Yii sticks to the singular **vendor**.)

Composer naturally defaults to this structure as well. (A clear theme is: use Composer when possible!)

If the library you want to use is installable via Composer, add it as a dependency for a project. That's accomplished by creating a file of JSON (JavaScript Object Notation) data named **composer.json**. The sample Yii application creates this file for you, in the base application directory.

Among other uses, the **composer.json** file identifies the requirements for the project. This goes within a "require" section, using the syntax **package:version**:

```
{  
    "require": {  
        "library": "x.y.z"  
    }  
}
```

The package name includes the vendor name and the project name, with the two often being the same. You can find a list of available packages at [Packagist](#). This is the main repository for Composer packages, although some frameworks use their own.

To add repositories for Composer to use, change the JSON syntax to first list the additional repositories, such as the one for the Zend Framework:

```
{  
    "repositories": [  
        {  
            "type": "composer",  
            "url": "https://packages.zendframework.com/"  
        }  
    ],  
    "require": {  
        "zendframework/zend-mail": "2.0.*"  
    }  
}
```

The above adds the Zend Framework's repository to the list of repositories for Composer use, allowing Composer to install libraries from it, too.

As for the version, if you use a specific version, such as 1.2.3, only that version would ever be installed. If you use an asterisk for any number, that's a wildcard: 1.2.* installs the latest version within the 1.2 family. If you want the latest version regardless of the numbers, you'd use .. Composer only installs stable versions, by default.

The **composer.json** file included in the base Yii application requires several Yii libraries:

```
"require": {  
    "php": ">=7.4.0",  
    "yiisoft/yii2": "~2.0.45",  
    "yiisoft/yii2-bootstrap5": "~2.0.2",  
    "yiisoft/yii2-symfonymailer": "~2.0.3",  
},
```

To add third-party libraries, edit that section accordingly:

```
"require": {  
    "php": ">=7.4.0",  
    "yiisoft/yii2": "~2.0.45",  
    "yiisoft/yii2-bootstrap5": "~2.0.2",  
    "yiisoft/yii2-symfonymailer": "~2.0.3"  
    "aws/aws-sdk-php": ">=3.258.1"  
},
```

That additional line of code adds the [AWS SDK for PHP](#) package as a dependency.

With Composer installed and the dependencies for the project identified, the final step is to have Composer install the dependencies. For this, use the command line:

```
php /path/to/composer.phar <command>
```

If you're running Composer for the first time, the command is **install**. That command also creates a **composer.lock** file in the same directory as **composer.json**. This file acts as a record of what was installed when.

If Composer has already been run, the **composer.lock** file already exists, and the proper command is **update**.

To install dependencies: 1. Access your computer from a command-line interface. 2. Move to the directory with the **composer.json** file. 3. Execute the following command (**Figure 20.1**):

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



```
larry@utm-ubuntu-20: /var/www/html/yii-demo — ssh larry@192.168.64.2 — 105x29
[larry@utm-ubuntu-20:/var/www/html/yii-demo$ php /home/larry/composer.phar update
Loading composer repositories with package information
Info from https://repo.packagist.org: #StandWithUkraine
Updating dependencies
Lock file operations: 6 installs, 42 updates, 0 removals
- Locking aws/aws-crt-php (v1.0.2)
- Locking aws/aws-sdk-php (3.258.1)
- Upgrading bower-asset/bootstrap (v5.2.1 => v5.2.3)
- Upgrading bower-asset/jquery (3.6.1 => 3.6.3)
- Upgrading bower-asset/viiz-pjax (2.0.7.1 => 2.0.8)
- Upgrading codeception/codeception (5.0.2 => 5.0.7)
- Upgrading codeception/lib-asserts (2.0.0 => 2.0.1)
- Upgrading codeception/lib-innerbrowser (3.1.2 => 3.1.3)
- Upgrading codeception/stub (4.0.2 => 4.1.0)
- Upgrading doctrine/instantiator (1.4.1 => 2.0.0)
- Upgrading doctrine/lexer (1.2.3 => 3.0.0)
- Upgrading egulias/email-validator (3.2.1 => 4.0.1)
- Upgrading ezyang/htmlpurifier (v4.14.0 => v4.16.0)
- Upgrading fakerphp/faker (v1.20.0 => v1.21.0)
- Locking guzzlehttp/guzzle (7.5.0)
- Locking guzzlehttp/promises (1.5.2)
- Upgrading guzzlehttp/psr7 (2.4.1 => 2.4.3)
- Locking mtdowling/jmespath.php (2.6.1)
- Upgrading nikic/php-parser (v4.15.1 => v4.15.3)
- Upgrading phpunit/php-code-coverage (9.2.17 => 9.2.24)
- Upgrading phpunit/phpunit (9.5.24 => 9.5.28)
- Locking psr/http-client (1.0.1)
- Upgrading psy/psysh (v0.11.8 => v0.11.12)
- Upgrading sebastian/comparator (4.0.6 => 4.0.8)
```

Figure 20.1: Using Composer.

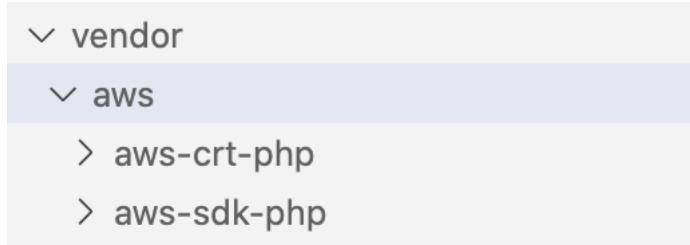


Figure 20.2: Composer installed these files.

This line uses the **composer.phar** script to install the necessary dependencies. Depending on your setup and your operating system, you'll most likely need to explicitly set the path to the PHP executable and/or the path to **composer.phar**.

{TIP} Ideally, you should add your PHP executable to your environment's path. Look online to find instructions for your operating system.

4. Look within the **vendor** directory to find the installed stuff (**Figure 20.2**).

After changing the requirements in **composer.json**, or when packages are updated, you'll want to have Composer update your installation. This is done via the Composer's **update** command:

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

{NOTE} Composer installs libraries for a specific project only, it does not perform a global installation.

Any library installed via Composer is automatically loaded to use in your Yii application!

If a library isn't neatly packaged up for Composer installation, you'll need to install it manually. To start:

1. Download the library.
2. Expand the library (assuming it came in a compressed format).
3. Move the expanded folder of items to your application's **vendor** directory.

Once you've installed the requisite third-party library, how easy or hard it is to use the third-party library's code is entirely dependent upon how complex the library is, and the status of its autoloader. To appreciate what steps you'll need to take, let's look at how Yii works and what could go wrong.

Yii relies upon *autoloading* to grab required class files on demand. Take the following bit of code:

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

When Yii executes that code, it knows that it needs to load the **User.php** file that defines the **User** class. (Presumably, that's **models/User.php**.) Yii is aware of the existence of that file thanks to this line in the application's configuration:

```
# config/web.php
'components' => [
    'user' => [
        'identityClass' => 'app\models\User',
        'enableAutoLogin' => true,
    ],
],
```

When Yii cannot find a class definition, it'll throw an exception (**Figure 20.3**).

This is true of PHP in general, of course.

When it comes to using third-party libraries, the most common problem will be an inability for Yii to find the corresponding class file for the object type you're trying to create. This is particularly true when a library has a very complex class hierarchy or just uses multiple classes. But the right Yii code will resolve these issues.

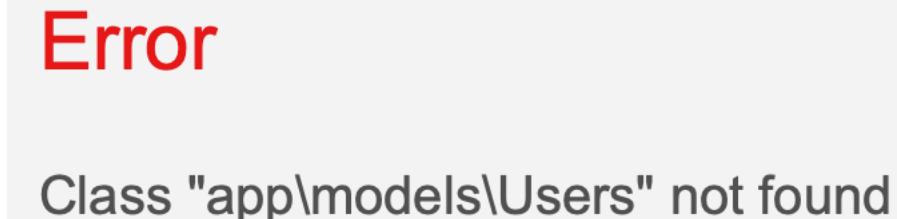


Figure 20.3: The misspelling of the class name results in an exception.

Over the next several pages I'll introduce and explain many different ways to make third-party library classes available to your Yii application. If it's still not clear, subsequent specific examples should hammer the points home.

The most important lesson is that when you see an exception like that in Figure 20.3, the reason is almost always one of the following:

- You misspelled the class name
- Yii cannot find the file where that class is defined

To debug such problems, start by confirming the class name. If that doesn't clearly resolve the issue, you'll need to fix where Yii looks for files.

{TIP} An even bigger lesson is your Yii development life is a lot happier when you can use Composer-based libraries!

One way to include a library class definition in your Yii application is the standard PHP approach: using `require()`.

```
require('vendor/VendorName/ClassName.php');
```

Having included that library, you can now create objects of the `ClassName` type:

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

One downside to this approach is that it automatically includes the file at the invocation of `require()`, even if the class definition isn't used until much later, or not at all.

A bigger problem with this approach is that it won't work with especially complicated class structures. For example, if `ClassName` is just one of many classes used by the library, and objects of other class types are created on the fly, this code won't work.

Third, and most trivially, using `require()` is not in keeping with the general Yii approach.

A better solution is to autoload the class definition into the application, as Yii already does with the fundamental application models and controllers.

How a Yii application autoloads a third-party library depends upon how the library is written. Any library with a complex structure should have its own autoloader mechanism. The only remaining trick is getting Yii to use it. Often this is as simple as including the autoloader file that comes with the library:

```
$al = dirname(__FILE__) .  
    '/vendor/vendorname/autoload.php';  
require_once($al);
```

In many cases, that will suffice, especially if usage of the library is localized.

If the library could be used anywhere in the application, you can register the autoloader in the bootstrap file (or entry script) of the application:

```
# yii  
#!/usr/bin/env php  
<?php  
defined('YII_DEBUG') or define('YII_DEBUG', true);  
defined('YII_ENV') or define('YII_ENV', 'dev');  
  
// Include the vendor's autoloader:  
$al = dirname(__FILE__) .  
    '/vendor/vendorname/autoload.php';  
require_once($al);  
  
require __DIR__ . '/vendor/autoload.php';  
...
```

You'd want to place this after the application has been configured but before Yii's own autoloader is required.

If the library does not have its own autoloader but wisely follows the [PSR-4 Autoloader naming scheme](#), Yii can autoload its classes. For that to work, you must configure a root alias for each of the library's namespaces:

```
# config/web.php  
$config = [  
    'aliases' => [  
        '@someNamespace' => '@vendor/vendorname/someNamespace',  
    ],
```

Yet even one more way to resolve the issue of being unable to find classes is to tell Yii about your classes via class mapping. This is simply a matter of creating aliases so that when you attempt to create an object of type X, Yii knows to use the definition found in **path/to/X.php**.

Class mapping is accomplished by assigning an array to the `$classMap` property of the `Yii` class:

```
Yii::$classMap = [
    'Stripe' => 'vendor/stripe/lib/Stripe.php',
    'Another' => 'vendor/another/lib/Another.php'
];
```

Class mapping is a last resort, as proper use of autoloaders should suffice, but I wanted to at least mention that this option exists.

Thanks to support for namespaces in PHP 5.3, many PHP frameworks have turned to a namespace structure. This is true for Zend Framework 2, Symfony 2, and will be the case with Yii 2 as well.

The PHP Specification Request level 0 (PSR-0) defines the rules for which PSR-0 compliant libraries should autoload classes. If you want to use a third-party library that abides by PSR-0, then you can use namespace references to the library's classes in your Yii application (assuming you're using PHP 5.3 or greater).

First, you'll want to create an alias to the library's root folder:

```
Yii::setAlias('@VendorName',
    Yii::getAlias('@vendor/vendornamespace'));
```

Now you can use the namespace syntax to create objects, starting with `@VendorName` as the root namespace. Again assuming that there is a **VendorName/Subdir/Charlie.php** file, you can now do this:

```
$charlie = new VendorName\Subdir\Charlie();
```

Of course this only works with a properly setup autoloader.

[Symfony](#) is a popular PHP framework that's been around since 2005 and is in version 6 at the time of this writing (2022). Supported by [SensioLabs](#), Symfony has numerous useful components, pretty good documentation, and is easily used as a third-party tool in a Yii-based application.

For a Symfony example, I'm going to use its [DomCrawler](#) component. It provides an easy and powerful way to traverse HTML and XML documents. The specific example could be the basis of a crawler you create to index web pages (perhaps in conjunction with the Elasticsearch example from Chapter 13).

I'm not going to discuss Symfony in detail. For more information on this framework, see the Symfony site or search online. The focus here is on using Symfony within Yii.

To install the necessary Symfony component, add the following to your **composer.json** file:

```
{  
    "require": {  
        "symfony/css-selector": "*"  
        "symfony/dom-crawler": "*"  
    }  
}
```

The DomCrawler component is the key one, but its extended functionality uses the Symfony CSSSelector component.

Once you've configured Composer, run its `update` command, as already explained. The result will be a **vendor/symfony** folder, with appropriate subfolders.

Now Symfony is ready to use!

To use Symfony, create the appropriate controller and action. For simple testing purposes, I normally just throw these attempts into a new action in the “Site” controller:

```
public function actionSymfony() {  
    // Do stuff here!  
}
```

For this example, I want to read in a page of HTML, find every link, and pass those links to the view file to be displayed. Alternatively, you might:

- Read in the page
- Index the site's content for your search engine
- Pull all of the links out of the content
- Repeat these steps for each link on that same site

So here's what the complete function would look like, with inline comments:

```
public function actionSymfony() {  
    // Read in the HTML:  
    $html = file_get_contents('http://localhost/');  
  
    // Find all the links on the page:  
    $crawler = new Crawler($html);
```

Error

Class "app\controllers\Crawler" not found

Figure 20.4: *Yii can't find the Crawler definition.*

```
// Filter by "a" tag:
$found = $crawler->filter('a');

// Store the links in an array:
$links = [];
foreach ($found as $link) {
    // $link->nodeValue will be the link text
    $links[$link->nodeValue] = $link->getAttribute('href');
}

// Pass the links to the view file:
return $this->render('symfony', ['links'=>$links]);
}
```

Even without knowing Symfony, the commented code above should be easy enough to follow. There's one catch, however. The `Crawler` class is namespaced in Symfony, meaning that the Composer autoloader won't be able to find it (**Figure 20.4**).

The solution is to tell Yii about the location of `Crawler` within the namespace. That's done via PHP's `use` command:

```
use Symfony\Component\DomCrawler\Crawler;
```

The next catch, though, is that you can't place this code within a function, as PHP won't let you use `use` within a block. That line must go outside of the class definition:

```
<?php
use Symfony\Component\DomCrawler\Crawler;
class SiteController extends Controller
```

The final step is to create the view file that displays the links:

Links Found by Symfony

My Application: /index.php

Home: /index.php?r=site%2Findex

About: /index.php?r=site%2Fabout

Contact: /index.php?r=site%2Fcontact

Login: /index.php?r=site%2Flogin

Get started with Yii: <http://www.yiiframework.com>

Yii Documentation »: <http://www.yiiframework.com/doc/>

Yii Forum »: <http://www.yiiframework.com/forum/>

Yii Extensions »: <http://www.yiiframework.com/extensions/>

Yii Framework: <https://www.yiiframework.com/>

Figure 20.5: The names and URLs of the found links.

```
<h2>Links Found by Symfony</h2>

<div>
<?php foreach ($links as $name => $url) {
    echo '<p><strong>' . $name . '</strong>: ' . $url . '</p>';
}
?>
</div>
```

Figure 20.5 shows the output after reading in the default Yii template home page.

Chapter 21

Testing Your Applications

Testing your code is simple in theory, but complex in reality, requires a decent amount of work, and adds a lot of code. Those are the negatives. What you get in return is a more reliable and less buggy application, both now and as you update the code in the future.

Testing was adopted relatively late in the PHP community, which is unfortunate, but at least it is being used more and more. Naturally, Yii supports testing, too.

There are two primary kinds of testing you implement in Yii. The first is *unit testing*. The premise behind unit testing is that you write tests that verify that your code does exactly what it should on the smaller-picture, behind-the-scenes level. These tests should inspect every little component of the application—every class and method, asking the question: is the result of executing this code always what it should be?

The second primary type of testing is *functional*. Functional testing takes a big-picture approach to the application, verifying that it *behaves* as it should.

Loosely speaking, you can think of unit testing as focusing on the code itself, and functional testing as focusing on the user interface. Or you could say that unit tests focus primarily on the models and functional tests look at the views and controllers.

{NEW} Yii 2 adds support for *acceptance tests*. Acceptance tests are similar in purpose to functional tests but use a real or virtual browser. Acceptance tests are slower to execute, and can be finicky, so it's best to reserve them for situations in which unit and functional tests do not provide confident coverage.

Testing starts off simple, but real-world and thorough testing is an involved process. But by applying tests to your application, your code and site should be more predictable. Further, subsequent code changes or alterations won't be able to create new bugs in existing code, a common problem as projects are expanded and modified.

It can take some time to become completely fluent in testing, especially when it comes to effectively testing complex structures and processes. But in this chapter, I'll introduce the basic concepts of the two testing types, and explain how to begin adding testing to your Yii projects. I'll also include some tips and tricks, and recommend some resources for learning more about testing in general.

{NEW} Yii 2 has built-in support for testing using the [Codeception](#) testing framework.

After creating a new Yii basic or advanced application using Composer, you'll find a **tests** directory within the base application folder. By default, that folder contains:

- **acceptance**, stores acceptance tests
- **bin**, has the executable for running tests
- **functional**, stores functional tests
- **unit**, stores unit tests

Three additional folders are for test input and output:

- **__data**, stores fake data used by tests
- **__output**, stores test coverage reports
- **__support**, stores helper methods

Finally, there are three configuration files, one for each test type:

- **acceptance.suite.yml.example**
- **functional.suite.yml**
- **unit.suite.yml**

These files configure how Codeception tests work. By default, the base application does not enable the acceptance tests but does enable and configure functional and unit tests.

The unit testing configuration contains:

```
actor: UnitTester
modules:
    enabled:
        - Asserts
        - Yii2:
            part: [orm, email, fixtures]
```

Actor is a Codeception concept the documentation goes into in more detail. You may not even need to think about it. Next, the configuration enables the Asserts and Yii2 modules. Asserts defines all the assertions used in tests. The Yii configuration specifically enables only the object-relational mapping (i.e., Active Record), email, and fixtures modules (of Codeception).

In the application root directory you'll find **codeception.yml**, which configures the behavior of the testing framework as a whole.

You'll write your unit tests in code stored in the **unit** directory, functional tests in the **functional** directory, and define any fixtures required—to be covered later in the chapter—in ****_data**. If you generate coverage reports (also covered later), those will be written into **_output**** (much like Yii itself uses the **runtime** directory to write data on the fly).

The other configuration files to be aware of are **config/test.php** and **config/test_db.php**. The test suites have their own bootstrap file, which sets **YII_ENV** to “test”. This tells the Yii application to load the test configuration file instead of the normal web (or console) one.

There are two key implications to this structure:

- As you change the primary configuration file, you may also need to duplicate those changes in **test.php**
- Your test database must have the exact same structure as your production database (but not contain production data)

Since the base Yii application comes with defined tests, you can already see what it's like to run them.

To run tests: 1. Access your computer from a command-line interface. 2. Move to the root application directory. 3. Execute the following command (**Figure 21.1**):

```
./vendor/bin/codecept run
```

That command automatically runs all defined and enabled tests, regardless of type. For each test you'll see:

- Test status, an “E” for error or a checkmark
- Test suite name
- Specific test name
- Time it took to execute the test

You'll also see how many total tests there are, grouped by type, how much total time it took to run (not much!), and the memory required.

For any test that creates an error you can see the detailed information after the summary (**Figure 21.2**):

```

larry@utm-ubuntu-20:/var/www/html/yii-demo — ssh larry@192.168.64.2 — 110x35
larry@utm-ubuntu-20:/var/www/html/yii-demo$ ./vendor/bin/codecept run
Codeception PHP Testing Framework v5.0.7 https://helpukrainewin.org

[Functional Tests (10) -----]
E ContactFormCest: Open contact page(0.02s)
E ContactFormCest: Submit empty form(0.00s)
E ContactFormCest: Submit form with incorrect email(0.00s)
E ContactFormCest: Submit form successfully(0.00s)
✓ LoginFormCest: Open login page(0.01s)
✓ LoginFormCest: Internal login by id(0.00s)
✓ LoginFormCest: Internal login by instance(0.00s)
✓ LoginFormCest: Login with empty credentials(0.01s)
✓ LoginFormCest: Login with wrong credentials(0.00s)
✓ LoginFormCest: Login successfully(0.00s)

[Unit Tests (21) -----]
✓ ContactFormTest: Email is sent on contact(0.01s)
✓ LoginFormTest: Login no user(0.00s)
✓ LoginFormTest: Login wrong password(0.00s)
✓ LoginFormTest: Login correct(0.00s)
✓ UserTest: Find user by id(0.00s)
✓ UserTest: Find user by access token(0.00s)
✓ UserTest: Find user by username(0.00s)
✓ UserTest: Validate user(0.00s)
✓ AlertTest: Single error message(0.00s)
✓ AlertTest: Multiple error messages(0.00s)
✓ AlertTest: Single danger message(0.00s)
✓ AlertTest: Multiple danger messages(0.00s)
✓ AlertTest: Single success message(0.00s)
✓ AlertTest: Multiple success messages(0.00s)
✓ AlertTest: Single info message(0.00s)
✓ AlertTest: Multiple info messages(0.00s)
✓ AlertTest: Single warning message(0.00s)
✓ AlertTest: Multiple warning messages(0.00s)

```

Figure 21.1: Running tests with Codeception.

```

larry@utm-ubuntu-20:/var/www/html/yii-demo — ssh larry@192.168.64.2 — 101x28
There were 4 errors:
1) ContactFormCest: Open contact page
Test tests/functional/ContactFormCest.php:openContactPage

[yii\base\InvalidConfigException] Either GD PHP extension with FreeType support or ImageMagick PHP
extension with PNG support is required.

Scenario Steps:
1. $I->amOnRoute("site/contact") at tests/functional/ContactFormCest.php:7

#1 /var/www/html/yii-demo/vendor/yiisoft/yii2/captcha/Captcha.php:175
#2 /var/www/html/yii-demo/vendor/yiisoft/yii2/captcha/Captcha.php:94
#3 /var/www/html/yii-demo/vendor/yiisoft/yii2/base/BaseObject.php:109
#4 yii\base\BaseObject->__construct
#5 /var/www/html/yii-demo/vendor/yiisoft/yii2/di\Container.php:419
#6 /var/www/html/yii-demo/vendor/yiisoft/yii2/di\Container.php:170
#7 /var/www/html/yii-demo/vendor/yiisoft/yii2/BaseYii.php:365
#8 /var/www/html/yii-demo/vendor/yiisoft/yii2/base/Widget.php:143
#9 /var/www/html/yii-demo/vendor/yiisoft/yii2/widgets/ActiveField.php:795
#10 /var/www/html/yii-demo/views/site/contact.php:54
Artifacts:

Yii-log: [yii\web\Session::open] 'Session started'

```

Figure 21.2: The error indicates I don't have the necessary PHP module installed.

Codeception allows you to run all the tests of a type using the command:

```
./vendor/bin/codecept run <unit|functional|acceptance>
```

Or you can run a specific suite of tests with:

```
./vendor/bin/codecept run <unit|functional|acceptance> <ClassName>
```

For example, to run the login tests, use:

```
./vendor/bin/codecept run unit LoginFormTest
```

To rerun only the tests that failed (Codeception tracks this), use:

```
./vendor/bin/codecept run -g failed
```

If your base Yii application already fails some tests, you should fix those before proceeding!

Unit testing in Codeception is built upon the most popular tool for unit testing in PHP, [PHPUnit](#). Over the next several pages, you'll use Codeception and PHPUnit to implement basic tests of your code. For additional information on both, see their corresponding documentation, starting with Codeception (which has dedicated content on using Codeception with Yii).

Since Codeception is already installed and working, the next step is to begin defining tests. Let's look at the basic concept, and then expand on that into real-world examples. Once again, I'll point out that knowing how to use Codeception or PHPUnit itself is necessary in order to fully comprehend everything involved.

To create a unit test, you define a class:

- Whose name ends with the string “Test”
- That extends a `Codeception\Test\Unit` class
- Saved in a file named `ClassName.php`, in `tests/unit` (or a subdirectory)

For example, the test of `LoginForm` class from the default Yii example, is in `tests/unit/models/LoginFormTest.php` file:

```
<?php
namespace tests\unit\models;
use app\models\LoginForm;
class LoginFormTest extends \Codeception\Test\Unit
```

Each individual test is defined as a method of the test class. These methods are given the name “testFoo”, where “Foo” refers to the method or functionality in the associated class that’s being tested:

```
public function testLoginNoUser() {}
public function testLoginWrongPassword() {}
public function testLoginCorrect() {}
```

Give your methods meaningful names as they’ll be broken into descriptive phrases during execution (see Figure 20.2).

{TIP} The testing class constitutes a “suite” of test cases.

Within each test method, you’ll run one assertion for every possible scenario you’ll want to test. The assertions come from Codeception and PHPUnit, and are enumerated in their documentation. These assertion methods are run on the `$this` object, as `$this` refers to an instance of the test class within the testing method. Since that test class extends from `\Codeception\Test\Unit`, it has all the PHPUnit assertions.

For a completely unnecessary example, just to get the process started, here’s a dummy test class:

```
<?php
class DummyTest extends \Codeception\Test\Unit {
    public function testTrue() {
        $var = true;
        $this->assertTrue($var);
    }
}
```

The class has one test that confirms that a variable has a true value. (Later in the chapter, you’ll create real tests.)

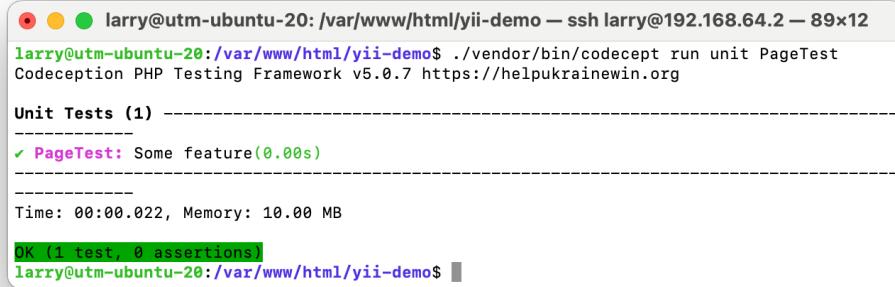
You could manually write tests, but as with many things in Yii, Codeception can bootstrap this process for you. Autogenerate test suites using:

```
./vendor/bin/codecept generate:test <unit|functional|acceptance> <ClassName>
```

For example, to bootstrap a test suite for `Page` class of the CMS example, use:

```
./vendor/bin/codecept generate:test unit Page
```

The result is the file `tests/PageTest.php` with this code:



```
larry@utm-ubuntu-20: /var/www/html/yii-demo — ssh larry@192.168.64.2 — 89x12
larry@utm-ubuntu-20:/var/www/html/yii-demo$ ./vendor/bin/codecept run unit PageTest
Codeception PHP Testing Framework v5.0.7 https://helpukrainewin.org

Unit Tests (1) -----
-----
✓ PageTest: Some feature(0.00s)
-----
Time: 00:00.022, Memory: 10.00 MB
OK (1 test, 0 assertions)
larry@utm-ubuntu-20:/var/www/html/yii-demo$
```

Figure 21.3: Running the new (empty) test.

```
<?php
namespace Unit;
use \UnitTester;
class PageTest extends \Codeception\Test\Unit {
    protected UnitTester $tester;
    protected function _before() {}
    // tests
    public function testSomeFeature() {}
}
```

The generated test even runs (**Figure 21.3**)!

The autogenerated code is somewhat minimal, so it may be to your liking or you may prefer to duplicate an existing test file or you could just create new tests from scratch.

Real-world testing often requires that certain things happen before a test can be run. For example, perhaps an object needs to be created or a connection to a database established. When tests require certain things to exist or have happened before being run, you can use a “setup” method to do the preparation, rather than doing the work within each individual test method.

To create a setup method, define a method in your class named `_before()`. This method is automatically called once before each test is run. Continuing with the unnecessary example, the `$var` variable had been created within `testTrue()`. If multiple tests might use that variable, it could be assigned a value within `_before()`:

```
class DummyTest extends \Codeception\Test\Unit {
    public $var;
    public function _before() {
        $this->var = false;
```

```
    }
    public function testTrue() {
        $this->assertTrue($var);
    }
}
```

This, of course, is still trivial, but the point should be sufficiently clear.

Conversely, the `_after()` method is executed after each test is run. You won't need a `_after()` method as frequently as you do `_before()`, but if you tied up resources in the `_before()` method, such as open a file or network connection, you could free up those resources (e.g., close the file or connection) in `_after()`.

{NOTE} The `**_after()**` method is always called after a test is run, whether or not the test succeeded.

Understand that the `_before()` and `_after()` methods are called *before and after each test case* (i.e., each method is run). Codeception also has the `_beforeSuite()` and `_afterSuite()` methods. These are each only called once, regardless of how many test methods you have. The `_beforeSuite()` method is executed before the running of the first test case, and `_afterSuite()` is executed after the running of the last test case.

{TIP} In some testing frameworks these are called “set up” and “tear down” methods.

The `_before()` and `_after()` methods just explained help define, and clear up, a state of being for when tests are run. For example, an object of a certain type may need to exist. This state, in unit testing, is called a *fixture*.

The previous example isn't much of a fixture: a simple variable was populated. Real-world testing requires more complex fixtures, and fixtures are sometimes used by more than one test suite (i.e., class). This most often occurs in cases where a database is involved.

To understand the need for fixtures, it may help to work the logic backwards:

1. An application relies upon data, so tests need access to data to truly evaluate the application.
2. You should not run tests on real data (or do anything with real data except for real uses on a real site).
3. Therefore, the tests need access to data that is real-like.

This is where a fixture comes in: as a mechanism to provide real-like data for tests.

To create a fixture, you define a class:

- Whose name ends with the string “Fixture”
- That extends a `yii\test\Fixture` or `yii\test\ActiveFixture` class
- Saved in a file named **ClassName.php**, in `tests/fixtures` (you may need to create this directory)

For fixtures tied to database tables, use `ActiveFixture`, just as you use Active Record for the model.

```
<?php
namespace app\tests\fixtures;
use yii\test\ActiveFixture;
class PageFixture extends ActiveFixture {
    public $modelClass = 'app\models\Page';
}
```

That code defines a fixture based upon the existing `Page` model (which, in turn, represents the `page` database table).

The above code creates the necessary object type, but not the data. The next step is create one or more “fixture files” in `tests/fixtures/data`. Each file should return an array of representative data for a table row, and each file would use the same name as the table or class for which it’s a fixture.

{WARNING} The directory structure as created by the base Yii application and that explained in the Yii manual is a tad confusing. Yii creates a `tests/_data` directory; however the framework expects fixture files to be in a `data` subdirectory of the directory where fixture classes are defined. It’s easiest to ignore `tests/_data` and to just create and use `/tests/fixtures/data` instead.

For example, the `Comment` class in the on-going CMS example has the following properties (corresponding to the same-named columns in the `comment` table):

- `id`
- `user_id`
- `page_id`
- `comment`
- `date_entered`

A fixture file for that class would look like so:

```
<?php
# tests/fixtures/data/comment.php
return [
```

```
'comment1' => [
    'user_id' => 23,
    'page_id' => 14,
    'comment' => 'This is the comment.',
    'date_entered' => '2022-07-20 12:09:23'
],
'comment2' => [
    'user_id' => 3,
    'page_id' => 8,
    'comment' => 'This is another comment.',
    'date_entered' => '2022-08-01 19:43:08'
],
];
```

Each subarray represents one row of data, and each is given an alias as a reference point. Each subarray's element is indexed using the table's column names. If a database table has certain automatic behavior, such as using an auto-incremented integer for the primary key, you don't have to provide that value.

With the fixture class and file defined, when tests are run, Codeception will:

- Before any tests are run, reset all tables involved to a known state
- Before a specific test is run, reset any involved tables to a known state
- Populate the table with the temp data

Understand that having defined this file, Codeception runs a `TRUNCATE TABLE comment` command when first accessed, and then inserts those new rows of data into the table. Since Yii runs Codeception under the test environment, the `test_db.php` configuration is used, so this is completely safe.

{TIP} Codeception automatically uses properly defined fixtures. If not using Codeception, Yii provides additional tools to utilize fixture (e.g., in other test frameworks).

Before using the fixture, you must configure the database connection in Yii such that you're using a test database, not a production one.

The `bin/yii` file within the `tests` directory (implicitly) includes the `config/test.php` file. The test configuration file includes `test_db.php`. It looks like so, by default:

```
<?php
$db = require __DIR__ . '/db.php';
// test database! Important not to run tests on production or development databases
```

```
$db['dsn'] = 'mysql:host=localhost;dbname=yii2basic_test';

return $db;
```

Before using fixtures, ensure the DSN value is correct for your test database. Next, ensure the test database has the same structure—but not data!—as the production database. You can do this by manually executing SQL commands or using `yii migrate` on the test database (if you’re already supporting migrations).

With the fixture class and data defined, and the test database configured, you can now use fixtures in a test suite. They are accessible through `$this->test`.

For example, because the orm module is enabled you can use these methods:

- `haveRecord()`, inserts a record (kind of confusing name)
- `grabRecord()`, retrieves a record
- `seeRecord()`, confirms that a record does exist
- `dontseeRecord()`, confirms a record does not exist

For example, this test confirms that the first fixture record was created in the database:

```
<?php
namespace Unit;
use \UnitTester;
class CommentTest extends \Codeception\Test\Unit {
    protected UnitTester $tester;
    public function testFetchComment() {
        $this->tester->seeRecord('app/model/Comment', ['user_id' => 23]);
    }
}
```

This could also be used after invoking a model’s `save()` method to confirm the save worked. You could also test:

- Record updates
- Record deletions
- Data validation routines
- Related models

{TIP} Another way of testing using data is to create stubs and mocks.
See the Codeception documentation for details.

In this chapter, I'm essentially covering the fundamentals of testing in Yii, walking through the mechanics more than the theory. That's because the theory can take months to master, and just an example or two or three can't really ingrain testing into you. That being said, I'll end this section with some general tips.

When it comes to designing unit tests for your code, your goals should be making your tests:

- Thorough
- As atomic as possible
- Easy to read, write, and execute
- Never superfluous (i.e., don't use any unnecessary assertions)

Your unit tests should never be used to validate user input or handle problems that could occur unexpectedly on a live site. That's what exception handling is for, after all. The point of unit tests is to confirm valid results when valid data is used and appropriate results when invalid data is used. In other words: is the code doing what it should for all possible cases? Test what absolutely should happen and what absolutely shouldn't.

Finally, to further your studies, learn as much as you can about PHPUnit. For additional expert advice on testing your PHP applications, check out [Grumpy Learning](#), and pretty much everything that Chris Hartjes does.

{TIP} Testing can be taken further to the concept of Test-Driven Development (TDD). With TDD, you define your tests first, and then write code that passes the tests.

The second testing component to be explained in this chapter is *functional* testing: looking at how the site operates in the browser. Put another way, functional testing helps verify that the site works for end users as it should by emulating GET and POST requests.

{TIP} To better understand the difference between unit and functional tests, look at the generated tests for the login and contact form pages.

Just as Codeception builds its unit testing on top of PHPUnit, Codeception builds its functional testing on top of Symfony's [BrowserKit](#) library, for simulating browser behavior.

To perform functional tests using Codeception, I recommend reviewing the Codeception documentation. Even though I'm not discussing acceptance tests in this chapter, you should also read that section of the documentation to understand all of the concepts.

That being said, creating and executing functional tests in Yii is quite similar to unit tests.

First, you define a class:

- Whose name ends with the string “Cest”
- That DOES NOT extend another class
- Saved in a file named **ClassName.php**, in **tests/functional**

“Cest” is not a typo; it’s short for “Codecept” + “Test”!

Again, within the class, each individual test is defined as a method. These methods are given meaningful names indicating the specific functionality being tested. Note that because functional tests aren’t normally tied to specific class methods, the names likely reflect either the view page being tested, or specific capabilities (e.g., login and logout).

Each method expects an argument of type **FunctionalTester**. This is an [actor](#) Codeception uses, a type of virtual identity.

```
<?php
class PageCest {
    public function testViewPage(\FunctionalTester $I)
    {}
    public function testEditPage(\FunctionalTester $I)
    {}
}
```

Within each test method, you’ll again run one assertion for every possible scenario you’ll want to test. With the assertions, you’ll also use Codeception actions, assertions, and grabbers (in their parlance). For example, you’ll want to load a specific page, maybe click on a link, enter values in a form, and so forth.

Good understanding of these options are best found under the [acceptance tests](#) **PhpBrowser** documentation. All are executed from the received **FunctionTester** argument, **\$I** as a norm.

You can see some example tests in the **tests/functional/ContactFormCest.php** file:

```
class ContactFormCest {
    public function _before(\FunctionalTester $I) {
        $I->amOnRoute('site/contact');
    }
    public function openContactPage(\FunctionalTester $I) {
        $I->see('Contact', 'h1');
    }
    public function submitEmptyForm(\FunctionalTester $I) {
        $I->submitForm('#contact-form', []);
        $I->expectTo('see validations errors');
        $I->see('Contact', 'h1');
        $I->see('Name cannot be blank');
```

```
$I->see('Email cannot be blank');
$I->see('Subject cannot be blank');
$I->see('Body cannot be blank');
$I->see('The verification code is incorrect');
}
```

The `_before()` method performs the necessary set up; in this case loading the specific page.

The first test method submits the form without providing any data. The test then creates a comment, using the `expectTo()` method. This isn't strictly necessary, but goes in keeping with functional testing as a narrative.

The test then confirms that various bits of text are present, including the word "Contact" within an HTML H1 tag. The remaining phrases are all expected when an empty contact form is submitted.

By comparison, this method tests proper submission of the contact form:

```
public function submitFormSuccessfully(\FunctionalTester $I) {
    $I->submitForm('#contact-form', [
        'ContactForm[name]' => 'tester',
        'ContactForm[email]' => 'tester@example.com',
        'ContactForm[subject]' => 'test subject',
        'ContactForm[body]' => 'test content',
        'ContactForm[verifyCode]' => 'testme',
    ]);
    $I->seeEmailIsSent();
    $I->dontSeeElement('#contact-form');
    $I->see('Thank you for contacting us. We will respond to you as soon as possible.');
}
```

The form is submitted with proper data and then the tests validate expected page responses.

Other useful action methods include:

- `click()`
- `fillField()`
- `setCookie()`
- `amLoggedInAs()`

With these, and the other commands and assertions, you can have Codeception literally click a checkbox that triggers some JavaScript and then confirm the result. Or you could see if text is present when authenticated.

After you've put so much time and effort into writing tests, you can become complacent, thinking you'll never have another bug. That's only true if your tests cover every bit of code. But how can you be sure of that? Well, Codeception has the ability to generate coverage reports that detail how much of your code is tested.

To create a report, your PHP installation needs to support [Xdebug](#) or [PCOV](#).

If you meet that requirement, enable code coverage in Codeception by editing the `codeception.yml` file:

```
coverage:
    enabled: true
    whitelist:
        include:
            - models/*
            - controllers/*
            - commands/*
            - mail/*
```

Without specifying the `include` parameter, Codeception checks code coverage for the entire application, including the entire `vendor` directory. For that reason it's best to limit code coverage checks to specific parts of your application, starting with your models and controllers.

With code coverage enabled, run (**Figure 21.4**):

```
./vendor/bin/codecept run --coverage
```

```
● ● ● larry@utm-ubuntu-20: /var/www/html/yii-demo — ss...
Code Coverage Report:
2023-02-06 22:33:27

Summary:
Classes: 75.00% (3/4)
Methods: 81.82% (18/22)
Lines: 89.11% (90/101)

app\controllers\SiteController
    Methods: 50.00% ( 4/ 8) Lines: 80.00% ( 44/ 55)
app\models>ContactForm
    Methods: 100.00% ( 3/ 3) Lines: 100.00% ( 18/ 18)
app\models\LoginForm
    Methods: 100.00% ( 4/ 4) Lines: 100.00% ( 15/ 15)
app\models\User
    Methods: 100.00% ( 7/ 7) Lines: 100.00% ( 13/ 13)
Remote CodeCoverage reports are not printed to console

Time: 00:00.240, Memory: 32.00 MB

OK (32 tests, 132 assertions)
larry@utm-ubuntu-20:/var/www/html/yii-demo$
```

Codeception spits the results out in the console. For a nicer interface, have it generate an HTML version:

```
./vendor/bin/codecept run --coverage-html
```

Codeception writes a coverage report to the `**tests/_output/coverage**` directory. Understand that while you might think 100% coverage is ideal, a target of 80% coverage or higher suffices in real-world applications.

{TIP} Many organizations use automated testing, not allowing code be moved to production until it has complete test coverage, and passes all tests.

Part IV

Completing Projects

Chapter 22

Creating a CMS

In my [other books](#), example chapters have been popular additions. Instead of trying to cover new material, the primary goal in example chapters is to present previously explained content within a more complete context. Accordingly, I've decided to include two sample chapters in this book as well.

In this sample chapter, I walk through the creation of a Content Management System (CMS). A CMS has been used as a hypothetical example throughout most of the book, so putting it all together in one place makes sense. One could say this is more of a blog than a full CMS, but the core concepts still apply regardless of the name.

I'll undoubtedly repeat this next sentiment frequently, but do understand that even this is not a complete example. The goal with the code and the chapter is to focus on the heart of the application, and how one might go about developing one. There are myriad ways this code and example could be improved, made more secure, made to perform better, and so forth. But it is a reasonable, working example that should be educational.

Every project begins not with a computer, but with pen and paper. (Or a note-taking application on your computer. Or a whiteboard.) You need to identify the project's goals, needs, target audience, and so on. The basis of this chapter's example is a CMS or a blog (more of a blog, really). Unlike the variations on this example that have littered the book thus far, in this chapter, the blog features:

- Ability for anyone to comment on a page (not just registered users)
- Three user roles: author, editor, and admin
- Password management using the most secure method
- Comments shown on each page
- HTML WYSIWYG editor for creating and editing posts
- More SEO-friendly URLs
- A custom widget to browse for posts by month

There's a lot to this application, even though it's not 100% finished.

At the end of the chapter, you'll see some of the notes and thoughts I have for how the example could be completed or expanded.

{NEW} The first edition of this book included instructions for applying Twitter Bootstrap to the application's design and layout. Yii 2 comes with that framework already enabled, so no additional steps are required.

The SQL commands for creating the database are as follows, with a few comments following each.

```
CREATE TABLE `page` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `user_id` int(10) unsigned NOT NULL,
    `live` tinyint(1) unsigned NOT NULL DEFAULT '0',
    `title` varchar(100) NOT NULL,
    `content` longtext,
    `date_updated` datetime DEFAULT NULL,
    `date_published` date DEFAULT NULL,
    PRIMARY KEY (`id`),
    KEY `fk_page_user_idx` (`user_id`),
    CONSTRAINT `fk_page_user` FOREIGN KEY (`user_id`)
        REFERENCES `user` (`id`) ON DELETE NO ACTION
        ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The `page` table is the most important one, representing a page of content (aka, a blog post). It references the `user` table, which reflects the page's author. The `live` column allows pages to be created in draft mode. There are also two dates: one for when the page was last updated and another for when it was, or will be, published.

```
CREATE TABLE `comment` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `page_id` int(10) unsigned NOT NULL,
    `username` varchar(45) NOT NULL,
    `user_email` varchar(60) NOT NULL,
    `comment` mediumtext NOT NULL,
    `date_entered` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    KEY `fk_comment_page_idx` (`page_id`),
    KEY `date_entered` (`date_entered`),
    CONSTRAINT `fk_comment_page` FOREIGN KEY (`page_id`)
        REFERENCES `page` (`id`) ON DELETE CASCADE
        ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The `comment` table links to the `page` table. Each comment has a username, a user's email address, the comment itself, and the date the comment was entered.

```
CREATE TABLE `user` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `username` varchar(45) NOT NULL,
    `email` varchar(60) NOT NULL,
    `pass` varchar(255) DEFAULT NULL,
    `type` enum('author','editor','admin') NOT NULL,
    `date_entered` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    UNIQUE KEY `username_UNIQUE` (`username`),
    UNIQUE KEY `email_UNIQUE` (`email`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

The `user` table represents the site's registered users: administrators, editors, and authors. Readers would be non-registered users in this version of a CMS.

With the database created, it's time to get started creating the application. Logically, this begins by using Composer to create a new basic application:

```
composer create-project
    --prefer-dist yiisoft/yii2-app-basic
    /path/to/yiobook-cms-ch22
```

For my own personal development, at this point I also:

- Create a new virtual host so that I can access the site using something like `http://ch22`
- Initialize Git

Next, open up the permissions on two directories: * `web/assets` * `runtime`

Next, edit the `db.php` and `web.php` configuration files. The most important steps are to:

- Configure the database connection
- Set the `cookieValidationKey` seed
- Enable SEO-friendly URLs

Next, use Gii to generate the models and create the CRUD functionality. Note that the new `User` class replaces the one included in the basic application.

All of the above is described in great detail in earlier book chapters.

At this point in the process, with very little actual work done, the heart of the application is in place. From here on out, it's mostly about editing (which is one thing I love about Yii). First, let's edit the model classes, specifically focusing on the rules and labels.

For the `User` class, the final rules are:

```
// Required fields when registering:  
[['username', 'email', 'pass'], 'required', 'on'=>'register'],  
// Required fields when logging in:  
[['username', 'pass'], 'required', 'on'=>'login'],  
// Encrypt the password when registering:  
[['pass'], 'encryptPassword', 'on'=>'register'],  
// Username must be unique and less than 45 characters:  
[['username'], 'unique'],  
[['username'], 'string', 'max' => 45],  
// Email address must be unique, an email address,  
// and less than 60 characters:  
[['email'], 'unique'],  
[['email'], 'email'],  
[['email'], 'string', 'max' => 60],  
// Set the type to "author" by default:  
[['type'], 'default', 'value' => 'author'],  
// Type must also be one of three values:  
[['type'], 'in', 'range' => ['author', 'editor', 'admin']]
```

Most of the rules come from the database constraints. There are specific rules that apply to registration (insertion of a new record), as the username and date wouldn't be required when logging in. The user type has three defined possible values—"author", "editor", and "admin", with "author" being the default. The password field is run through the `encryptPassword()` function (well, rule), upon registration. I'll return to that shortly.

The rules for the `Page` model are:

```
// Only the title is required from the user:  
[['title'], 'required'],  
// User must exist in the related table:  
[['user_id'], 'exist', 'skipOnError' => false, 'targetClass' => User::class,  
 'targetAttribute' => ['user_id' => 'id']],  
// Live needs to be Boolean; default 0:  
[['live'], 'integer'],  
[['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'], 'string'],  
[['content'], 'default', 'value' => NULL],
```

Only the page title is required. The user ID value must exist in the `user` table. By default, the `live` property is 0 (not live). The title is run through the `strip_tags()` function to remove any HTML.

Later in the `Page` definition, for the `user_id` label, the value is set to “Author”.

Finally, the rules in the `Comment` class:

```
// Required attributes (by the commentor):  
[['username', 'user_email', 'comment'], 'required'],  
// Must be in related tables:  
[['page_id'], 'integer'],  
[['page_id'], 'exist', 'skipOnError' => true, 'targetClass' => Page::class,  
 'targetAttribute' => ['page_id' => 'id']],  
// Strip tags from the comments:  
[['comment'], 'string'],  
[['comment'], 'filter', 'filter' => 'strip_tags'],  
// Username limited to 45:  
[['username'], 'string', 'max' => 45],  
// Email limited to 60 and must be an email address:  
[['user_email'], 'string', 'max' => 60],  
[['user_email'], 'email'],
```

Remember that in this version of the application, the commenter is not a registered user. The `username`, `email`, and `comment` values are required. The `page_id`, which wouldn't be set by the user, must match an existing page in the system. The comments are stripped of any PHP, HTML, or JavaScript using `strip_tags()`.

And that's it for editing the rules in the main models. You'll also need to edit the rules in the corresponding search models.

The site requires registered users to create and edit pages of content, so the next step is to implement registration, login, and logout functionality. The default Yii application defines all this capability, as explained in Chapter 11, “[User Authentication and Authorization](#)”. For this application the generated code just needs to be tweaked to use a database and encrypted password instead of static values.

Login requires an email address—not a username—and password. The `views/site/login.php` page is edited towards that end (**Figure 22.1**).

```
<?php $form = ActiveForm::begin([  
    'id' => 'login-form',  
    'layout' => 'horizontal',
```

Login

(Only site writers that have previously been asked to register can log in.)

Email

Pass

Remember Me

Login

Figure 22.1: The login form.

```
'fieldConfig' => [
    'template' => "{label}\n{input}\n{error}",
    'labelOptions' => ['class' => 'col-lg-1 col-form-label mr-lg-3'],
    'inputOptions' => ['class' => 'col-lg-3 form-control'],
    'errorOptions' => ['class' => 'col-lg-7 invalid-feedback'],
],
]); ?>
<?= $form->field($model, 'email')->textInput(['autofocus' => true]) ?>
<?= $form->field($model, 'pass')->passwordInput() ?>
<?= $form->field($model, 'rememberMe')->checkbox([
    'template' => "<div class=\"offset-lg-1 col-lg-3 custom-control
    custom-checkbox\">{input} {label}</div>\n
    <div class=\"col-lg-8\">{error}</div>",
]) ?>
<div class="form-group">
    <div class="offset-lg-1 col-lg-11">
        <?= Html::submitButton('Login', ['class' => 'btn btn-primary',
            'name' => 'login-button']) ?>
    </div>
</div>
<?php ActiveForm::end(); ?>
```

(Note that I use “pass”, not “password”, which I prefer because the `User` table uses the former.)

The login form is associated with the `LoginForm` model. The model needs to be updated to use the email address instead of the username. This is really just a matter of replacing uses of “username” with “email”. The same goes for replacing “password” with “pass”. Similar edits need to be made in `app/controllers/Site`.

For security purposes, disable auto-login in the configuration file:

```
'user' => [
    'identityClass' => 'app\models\User',
    'enableAutoLogin' => false,
],
```

The `User` class performs the rest of the login functionality. Chapter 11 explains how the default application defines an authentication class by implementing `IdentityInterface`. This application uses an Active Record model, so it needs to be updated to implement that interface, too.

```
class User extends \yii\db\ActiveRecord implements \yii\web\IdentityInterface
```

Next, the model needs to define five methods (to fulfill the requirements of the interface):

- `findIdentity()`
- `findIdentityByAccessToken()`
- `getId()`
- `getAuthKey()`
- `validateAuthKey()`

Any of those methods that won’t be used can be defined with an empty body (but the same name and parameters). I only chose to define `findIdentity()` and `getId()`.

```
# app/models/User.php
public function getId() {
    return $this->id;
}
public static function findIdentity($id) {
    return User::findOne($id);
}
```

To understand validation in this example, let’s look at the logic flow (you may want to look at your local, generated code or the GitHub repo to follow along):

1. The login form is presented to the user and submitted.
2. The Site controller handles the login form submission, and calls the `LoginForm` model's `login()` method.
3. The `login()` method calls the model's `validate()` method, which validates all the model data against its rules. This includes running the password through the `validatePassword()` method.
4. The `LoginForm` model's `validatePassword()` method loads the user instance and calls the `User` model's `validatePassword()` method. This is where the code actually compares the stored, encrypted password against the submitted, login password.

```
# app/models/User.php
public function validatePassword($loginPass) {
    return password_verify($loginPass, $this->pass);
}
```

That code assumes the users password was encrypted with `password_hash()` before storage. For that to work, the `encryptPassword()` method within `User` that properly encrypts the password upon registration. The class's rules already require its usage. Here's its definition:

```
# app/models/User.php
public function encryptPassword($attr, $params) {
    $this->pass = password_hash($this->pass);
}
```

Finally, if the user provides the correct credentials, the application should store their type, ID, and username in the session state (for example, for use when creating new pages). This should happen further up the chain of events, normally within a controller. But since the Site controller doesn't have the `User` model instance, the code can go in the `LoginForm` model:

```
# app/models/LoginForm.php
public function login() {
    if ($this->validate()) {
        $user = $this->getUser();

        // Store values in the session:
        $session = Yii::$app->session;
        if (!$session->isActive) $session->open();
        $session['type'] = $user->type;
        $session['user_id'] = $user->id;
        $session['username'] = $user->username;
```

```
        return Yii::$app->user->login($user, $this->rememberMe ? 3600*24*30 : 0);
    }
}
```

Now the site has proper user registration, login, and logout functionality.

The next step is to implement the heart of the application's functionality: the ability to create and display pages of content. While developing the project, I did this in two steps:

- Implement core capability
- Add in WYSIWYG support

{TIP} When developing sites, I normally find it most helpful to populate the database with sample records first, and then implement record creation via PHP.

Towards the first goal, start by updating **page/_form.php**:

- Remove the “user ID” element (whose value comes from the user storage)
- Convert the “live” element to a checkbox
- Remove the “date updated” element (because it’s populated by the rules)

At this point there’s a simple, but usable, HTML form. Later on, the WYSIWYG editor is added (**Figure 22.2**).

Finally, the **Page** model needs to pull in the user ID in order to represent the author of the page. That’s accomplished by adding this method to **models/Page.php**:

```
public function beforeValidate() {
    if(empty($this->user_id)) {
        $session = Yii::$app->session;
        if (!$session->isActive) $session->open();
        $this->user_id = $session['user_id'];
    }
    return parent::beforeValidate();
}
```

For new pages, where the **user_id** property would be empty, the current user’s ID is assigned to it. For existing pages, perhaps being updated by an editor or administrator, the **user_id** property is not touched. Thus the tie between a page and its original author is maintained.

Next, let’s change the page view from the default **DetailView** widget. First, add a method to **Page** that outputs the date in a formatted way:

Create Page

Live

Title

Content



A horizontal toolbar with various icons for text editing, including bold, italic, underline, and alignment options.

Date Published

Save

Figure 22.2: The add page form, with the WYSIWYG editor.

Alice in Wonderland

By larry | October 20, 2024

All the time they were playing the Queen never left off quarreling with the taken into custody by the soldiers, who of course had to leave off being the King, the Queen, and Alice, were in custody and under sentence of e

Figure 22.3: The start of a page's display.

```
# app/models/Page.php
public function formattedPublishedDate() {
    if ($this->date_published) {
        $formatter = \Yii::$app->formatter;
        return $formatter->asDate($this->date_published, 'long');
    }
}
```

Next, remove the widget from the view file and update the HTML as you see fit (**Figure 22.3**):

```
<div class="page-view">
    <h1><?= Html::encode($this->title) ?></h1>
    <p>By <?= $model->user->username ?> | <?= $model->formattedPublishedDate() ?></p>
    <div><?= $model->content ?></div>
</div>
```

Now that pages can be viewed, it's time to support the ability to add and view comments. This is a bit more complicated than some of the other examples in the book because the comment form and display of existing comments needs to happen within the context of the page view files.

Working backwards from the view files, the comment form needs to be pulled into the page view (**Figure 22.4**):

```
<div>
    <h3>Leave a Comment</h3>
    <?php if (Yii::$app->session->hasFlash('commentSubmitted')): ?>
        <p class="alert alert-success">
            <?php echo Yii::$app->session->setFlash('commentSubmitted'); ?>
        </p>
    <?php else: ?>
```

Leave a Comment

Username

User Email

Comment

Post Comment

Figure 22.4: The comment form.

```
<?php echo $this->render('@app/views/comment/_form', [
    'model'=>$comment,
]); ?>
<?php endif; ?>
</div>
```

The `render()` method pulls in the comment form from the `views/comment` directory. The `$comment` instance is required on that view, so it'll need to be created in the controller and passed along. Just before the `render()` code, a flash message is used to report upon a successful comment post.

For the comment form itself, two updates are required from the Gii-generated code:

- Ajax validation is enabled
- The submit button's label is changed to "Post Comment"

```
# views/comments/_form.php
<?php $form = ActiveForm::begin([
    'id' => 'comment-form',
    'enableAjaxValidation' => true,
]); ?>
```

The Ajax validation isn't obligatory, but it's a nice addition that Yii supports out of the box.

The most complicated part of handling comments within a page view takes place in the "Page" controller. Within the `actionView()` method, a `Comment` object needs to be created. This object could be empty upon first viewing a page or come from the comment form data upon a comment submission. Create a new method within the "Page" controller for handling both scenarios. The "view" action is then updated, too:

```
# controllers/PageController.php
public function actionView($id) {
    $page = $this->findModel($id);
    $comment = $this->newComment($page);
    return $this->render('view', [
        'model' => $page,
        'comment' => $comment,
    ]);
}
```

The key line there is `$comment = $this->newComment($page);`. The `newComment()` method, also defined within the "Page" controller, looks a lot like the "create" action would in the "Comment" controller:

```
public function newComment($page) {
    $comment=new Comment;
    $comment->page_id = $page->id;
    return $comment;
}
```

The changes to this point allow a page to display a new comment form. Now code needs to handle that form's submission. Update the page view method accordingly:

```
public function actionView($id) {
    $page = $this->findModel($id);
```

```
$comment = $this->newComment($page);

if (\Yii::$app->request->isAjax && $comment->load(\Yii::$app->request->post())) {
    \Yii::$app->response->format = Response::FORMAT_JSON;
    return ActiveForm::validate($comment);
} elseif ($this->request->isPost) {
    if ($comment->load($this->request->post()) && $comment->save()) {
        \Yii::$app->session->setFlash('commentSubmitted', 'Thank you for your comment');
        return $this->render('view', [
            'model' => $page,
            'comment' => $this->newComment($page),
        ]);
    }
} else {
    return $this->render('view', [
        'model' => $page,
        'comment' => $comment,
    ]);
}
}
```

Finally, the page ought to display existing comments, too. The `Page` model already has a defined relationship with `Comment`, so the view file can use lazy loading to find any comments for that page.

```
<div class="comments">
    <h3>Comments</h3>
    <?php
    $comments = $model->comments;
    if (count($comments) >= 1) {
        $data = new ArrayDataProvider([
            'allModels' => $comments,
        ]);
        echo ListView::widget([
            'dataProvider' => $data,
            'itemView' => '_comment',
        ]);
    } else {
        echo '<b>Be the first to comment on this page!</b>';
    }
    ?>

</div>
```

An easy way to display the comments is to use the `ListView` widget. It needs to be

Comments

Showing 1-2 of 2 items.

 Lorem ipsum dolor sit amet, c
 accusam et justo duo dolores
 elitr, sed diam nonumy eirmod
 gubergren, no sea takimata sa
 dolore magna aliquyam erat, s
 sit amet.

testing

What a wonderful thing.

Jane Doe

Figure 22.5: The display of comments.

provided with a data source, populated by the returned `Comment` objects.

The `ListView` widget uses the `views/page/_comment` file as the item display. It can be as simple or complex as you'd like (**Figure 22.5**).

```
<?php
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
?>
<hr>
<div class="comment">
    <p><?= HtmlPurifier::process($model->comment) ?></p>
    <p><?= HtmlPurifier::process($model->username) ?></p>
</div>
```

Thanks to the built-in Yii widget, displaying a paginated list of comments is a breeze.

Content

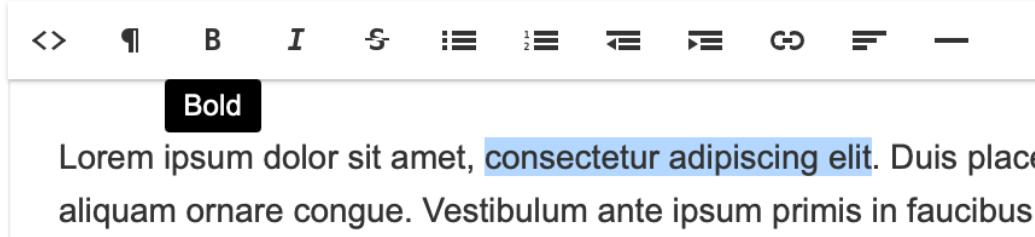


Figure 22.6: The WYSIWYG editor for pages.

The blog needs a WYSIWYG editor, of course, so the next step is to install it. In Chapter 13, “[Using Extensions](#),” used CKEditor, via the [editMe](#) extension. This example uses [Imperavi’s Redactor](#) tool, which gets a lot of love and attention these days. Although Redactor is a commercial product, Yii has purchased an OEM license for it, and it can be installed through the [Imperavi Redactor widget](#).

First, install the widget using Composer:

```
composer require --prefer-dist vova07/yii2-imperavi-widget "*"
```

(You can also add it to your Composer file and then do an update.)

Next the page’s `_form.php` file should be updated to use the widget for the content text area:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;
use vova07\imperavi\Widget;
?>

<div class="page-form">

    <?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'title')->textInput(['maxlength' => true]) ?>

    <?= $form->field($model, 'content')->widget(Widget::className()); ?>
```

Basic WYSIWYG functionality is that simple! This creates a barebones editor (**Figure 22.6**).

You can tweak its functionality and behavior using settings explained in the widget and Redactor docs.

The steps to this point focus on the big, core issues, but there are lots of little things to change. Let's quickly look at a few.

When developing an application like this, you'll find that the generated access rules often conflict with what you're trying to do. You'll create new actions that, by default, are denied, and you'll leave open access to actions that won't actually be used. In the course of developing and testing, you'll quickly catch the former problems, but the latter won't be obvious. For that reason, you'll want to go through your access rules and confirm that you've adjusted all of them accordingly. If an action shouldn't be executable at all—for example, the site shouldn't allow for the deletion of users, then remove the action method itself.

In this particular example, there's one more permission to be tweaked, this one being more contextual. Any editor or administrator should be able to edit any page, but the author of the page should be the only “author” type that can edit it.

Restricting updates to logged-in users is easily done in the access rules:

```
# app/controllers/PageController.php
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::class,
            'only' => ['admin', 'create', 'update'],
            'rules' => [
                [
                    'allow' => true,
                    'actions' => ['admin', 'create', 'update'],
                    'roles' => ['@'],
                ],
            ],
        ],
    ];
},
```

But to enforce the restriction that only the page's author can edit the page requires logic within the “update” action itself:

```
public function actionUpdate($id) {
    $model=$this->loadModel($id);
    if ((\Yii::$app->user->type == 'author') &&
        (\Yii::$app->user->id != $model->user_id)) {
        throw new CHttpException(403, 'You do not have
            permission to edit this page.');
}
```

Showing 1-2 of 2 items.

#	ID	Page	Username	User Email	Comment	
1	1	This is the First Test	testing	blah@blah.com	Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.	  
2	2	This is the First Test	Jane Doe	jane@example.net	What a wonderful thing.	  

Figure 22.7: The admin page for comments.

If the user type is “author”, but the current user’s ID doesn’t match the page’s author ID, an exception is thrown.

The default `GridView` used on the admin pages is a pretty good way for administering records. You won’t want to keep using it for all projects, I wouldn’t think, but when you do, you certainly want to make a few edits. For this example, I made slight changes to the comments, users, and pages admin `GridView` widgets.

On the comments admin page, the page’s title would be more useful than the ID (**Figure 22.7**):

```
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        'id',
        [
            'header'=>'Page',
            'value'=> function($data) {
                return $data->page['title'];
            }
        ],
        'username',
        'user_email:email',
        'comment:ntext',
        // 'date_entered',
        [
            'class' => 'yii\grid\ActionColumn'],
        ],
    ]); ?>
```

On the users admin page, the user type should be filter-able by the allowed values.

Figure 22.8: The pages admin page.

On the pages admin page, additional changes are needed (**Figure 22.8**):

- The author's name should be shown, not the author's ID
 - The live column should say "Live" and "Draft", not 1/0
 - A preview of the page should be shown for the content, not the entire page

Here's how those first changes are made, using what was previously explained in Chapter 12:

```
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        'id',
        [
            'header'=>'Author',
            'value'=> function($data) {
                return $data->user->username;
            }
        ],
        [
            'header'=>'Status',
            'value'=> function($data) {
                return ($data->live == 1) ? "Live" : "Draft";
            }
        ],
    ],
])
```

```
'title',
'content:ntext',
//'date_updated',
//'date_published',
['class' => 'yii\grid\ActionColumn'],
],
]); ?>
```

To present a preview of a page, instead of all the content, define a `getPreview()` method in the `Page` class that returns just the first few sentences of text. Then use call that method as the value for that column.

This chapter walks through the main functionality and issues for creating a fuller CMS example. Of course, there are numerous ways the application could be implemented differently, and even with all the code and changes made so far, the application is still not production-ready. Here, then, are some ways the application could be further improved:

- Making a proper home page
- Create a page synopsis option to be used for SEO purposes and page introductions
- Support an approval mechanism for comments
- Incorporate ElasticSearch, using the detailed instructions explained from Chapter 19
- Support tags
- Remove the “delete” action for pages and users
- Restrict access to the user registration page so only known people (that is, those to become authors or editors) can register
- Show recent comments in a sidebar
- Add a file upload mechanism for the HTML editor
- Ajax-ify more functionality
- Cache, cache, cache!
- Use blog titles in all URLs
- Add links to edit the content to the public view of a blog page for logged in users
- Complete the user functionality (such as recover password, change password, and the ability to change a user’s type)

Chapter 23

Making an E-commerce Site

As with the previous chapter, the primary goal in this chapter is to present already explained content within a more complete context. This chapter walks through the creation of an e-commerce site.

As before, the goal with the code and the chapter is to focus on the heart of the application, and how one might go about developing an e-commerce site. There are myriad ways this code and example could be improved, made more secure, made to perform better, and so forth. But it is a reasonable, working example, that should be educational.

[[TODO]]

You can find the complete code in a [GitHub repository](#). You can also use the commit history to roughly track the development of this application. The rest of the chapter approximately maps to the same development path as that commit history.

A mitigating factor when developing an e-commerce site is distinguishing between selling physical goods and selling digital goods. With the former, you have to worry about inventory, shipping and handling, accepting shipping addresses, and much, much more. Not to mention, an actual person has to handle the receiving of new inventory and shipping out orders.

Conversely, at most, a purveyor of digital goods only needs to worry about licensing or Digital Rights Management (DRM). For this reason, a virtual e-commerce site practically runs itself once developed.

Considering the constraints of time and space, this chapter creates an e-commerce site that sells digital goods. Specifically, it sells books in PDF format.

The features to implement include:

- List of books to purchase on the main page
- Detail book item view
- Shopping cart for visitors

- Purchases made through a separate module
- Use of a helper class
- PDF downloads

This site uses the default Yii layout. But you need to copy the **images** directory (in the webroot), from the GitHub repo, as that contains the book images. And the **books** directory (on GitHub) has the PDFs of the books themselves (these aren't actual PDFs of my books, they're dummy files).

As with the previous chapter, there's a lot to this application, even though it's not 100% finished. At the end of the chapter, you'll see some notes for how the example could be completed or expanded.

You can find the SQL statements required to create the database in the application's **data/yiobook2-ch23.sql** file, but the chapter quickly runs through them here as well. The only table not referenced below is the one created by the "pay" module extension, to be covered later in the development sequence.

The name of the database is "yiobook2_ch23".

The main product table is **book**:

```
CREATE TABLE IF NOT EXISTS `yiobook2_ch23`.`book` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(255) NOT NULL,
  `price` INT UNSIGNED NOT NULL,
  `description` TEXT NULL,
  `author` VARCHAR(60) NOT NULL,
  `filename` VARCHAR(60) NOT NULL,
  `date_published` DATE NOT NULL,
  PRIMARY KEY (`id`)
)
ENGINE = InnoDB;
```

Everything should be pretty obvious there. Note that all prices in the database are stored as integers. The **filename** field refers to the file name of the PDF that the customer would actually purchase (these are found in the **books** directory).

In a more fleshed-out example, create a separate **author** table, and link **author** to **book** through an intermediary table.

Populate the **book** table using these insert commands (use the SQL file, found in the code on GitHub, if you'd rather):

```
INSERT INTO `book` VALUES
(1, 'The Yii Book', 2000, 'Lorem ipsum...', 'Larry Ullman',
 'yiobook.pdf', '2024-12-31'),
(2, 'Effortless E-commerce with PHP and MySQL (2nd Edition)', 4499,
 'Lorem ipsum...', 'Larry Ullman', 'effortless-ecom-2nd.pdf', '2013-11-30'),
```

```
(3,'PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide (4th  
Edition)',40,'Lorem ipsum...', 'Larry Ullman', 'php-mysql-4th.pdf','2012-08-16'),  
(4,'PHP for the Web: Visual QuickStart Guide (4th Edition)',2499,  
'Lorem ipsum...', 'Larry Ullman', 'php-4th.pdf','2012-01-10'),  
(5,'Modern JavaScript: Develop and Design',4499,'Lorem ipsum...',  
'Larry Ullman', 'modern-javascript.pdf','2013-02-08'),  
(6,'Advanced PHP and Object-Oriented Programming: Visual QuickPro  
Guide (3rd Edition)',4499,'Lorem ipsum...', 'Larry Ullman',  
'adv-php-3rd.pdf','2012-03-19');
```

The `cart` table stores metadata about the shopping cart. To support the ability to have a shopping cart without being registered, a session ID, stored in a cookie, represents the customer.

```
CREATE TABLE IF NOT EXISTS `yiibook2_ch23`.`cart` (  
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,  
  `customer_session_id` CHAR(32) NOT NULL,  
  `date_modified` TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP,  
  PRIMARY KEY (`id`),  
  UNIQUE INDEX `customer_session_id_UNIQUE` (`customer_session_id` ASC)  
) ENGINE = InnoDB;
```

The `cart_content` table stores the particular items in a given shopping cart. That is the book ID and the quantity:

```
CREATE TABLE IF NOT EXISTS `yiibook2_ch23`.`cart_content` (  
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,  
  `cart_id` INT UNSIGNED NOT NULL,  
  `book_id` INT UNSIGNED NOT NULL,  
  `quantity` TINYINT UNSIGNED NOT NULL DEFAULT 1,  
  PRIMARY KEY (`id`),  
  INDEX `fk_cart_idx` (`cart_id` ASC),  
  CONSTRAINT `fk_cart_content`  
    FOREIGN KEY (`cart_id`)  
    REFERENCES `yiibook_ch23`.`cart` (`id`)  
    ON DELETE NO ACTION  
    ON UPDATE NO ACTION,  
  INDEX `fk_cart_books_idx` (`book_id` ASC),  
  CONSTRAINT `fk_cart_books`  
    FOREIGN KEY (`book_id`)  
    REFERENCES `yiibook_ch23`.`book` (`id`)  
    ON DELETE NO ACTION  
    ON UPDATE NO ACTION)  
ENGINE = InnoDB;
```

The `customer` table represents people that have made a purchase. It stores an email address, the password, and whether or not the customer wants to receive email updates. That functionality isn't defined in the application yet, but could be easily added.

```
CREATE TABLE IF NOT EXISTS `yiibook2_ch23`.`customer` (
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
    `email` VARCHAR(80) NOT NULL,
    `pass` VARCHAR(255) NULL,
    `get_emails` TINYINT(1) UNSIGNED NOT NULL DEFAULT 0,
    `date_entered` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    UNIQUE INDEX `email_UNIQUE` (`email` ASC)
)
ENGINE = InnoDB;
```

The `order` table stores the metadata about an individual order. This includes the:

- Customer ID
- Payment ID
- Order total
- Date

The specifics of an order (what was purchased) is stored separately:

```
CREATE TABLE IF NOT EXISTS `yiibook2_ch23`.`order` (
    `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
    `customer_id` INT UNSIGNED NOT NULL,
    `payment_id` INT UNSIGNED NOT NULL,
    `total` INT UNSIGNED NOT NULL,
    `date_entered` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    INDEX `fk_orders_users_idx` (`customer_id` ASC),
    CONSTRAINT `fk_orders_users`
        FOREIGN KEY (`customer_id`)
        REFERENCES `yiibook2_ch23`.`customer` (`id`)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT
)
ENGINE = InnoDB;
```

The `order_content` table stores the particular items included in an order: the book, the quantity of that book, and the price paid per copy.

```
CREATE TABLE IF NOT EXISTS `yiibook2_ch23`.`order_content` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `order_id` INT UNSIGNED NOT NULL,
  `book_id` INT UNSIGNED NOT NULL,
  `quantity` TINYINT UNSIGNED NOT NULL DEFAULT 1,
  `price_per` INT UNSIGNED NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_order_content_order1_idx` (`order_id` ASC),
  INDEX `fk_order_content_book1_idx` (`book_id` ASC),
  CONSTRAINT `fk_order_content_order1`
    FOREIGN KEY (`order_id`)
    REFERENCES `yiibook_ch23`.`order` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_order_content_book1`
    FOREIGN KEY (`book_id`)
    REFERENCES `yiibook_ch23`.`book` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

The SQL file in the application code (found on GitHub) includes two other tables, not used by the code. The end of the chapter explains the intent of both.

(These next instructions are almost verbatim from that in the previous chapter.)

With the database created, it's time to start creating the application. Logically, this begins by using Composer to create a new basic application:

```
composer create-project
  --prefer-dist yiisoft/yii2-app-basic
  /path/to/yiibook-cms-ch23
```

For my own personal development, at this point I also:

- Create a new virtual host so that I can access the site using something like **http://ch23**
- Initialize Git

Next, open up the permissions on two directories: * **web/assets** * **runtime**

Next, edit the **db.php** and **web.php** configuration files. The most important steps are to:

- Configure the database connection

- Set the `cookieValidationKey` seed
- Enable SEO-friendly URLs

Next, use Gii to generate the models and create the CRUD functionality.

All of the above is described in great detail in earlier book chapters.

Having implemented all the models and CRUD functionality, it's editing time! Surprisingly, in this case, I didn't make any major changes to the models at this point. Later in the development, new methods will be added. But based upon the database table definitions, and particularly the foreign key constraints that turn into model relations, no major model edits are needed yet.

The home page for the site shows a list of books available for sale (**Figure 23.1**).

This is done through the "book" controller, so the site should be configured to use it as the default:

```
# config/web.php
'defaultRoute' => 'book',
```

With that in place, the "index" action of the "book" controller becomes the default (because "index" is the default action of any controller). That method creates a data provider of Book objects and passes it to the view file:

```
public function actionIndex() {
    $searchModel = new BookSearch();
    $dataProvider = $searchModel->search($this->request->queryParams);

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

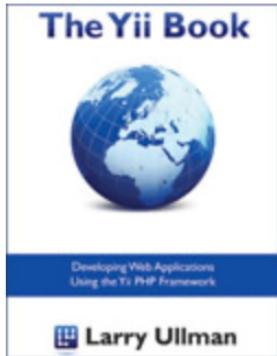
By default, the "index" view file uses the `GridView` widget to render its content:

```
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    ...
```

`GridView` is more of an administrative-type widget; the home page is better served by switching to `ListView`:

Books

Showing 1-6 of 6 items.

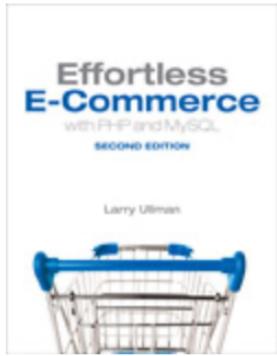


Title: The Yii Book

Price: USD 20.00

Author: Larry Ullman

Date Published: December 31, 2024



Title: Effortless E-commerce with PHP and MySQL (2nd Edition)

Price: USD 44.99

Author: Larry Ullman

Date Published: November 30, 2013

Figure 23.1: Books listed on the home page.

```
<?= ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_book'
]); ?>
```

With `ListView` in place, individual records are displayed by creating `views/-book/_book.php`:

```
<?php
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
use app\components\Utilities;
?>

<div class="view">
    <?= Html::a('', [
        'book/view', 'id' => $model->id]); ?>
    <br />
    <b><?= $model->getAttributeLabel('title'); ?></b>
    <?= Html::encode($model->title); ?>
    <br />
    <b><?= $model->getAttributeLabel('price'); ?></b>
    <?= Utilities::formatAmount($model->price); ?>
    <br />
    <b><?= $model->getAttributeLabel('author'); ?></b>
    <?= Html::encode($model->author); ?>
    <br />
    <b><?= $model->getAttributeLabel('date_published'); ?></b>
    <?= Utilities::formatDate($model->date_published); ?>
    <br />
</div>
```

First, the book's image is shown, and hyperlinked to the book's individual view page. For ease of implementation, each book's image is the book's ID plus the ".jpg" extension, found in the `images` directory.

Two pieces of information are formatted: the price and the date published. Because the prices are in integers in the database, they need to be formatted to cents in multiple places on the site. Many dates—a book's published date, an order date, and so on—need to be formatted, too.

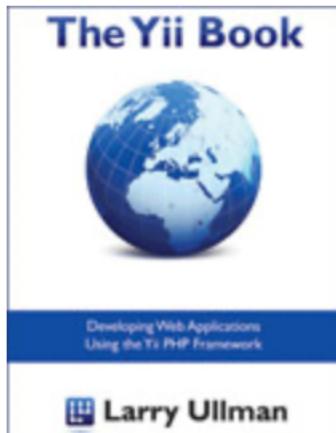
To avoid replicating code in multiple places, create a `Utilities` class. It'll be defined shortly. But, here, two methods in that class are called to format the price and date, accordingly.

By clicking on the image, the user is taken to the book's details page. Obviously you could do much more with the aesthetics and the user interface here, but the

functionality is sufficient for the purposes of this example. (And thanks to the use of the widget, making bold changes to the display is easy.)

Individual books are shown on the book’s “view” page, which is also where a book can be added to the shopping cart. The default code uses `DetailView`. Instead, this example mostly replicates what’s shown on the home page, with the additional description of the book and a link to add the book to the cart (**Figure 23.2**).

The Yii Book



Title: The Yii Book

Price: USD 20.00

Author: Larry Ullman

Date Published: December 31, 2024

Description: Lorem ipsum dolor sit am
vero eos et accusam et justo duo dolor
sadipscing elitr, sed diam nonumy eirm
Stet clita kasd gubergren, no sea takim
invidunt ut labore et dolore magna aliqu
est Lorem ipsum dolor sit amet.

[Add to Cart](#)

```
<div class="book-view">
    <h1><?= Html::encode($this->title) ?></h1>
    <br />
    <p>
        <b><?= $model->getAttributeLabel('title'); ?></b>
        <?= Html::encode($model->title); ?>
    </p>
    <p>
        <b><?= $model->getAttributeLabel('price'); ?></b>
        <?= Utilities::formatAmount($model->price); ?>
    </p>
    <p>
        <b><?= $model->getAttributeLabel('author'); ?></b>
        <?= Html::encode($model->author); ?>
    </p>
    <p>
        <b><?= $model->getAttributeLabel('date_published'); ?></b>
        <?= Utilities::formatDate($model->date_published); ?>
    </p>
    <p>
        <b><?= $model->getAttributeLabel('description'); ?></b>
        <?= Html::encode($model->description); ?>
    </p>
    <p>
        <?= Html::a('Add to Cart', ['/cart/add', 'id' => $model->id],
                    ['class' => 'btn btn-primary']) ?>
    </p>
</div>
```

As you can see, the “Add to Cart” button at the bottom goes to the “cart” controller’s “add” action, passing along the book ID. And, as with the home page, this could be made to look a lot nicer, which would be easy to do.

As already stated, some functionality is required multiple times in the application:

- Conversion of amounts from integers to decimals (that is, from cents to dollars and cents)
- Proper formatting of dates
- Cart retrieval

Rather than duplicating the same code in multiple models, it makes sense to create a helper class instead. This is just a PHP class that does not extend any Yii classes. It has only static methods:

```
<?php
namespace app\components;
class Utilities {
    public static function formatAmount($amount) {
        return \Yii::$app->formatter->asCurrency($amount/100, 'USD');
    }

    public static function formatDate($date, $format = 'long') {
        return \Yii::$app->formatter->asDate($date, $format);
    }
}
```

The `formatAmount()` method takes an amount and returns it divided by 100, formatted as USD using Yii's built-in formatter. By creating this method, it wouldn't be hard to support multiple currencies, passing a currency flag as the second argument, and changing the formatting and currency symbol accordingly.

The `formatDate()` method takes a date and a format as its two arguments. The format is the display date's format, which can vary based upon the destination. The method then returns a formatted date. Again, because this has been defined in a helper class, it'd be easy to make locale-aware, if you so chose.

The `Utilities` class file—`Utilities.php`—needs to be in the `protected/components` folder to make the application aware of its existence.

The third method in this class—`getCart()`—is explained separately next, as it's the most complicated helper method.

Successful e-commerce provides a product the customer wants to purchase and then gets out of the way of them buying it. From a user experience and development perspective, anything you can do to facilitate completing a purchase is a good thing; any obstacle you can remove is a benefit. Towards that end, the ability to use a shopping cart before registration and login is a plus.

To do that, a cart session identifier is stored in a cookie and in the database. That's the heart of the functionality. Then, when a user accesses the site, some code must look for the presence of the cookie. If one exists, the current cart session is used and extended. If no cookie exists, a new cart session is created. Because many of the site's controllers need access to the cart, the ability to get a reference to the cart is placed in the `Utilities` class. The comments inline explain what the code is doing:

```
# components/Utilities.php
public static function getCart() {

    // Get or create the cart session ID:
    $cookies = Yii::$app->request->cookies;
    if ($cookies->has('SESSION')) {
```

```
        $sess = $cookies['SESSION']->value;
    } else {
        $sess = bin2hex(openssl_random_pseudo_bytes(16));
    }

    // Send the cookie:
$cookies = Yii::$app->response->cookies;
$cookies->add(new \yii\web\Cookie([
    'name' => 'SESSION',
    'value' => $sess,
    'expire' => time()+(60*60*24*30)
]));

$cart=Cart::find()->where(['customer_session_id' => $sess])->one();
if($cart==null) {
    $cart = new Cart();
    $cart->customer_session_id = $sess;
    $cart->save();
}
return $cart;

}
```

This code can be used like so:

```
use app\components\Utilities;
$cart = Utilities::getCart();
```

Because the method only relies upon the cookie, it can be called without being passed any parameters. Either an existing cart or a new, empty, cart is returned (as a `Cart` object in either case).

With the ability to create or recreate a cart in place, the next step is to implement “add to cart” functionality. The “cart” controller’s “add” action accepts a product ID as a parameter (specifically, the ID of the product being added to the cart). The method first gets a reference to the cart:

```
public function actionAdd($id) {
    $cart = Utilities::getCart();
```

Next, the method needs to see if the item is already in the shopping cart:

```
$item = CartContent::find()
    ->where(['cart_id' => $cart->id, 'book_id' => $id])
    ->one();
```

The item is already in the cart if the book ID matches the passed ID and the cart ID matches the user's cart ID.

Next, if the item is in the cart, the quantity is updated. Thanks to this code, if the customer adds an item to the cart once, and then adds it again, the customer has 2 of that item in the cart.

If the item is not in the cart, it needs to be added. This is done through the `CartContent` model instance:

```
if($item!==null) {
    $item->quantity = $item->quantity + 1;
} else {
    $item = new CartContent();
    $item->cart_id = $cart->id;
    $item->book_id = $id;
    $item->quantity = 1;
}
```

Finally, the new item needs to be saved, and the “view cart” view file rendered:

```
$item->save();
return $this->render('view',array(
    'model'=>$cart
));
```

The “view cart” view file is the only cart view file used (that is, there's no “update”, “create”, or “index” view file used by the application). The “view cart” file is shown when a new product is added to the cart or when the customer clicks a “view cart” link. The view is also used upon updating the cart.

The default “view” file uses a `DetailView` widget, but `GridView` is more appropriate for outputting rows of information. This is a pretty heavy customization of the widget, explained in detail.

Before getting into the view, a method must be defined on the `Cart` class that returns the cart contents. Specifically, the method returns the contents as an `ActiveDataProvider` instance of `CartContent` objects for the given cart. The existing `getCartContents()` method, generated by Yii, already returns an `ActiveQuery` of related `CartContents` objects for the cart. Simply use that method as the query source for the new `ActiveDataProvider`:

```
# models/Cart.php
public function getCartContents() {
    return $this->hasMany(CartContent::class, ['cart_id' => 'id']);
}
public function getContents() {
    return new ActiveDataProvider([
        'query' => $this->getCartContents()
    ]);
}
```

(Alternatively, one could update `getCartContents()` to return the `ActiveDataProvider` but as this method might be used in other situations, creating a second, more specific method, makes more sense.)

It also helps if the `Cart` class has a method for returning the order total. This can efficiently be calculated using a SQL command:

```
// Returns the total of the items in a cart (as an integer).
public function getTotal() {
    $q = new Query();
    return $q->select('SUM(quantity * book.price)')
        ->from('cart_content')
        ->join('INNER JOIN', 'book', 'book_id=book.id')
        ->where(['cart_id' => $this->id])
        ->scalar();
}
```

Both of these new `Cart` class methods are used by the view file. Again, the view file uses a `GridView` widget to display the cart contents:

```
# views/cart/view.php
<?= GridView::widget([
    'dataProvider' => $model->getContents(),
    'columns' => [
        [
            'header' => 'Title',
            'enableSorting' => false,
            'format' => "html",
            'value' => function ($data) {
                return Html::a($data['book']['title'], ['book/view',
                    'id' => $data['book']['id']]);
            },
        ],
        [
            'header' => 'Price',
        ],
    ],
])
```

```
'enableSorting' => false,
'value' => function ($data) {
    return Utilities::formatAmount($data['book']['price']);
},
],
[
    'header' => 'Quantity',
    'enableSorting' => false,
    'value' => 'quantity',
],
```

The `dataProvider` for the widget is `Cart::getContents()`, the method just explained.

For the columns, the widget first displays the book's title, price, and the quantity of that item within the cart. The title is linked to the view page for the book. To properly format the book's price, it's run through the `formatAmount()` utility method (**Figure 23.3**).

Title	Price	Quantity
The Yii Book	USD 20.00	7
Modern JavaScript: Develop and Design	USD 44.99	5

In the last column, the widget is going to display buttons. By default, the buttons are to view, edit, and delete the item. For the time being, just delete option is implemented. The delete link goes to a custom action for removing items from the cart:

```
[

    'header' => 'Remove',
    'class' => 'yii\grid\ActionColumn',
    'template' => '{remove}',
    'buttons' => [
        'remove' => function ($url, $model, $key) {
            return Html::a('Remove', ['cart/delete',
                'book_id' => $model['book_id']]);
        },
    ],
]
```

The “delete” item links to “cart/delete”, passing along the book's ID.

(The widget is then completed with `) ,)); ?>`.)

Here is the “actionDelete” definition, from the “Cart” controller:

Total: USD 364.95

[Checkout](#)

Figure 23.2: The end of the cart display.

```
public function actionDelete($book_id) {  
  
    // Need the cart:  
    $cart = Utilities::getCart();  
    $cmd = \Yii::$app->db->createCommand()->delete('cart_content',  
        ['cart_id' => $cart->id, 'book_id' => $book_id]);  
    $cmd->execute();  
  
    return $this->render('view', array(  
        'model' => $cart  
    ));  
  
}
```

This method runs a DELETE query on the database, for the given cart ID and book ID.

With all this in place, users can now add items to their cart, view the cart, remove items from the cart, and quickly go to view an item in their cart. There's still room for improvement, of course: the ability to update quantities within the cart would be necessary, and a quick one-click "clear cart" option would make sense. (The "clear cart" functionality in terms of clearing the database is implemented as part of the recording orders process; it just needs to be connected to a cart controller method.)

Finally, the view cart page shows the order total and creates a link to checkout (**Figure 23.4**):

```
<p><b>Total</b>: <?php echo Utilities::formatAmount($model->getTotal()); ?></p>  
<p><?php echo Html::a('Checkout', array ('/pay')) ; ?></p>
```

To complete the checkout process, the customer has to pay. To do that, this app uses the Stripe payment module developed and explained in Chapter 19, “[Extending Yii](#)”. In the process of integrating the payment module, I've slightly modified the original extension. The changes:

- Fill in some gaps from the original incarnation

- Demonstrate how extensions might evolve in time
- Make the extension more production ready

{NOTE} For detailed explanation on the extension itself, see Chapter 19. I'll point out that I worked for Stripe from 2013 until 2022. Also, while Stripe isn't available in every country, you can use a non-active test account in any country, free of charge.

To install the extension:

1. Grab the code from this chapter's [GitHub repository](#), if you haven't already.
2. Copy "modules/pay" to your application directory.
3. Add the module to your configuration:

```
# config/web.php
'modules' => [
    'pay' => [
        'class' => 'app\modules\pay\Module',
    ],
],
```

Next, there's a migration to be run that creates the `payment` database table:

```
$this->createTable('{{%payment}}', [
    'id' => $this->primaryKey(),
    'payment_id' => $this->string(100)->notNull(),
    'email' => $this->string(100)->notNull(),
    'amount' => $this->integer()->unsigned()->notNull(),
    'created_at' => $this->timestamp()->notNull(),
]);
$this->createIndex('payment_id', 'payment', 'payment_id', true);
$this->createIndex('email', 'payment', 'email');
```

This table is particular to the extension. It stores the pertinent details about each Stripe charge.

To apply the migration, run this command from your application directory:

```
yii migrate --migrationPath=@app/modules/pay/migrations
```

Next, the module is configured in the main configuration file:

```
# config/web.php
'modules' => [
    'pay' => [
        'class' => 'app\modules\pay\Module',
        'publishable_key' => 'pk_test_XXXX',
        'secret_key' => 'sk_test_XXXX',
        'redirectOnSuccess' => '/order/create'
    ],
],
],
```

The public and private keys are required, and come from the Stripe account. The `redirectOnSuccess` property is helpful addition: It allows you to set a new destination for the user upon a successful charge.

The module now has these properties:

```
class Module extends \yii\base\Module {
    public $publishable_key;
    public $secret_key;
    public $redirectOnSuccess;
    public $amount;
```

The amount needs to be set dynamically. The `PayModule` class's `init()` method is where that should happen, as it's designed for code that initializes the extension:

```
public function init() {
    parent::init();

    if ($this->publishable_key === null) {
        throw new InvalidConfigException("Your Stripe 'publishable_key' must be set.");
    }
    if ($this->secret_key === null) {
        throw new InvalidConfigException("Your Stripe 'secret_key' must be set.");
    }

    $cart = Utilities::getCart();
    $this->amount = $cart->getTotal();
    if($this->amount < 50) {
        throw new UserException('You have nothing to purchase.');
    }
}
```

With this code setting the amount dynamically, the extension is updated to use the `amount` property (this value was hard-coded in the Chapter 19 version):

```
# pay/controllers/DefaultController.php
public function actionIndex() {
    $module = \Yii::$app->controller->module;

    $model = new Payment;
    if ($model->amount === null) {
        $model->amount = \Yii::$app->controller->module->amount;
    }
}
```

The last changes to the extension include storing the payment ID in the session and then allowing the module to redirect to another page, if `redirectOnSuccess` is set:

```
// Handle the form submission:
if ($model->load(\Yii::$app->request->post()) && $model->save()) {
    // Store values in session:
    \Yii::$app->session['payment_id'] = $model->id;

    // Redirect:
    if (!empty(\Yii::$app->controller->module->redirectOnSuccess)) {
        return $this->redirect(\Yii::$app->controller->module->redirectOnSuccess);
    } else {
        return $this->render('thanks', ['model' => $model]);
    }

    // Present the form:
} else {

    \Stripe\Stripe::setApiKey($module->secret_key);

    $intent = \Stripe\PaymentIntent::create([
        'amount' => $model->amount,
        'currency' => 'usd',
    ]);
    $model->payment_id = $intent->id;

    return $this->render('index', ['model' => $model, 'intent' => $intent, 'publishable_']);
}
```

Now, if the Stripe charge succeeds, the payment details are stored in the database, the payment ID is stored in the session, and the customer is redirected to “order/create”.

As a reminder, the extension itself creates a simple form for taking the customer’s details (**Figure 23.5**). The page also defines the proper JavaScript to send the payment request securely to Stripe.

The form consists of several input fields and a submit button. At the top left is a small icon of a credit card followed by the label "Card number". To its right is a field for "MM / YY CVC". Below these is a label "Email" next to a large, empty rectangular input field. At the bottom left is a rectangular button with the text "Submit Payment" in white.

Figure 23.3: The payment form.

(With this better version of the extension, usage of it only requires configuration of the extension in the main configuration file and, optionally, setting the amount dynamically in the controller.)

After the customer pays, a few things have to happen:

1. The customer record needs to be created in the database.
2. The order itself needs to be recorded in the database.
3. The actual customer needs to be able to download the purchase(s).

Upon successful payment, the customer is sent to “order/create”, so the code for the above steps goes within that method. The first thing the method does is retrieve the payment details:

```
public function actionCreate() {

    // Must have processed a charge before coming here:
    if (!isset(\Yii::$app->session['payment_id'])) {
        throw new CHttpException(400, 'You have not made a purchase.');
    }

    // Get the payment info:
    $cmd = \Yii::$app->db->createCommand('SELECT * FROM payment
WHERE id=:id');
    $payment_id = \Yii::$app->session['payment_id'];
    $cmd->bindParam(':id', $payment_id, PDO::PARAM_INT);
    $payment = $cmd->queryRow();
    if ($payment === null) {
        throw new CHttpException(404, 'You have not made a purchase.');
    }
}
```

Next, the customer record should be recorded in the database. Because the customer may have purchased something in the past, the code first checks for the customer's existence already, using the email address provided during the payment process:

```
// Fetch the customer, if existing:  
$customer = Customer::findOne(['email' => $payment['email']]);// If no customer, create a new one:  
if($customer==null) {  
    $customer = new Customer;  
    $customer->email = $payment['email'];  
    $customer->save();}  
}
```

With the customer record either created or retrieved, the next step is to store the order in the database.

Orders are stored in two tables: `order` and `order_content`. The “create” action of the “Order” controller first creates and save the `Order` instance:

```
$order=new Order;  
$order->customer_id = $customer->id;  
$order->payment_id = $payment['id'];  
$order->total = $payment['amount'];  
$order->date_entered = $payment['date_added'];  
$order->save();
```

Next, the order details—the contents of the order—need to be saved in the `order_content` table. This should happen immediately after the order is saved.

```
$cart = Utilities::getCart();  
$cmd = \Yii::$app->db->createCommand('INSERT INTO order_content  
(order_id, book_id, quantity, price_per)  
SELECT :order_id, cc.book_id, cc.quantity, b.price  
FROM cart_content AS cc, book AS b  
WHERE (b.id=cc.book_id) AND (cc.cart_id=:cart_id)');  
$order_id = $order->id;  
$cart_id = $cart->id;  
$cmd->bindParam(':order_id', $order_id, \PDO::PARAM_INT);  
$cmd->bindParam(':cart_id', $cart_id, \PDO::PARAM_INT);  
$cmd->execute();  
$cart->clear();
```

The code itself should be pretty self-explanatory. The underlying SQL command is:

```
INSERT INTO order_content (order_id, book_id, quantity, price_per)  
SELECT :order_id, cc.book_id, cc.quantity, b.price  
FROM cart_content AS cc, book AS b  
WHERE (b.id=cc.book_id) AND (cc.cart_id=:cart_id)
```

This is an example of a `INSERT INTO...SELECT` query. The values selected are used for the insert. Those values need to be the:

- Order ID
- Book ID
- Quantity
- Price per book

The order ID is available within the method through `$order->id`. That value is selected as a static value (as opposed to selecting it from a table). The other values come from the `cart_content` and `book` tables.

The final step in the method is to clear out the cart's contents (since the items have all been purchased): `$cart->clear()`. The `clear()` method is defined within the `Cart` class:

```
public function clear() {
    $cmd = \Yii::$app->db->createCommand('DELETE FROM cart_content
        WHERE cart_id=:cart_id');
    $cart_id = $this->id;
    $cmd->bindParam(':cart_id', $cart_id, \PDO::PARAM_INT);
    $cmd->execute();
    $cmd = \Yii::$app->db->createCommand('DELETE FROM cart WHERE id=:cart_id');
    $cmd->bindParam(':cart_id', $cart_id, \PDO::PARAM_INT);
    $cmd->execute();
}
```

The method has to clear out both the `cart` and `cart_content` tables.

{TIP} If you want to create a “clear cart” functionality for the customer, the button or text just needs to link to an action that calls this `clear()` method of the `Cart` class.

Returning to the “Order” controller, the last step is to render the view file that shows the order:

```
return $this->render('view', [
    'model' => $order,
]);
```

The view file starts with a `DetailView` widget to display the order details (**Figure 23.6**).

Order Confirmation #19

Order Number	19
Email	another@test@example.com
Total	USD 364.95
Order Date	November 18, 2024

Figure 23.4: The start of the view order page.

```
<?= DetailView::widget([
    'model' => $model,
    'attributes' => [
        [
            [
                'label' => 'Order Number',
                'value' => $model->id,
            ],
            'customer.email',
            [
                [
                    'label' => 'Total',
                    'value' => Utilities::formatAmount($model->total),
                ],
                [
                    [
                        'label' => 'Order Date',
                        'value' => Utilities::formatDate($model->date_entered),
                    ],
                ],
            ],
        ],
    ])>
```

Next, a link to download each book is shown (**Figure 23.7**).

Download Purchases

- [The Yii Book](#)
- [Effortless E-commerce with PHP and MySQL \(2nd Edition\)](#)

:

```
<h2>Download Purchases</h2>
<ul>
<?php $items = $model->orderContents;
foreach ($items as $item) {
    echo '<li>' . HTML::a($item->book->title, ['/book/download',
        'id' => $item->book_id]) . '</li>';
}
?>
</ul>
```

The link goes to “book/download”, passing along the book ID.

The “download” action of the “book” controller sends the PDF files to the user through the browser. It requires a book ID. The trick to sending the PDF is that the file itself is stored in another directory, presumably not within the web root. To send the PDF to the browser, the right header lines have to be sent. Let’s walk through this method a few lines at a time:

```
public function actionDownload($id) {
    $model = $this->findModel($id);
```

First the method loads the book using the provided ID. If no such book exists, the `findModel()` method would throw an exception.

A complete example would have code here that checks that the current user has the right to download the given book. Likely this would be accomplished by storing the customer ID in the session (after completing a purchase or logging in), and then checking the `order_content` table for the combination of that customer ID and this book ID. If no match was found, an exception would be thrown.

Next, information about the book file has to be retrieved:

```
$file = Yii::getAlias('@app/assets/pdfs/' . $id . '.pdf');
$title = $model->filename;
```

The book uses the name “X.pdf”, where X is the book’s ID (similar to how the book’s images are named). The PDFs are stored in the `assets/pdfs` directory. The file’s title is pulled from the database. This allows the code to change the downloaded file to use, for example, “yiobook.pdf”, as the file’s name, instead of “1.pdf”.

Next, a single method, added in Yii 2, makes sending the file easy:

```
return \Yii::$app->response->sendFile($file, $title);
```

That code sends the necessary headers and then transmits the file itself.

This is the heart of a mostly complete e-commerce site, with shopping cart functionality, actual payments, customer creation, and delivery of goods. Still, it's not 100% complete (nor was it intended to be). Just a few things you'd likely implement are:

- Removal of all unused controllers and actions
- Fixing of permissions on all controller actions
- Ability to update the quantity of items in the cart
- Complete customer account management (registration after purchase, ability to change a password, forgotten password tool, and so on)
- Deny access to books unless the user has purchased them
- Handle all exceptions nicely
- A custom layout and design

The SQL commands also create definitions for two more tables: `download` and `password_token`. The former can be used to track the downloads of books, which may be a useful metric. The latter would be used as part of a “forgot password” logic.

From a programming perspective, there are different steps you'll need to take to create and run a site that sells physical goods compared with one that's only digital.

First, every e-commerce site needs unique identifiers—call them “product IDs” or “SKUs”—for each item sold. Unique identifiers is the only way you can track shopping carts, manage inventory, and deliver goods.

Second, with physical goods, you need to track inventory:

- Increase inventory when new stock is received
- Decrease inventory when items are sold
- Display inventory unavailability (that is, “out of stock”) on the site

This is not hard to do, but you'll need to establish some policies. For example, if putting something in a cart decreases the inventory on hand, you'll never have the bad experience of a user suddenly not being able to buy something because the last one was sold while it was in their cart. On the other hand, you'll lose some sales, as the reduced inventory would reflect what people *think* they're going to buy, not what has actually been purchased. How you handle this differs from one business model to another.

Similarly, just being out of stock of something may not rule out it being purchased. It could be an item that gets replenished on a frequent and reliable basis.

With physical goods, you also have to handle shipping and delivery. This means knowing the:

- Origination and destination addresses
- Package's size and weight

- Shipping costs by service
- Approximate shipping time

For the package's size and weight, you'd want to store this in the database, along with the other product details. That way, when a user buys 2 of Widget A and 1 of Widget D, you can know how much it weighs in total.

For calculating the costs, the clear best approach would be to use a third-party service, such as a national or international carrier. They all have APIs through which you send the requisite information, and the API returns the cost options. There's no better way to handle this.

Finally, a site that sells physical goods needs people to actually do all of the above: receive inventory, package up outgoing parcels, order more inventory, etc. This would likely not be your problem (if you're just the developer), but it does mean you'll need to create administrative interfaces to help in this area:

- Show orders to be shipped
- Create shipping labels (perhaps)
- Show which items need to be re-ordered (where there's problematically low inventory)

Obviously there are many differences between selling virtual products and physical goods. But even within each broad category, a multitude of differences exist: virtual products that require licenses (such as software) need extra code; physical goods that are perishable have to be handled more cautiously; for other products, different local, state, and national laws may apply. But hopefully through this chapter, you have a good taste as to some of the components of implementing e-commerce in Yii.

Chapter 24

Shipping Your Project

You've done it: you've wrapped up your Yii project! You've tested it, gone through reviews and edits with your client (or by yourself), and polished it thoroughly. The project is done. Well, the *development* version is done. Now you have to take the site live. How do you do that?

In this chapter, I'll walk through many of the steps you should take in order to ship your project. Admittedly, some of these ideas could be implemented from the onset. If so, that's something you can apply to your next Yii project!

{NEW} Since Yii 2 uses a single public directory—**web**, taking a site live requires slightly fewer precautions than in Yii 1.

A common misstep when taking a site live is not having the proper permissions set on the various directories. This can lead to problems with your logs and assets, as Yii needs to write data to those directories (**runtime**, in the case of log files).

In Unix permissions terms, you should establish 0755 (directory read + write) permissions on:

- **runtime**
- **web/assets**

You may also find you'll want to restrict the permissions on directories that were previously more open (all in the application base directory):

- **controllers**
- **data**
- **models**
- **views**

Those should all be fine with 0644 permissions.

You may be tempted to copy over the *contents* of your **web/assets** folder when you take your site live. Do not do this! The Yii **AssetBundle** class takes care of your assets for you. Just make sure the **web/assets** folder exists on your live server and that it has writable permissions by the web server. That's all you need to do; Yii takes care of the rest.

If you'd rather store your assets in a different directory, you just need to configure the assets manager. This is done in the **assets/AppAsset** class definition. Set a new base path and base URL:

```
class AppAsset extends AssetBundle {  
    public $basePath = '@webroot/_';  
    public $baseUrl = '@web/_';
```

That code tells Yii to publish and serve the assets from the **webroot/_/assets** directory instead of the default **webroot/assets**.

Note that even when making a change like this you still want to have assets be served from a public, writable directory within the URL structure.

The default bootstrap file (**web/index.php**), created by the `yii` command, is coded for easier development. Specifically, it's written such that debugging is enabled. That's not something you'll want on a live site, though, so you'll need to delete this line:

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

Removing that line disables the debugging module thanks to code in **config/web.php**:

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

This change also has the net effect of disabling the bootstrap module. You'll want to do that for both performance and security reasons.

Next, change the environment value from **dev** to **prod**:

```
defined('YII_ENV') or define('YII_ENV', 'prod');
```

(Technically you could just remove that line as `prod` is the default, but sometimes it's best to be explicit instead of relying upon implicit behavior.)

If you're lucky, or just well organized, you have a development installation of your site and a production version. If so, then you'll never end up making code changes or (again, hopefully) debugging a live site. But if, for whatever reason, you do have that need, there's a trick to enable debugging for only yourself.

The bootstrap file originally has this line:

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

You could replace it with something like this:

```
if (isset($_GET['debug'])) define('YII_DEBUG', true);
```

If you do that, then to debug any page on the fly, change the URL from:

example.com/index.php/site/contact

To:

example.com/index.php/site/contact/debug/true.

If you do go this route, do yourself a favor and make the term less obvious than "debug", and remove that line (or comment it out) once you're done.

The only downside to this approach is that it won't allow you to test the submission of forms (because `$_GET['debug']` won't be passed upon submission). If that's a need, check for the presence of a cookie to enable debugging, and then set that cookie for yourself.

By default, Yii creates multiple configuration files for an application:

- **web.php**, the primary one
- **console.php**, for use with console applications
- **test.php**, used for tests
- **params.php**, for other parameters, such as email addresses
- **db.php**, the live database connection
- **test_db.php**, the test database connection

You may find this arrangement to be sufficient. However, in more complex applications, other structures might work better. Further, if you tend to use common configurations among all your Yii projects, having a new default configuration file, to be amended and appended by the main configuration file, may make more sense. You might also like multiple configuration files for different environments:

- Development

- Staging
- Production (aka live)

There are a couple of ways you can make use of multiple configuration files in Yii.

For the environment situation, a common solution is to watch for a server variable in the bootstrap file and include the appropriate configuration file based upon that variable's value:

```
switch ($_SERVER['HTTP_HOST']) {
    case 'localhost':
        $config = 'localhost';
        break;
    case 'dev.example.com':
        $config = 'development';
        break;
    default:
        $config = 'production';
        break;
}
$config = require __DIR__ . '/../config/' . $config . '.php';
```

{TIP} Watching for a server variable can also be used to dynamically set the **YII_ENV** value.

The above approach dynamically switches the primary configuration file based upon context. You can also break the configuration files into additional ones (which can be used with or without the above technique). The default behavior already does this by including the parameters and database configuration files:

```
<?php
$params = require __DIR__ . '/params.php';
$db = require __DIR__ . '/db.php';
$config = [
    'components' => [
        'db' => $db,
        ...
    ],
    'params' => $params,
];
```

You can use this same approach to separate out anything that you're starting to find is cluttering up your configuration:

- URL routing rules

- Application parameters (i.e., not specific to components)
- Module inclusions and configurations
- Logging settings

Simply use the same syntax in the primary configuration file and have the offloaded content be returned as an array in the new file (see **params.php** for an example).

In unfortunate and rare situations you may need to put your production site into “maintenance mode”. A well-managed and hardened site should handle changes and fixes on the fly, but things don’t always go as planned.

In maintenance mode the site disables all functionality and shows a simple page to every visitor indicating the outage, plan, etc. This can be handled easily in Yii by configuring the application’s **catchAll** property.

```
# config/web.php
<?php
$config = [
    'catchAll' => [ 'site/maintenance'],
    ...
]
```

Assign to this property the controller action (i.e., class and method) to catch all requests.

Then define the functionality in the controller:

```
# controllers/SiteController.php
public function actionMaintenance() {
    return $this->render('maintenance');
}
```

Finally, create the proper view file.

When the site can be taken out of maintenance mode, just remove **catchAll** from the configuration. You can leave the controller and view modifications for future use.

During development of a project, you normally want to enable logging and the debug panel, so that Yii immediately displays the details of any problems to you. On a live site, however, you can’t do that (for many obvious reasons). But using logging is still important on live sites; you just need to be smarter in how you approach it.

The most important thing to keep in mind is that logging hurts the performance of your site, but when things go wrong—bugs or security breaches, having good logs makes your life vastly easier. Towards that end, you need to make good decisions as to:

- What events you log

- What data you log when those events occur
- Where that log data goes

For the first question, you can choose to log or not log:

- All access (as your web server already does)
- Changes to user accounts
- Warnings
- Errors

For the second question, you need to specifically decide if you want to log some or all existing variables (and their values). In other words: how much context is useful retroactively?

For the third question, you can choose to have the log item:

- Emailed
- Displayed in the browser (which you don't want to do on a production site)
- Written to a file
- Stored in a database

There's no one right answer, nor one right solution. You could choose to write some logs to files, but be emailed for serious occurrences.

Log messages are generated by invoking a top-level method:

- `Yii::debug()`
- `Yii::info()`
- `Yii::warning()`
- `Yii::error()`

Each method takes a message as its first argument and a “category” as its second. I'll return to “category” shortly.

For example, to write an “info” log, you'd do this:

```
Yii::info('Somebody did something.');
```

So what happens with that log? Well, it depends upon your configuration!

That line of code adds the log item to memory. Yii then outputs it to the proper target. Targets are defined by configuration the “log” component:

```
# config/web.php
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'targets' => [
                [
                    'class' => 'yii\log\DbTarget',
                    'levels' => ['info', 'warning'],
                ],
                [
                    'class' => 'yii\log\EmailTarget',
                    'levels' => ['error'],
                    'message' => [
                        'from' => ['log@example.com'],
                        'to' => ['admin@example.com'],
                        'subject' => 'Error at example.com',
                    ],
                ],
            ],
        ],
    ],
];
```

Note that you must first add “log” to the bootstrap configuration so the Yii application loads it. After that, define one or more targets. Each target is a combination of log type and target type. The above sends all info and warning log messages to the database and all errors to an email address.

Each target is identified by a class:

- `yii\log\DbTarget`
- `yii\log>EmailTarget`
- `yii\log\FileTarget`
- `yii\log\SyslogTarget`

Each route is further customized by assigning properties to the class’s attributes. Each class extends `yii\log\Target`, which defines the `levels` property, among others.

The log routing classes also have `categories` and `except` properties that can be used to create more nuanced logging. For example, say that you want to handle problems with the database different from any other problem. The following code logs most errors to files, but sends emails for the database category:

```
'log' => [
    'targets' => [
        [
            'class' => 'yii\log\FileTarget',
            'levels' => ['error', 'warning'],
            'except' => 'yii\db\*',
        ],
        [
            'class' => 'yii\log\EmailTarget',
            'levels' => ['error'],
            'categories' => ['yii\db\*'],
            'message' => [
                'from' => ['log@example.com'],
                'to' => ['admin@example.com'],
                'subject' => 'Database error at example.com',
            ],
        ],
    ],
],
```

The categories are strings in the format `xxx\yyy\zzz`, like namespaces. They are treated hierarchically, allowing you to have a logging approach for `app\controller*` but exempt `app\controller\SiteController`, if you so choose.

For category configurations to work, the category must be identified at the time of log creation. This is easily done using PHP's `__METHOD__` constant:

```
Yii::info('Somebody did something.', __METHOD__);
```

For example, if that code is within the `actionAbout()` method of the “Site” controller, then `__METHOD__` is `app\controllers\SiteController::actionAbout`.

The last thing to know about logging is how to add contextual information such as the variables that exist at the time of the logging. Contextual data is added via the `logVars` attribute of the target class. To simply enable it, provide it in the configuration:

```
'targets' => [
    [
        'class' => 'yii\log\FileTarget',
        'levels' => ['error', 'warning'],
        'except' => 'yii\db\*',
        'logVars' => ['_REQUEST'],
    ],
]
```

Chapter 17, “Improving Performance,” covers material that’s even more critical in production sites. Review that chapter before deploying, ensuring you’re following best practices. This applies particularly to caching.

Beyond the caching mechanisms Yii provides, in production mode you should consider enabling a bytecode cache, such as PHP OPcache or APC, to minimize the time needed for including and parsing PHP files.

To complete this chapter (and the book!), here’s a quick checklist of things you’ll want to do as part of shipping a project:

- Make sure all your tests pass, if you’ve created them
- Disable debugging
- Disable Gii and any other modules or extensions that are no longer needed
- Disable any back-doors or potential security holes you may have created for “convenience” while developing
- Check the permissions on the directories
- Properly configure logging
- Minify your CSS and JavaScript
- Use CDN-hosted versions of libraries, if possible
- Hammer your site with problematic input to confirm nothing terrible happens or is shown
- Cache, cache, cache
- Create a back up plan!
- Start making a list of things to change, add, and fix in the next revision
- Celebrate!