Community Experience Distilled

Web Application Development with Yii 2 and PHP

Fast-track your web application development using the new generation Yii PHP framework

Mark Safronov Jeffrey Winesett

[PACKT] open source*
PUBLISHING
community experience distilled

Web Application Development with Yii 2 and PHP

Fast-track your web application development using the new generation Yii PHP framework

Mark Safronov

Jeffrey Winesett

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Web Application Development with Yii 2 and PHP

First published: August 2010

Second edition: November 2012

Third edition: September 2014

Production reference: 1190914

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-188-5

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Authors
Mark Safronov
Jeffrey Winesett

Reviewers
Ajay Balachandran
Maher Elaissi
Md. Rashidul Hasan Masum
Mohammed Hussein Othman

Commissioning Editor
Usha Iyer

Acquisition Editor
Harsha Bharwani

Content Development Editor
Madhuja Chaudhari

Technical Editors
Veronica Fernandes
Pramod Kumavat

Copy Editors
Roshni Banerjee
Sarang Chari
Gladson Monteiro
Adithi Shetty

Project Coordinators
Venitha Cutinho
Akash Poojary

Proofreaders
Simran Bhogal
Stephen Copestake
Maria Gould
Ameesha Green
Paul Hindle
Joel T. Johnson
Jonathan Todd

Indexers
Mariammal Chettiyar
Monica Ajmera Mehta
Tejal Soni

Graphics
Disha Haria

Production Coordinators
Kyle Albuquerque
Melwyn D'sa
Saiprasad Kadam

Cover Work
Melwyn D'sa

About the Authors

Mark Safronov is a professional web application developer from the Russian Federation, with experience and interest in a wide range of programming languages and technologies. He has built and participated in building different types of web applications, from pure computational ones to full-blown e-commerce sites. He is also a proponent of following the current best practices of test-first development and clean and maintainable code.

He is currently employed at Clevertech and is working on Yii-based PHP web applications. He was also a maintainer of the popular YiiBooster open source extension for some time.

Back in 2008, he translated the book *Visual Prolog 7.1 for Tyros*, *Eduardo Costa*, in Russian with a totally new color layout. In 2013, along with *Jacob Mumm*, he co-authored the book *Instant Yii Application Development Starter*, *Packt Publishing*.

Jeffrey Winesett is a partner at SeeSaw Labs in Austin, Texas, and has over 10 years of experience building large-scale, web-based applications. He is a strong proponent of using open source development frameworks when developing applications, and a champion of the Yii framework in particular since its initial alpha release. He frequently presents on, writes about, and develops with Yii as often as possible.

I would like to thank Qiang Xue for creating this amazing framework, and the entire Yii framework development team who continually improve and maintain it. I thank all of the technical reviewers, editors, and staff at Packt Publishing for their fantastic contributions, suggestions, and improvements. I would also like to thank my family, who have provided encouragement and unconditional support, and to my many colleagues over the years for providing invaluable opportunities for me to explore new technologies, expand my knowledge, and shape my career.

About the Reviewers

Ajay Balachandran is a hardcore PHP developer and an avid Yii lover from India. He is a huge advocate of writing modular, reusable, and standards-based code, which leads to his love for the Yii framework.

He is an expert in federated authentication using OpenID Connect, and now specializes in providing single sign-on and analytics solutions for the enterprise customers on behalf of his company, HiFX IT & Media Services.

Having a UI/UX background, Yii and its robust Web 2.0 oriented development has enabled Ajay to easily write applications ranging from simple shopping carts to robust APIs.

Maher Elaissi is a web developer based in Canada. He has good knowledge of object-oriented analysis and designs and specializes in PHP programming. His first experience with the Yii framework was in 2012, with a startup company Cisha GmbH based in Germany, to create an online chess game (`www.chess24.com`).

I would like to thank the Super Mario team (dev team) for all their support and help in producing this book.

Md. Rashidul Hasan Masum is a professional Software Engineer. Over the last 6 years, he has designed and developed a wide range of desktop and web applications using the enterprise framework Spring.NET NHibernate and websites using HTML, DHTML, JavaScript, jQuery, SignalR, Ext JS 4, ASP.NET (C#), PHP (Yii framework), Spring.NET, NHibernate, Google App Engine (Java), OpenLayer, Android with MSSQL, MySQL, and Bigtable, including sites for startup companies and small businesses. His core competency lies in complete end-to-end management of a new application development.

He also has experience in the following areas: OOP, AOP, DI, ORM, SOA, n-Tire, highly configurable applications, neural networks, and software design and testing.

He now works at OnnoRokom Software Ltd. as a Software Architect. From the beginning, they have been using the Yii framework for their large-scale web application development. S. M. Quamruzzaman Rahmani (`www.byronbd.com`), Project Manager, and GM Nazmul Hossain, (`www.gmnazmul.com`), System Analyst, have been working with him. The three of them are a super combination for teamwork according to their personality profiles.

I'd like to thank Venitha Cutinho and Akash Poojary for their coordination. Also, I'd like to thank my friend Maruf Maniruzzaman who works at Microsoft. He has taught me a lot about computer engineering. Thank you to my friend K. M. Masum Habib. I'd also like to thank Packt Publishing. I have read lots of e-books published by Packt Publishing.

Mohammed Hussein Othman is a Software Engineer who has graduated from Damascus University in Syria. He has 4 years of experience in working with the Yii framework in a variety of small and enterprise projects. Mohammed has also been working on various modern web technologies, such as PHP, ASP.NET, Ruby on Rails, Node.js, and many others. Currently, he works as a Senior Web Developer and Project Manager at Flex Solutions, which specializes in enterprise web applications.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

PACKTLIB™

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

This book is a guide that describes the process of incremental, test-first development of a web application using Yii framework Version 2.

The Yii framework, hosted at `http://www.yiiframework.com/`, is a PHP-based application framework built around the Model-View-Controller composite pattern. It is suitable for building both web and console applications, but its feature set makes it most useful for web applications. It has several code generation facilities, including the full create-read-update-delete (CRUD) interface maker. It relies heavily on the conventions expressed in its default configuration settings.

Overall, if all you need is a fancy interface for the underlying database, there is probably nothing better for you than the Yii framework. Given the extensive configuration options, any kind of application can be made from Yii.

Version 2 of the Yii framework is built utilizing the latest improvements in the PHP infrastructure collected over the years. It uses the Composer utility (see `https://getcomposer.org/`) as a primary distribution method, PSR levels 1, 2, and 4 from the PHP Framework Interop Group guidelines (see `http://www.php-fig.org/`), and PHP 5.4+ features, such as short array syntax and closures.

At the time of writing, Yii 2 is at the stage of late beta. Some changes are expected, but there should not be backward-compatibility-breaking changes anymore. Thus, the content of this book can be reasonably trusted even if the framework can attain some additional features after this book is published.

What this book covers

Chapter 1, *Getting Started*, covers the simplest possible methods to raise a working web application completely from scratch using the Yii framework.

Chapter 2, *Making a Custom Application with Yii 2*, shows how the process of implementation of a web application with a single, working, tested feature can be done from scratch using the Yii framework.

Chapter 3, *Automatically Generating the CRUD Code*, shows how we can implement a working, tested feature in an existing web application using only the code generation facilities and not a line of custom code written.

Chapter 4, *The Renderer*, describes the details of how the framework renders its output and presents some tricks to introduce customizations to the rendering process.

Chapter 5, *User Authentication*, discusses the tools to provide authentication to application visitors.

Chapter 6, *User Authorization and Access Control*, explains the ways to control access for application visitors, and, especially, about the role-based access control system.

Chapter 7, *Modules*, returns from the exact features of the framework to its fundamentals. Here we will clearly understand the internal structure and logic of the Yii-based application and how it influences the overall design.

Chapter 8, *Overall Behavior*, is about the infrastructure of the Yii-based application. We will learn about several features that affect the application as a whole.

Chapter 9, *Making an Extension,* tells us how to make the extension to the Yii 2 framework and prepare it so that it is installable in the same way as the extensions built-in to the basic distribution of the framework itself.

Chapter 10, *Events and Behaviors*, investigates the intricacies of the system inside the Yii 2 framework allowing us to attach custom behavior to many of the usual activities of the application, such as fetching a record from the database or rendering a view file.

Chapter 11, *The Grid*, has two purposes. First, it explains the powerful and complex GridView widget, which allows you to make complicated, table-based interfaces relatively easily. Second, it presents a *different approach* in developing applications using the Yii 2 framework, the one that is customary in its community, so you can see both the advantages and the disadvantages of both approaches.

Chapter 12, *Route Management*, explains the top level of the framework, that is, how it responds to HTTP requests from actual visitors.

Chapter 13, *Collaborative Work*, concludes the book by presenting the methods that help to manage the code base of a Yii-based application when there are several developers working on it.

Appendix A, *Deployment Setup with Vagrant*, shows a simple way to construct a virtual machine for your local development, which you can use for building the examples from this book.

Appendix B, *The Active Form Primer*, contains an extension to *Chapter 11*, *The Grid*, in which we use another powerful user interface widget of Yii 2, the ActiveForm. It was excluded from the chapter text because it's not directly related to the GridView widget, but we could not gloss over it completely. Without the ActiveForm, the feature we were building in *Chapter 11*, *The Grid*, is not complete.

Through the course of the book, starting from *Chapter 2*, *Making a Custom Application with Yii 2*, we'll be working with a single code base. Later chapters will build over the work previously done, so the book is expected to be read sequentially, without skipping or changing order.

Who this book is for

The text is targeted at existing, established developers who want to learn quickly whether the Yii framework can fit their demands and, especially, workflow. It is not a reference, but rather a guide. More than that, the reader is expected to have a copy of the source code and an official documentation as supplementary material while reading.

We will expect some relatively high qualifications from the reader, as several basic development concepts such as POSIX-compatible command line, version control system, deployment pipeline, automated testing harness, and an ability to navigate through the code base by fully qualified names of classes are assumed as obvious and not requiring any explanation.

What you need for this book

A workstation with the full LAMP stack installed, that is, having Apache web server, MySQL relational database management system, and PHP runtime installed over some Linux-based distribution. If the reader is capable enough, then any of these requirements can be swapped for different vendors, except PHP, which is quite obvious.

You have to use PHP Version 5.4 and higher, because it's a requirement for Yii 2, and generally, you don't have any reason to use previous versions anymore.

Even if you don't use a POSIX-compatible OS, such as any Linux-based distribution or Mac OS X, you should have a Bash-like shell, as all command-line examples in this book assume the availability of this shell.

An Internet connection is required to download many necessary libraries used throughout this book. Even if you don't update anything, you'll download approximately 320 MB of libraries, so mobile Internet probably will not cut it.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Now run the following command to create a subdirectory called `basic`, and fill it with the basic application template."

A block of code is set as follows:

```
require_once(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
new yii\web\Application(
    require(__DIR__ . '/../../config/web.php')
);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
require_once(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
new yii\web\Application(
    require(__DIR__ . '/../../config/web.php')
);
```

Any command-line input or output is written as follows:

```
$ php composer.phar require --prefer-dist yiisoft/yii2-debug "*"
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "You should fill the available fields as shown in the following table, and hit the **Preview** button."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started

Let's see how we can set up a website with Yii 2 from scratch and with a minimum amount of effort. The goal is to learn about the installation procedure of the Yii application boilerplates offered by developers and the starting set of features provided in them.

A basic application

The most basic and straightforward way to get started with Yii 2 is to use the application boilerplate published by the Yii 2 team on their GitHub repository (`https://github.com/yiisoft/yii2`) and available through the Composer tool. In the prior versions of Yii, you had to manually download and extract the archive with the framework contents. While you can still do the same in Yii 2, this version is carefully crafted so that it's especially simple to install it using the Composer tool.

Installation of a basic application template

Find a suitable directory in your hard drive and fetch the Composer **PHP archive** (**PHAR**) into it in any way suitable for you, for example, using the following command:

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

Now run the following command to create a subdirectory called `basic`, and fill it with the basic application template:

```
$ php composer.phar create-project --prefer-dist --stability=dev \
yiisoft/yii2-app-basic basic
```

> Please note that Yii 2 specifies several system-wide dependencies for itself. You probably will need to consult the `composer.json` file inside their GitHub repository to learn about them beforehand (`https://github.com/yiisoft/yii2`). But in any case, Composer will tell you what you need to install to make Yii 2 workable. Yii 2 gets new updates quite often and its requirements are a moving target.

It's a line split into two lines for readability, with a slash denoting the overflow of the command line to the next line of text. Shell interpreters on Unix-like systems should understand this convention, so you can probably just copy and paste the code verbatim and it'll be executed correctly. You better check the documentation of Composer for the meaning of the beginning of the preceding command, but the part relevant to us is `yiisoft/yii2-app-basic basic`, which means "copy contents of the repo published at `https://github.com/yiisoft/yii2-app-basic` to our local folder named `basic`." The command will install the project skeleton in the form of a set of predefined folders, and among them the `vendor` subdirectory, which contains quite a lot of other Composer packages. Inside the `basic` folder will be your application root.

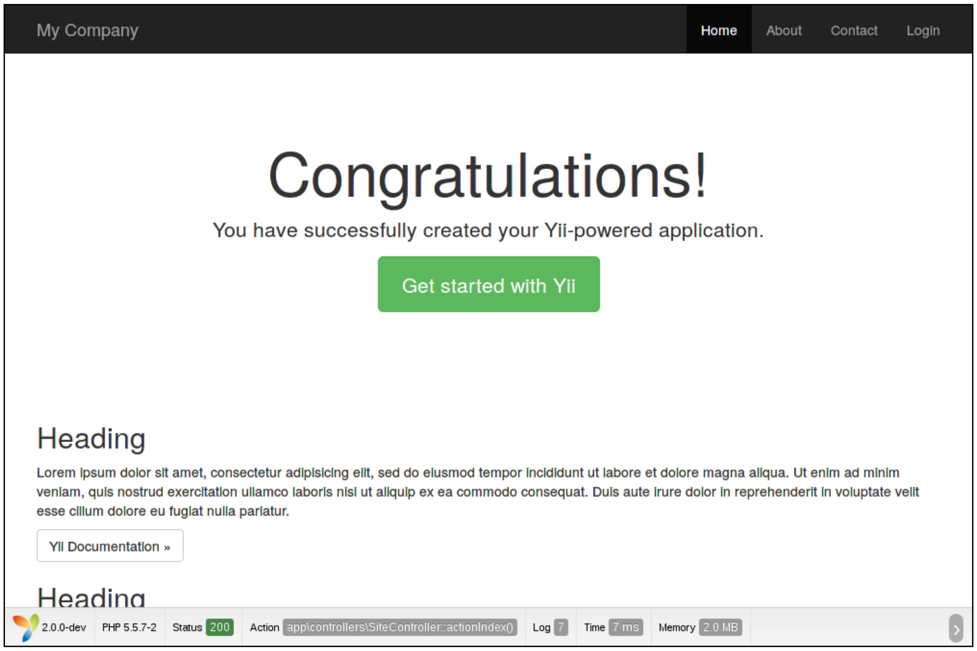
After Composer finishes installing the required packages, you can just issue the following command:

```
$ php -S localhost:8000 –t basic/web
```

Here, `8000` is the port number, which you can change to anything you want. This will launch the web server built into PHP.

> This is not the preferred setup for PHP-based web applications, of course. The built-in web server was used only as a "smoke test" to verify that everything in general works. It is suitable only for local development without a heavy load. The next chapter will deal with the real-world deployment of a Yii-based web application.

Point your web browser at the `http://localhost:8000/` URL. You should see the welcome page for the Yii 2 application, which means that you're done setting things up.



Specifics of the basic application template

You can get a comprehensive overview of the various folders inside the basic template by reading the README file provided with the template (`https://github.com/yiisoft/yii2/blob/master/apps/basic/README.md`), or by reading the global Yii 2 documentation page describing basic applications (`http://www.yiiframework.com/doc-2.0/guide-start-installation.html`).

The most important thing you should understand is that the publicly available web root directory is just one folder in the overall code base. It's the `web` directory for our basic application. Every other folder is outside the web root directory, that is, out of reach for the web server.

As you have already seen in the installation description, given that you have PHP and, optionally, curl, this code base is ready to use right from the start; no specific environment is needed to be set up. All the dependencies are managed through the Composer tool.

A three-tier automatic testing harness is already set up in the basic template. It contains acceptance, functional, and unit tests covering most of the functionalities. The tests already included in the template are useful as examples that show how to utilize the testing framework used, which is Codeception (`http://codeception.com/`).

The template can be really useful to you if all you need is something like a news feed feature or a web tool spanning a couple of pages. However, the absence of separation by subsystems, such as the administrative backend and public frontend, will start to hinder you on a larger web application; probably, an application with more than just 10 unique routes.

Note that by reading the project dependencies in the `composer.json` file, Yii 2 has several important parts separated out as pluggable packages, and they are already included in your code base by Composer. These packages are listed as follows:

- Gii, the code generator, which we will discuss in detail in *Chapter 3, Automatically Generating the CRUD Code*
- The debug console that is already enabled on the basic application template
- A wrapper around the Codeception testing framework
- A wrapper around the SwiftMailer library (`http://swiftmailer.org/`), which can be found at `https://github.com/yiisoft/yii2-swiftmailer`
- The Twitter Bootstrap UI library packaged as a Yii 2 asset bundle (it's practically ubiquitous nowadays, but here's the link anyway: `http://getbootstrap.com/`)

The first three are set up so that you get them only when you are developing the application, since they are useless and even harmful in the production environment. Most probably, you'll need all of these on any serious project though.

A short overview of the basic application installation:

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar create-project --prefer-dist \
--stability=dev yiisoft/yii2-app-basic basic
$ php -S localhost:8000 -t basic/web
```

An advanced application

Apart from the basic application template, Yii 2 has an advanced application template. It's geared more towards medium-sized applications (such as applications that are really useful to businesses), and its main feature is two separate web interfaces: one dedicated to content management and the other to presenting this content to visitors. So, you get an almost complete CMS skeleton with this template.

Installation of an advanced application template

The first steps are the same as for a basic template. You need to fetch the Composer executable and set up a new project using it:

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar create-project --prefer-dist --stability=dev \
yiisoft/yii2-app-advanced advanced
```

You can see that the difference is just that instead of the word "basic", we use the word "advanced".

Now, let's make further changes. First, go to the newly created directory named `advanced`. After this, you need to generate the required local configuration by running the following command:

```
$ ./init
```

Yes, it's just the `init` script from the root of the code base. It'll ask you whether you want a development mode or a production mode and create all the necessary auxiliary configuration snippets and entry scripts. To be precise, it just copies the contents of the `dev` or `prod` folders from the `environments` subdirectory depending on whether you selected the development environment or the production one. Just open the `environments` subdirectory and you'll understand how it works.

Next, you need to create the database to be used by this application. By default, for configuration of a development environment, you have to set up a MySQL database named `yii2advanced` accessible from the `localhost` at the default MySQL port for user `root` without any password. You can see the details in the `common/config/main-local.php` file.

Given that you have the database set up, you need to run migrations. We'll talk about migration scripts in the next chapter (and we will even write several scripts ourselves), but if the very concept of **database migrations** is foreign to you, you can read about it in the official documentation at the Yii 2 website (at the time of writing this, there is no official documentation, but the framework has the docs included in the GitHub repository at `https://github.com/yiisoft/yii2/blob/master/docs/guide/db-migrations.md`).

Just run the following command anyway:

```
$ ./yii migrate
```

It'll present you with a list of exactly one migration and ask you for confirmation before doing its job.

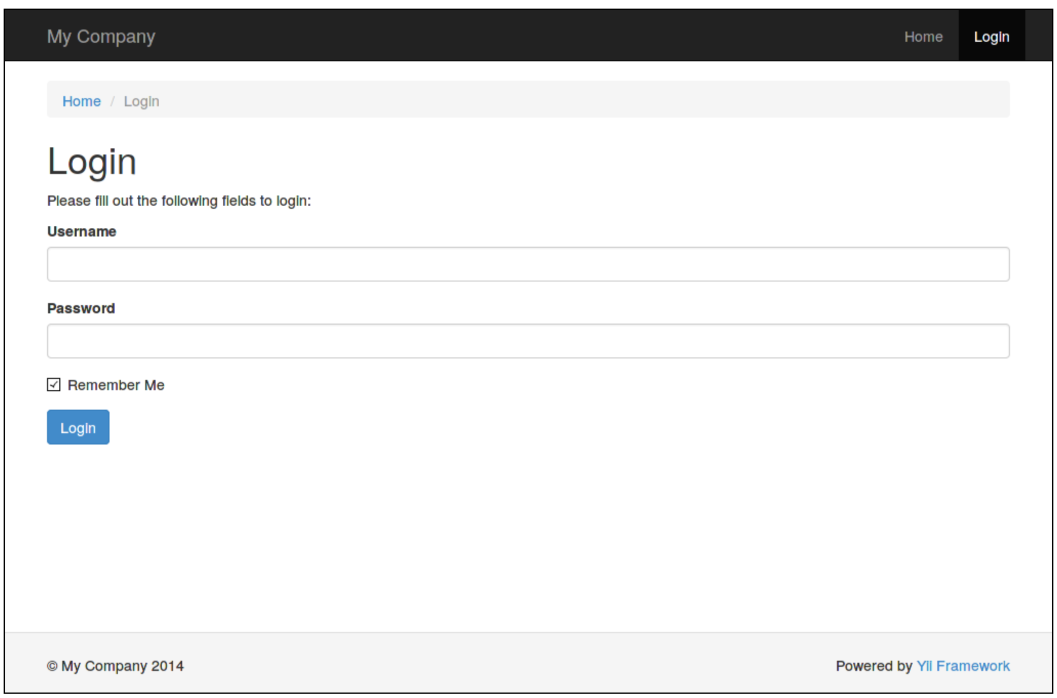
Now, you are ready. Make both the sides of the application accessible for you by executing the following commands:

```
$ php -S localhost:8080 -t frontend/web
$ php -S localhost:8081 -t backend/web
```

As with the basic application template, we are using the built-in PHP web server just because it's a lot simpler to be demonstrated in the book than explain how to set up Apache or some other web server to serve from these folders.

Now you'll have the backend side of the application intended to be used by content managers and the frontend side of the application intended to be the website your visitors will see. Also, note that you have a completely controllable console runner launched by the `yii` script you used when doing migrations.

Advanced application has exactly the same frontend as basic application. Here is how its backend looks like:



The advanced template backend is locked down initially. After logging in, you get the same page as the one from the basic template or the advanced template frontend, but with only the login feature in the menu.

Specifics of the advanced application template

The most important thing about the advanced application template is that it is three *basic* application templates wired together as one:

- Inside the `frontend` folder is the application structure for the public-facing side of your website. Real functionalities and the content of your website or web application is expected to be placed here.
- The folder named `backend` is for your CMS, protected from unauthorized access. You are expected to place all of your admin-accessible CRUD here.
- The `console` folder is mainly for your custom console commands, in hope you'll have any, and a lot more likely, for your migration scripts.
- The `common` folder contains code that will be used by all the entry points, since it's a single application after all.

Of course, nobody forces you to use the frontend and backend sides as described. You just have two web frontends sharing the same code base, so you are free to use them as you wish. However, the UI already prepared for the advanced template that has a password-protected backend right from the start.

You should know about the login feature for the freshly installed advanced application template. Initially, it had no users defined, and you had to create one by utilizing the signup feature at the frontend. After that, you'll be able to login to both backend and frontend using the created credentials. The frontend is identical to the basic application, and the backend is stripped of everything except the login feature and front page, so everything is up to you.

A short overview of the installation of the advanced application:

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar create-project --prefer-dist \
--stability=dev yiisoft/yii2-app-advanced advanced
$ cd advanced
$ ./init
$ mysql -u root -e 'create database yii2advanced'
$./yii migrate
$ php -S localhost:8080 -t frontend/web
$ php -S localhost:8081 -t backend/web
```

Summary

Yii 2 allows you to configure almost all paths used by the framework, and hence you can create any directory tree you wish. By utilizing the PHP 5.3 namespaces wisely, you can even have a physical structure of your project different from the logical one, that is, your files will lie in folders differently from how your classes are structured by namespaces. This will surely be quite tedious to do though.

In the next chapter, we'll look at how we can utilize Yii in a (albeit small) real-world project, built completely from scratch, and without using the templates we saw in this chapter.

2

Making a Custom Application with Yii 2

In this chapter, we'll see how Yii can help us build web applications. The example will be reasonably small, but it will be done using proper software engineering disciplines. We'll go through all the steps of application development, each step backed by the bleeding-edge best practice described in the definitive books on this topic:

- **Building the domain model**: This is explained in *Domain-Driven Design: Tackling Complexity in the Heart of Software*, *Eric Evans*, *Addison-Wesley Professional*
- **Setting up the testing harness**: We'll follow the acceptance test-driven development practice described in *Growing Object-oriented Software, Guided by Tests*, *Steve Freeman and Nat Pryce*, *Addison-Wesley Professional*
- **Setting up the deployment pipeline**: This is explained in the following books:
    - *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, *Jez Humble and David Farley*, *Addison-Wesley Professional*
    - *Continuous Integration: Improving Software Quality and Reducing Risk*, *Paul M. Duvall*, *Steve Matyas*, and *Andrew Glover*, *Addison-Wesley Professional*

- **The Red-Green-Refactor development cycle**: This is explained in depth in the following books:
    - *Clean Code: A Handbook of Agile Software Craftsmanship*, *Robert Martin*, *Prentice Hall*
    - *Test-Driven Development by Example*, *Kent Beck*, *Addison-Wesley Professional*
- **Deployment and manual tests**: This fits into the *Continuous Delivery* paradigm too, but these steps are inevitable anyway

Stay focused.

The design stage

Pay attention to the fact that we will be using this example application through the whole book. In this section we'll define the landscape for the whole adventure before us.

Task at hand

Let's pretend we are a small business providing some services. We have a particular number of clients we have connections with, and the amount is so large that managing it on paper and business cards is too unwieldy. So, we need some sort of automated way of finding the full profile about a given client.

For starters, we need some sort of create-read-update-delete (**CRUD**) user interface for simple records describing the most essential attributes of our clients.

It's obvious that as we as a business will grow and evolve, the same will happen with our client management, and so our application will grow and evolve too. We should account for changes right from the start.

As we will be eating our own dog food, this software better be of the highest possible quality.

Domain model design

Obviously, we will be dealing with customer models in our application. Between the "customer" and "client" terms, we choose "customer" for accuracy.

A customer is a person who has a name, address, e-mail, and phone number at the minimum. We provide services for customers, which are counted in hours, and we are being paid fees for these hours according to the contract. That's what we will include in the first iteration of designing.

Our primary assumption is that each customer is a single person, so we don't deal with companies that can have multiple people as contacts. The name is an incredibly complex construct; if we are to delve into the details of its structure, such as honorifics, titles, nicknames, middle names, patronymics, and so on. But we aren't really interested in the structure of the customer's name, we need it only for identification purposes. So, we will represent it as a line of text, allowing us to write a name in free format. The address is a construct of the same complexity, but this time we have to retain the structure instead of using plain strings again, because we will need to do two things with addresses:

- Calculate some statistics, such as how many customers we have in a particular city
- Properly generate the address lines according to the postal rules in different cultures

So, we decide on the structure as follows:

- Purpose (for example, billing address, shipping address, home address, or work address)
- Country
- State, for countries partitioned into several regions, such as the USA
- City
- Street
- Building
- Apartment/office
- Receiver name
- Postal code

We should note that an address can be for an apartment, postal box, office in an office building, employee in an organization, or for a whole building. Also, a customer can have many addresses.

The phone entity has the following attributes:

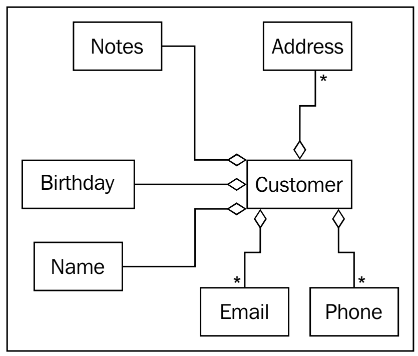
- Purpose (personal or work)
- Number

A customer can have several phone numbers, hence the "purpose" field.

Apart from names, addresses, and phone entities, our staff will surely need a way to assign a free-form textual description of a customer, which we'll simply name notes. Also, taking note of a birthday would be cool too. And e-mail, of course. By the way, a single customer can have several e-mails.

Let's stop here.

We can now figure out the complete aggregate for the customer model, which is depicted in the following diagram:



According to the *Eric Evans* book *Domain-Driven Design*, the customer is an entity, that is, an object whose state will surely change over time, and we care about its identity across the system. Everything else is a value object, that is, an object whose state will not change after initial creation, and thus they are completely interchangeable.

For the sake of simplicity, we will not detail how business is done with the customer, because we will just not be able to cover it all across this book. However, let's mention that we have some sort of services we provide to our customers, and it'll be useful to maintain records of them too. This model will be used in the following chapter.

Target feature

Let's fulfill one specific task. Given that someone called us by phone (assuming we have identified the number), we want to get all the details we have gathered so far about the person who's calling. If we don't have such a number associated with someone registered in our application, then we know he or she is not our customer (yet). If we do have the number registered, then we can at least greet this person by his or her name, which is superb customer service.

We should understand that to make queries to the database, we need a way to at least insert data into it and potentially a way to edit and delete it. Thus, our feature set for the first iteration of development will be as follows:

- To record info about a customer into the database
- To edit info about a customer in the database
- To delete info about a customer from the database
- To query info about a customer by his or her phone number from the database

Building the way to make generic queries to a database is not the goal. We will only deal with the queries containing a phone number.

Let's begin then.

Initial preparations

We will be working on the same application in all of the following chapters until the end of the book, so the preparations are to be done only once.

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Setting up project management

Our example application is essentially a Customer Relation Management one, that is, a CRM application. Thus, we will start with a folder named `crmapp`.

> Please note that all examples of command-line invocations *through the whole book* assume that your current working directory is the `crmapp` folder and not anywhere deeper or higher in the filesystem hierarchy.

Version 2 of Yii has the Composer package manager as the preferred method of installation, so we will use this tool too. While you can read the details in Composer's full documentation, here's a crash course of it:

- All packages installed by Composer are kept inside a special subdirectory under your project's root called `vendor`.
- All dependencies and other information about your data relevant to Composer are kept in the manifest file called `composer.json` under your project's root. As long as you have the dependencies declared there, you can safely purge the `vendor` folder at any time as it'll be repopulated by the next call to `php composer.phar install` or `php composer.phar update`.

The documentation for Composer presents a nice one-line command that will get the Composer executable to you:

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

Of course, if you don't have CURL lying somewhere in your PATH, you can just go to the official site of Composer at `https://getcomposer.org/` and grab the PHAR archive from there.

After that, Composer will be invoked whenever `composer.phar` is called:

```
$ php composer.phar <command>
```

It is assumed that you use some version control system for your code base. The code bundle for this book uses Git (`http://git-scm.com/`).

> **Preparations in essence**
>
> ```
> $ mkdir crmapp
> $ cd crmapp
> $ curl -sS https://getcomposer.org/installer | php
> $ git init
> ```

Setting up the testing harness

As we declared in the opening paragraph of this chapter, we will follow the test-first development practice based on acceptance tests. The reasons behind this decision are as follows:

- We will have a way to check whether the application works as intended without resorting to tedious manual testing
- We do not have a real need for deep unit testing because most of our work will involve wiring together already-existing components with rudimentary logic in between, so end-to-end acceptance tests through the UI is the most simple and viable solution

We need some form of acceptance tests in the system anyway if we care about user feature requests at all.

Yii 2 has support for the **Codeception** testing framework built-in, with `http://codeception.com/` being the official website for it. We will never use it in this book, but the extension named `yii2-codeception` (see `https://github.com/yiisoft/yii2-codeception`) provides a set of helper classes to integrate your tests more deeply with the Yii framework.

Let's declare that we want Codeception available in our project. Run the following command:

```
$ php composer.phar require "codeception/codeception:*"
```

Wait a bit until Composer finishes.

> Here are the contents of your `composer.json` at this point:
>
> ```
> {
>     "require": {
>         "codeception/codeception": "*",
>     }
> }
> ```
>
> The `php composer.phar require <packagename:version>` command is just a helper method to insert lines inside the `require` block of the manifest and call the update routine.

Of course, we'll need to add Yii 2 as a dependency at some point, but for now, let's do one thing at a time.

> Now, we have the Codeception executable available at `./vendor/bin/codecept`. This location is a bit long to type, so a POSIX-compatible shell like bash allows you to reduce it as follows:
>
> ```
> $ alias cept="./vendor/bin/codecept"
> ```
>
> It's better now. In all of the following command-line examples in this chapter, we assume you have done this substitution.

Codeception is a complex system, so we'll need to rely on its self-building system for now. To not delve unnecessarily into the inner workings of Codeception, let's just stick to defaults. Call the following command:

```
$ cept bootstrap
```

This will generate a `tests` directory and configuration tree for Codeception.

Now, let's create the first dummy acceptance test to check the top level of our test harness:

```
$ cept generate:cept acceptance SmokeTest
```

This command will generate `SmokeTestCept.php` in the `tests/acceptance` directory. When you open it, you'll see something like the following, depending on the version of Codeception:

```
$I = new AcceptanceTester($scenario);
$I->wantTo('perform actions and see result');
```

`AcceptanceTester` is the class of objects holding all the methods we can use to test our application imitating a real user behind the browser. Codeception also has `CodeGuy` for unit tests and `TestGuy` for functional tests, but that's for later.

When we say `AcceptanceTester.wantTo("do something")`, we just create a title (enclosed in double quotes) for the test actions following this invocation.

Let's change the dummy test to a simple smoke test that our landing page is up for:

```
$I = new AcceptanceTester($scenario);
$I->wantTo('See that landing page is up');
$I->amOnPage('/');
$I->see('Our CRM');
```

So, we expect to see the line **Our CRM** when we access the landing page of our future application. Let's pretend that we'll have such a title somewhere in there.

Now we run the test:

```
$ cept run
```

We see it fail because we don't have the web server serving anything on the `/` request. Thus, we arrive at the point where we need to write production code to satisfy our tests. However, right now, it's not the production code we need, but the infrastructure to serve it. We need a machine to deploy to.

Setting up the deployment pipeline

The problem is described here. The web acceptance tests that we will be writing imitate a real user who opens the web application in the browser and interacts with it using the visible UI. So, for them to work, we need an application that is fully deployed to somewhere accessible from the machine on which we will run acceptance tests.

> In most cases, you'll decide to just run the application on the same machine on which you'll do the source code editing. Wrong! Don't do it.

Most probably your own workstation is not the same as the machine your web application will ultimately run on. This has been an ongoing problem in the industry for decades now, and you can be sure that when your application's lifetime is measured in years, you will get the same integration problems if you test your application on the machine with a different environment than the production server. Of course, this doesn't apply to the prepackaged software that you sell to various users and when you require portability. In our case, we assume a stationary web application for a single deploy point, so portability is not an issue, but reproducible tests are an issue.

Ultimately, your acceptance testing will consist of the following steps:

1. Deploy the application to the test server.
2. Run acceptance tests on your machine.

Of course, you can run acceptance tests on the test server. To do so, you just need to configure tests to use the usual loopback network interface, `localhost`. However, it will require you to install additional software for your test server irrelevant to the application itself. For example, if you decide to run full-stack, in-browser tests using **Selenium**, you'll probably need to install a web browser, Java runtime, and virtual framebuffer software on the test server, and this will lead to installation of a significant amount of system-related libraries, which probably is just a waste. It's a lot more efficient to use your own desktop environment to run the web acceptance tests.

> This cannot be said about unit and functional tests, of course. Unit tests, due to their nature, are run on the raw code base, without the need to deploy at all. Functional tests should be run on the deployed application because they are required to test the validity of interactions between the configured and working parts of the final application.

In any case, ideally you should end with a simple command, named something like `deploy`, which will do the following:

1. Access and launch the target machine (especially if it's a virtual machine instance).
2. Ensure that there is a valid environment expected by the application.
3. Copy the current state of the code base to the target machine.

4. Configure the copied code base for the environment on the target machine.
5. Launch the application.

You should be able to do all of the preceding steps by typing `deploy` in the command line and hitting *Enter*. As Martin Fowler said in his definitive article *Continuous Integration* (seen last time at `http://martinfowler.com/articles/continuousIntegration.html`), this should become a non-event for you. Ideally, deployment should happen automatically when you launch the acceptance test harness.

In this book, we'll concern ourselves with only the last two steps of the procedure. As we're working with a PHP application, the "launch the application" step typically will be completed as soon as we have a web server running on a target machine.

This is a book about web application development and not about system maintenance, and it's targeted at web developers and not operation engineers. However, in *Appendix A*, *Deployment Setup with Vagrant*, we prepared the description of one setup based on the usage of a virtual machine, which you can easily repeat on just any desktop workstation. You will not need a separate physical machine, and you will still be able to imitate a real-world deploy procedure. If you don't have other options, you are strongly encouraged to read it. In fact, all of the code in this book was prepared using the setup described there. Let's pretend that you have an environment prepared with a `deploy` command, and for simplicity, we assume it'll be run before each run of the acceptance testing suite. The result of your deploy should be the single URL accessible from your machine, which the acceptance testing harness will use as the entry point to your application.

Now, let's go to the section of Codeception configuration that is relevant for the acceptance test suite within the file `tests/acceptance.suite.yml`, and add that URL in the `modules.config.PhpBrowser.url` token. The file, assuming you did not modify anything else and nothing has changed in the default Codeception installation since this chapter was written, should look like the following:

```
class_name: AcceptanceTester
modules:
    enabled:
        - PhpBrowser
        - WebHelper
    config:
        PhpBrowser:
            url: 'http://YOUR.APPLICATION.URL'
```

For example, if you configure the target machine with the Apache web server using the IP-based virtual host technique (as described at `https://httpd.apache.org/docs/2.2/vhosts/ip-based.html`), the `modules.config.PhpBrowser.url` value can look like `http://127.0.0.1:8000`.

As we change the configuration, we should rebuild the Codeception harness. Here is the command to do it:

```
$ cept build
```

Do not forget that `cept` is an alias we created ourselves. The real executable is in the `./vendor/bin/codecept` file.

If you run the tests now:

```
$ cept run
```

You will see an output as shown in the following screenshot:

You'll see that Codeception now shows something on the / route but not what we expected it to. It'll either be a 404 error or 403 error, depending on the version of Apache used, or maybe something else if you are using a different web server. Anyway, the root of the problem is simple, that is, we need an `index.php` file inside the web-accessible directory.

Making a web application entry point visible

Let's decide on the convention here: the only folder that will be accessible from the Web will be called `web`, placed at the root of the code base. For example, if your web server is Apache, it'll be the `web` folder's path that you put into the `DocumentRoot` directory.

Given that, just put the following content as the `index.php` file in the `web` subdirectory:

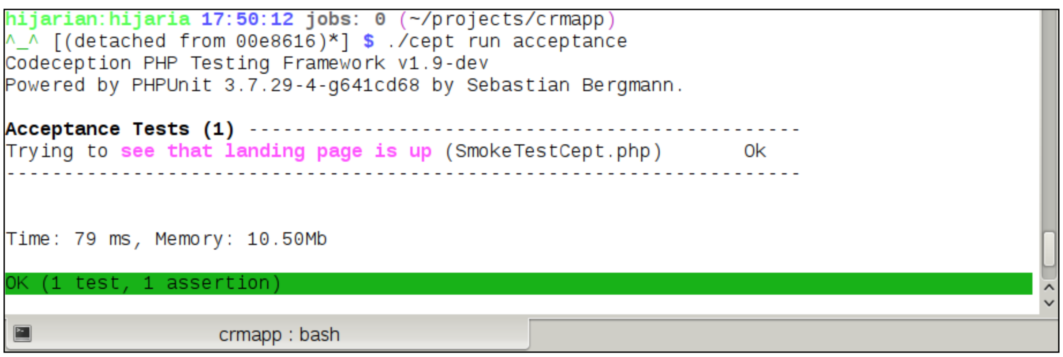
```
Our CRM
```

Yes, just a seven-character text file. After all, that's everything our acceptance test expected, right?

Then we run the tests:

```
$ cept run
```

We get the following output:



Now we need to allow ourselves to use Yii 2 in our project. An easy way to do this is just to write the full, end-to-end test, which describes our desired functionality.

Introducing the Yii framework into our application

Now that we have the entire supporting infrastructure we need to begin working with, let's return to our first feature we defined at the design stage and define the acceptance test for it.

First end-to-end test

The main point with end-to-end acceptance tests is that we have to deal with our application using only its UI. We don't have any way of direct access to the database or, worse, filesystem around the application. So, to test a query for some data in the database, this data should be inserted into the database first. And it should be done using the UI.

Here are the resulting test steps:

1. Open the UI for adding customer data to the database.
2. Add customer #1 to the database. You should see the Customer List UI with one record.
3. Add customer #2 to the database. You should see the Customer List UI with two records now.
4. Open the UI to query customer data by the phone number.
5. Query using the phone number of customer #1. You should see the query results UI with data for customer #1 but not for customer #2.

So, this test forces us to have three user-interface pages: the new customer records, the customer list, and the query UI. That's part of why it's called "end-to-end" testing.

Translated to the Codeception acceptance test suite, the procedure just described will look like this:

```
$I = new \AcceptanceTester\CRMOperatorSteps($scenario);
$I->wantTo('add two different customers to database');

$I->amInAddCustomerUi();
$first_customer = $I->imagineCustomer();
$I->fillCustomerDataForm($first_customer);
$I->submitCustomerDataForm();

$I->seeIAmInListCustomersUi();

$I->amInAddCustomerUi();
$second_customer = $I->imagineCustomer();
$I->fillCustomerDataForm($second_customer);
$I->submitCustomerDataForm();

$I->seeIAmInListCustomersUi();

$I = new \AcceptanceTester\CRMUserSteps($scenario);
```

```
$I->wantTo('query the customer info using his phone number');

$I->amInQueryCustomerUi();
$I->fillInPhoneFieldWithDataFrom($first_customer);
$I->clickSearchButton();

$I->seeIAmInListCustomersUi();
$I->seeCustomerInList($first_customer);
$I->dontSeeCustomerInList($second_customer);
```

Let's put it as is inside the `tests/acceptance/QueryCustomerByPhoneNumberCept.php` file. This is our goal for the current chapter.

Let's go over the not-so-obvious things in this test script.

First, we broke down the scenario into two logical parts and made two different subclasses of `AcceptanceTester` to stress this distinction. Codeception has a nice helper to automatically generate subclasses of different `Guy` classes, using which we created the `\AcceptanceTester\CRMOperatorSteps` class as shown in the following command:

```
$ cept generate:stepobject acceptance CRMOperatorSteps
```

Composer will prompt you for the names of the methods in the step where the object class is being generated. Just hit *Enter* there without writing anything to tell it that you want to start afresh.

This helper is used to support the **StepObject** pattern (see `http://codeception.com/docs/07-AdvancedUsage#StepObjects`), so it'll automatically append the `Steps` suffix to the `CRMOperatorSteps` class name. Of course, it's *a lot* more natural to reason about the subclasses of `AcceptanceTester` as having different roles than having some abstract containers of steps. However, if we forcefully rename the generated classes, removing the unnecessary suffix, we would lose the auto-loading ability that Codeception automatically provides us with, so we'll just tolerate it. The preceding incantation places the `CRMOperatorSteps.php` class file into the `tests/acceptance/_steps` subdirectory.

In the same way, the `CRMUserSteps` class can be generated.

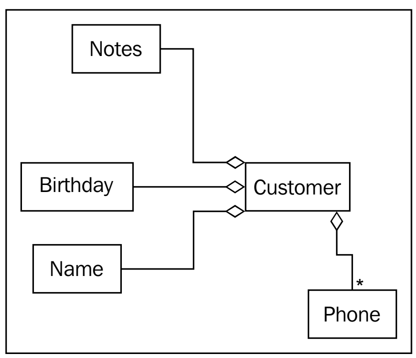
Now, let's define the steps mentioned in the test scenario. Almost all of these high-level steps will just be the containers of the more low-level steps built-in with the acceptance tester shipped with Codeception.

First, we will see the CRM operator steps.

The "I am in Add Customer UI" step will just be an opening of the route corresponding to our future Add Customer UI, so it'll look like this:

```
function amInAddCustomerUi()
{
    $I = $this;
    $I->amOnPage('/customers/add');
}
```

"Imagine Customer" is a helper to generate customer data to be entered in the Add Customer UI automatically and in a random way. Such placeholder data can be generated in any way. We'll be using the amazing **Faker** library (`https://github.com/fzaninotto/Faker`) to generate the correct-looking data for us. However, we'll look into that a bit later. Right now, the need to enter data in the Add Customer UI forces us to decide on the actual UI. We don't go after any fancy interface here; it'll be just a plain HTML form with the **Submit** button. But what fields should be there? Let's return to our **Customer** aggregate and see what parts of it we crucially need to fulfill our test scenario:



For simplicity, we leave off the **E-mail** and **Address** models for later. Of course, we don't take into account the **Contract** aggregate at all. For the task to be any "interesting", we include all the unique parts of the customer: **Name**, **Birthday**, and **Notes**. Do remember that **Name** is a structure and not just a line of text like **Notes**.

Now, let's settle on the fields in our **Add Customer** form. Please pay attention to the naming of form fields; it's not arbitrary and corresponds to our future database schema and the Yii 2 model's configuration. Have a look at the following table:

Field	Name in form
Full Name	`CustomerRecord[name]`
Birth Date	`CustomerRecord[birth_date]`
Notes	`CustomerRecord[notes]`
Phone Number	`PhoneRecord[number]`

Note that while in our design one customer can have several phones, here we allow only one. We cannot implement features until we have a basic test done for it, and our test does not explicitly check the ability to enter several phone numbers (yet).

So, we can now define the `CRMOperatorSteps.imagineCustomer` method. First of all, let's bring the Faker library into our project:

```
$ php composer.phar require "fzaninotto/faker:*"
```

Then, let's consider a customer with the attributes defined in the following method:

```
public function imagineCustomer()
{
    $faker = \Faker\Factory::create();
    return [
        'CustomerRecord[name]' => $faker->name,
        'CustomerRecord[birth_date]' => $faker->date('Y-m-d'),
        'CustomerRecord[notes]' => $faker->sentence(8),
        'PhoneRecord[number]' => $faker->phoneNumber
    ];
}
```

Our imagination here makes a structure for us, which we can easily use in our `fillCustomerData` method:

```
function fillCustomerDataForm($fieldsData)
{
    $I = $this;
    foreach ($fieldsData as $key => $value)
        $I->fillField($key, $value);
}
```

The method to submit the form will be straightforward; let's just name the button **Submit**:

```
function submitCustomerDataForm()
{
    $I = $this;
    $I->click('Submit');
}
```

Then, we need only two methods, one for checking whether we are inside the Customer List UI and another to actually go there:

```
    public function seeIAmInListCustomersUi()
    {
        $I = $this;
        $I->seeCurrentUrlMatches('/customers/');
    }

function amInListCustomersUi()
    {
        $I = $this;
        $I->amOnPage('/customers');
    }
```

In the Codeception ideology, assertion methods are expected to have the `see` prefix on the names, so we adhere to it.

We use `CurrentUrlMatches` to match URLs using regular expressions instead of the more strict `CurrentUrlEquals`, because we assume there'll be some query parameters at the end of the URL.

With all these methods defined in the `CRMOperatorSteps` class, we now have the first half of our test case completed (which means runnable).

Let's get done with the test steps for a CRM user, who'll do the querying. The following changes are to be done in the `CRMUserSteps` class. First, the obvious one is as follows:

```
function amInQueryCustomerUi()
{
    $I = $this;
    $I->amOnPage('/customers/query');
}
```

Let's just name the field to enter the phone number in the same way we named it in the **Add Customer** form as follows:

```
function fillInPhoneFieldWithDataFrom($customer_data)
{
    $I = $this;
    $I->fillField(
        'PhoneRecord[number]',
        $customer_data['PhoneRecord[number]']
    );
}
```

Let's name the button to start searching for customer data as the **Search** button as follows:

```
function clickSearchButton()
{
    $I = $this;
    $I->click('Search');
}
```

Then, we arrive at the duplication of `CRMOperatorSteps.seeIAmInListCustomersUi`:

```
function seeIAmInListCustomersUi()
{
    $I = $this;
    $I->seeCurrentUrlMatches('/customers/');
}
```

Let's just stick to the **Rule of Three** proposed in *Refactoring: Improving the Design of Existing Code, Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, Addison-Wesley Professional*, and let this method be as it is.

Lastly, here are our assertions:

```
function seeCustomerInList($customer_data)
{
    $I = $this;
    $I->see($customer_data['CustomerRecord[name]'], '#search_
results');
}
function dontSeeCustomerInList($customer_data)
{
    $I = $this;
```

```
        $I->dontSee($customer_data['CustomerRecord[name]'], '#search_
results');
    }
```

We should note that this is an extremely simple implementation, and it relies on several assumptions, which will be held true at this stage of development:

- All customers have a name defined
- No customers have identical names
- Search results are being rendered inside the HTML element with the ID `search_results`

Let's keep this test as it is for simplicity, but when we have more than one search result, we should think about how to properly check for a particular search result's existence (and most probably, the default semantics of the `see` method will not be enough for us).

Quite an important question is why we don't check for the customer data being shown in the Customer List UI after each addition of a new customer. After we query for the phone number, we end up in the same Customer List UI after all.

Well, *for now*, the reasoning is dead simple: our goal is to check that we can query for a customer by the phone number. Also, the existence of some assertions halfway through the test case would violate the **Single Assertion principle** (explained in detail in *Clean Code*, *Robert Martin*, *Prentice Hall*). However, as this is the end-to-end acceptance test, doing this is probably not such a bad thing. Anyway, nothing prevents us from extending this test further (again, it's an acceptance test imitating the behavior of a real user), but for now, let's stick with this simple scenario.

If you run the completed test scenario now, you should get the following error message:

```
1) Failed to add two different customers to database in
QueryCustomerByPhoneNumberCept
Sorry, I couldn't fill field "CustomerRecord[first_name]","Cheyanne":
Field by name, label, CSS or XPath 'CustomerRecord[first_name]' was
not found on page.

Scenario Steps:
2. I fill field "CustomerRecord[first_name]","Cheyanne"
1. I am on page "/customers/add"
```

We get this error because we don't have the HTML form served for the `/customers/add` request.

We have finally arrived at the Yii 2 installation procedure.

Yii 2 installation to the bare code base

We're going to make a totally custom application, which will not have the intention to rely much on the directory structure conventions of the Yii framework, instead just using its classes where it's convenient.

First of all, you need to declare Yii 2 as a dependency of your application.

Either add the required `require` line for Yii 2 manually in the `composer.json` file or run the following command:

```
$ php composer.phar require "yiisoft/yii2:*"
```

If you edited the manifest manually, do not forget to call the installation command:

```
$ php composer.phar install
```

Composer will bring Yii 2 to your code base after that. It should end inside the `vendor/yiisoft/yii2` folder.

Checking the requirements

Yii 2 includes an important feature, which is a built-in requirement checker. When you install one of the application templates discussed in *Chapter 1*, *Getting Started*, you get a script named `requirements.php` in the root of your code base. It's very useful, so make a copy and place it into the `web` subfolder. You can download this file from the Yii 2 repository at `https://github.com/yiisoft/yii2/blob/master/apps/basic/requirements.php`. After you get the file, you can run it from the command line by calling the following:

```
$ php web/requirements.php
```

Alternatively, you can just point your browser at the URL `http://<your_domain>/requirements.php` and get a nicely laid-out page with detailed explanations about whether your deploy target satisfies the framework requirements.

An introduction to Yii conventions

A really high-level explanation of things is as follows. To serve any of the requests coming to the application, Yii uses a single physical PHP script that instantiates a special object of the `\yii\web\Application` class. This object uses Yii's interpretation the of **Model View Controller (MVC)** composite pattern to process the request and display the result back to the sender. If you forgot or were unaware of the MVC before, then you probably want to read at least the official documentation for Yii for an in-depth explanation.

The Yii interpretation of MVC is as follows:

- **View** is a class that does the rendering of whatever you send to the client. Usually, it is the HTML page, but you are not restricted to this
- **Model** is a class that contains all the business logic
- **Controller** is a class that receives the user request, decides what to do with it, calls models if necessary to do the real work, and then uses a view to render and send the result back to the user

The most subtle part of this scheme is the model concept. Depending on the interpretations, a model is either what the controller uses to get data to put into a view or actually *is* what the controller puts into a view. Yii 2 does not enforce any approach, but its implementation of models assumes that the model is a container for some data, either transient (in-memory only) or persistent (with the support of the active record pattern).

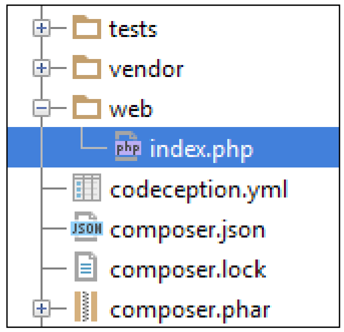
So, a request passes through the following steps:

1. The web server receives the request and passes it to the `index.php` script.
2. A Yii `Application` object is created. It decides which `Controller` class should be used to handle this request.
3. A `Controller` object is created. It decides what `Action` it should run (they can be either methods of `Controller` or separate classes), and run it, passing the request details there.
4. An action is performed, and, if it were properly constructed by the programmer, it returns something rendered by the view. It's not enforced by the framework in any way; you can have controller actions that do not render anything.
5. A special application component is responsible for formatting data before sending it back to the user.
6. Resulting data, be it HTML, JSON, XML, or an empty response, is sent to the user.

Knowing these steps, let's modify our current entry script, so it'll render the same thing, but using the Yii framework instead of displaying a raw text output. We'll look at a nice flowchart with all the details in *Chapter 12*, *Route Management*.

Building the wireframe code

Right now, we should have the following project structure:



We will start by introducing Yii 2 from the entry point script. At a reasonable minimum, the `index.php` file should look like this:

```
<?php
// Including the Yii framework itself (1)
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
// Getting the configuration (2)
$config = require(__DIR__ . '/../config/web.php');
// Making and launching the application immediately (3)
(new yii\web\Application($config))->run();
```

At (1) in the code, we bring all that Yii needs to function properly to our environment.

At (2) in the code, we get the configuration tree for the application. Yii application config is a massive PHP array describing the initial values of attributes of the application itself as well as its various components.

At (3) in the code, we make a new instance of the `Application` subclass representing web applications and immediately call the method called `run` on it.

At (2) in the code we are loading a nonexistent file `config/web.php`. Let's make it:

```
<?php
return [
    'id' => 'crmapp',
    'basePath' => realpath(__DIR__ . '/../'),
    'components' => [
        'request' => [
            'cookieValidationKey' => 'your secret key here',
        ],
    ],];
```

We must specify three settings:

- `id`: This is a mandatory identifier for our application. It is required because later we can split our application into different so-called modules, which we refer to using these IDs, and the top-level application obeys the same rules as a common module.
- `basePath`: This is mandatory because it's basically the way for Yii to understand where it exists in the host filesystem. All relative paths allowed in other settings should start from here.
- `components.request.cookieValidationKey`: This is a leak from the user authentication subsystem, which we will discuss in *Chapter 5*, *User Authentication*. This setting is the private key for validating users using the *remember me* feature, which relies on cookies. In earlier beta versions of Yii 2, this key was generated automatically. It was made visible by the 4e4e76e8 commit (`https://github.com/yiisoft/yii2/commit/4e4e76e8838cbe097134d6f9c2ea58f20c1deed6`). Apart from this setting, you can set `components.request.enableCookieValidation` to `false`, thus disabling cookie-based authentication completely. This will get your application working, too.

Next, we'll add some mandatory folders, because Yii just throws exceptions in case they are not there, and doesn't create them itself: `web/assets` and `runtime`. These folders are used by the framework when the application is running.

Adding a controller

Each controller class should have three distinct qualities:

- It must belong to the namespace defined in the `controllerNamespace` setting of the `Application` class.
- It must have a `Controller` suffix in the name.
- It must extend the `\yii\base\Controller` class. In the case of the controllers that are meant to be used by the web application and not the console one, we should extend the `\yii\web\Controller` class instead.

In addition, it's important to understand how Yii 2 will actually search for controller classes.

Yii 2, in general, utilizes the autoloader compatible with the PSR-4 standard (`http://www.php-fig.org/psr/psr-4/`). To put it simply, such an autoloader treats namespaces as paths in the filesystem, given that there is a special root namespace defined, which maps to the root folder of the code base.

In our case, Yii 2 defines the `\app` namespace for us, which maps to the root folder of the code base. As a result, for example, the default value of the `controllerNamespace` setting of the application, which is `\app\controllers`, will map to the directory called `controllers` at the root of code base, so all controller class definitions must be placed in there.

Also, each class to be available through the Yii 2 autoloader has to be put into its own file named exactly like the class itself.

So, let's create our first controller to satisfy the smoke test. We will not change the default controller namespace setting, and so we will write the following piece of code in the `controllers/SiteController.php` file:

```
namespace app\controllers;
use \yii\web\Controller;
class SiteController extends Controller
{
    public function actionIndex()
    {
        return 'Our CRM';
    }
}
```

This code heavily relies on the conventions of Yii. Without delving deep into the topic of routing in Yii, we can say that without special settings defined, Yii uses the `actionIndex` method of the controller named `SiteController` for the "/" request.

The most basic and straightforward way to define controller actions is to define them as public methods of the controllers having the name prefixed by `action`. To get into the `SiteController.actionIndex` method explicitly, you should request `site/index.php`.

So, we have our smoke test passing with the Yii-based routing. Let's add some helpers for ease of debugging.

Handling possible errors

You can get a lot of strange errors at this stage of development. Let's look at the ways you can use to set up your application quickly and gather as much feedback as possible.

First of all, when you make a really dire mistake, such as not defining `id` or `basePath` in the application config, you basically get a blank page as a response from the Yii application. The only place you can look at is the logs of your web server. For example, in Apache you can use the `ErrorLog` directive to specify the file in which such game-breaking error reports will end. Of course, any other application error will end there, regardless of whether it was rendered in the browser.

To fight off the "white screen," you can add a force override `display_errors` setting in your `index.php` entry point right *after* you require the Yii library, but *before* the creation and execution of the `Application` object as follows:

```
ini_set('display_errors', true);
```

Also, you should add one handy constant *before* you require the Yii library. It's extremely important to define it before you require the Yii library, because the Yii library will define it in the event that you don't define it yourself. That's it. Here's how you do it:

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

This will change the application to debug mode, and if some nasty exceptions are thrown, you'll get the generic status 500 page or blank screen but also a detailed report from Yii with the most important lines highlighted.

Lastly, you can add a custom logger to your application, which will log the errors happening within the application to a file. *Chapter 8*, *Overall Behavior*, explains this in great detail.

Making the data and application layers

Now, let's get started on the real work.

To satisfy our end-to-end test, we can start from many different levels of our application. But as we already know we will work with Yii, let's figure out what we should do in the controller.

We need to provide two routes: `/customers/add` and `/customers`. Here is the controller declaration we need for them:

```
namespace app\controllers;
use yii\web\Controller;

class CustomersController extends Controller
```

Obeying the default settings, route `/customers` equals to the route `/customers/index`. We need to provide the method named `actionIndex` to enable this route.

What will we do in this route? Our intention is to provide a list of customers recorded in the database. We have a separate route for querying the database for specific customers; thus we will have a special query parameter when accessing `/customers/index`. If this parameter is set, we filter the customer list. Otherwise, we display all customers from the database.

Here is the high-level representation of the `actionIndex` method:

```
public function actionIndex()
{
    $records = $this->getRecordsAccordingToQuery();
    $this->render('index', compact('records'));
}
```

> PHP's built-in function `compact('var_name_1', 'var_name_2', ...)` is immensely useful when using Yii. There are a lot of places where you should pass associative arrays to some functions, and the variables will have the same names as the keys of these arrays. If you did not know about this function already, we suggest that you consult the PHP function reference and learn about it.

Now we just need to figure out `getRecordsAccordingToQuery`. To get to this, we need to have our database first. But before that, let's settle on the customer aggregate definition.

Defining the customer model at the data layer

Why do we need the customer domain model separate from the handy ORM at all? For a simple reason: we will lay our data in the database in a different way than our domain model will be structured. Some parts of the customer domain model will be stuffed into a single database table. Some parts will be separated out to different tables, and maybe there'll be some additional tables to represent many-to-many relations. More than that, we definitely don't want the ORM details to leak to the layer where the controller is, because the very moment we do this, we'll bind ourselves to this ORM forever with no chance to replace it with anything modern.

A customer model is just a data holder class (for now), so it doesn't need any tests for it. Here's how we'll define it:

```
namespace app\models\customer;

class Customer {
    /** @var string */
```

```
        public $name;

        /** @var \DateTime */
        public $birth_date;

        /** @var string */
        public $notes = '';

        /** @var PhoneRecord[] */
        public $phones = [];

        public function __construct($name, $birth_date)
        {
            $this->name = $name;
            $this->birth_date = $birth_date;
        }
    }
```

As we place it in the `app\models\customer` namespace, for it to be found by the autoloader, we need to place the file in the `models/customer` subdirectory.

We imagined one additional domain object: `Phone`. Here's how it is defined:

```
namespace app\models\customer;

class Phone {
    /** @var string */
    public $number;
}
```

So, for a while, our `Customer` aggregate will be just a dumb data structure, holding other data structures consisting of primitive values. To create a `Customer` object, we must provide the name and birthdate, but apart from that, all fields are public, so anyone can do anything with the aggregate.

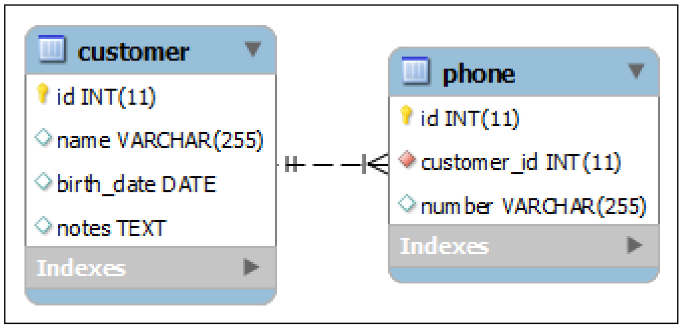
Now let's think about how we'll store this model in the database.

Setting up the database

Consider you ran the following query in your MySQL database:

```
create database `crmapp` default character set utf8 default collate
utf8_unicode_ci;
```

According to our `Customer` aggregate, we need the database structure depicted in the following figure (taken by using the MySQL Workbench, which is available for download at `http://dev.mysql.com/downloads/workbench/`).



As we certainly don't want to set up this database schema manually on each deploy, we need some kind of automated way of doing this. Yii supports the concept of migrations exactly for this purpose.

But to be able to use it, we need two things.

First, we need our own version of the console runner from Yii. With this, we'll be able to make our own console commands to automate things, but for now, all we need is the built-in `migrate` command. The script named `yii` should be placed in the root of the code base with the following content in it:

```
#!/usr/bin/env php
<?php
define('YII_DEBUG', true);

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);
```

This is basically a trimmed version of the `yii` launcher script bundled with the basic application template we set up in *Chapter 1*, *Getting Started*. You can see both the differences and similarities with `index.php` for the web application. *Don't forget to make this file executable!* Given that you're on a POSIX-compatible system this can be done with `chmod +x`.

This script requires the `config/console.php` configuration script. Let's make it as follows:

```
<?php
return [
    'id' => 'crmapp-console',
    'basePath' => dirname(__DIR__),
    'components' => [
        'db' => require(__DIR__ . '/db.php'),
    ],
];
```

Note that line starting with `db`? That's where our database is being introduced (at last). Settings for the `db` component are in a separate configuration snippet because we'll use exactly the same settings for our web application. Here's how the `db.php` file looks:

```
<?php
return [
    'class' => '\yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=crmapp',
    'username' => 'root',
    'password' => 'cheesy/hamburger'
];
```

Well of course, your username and password will be different. A database with that name should also exist in MySQL at this time.

Having all these three elements in place: `yii` launcher script, `console.php` configuration file for it, and the database settings separated out to `db.php`, we can now create a migration script using the following console command:

```
$ ./yii migrate/create init_customer_table
```

This will automatically generate the directory named `migrations` under the root of your code base and the class with a really lengthy name like `m140204_190825_init_customer_table` in its own file.

To keep it short, a Yii-style migration script is a class with two methods: `up` and `down`. In the `up` method, you describe the change you want to make in the database. In the `down` method, you describe how to revert the changes in the `up` method or specify by returning `false` that you give up; it's impossible, and you cannot revert from this particular migration.

When you run `yii migrate/up` or its shorthand `yii migrate`, Yii checks all classes inside the `migrations` subdirectory, which are sorted naturally by the datetime stamps in their names, and executes the `up` method in all of them, from first to last. It then writes the names of the migrations applied to the `tbl_migration` table in the database affected, so it will not rerun the same migration again next time. For safety, `yii migrate/down` does not have any shorthand and reverts only `one` migration at a time by default. You can specify the number of migrations to revert via call to `yii migrate/down <number>`.

With the migration scripts, you can just add the line `yii migrate` to your deploy script and can be sure that all changes you want to be done in the database will be there.

Here's the content of the `up` method, which we set up for the `customer` table:

```
        $this->createTable(
            'customer',
            [
                'id' => 'pk',
                'name' => 'string',
                'birth_date' => 'date',
                'notes' => 'text',
            ],
            'ENGINE=InnoDB'
        );
```

You revert this change with a simple line of code:

```
$this->dropTable('customer');
```

In another migration, we'll write the generation of the `phone` table, with the foreign key explicitly declared against the customer table:

```
        $this->createTable(
            'phone',
            [
                'id' => 'pk',
                'customer_id' => 'int unique',
                'number' => 'string',
            ],
            'ENGINE=InnoDB'
        );
        $this->addForeignKey('customer_phone_numbers', 'phone',
'customer_id', 'customer', 'id');
```

You write it in reverse order, or else MySQL will complain:

```
$this->dropForeignKey('customer_phone_numbers', 'phone');
$this->dropTable('phone');
```

Let's not bother with `Address` and `Email` tables for now. We wrote so much of the untested code already that it's giving me chills.

Don't forget to actually run these migrations you just created by this command:

```
./yii migrate
```

Now, while we're at it, let's set up the **object-relation mapping (ORM)** for these two tables.

Object-relation mapping in Yii

ORM in Yii is supported using the Active Record pattern. A developer declares the class of objects corresponding to the tables (so, you have one active record class definition per table), and then you have quite a lot of functionality already done for him. Yii implementation of this pattern helps to:

- Store active records in the database
- Validate data assigned to the active record's fields before storing it in the database
- Retrieve active record using the primary key, some specific attribute(s), custom queries, or just all of them at once

All you need to do is set up the active record definition properly. Let's begin with the `customer` table.

As we already have the customer name reserved for the domain model, we'll name the active record `CustomerRecord`, which is logical. We'll define the following class in the `app\models\customer` namespace:

```
namespace app\models\customer;

use yii\db\ActiveRecord;

class CustomerRecord extends ActiveRecord
{
    public static function tableName()
    {
        return 'customer';
    }

}
```

Again, for the autoloader to work, this file should be saved as `CustomerRecord.php` in the `models/customer` folder.

Well, honestly, that's all you need to define right now. If you worked with Yii 1.x before, you're probably as amazed as we were. We're lacking the validation rules, though. As we already have our database schema defined and loaded, nothing stops us from defining some validation rules:

```
public function rules()
{
    return [
        ['id', 'number'],
        ['name', 'required'],
        ['name', 'string', 'max' => 256],
        ['birth_date', 'date', 'format' => 'Y-m-d'],
        ['notes', 'safe']
    ];
}
```

That already reads like prose apart from the `safe` rule. Right now, it's sufficient to know that this rule means that we can assign anything to this field. Why does the safe rule exist at all, you may wonder; because of a special helper method in the `\yii\base\Model` class, called `setAttributes`, which accepts an associative array with attribute names and values. By default, if an attribute is not constrained by some validation rule and not marked as safe by the safe rule, it'll be ignored by this method. This feature allows us to pass the entire content of `$_POST` to the `setAttributes()` method and be sure that only expected values will be assigned to the model.

> You can read the full list of built-in validators in the Yii documentation: `https://github.com/yiisoft/yii2/blob/master/docs/guide/input-validation.md`.

The `PhoneRecord` class is a lot simpler:

```
namespace app\models\customer;
use yii\db\ActiveRecord;

class PhoneRecord extends ActiveRecord
{
    public static function tableName()
    {
        return 'phone';
    }
```

```
        public function rules()
        {
            return [
                ['customer_id', 'number'],
                ['number', 'string'],
                [['customer_id', 'number'], 'required'],
            ];
        }
    }
```

We declared the `number` field to be validated as string because we don't want to enforce the proper phone format at this stage. Later, we'll probably invent our own validator for this.

The `CustomerRecord. name`, `PhoneRecord.customer_id`, and `PhoneRecord.number` fields declared are required to prevent the possibility of submitting empty forms.

Both these classes should be placed in files in the `models/customer` subfolder.

Now, we've done everything related to the database preparation. Let's use it.

Decoupling from ORM

We need two pieces of functionality to satisfy our end-to-end test:

1. Get the customer record from the database that has the given phone number.
2. Store a new customer record in the database.

We don't even need to fetch the full list of customers, because we don't check it.

As we have our tiny domain model layer consisting of the `Customer` and `Phone` models, it would be wise to keep it isolated from the framework, so we need a translation layer between ORM from Yii 2 and the domain models. However, it would take a lot of pages to describe a proper setup of the **repository** pattern, and it's not really one of the topics of this book.

We'll settle on just two methods inside `CustomersController`, but overall, it's just a tradeoff only for our example application. In real-world, large-scale applications, you will surely need a proper translation layer for three reasons:

- ORM using active records is convenient to use, but it is very costly as it makes too many requests to database. At some point, you'll want to replace your usage of active records in some places to something at a lower level, such as Yii **DAO** (`https://github.com/yiisoft/yii2/blob/master/docs/guide/db-dao.md`) or maybe completely bypass Yii and use raw **PDO** calls. Without the repository, it's possible that your only option will be full text search for names of your `ActiveRecord`-based classes through the whole code base.
- It's possible that at some point you'll want to change the underlying database from something supported by Yii to something unsupported. Again, to make this change you'll need to do a lot of changes in various places.
- It's completely possible that your application will outlive Yii Version 2 and will meet Version 3, which will have random changes to the public API. Without the decoupling from the framework, at least its most invasive part, which is the database access layer, you'll get almost no possibility to upgrade.

To not bloat this chapter too much, let's make the following methods in `CustomersController`.

The first one is the method to store the customer model in the database as follows:

```
    private function store(Customer $customer)
    {
        $customer_record = new CustomerRecord();
        $customer_record->name = $customer->name;
        $customer_record->birth_date = $customer->birth_date-
>format('Y-m-d');
        $customer_record->notes = $customer->notes;

        $customer_record->save();

        foreach ($customer->phones as $phone)
        {
            $phone_record = new PhoneRecord();
            $phone_record->number = $phone->number;
            $phone_record->customer_id = $customer_record->id;
            $phone_record->save();
        }
    }
```

As you can see, we receive the `Customer` instance but are using the active records to record the data from it to the database. Note that the new instance of `CustomerRecord` magically gets the `id` field value after it was saved by the `save()` method.

The following is the method to convert from `ActiveRecord` instances to `Customer`:

```
private function makeCustomer(
    CustomerRecord $customer_record,
    PhoneRecord $phone_record
) {
    $name = $customer_record->name;
    $birth_date = new \DateTime($customer_record->birth_date);

    $customer = new Customer($name, $birth_date);
    $customer->notes = $customer_record->notes;
    $customer->phones[] = new Phone($phone_record->number);

    return $customer;
}
```

We accept a single `PhoneRecord` instance because we are dealing only with single phone number per customer right now. This method needs to be changed when there'll be a need to support several phones per customer.

These two methods are essentially the translation layer between Yii and our domain model.

Now, we will build the actual user interface and implement querying by phone number along the way.

Creating the user interface

With the `CustomersController` able to convert between domain models and active records, we move to the actual user interface pages at last.

The Add New Customer UI

What should Controller do for us when we arrive at the `/customer/add` URL? Well, it should just render the UI for us:

```
public function actionAdd()
{
    $this->render('add');
}
```

That's right, that's all. This code relies on the important convention of Yii describing from where the controller should get its views. Basically, if we have `CustomersController`, then its views are expected to be in the `views/customer` subdirectory. So, when we render something called `add` here, we're referring to the file `views/customer/add.php`. Let's create it.

What do we want on this page? According to our end-to-end test, there should be just the form to enter the customer information, along with their phone number and a **Submit** button.

Yii has a very extensive set of helpers to implement the web forms quickly and easily. The core concept behind them is the `ActiveForm`, which is the abstraction of the web form corresponding to some model from the Yii MVC interpretation. We initialize the `ActiveForm` and then use its methods to generate the HTML for fields corresponding to the model's attributes. Using the `ActiveForm`, here's how the `views/customers/add.php` view will look:

```
<?php
use app\models\customer\CustomerRecord;
use app\models\customer\PhoneRecord;
use yii\web\View;
use yii\helpers\Html;
use yii\widgets\ActiveForm;

/**
 * Add Customer UI.
 *
 * @var View $this
 * @var CustomerRecord $customer
 * @var PhoneRecord $phone
 */

$form = ActiveForm::begin([
    'id' => 'add-customer-form',
]);
```

```
echo $form->errorSummary([$customer, $phone]);
echo $form->field($customer, 'name');
echo $form->field($customer, 'birth_date');
echo $form->field($customer, 'notes');

echo $form->field($phone, 'number');

echo Html::submitButton('Submit', ['class' => 'btn btn-primary']);
ActiveForm::end();
```

That's basically the no-frills version; we don't provide any HTML wrappers besides the ones that will be generated by the `ActiveForm` and `Html` helper classes.

This is the only verbatim example of the view file in this book. We won't present the use clauses and documentation blocks again to save space.

The method called `errorSummary` is a nice shorthand, which will show all validation errors from the models in question in the event that the input is invalid according to the declared validation rules.

But to function, `ActiveForm` should have access to the model objects—in our case, `$customer` and `$phone`. This is done using the special mechanism of passing data to `View` from `Controller` via the second argument of the `render` method. Basically, we at least need the following in the `actionAdd` method for the form to get correctly rendered:

```
public function actionAdd()
{
    $customer = new CustomerRecord;
    $phone = new PhoneRecord;
    $this->render('add', compact('customer', 'phone'));
}
```

Now we need to solve two issues before we are able to actually look at our UI.

Routing 101

Default routing in Yii looks like this:

```
http://yourdomain/index.php?r=controller/action.
```

So, if we now have the `CustomersController` and `actionAdd` methods in it, we should access it using the URL `http://yourdomain/index.php?=customers/add`.

This is quite an unfortunate situation, and the Yii application object has a special setting in one of its components to change it to a more usual path format. You need to add the following to the `components` section of your application configuration:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'showScriptName' => false
]
```

With the `enablePrettyUrl` value set, URLs will become like this: `http://yourdomain/index.php/customers/add`.

With the `showScriptName` value set, URLs *being constructed* by Yii will not have the `index.php` starting point for them. For your web application to be able to *parse* such URLs, you should configure the corresponding URL rewriting in your web server. For example, Yii developers routinely hack Apache with the following lines in the `.htaccess` file lying in the `web` directory:

```
RewriteEngine on

# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

For any other web server, the logic should be the same. With rewriting a set correctly, you'll be able to use URLs such as `http://yourdomain/customers/add`. This is, of course, the best way to do it.

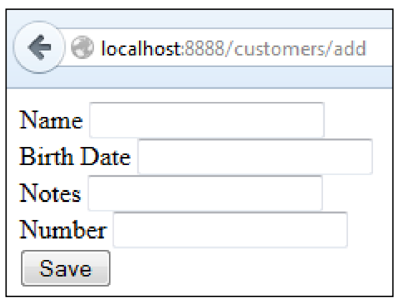
Layouts

We'll be talking about rendering in Yii in great detail in *Chapter 4*, *The Renderer*. For now, it's enough to know that by default, when you do `render` in the controller, it assumes that the view you're rendering is just a snippet wrapped by a particular script called the `layout`. By default, all controllers search for `layout` in the file `views/layouts/main.php`. The view rendering the result will be passed to the layout script as a variable named `$content`. Apart from that, layout is just another view script. It's expected that you have some parts of the web application UI common to all pages, and a layout is exactly the place to hold them.

We're pretty content with the defaults here, so let's just create some basic HTML5 boilerplate as the layout and see our **Add Customer** form at last. Place the following script as the `views/layouts/main.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>CRM</title>
</head>
<body>
    <?= $content; ?>
</body>
</html>
```

So, now you can open the `/customers/add` route with whatever method you employed when we were at the *Routing 101* section.



What? Did you expect some fancy CSS stuff at this stage of development? Our goal right now is to implement the bare functionality, so this basic HTML is more than enough.

Finishing the Add New Customer UI

Idiomatic Yii code assumes that forms send data to the same route that renders them. We'll utilize the same approach; thus, in the `actionAdd` method, we need to process possible input data. We will do it like this:

```
public function actionAdd()
{
    $customer = new CustomerRecord;
    $phone = new PhoneRecord;

    if ($this->load($customer, $phone, $_POST))
    {
```

```
            $this->store($this->makeCustomer($customer, $phone));
            return $this->redirect('/customers');
        }

        // stateful magic: both $customer and $phone will be validated
at this point
        return $this->render('add', compact('customer', 'phone'));
    }
```

Note the highlighted lines. When (and if) we successfully load the new data submitted to us via a POST request, we store the resulting customer model in the database using methods defined in our translation layer. A method called `load` is shorthand for the following four checks:

```
    private function load(CustomerRecord $customer, PhoneRecord
$phone, array $post)
    {
        return $customer->load($post)
            and $phone->load($post)
            and $customer->validate()
            and $phone->validate(['number']);
    }
```

Here are the four checks:

- `\yii\base\Model::load()` is a built-in method that allows the filling of the model attributes with the data from the `POST` request made by the `ActiveForm` widget.
- `\yii\base\Model::validate()` is a built-in method that checks the attribute values of the model against the validation rules defined in `\yii\base\Model::rules()`. Note how we can validate only a select few attributes instead of all of them. In our case, we need to validate only the `number` attribute because in the form data just submitted, there will be no `customer_id` field, which was declared required by our current rules.
- `\yii\db\BaseActiveRecord::save()` method calls `validate()` before actually saving the data in the database.

The idea behind the `validate()` implementation is that if there are any erroneous values, it will put an error message in the special attribute of any model called `\yii\base\Model::$errors`. This is what allows us to utilize the `errorSummary()` method on `ActiveForm`, because it will check the error messages one by one and print them nicely.

When the item is added successfully, we redirect to the customer list. It's probably the most interesting controller action so far.

```
    public function actionIndex()
    {
        $records = $this->findRecordsByQuery();
        return $this->render('index', compact('records'));
    }
```

This part was already discussed before. Now, we are ready to really get records according to the query. Usually, we'll just access the `$_GET` superglobal to get the query parameters, but Yii has an authentic wrapper around it, so let's use it. A special `Application` component called `Request` manages the query parameters and exports the method called `get`. With this method, we can check whether a certain `$_GET` parameter was set to a certain value:

```
private function findRecordsByQuery()
    {
        $number = Yii::$app->request->get('phone_number');
        $records = $this->getRecordsByPhoneNumber($number);
        $dataProvider = $this->wrapIntoDataProvider($records);
        return $dataProvider;
    }
```

We cheated here a bit: you don't know what the data providers are yet. To understand why we need to wrap the records fetched by the query to the data provider, we need to know how we are going to render the results.

Widgets

As themselves, data providers are not important. Their importance is that almost all widgets built in Yii use them as a source of models to render.

Widget can be seen just as the encapsulated view along with some supplementary logic. Typical widgets are:

- `ListView` for encapsulating a list of models.
- `DetailView` for encapsulating rendering of the attributes of models.
- `GridView` for encapsulating the tabular representation of the set of models. We'll discuss this enormously powerful UI element in *Chapter 11, The Grid*.

Widgets are used in views like this:

```
echo \yii\widgets\DetailView::widget($settings);
```

Settings are passed to the widget as an associative array of initial values of the widget member variables. Given the amount of self-documentation on all Yii classes, even without any documentation and code examples, you can just open the class definition of the widget and figure out how to configure it.

So, we need data providers because they encapsulate the act of finding the set of models appropriate to render right now. They do sorting, pagination, and filtering for you. They are most useful when you use the active records as a domain model in your application (that is, when you map tables to domain objects exactly). In our case, when we try our best to decouple from the ORM, we need data providers just as wrappers around our data to satisfy the widget's requirements.

The List Customers UI

The perfect data provider for our cause is `ArrayDataProvider`, which just gets the list of ready-made models and wraps them, allowing us to feed them to widgets.

So, here's what we do in `wrapIntoDataProvider`:

```
    private function wrapIntoDataProvider($data)
    {
        return new ArrayDataProvider(
            [
                'allModels' => $data,
                'pagination' => false
            ]
        );
    }
```

Setting the `pagination` setting to `false` means that we want to disable pagination capabilities as for now the pagination under the list looks just ugly.

The actual data to wrap into `DataProvider` and send to render is found like this:

```
    private function getRecordsByPhoneNumber($number)
    {
        $phone_record = PhoneRecord::findOne(['number' => $number]);
        if (!$phone_record)
            return [];

        $customer_record = CustomerRecord::findOne($phone_record-
>customer_id);
        if (!$customer_record)
            return [];

        return [$this->makeCustomer($customer_record, $phone_record)];
    }
```

So many wrappers for our `findByPhone` method!

Here, we meet the query methods of the `ActiveRecord` class for the first time. They are so numerous and the API in general is so vast, that it is probably better if you turn to the official documentation here: `https://github.com/yiisoft/yii2/blob/master/docs/guide/db-active-record.md`.

It's reasonable to construct the List Customers UI on the base of the `\yii\widgets\ListView` widget. This is how you can configure it in our case:

```
echo \yii\widgets\ListView::widget(
    [
        'options' => [
            'class' => 'list-view',
            'id' => 'search_results'
        ],
        'itemView' => '_customer',
        'dataProvider' => $records
    ]
);
```

That's the exact content of the `views/customers/index.php` file.

We need to set the `id` HTML attribute for the widget because we assume our search results are? inside the `#search_results` element in our end-to-end test.

The `itemView` element holds the name of a separate view, which will actually render each individual model from a given `DataProvider`. It's reasonable to define it in terms of the `DetailView` widget. The following is its implementation in the `views/customers/_customer.php` file:

```
echo \yii\widgets\DetailView::widget(
    [
        'model' => $model,
        'attributes' => [
            ['attribute' => 'name'],
            ['attribute' => 'birth_date', 'value' => $model->birth_
date->format('Y-m-d')],
            'notes:text',
            ['label' => 'Phone Number', 'attribute' =>
'phones.0.number']
        ]
    ]);
```

As we did not have a flat single-table-based `ActiveRecord` as our model, but a complex aggregate domain data object, it's a bit tricky to configure the fields as you can see in the code. However, even the array attributes can be referenced in the widget, which is just amazing, honestly.

Customer Query UI

The last piece of UI is laughably simple:

```
public function actionQuery()
{
    return $this->render('query');
}
```

We just show a custom handcrafted form in there:

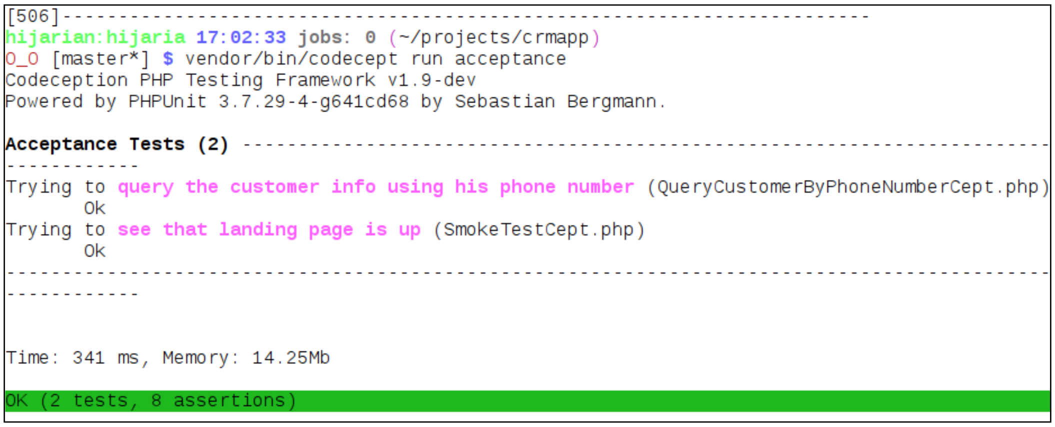
```
<?php
use yii\helpers\Html;

echo Html::beginForm(['/customers'], 'get');
echo Html::label('Phone number to search:', 'phone_number');
echo Html::textInput('phone_number');
echo Html::submitButton('Search');
echo Html::endForm();
```

We believe this code is straightforward enough to understand without any explanation. In general, the `Html` helper class makes writing view files really easy and declarative, because, as you probably noticed, we did not write any of the bare HTML code in any of our views except for the layout file.

Using the application

Now, we can look back at our end-to-end test. We assumed there that we can just use relative URLs in it like `/customers/add`, but as we learned in the *Routing 101* section, to do so, we need to tweak our web server to understand routes without the `index.php` related part. Let's assume you've done it already for simplicity.

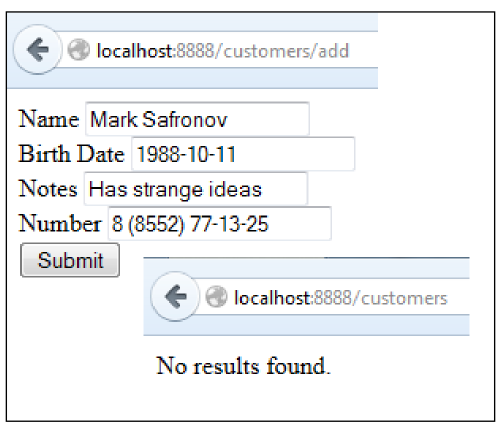
That's what you should get when you destroy your deploy machine, recreate it, and redeploy your application there, and run the acceptance tests from your own machine pointing to the deploy machine:



```
[506]----------------------------------------------------------------------
hijarian:hijaria 17:02:33 jobs: 0 (~/projects/crmapp)
O_O [master*] $ vendor/bin/codecept run acceptance
Codeception PHP Testing Framework v1.9-dev
Powered by PHPUnit 3.7.29-4-g641cd68 by Sebastian Bergmann.

Acceptance Tests (2) -------------------------------------------------------
------------
Trying to query the customer info using his phone number (QueryCustomerByPhoneNumberCept.php)
       Ok
Trying to see that landing page is up (SmokeTestCept.php)
       Ok
---------------------------------------------------------------------------
------------


Time: 341 ms, Memory: 14.25Mb

OK (2 tests, 8 assertions)
```

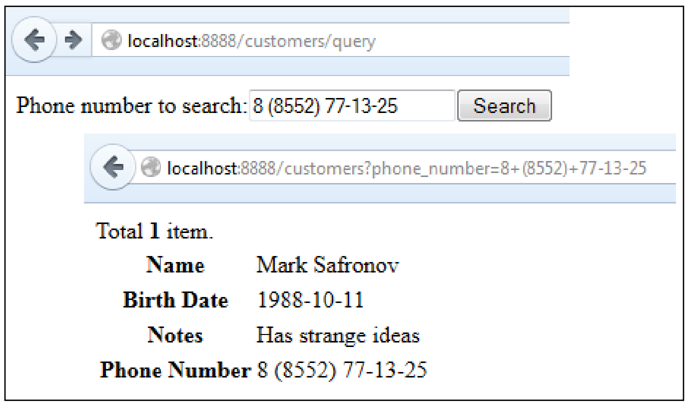
Let's walk through the UI manually, though.

For exactness, let's assume your deploy machine is reachable at `http://localhost:8080/`. Then, after the clean deploy, first you open up the Add New Customer UI at `http://localhost:8080/customers/add`, and fill the fields with something. Remember the phone number you entered. Then you click on the **Submit** button and end up at the List Customers UI, which tells you that no results were found. Have a look at the following screenshot:



It's clear that a feature is missing here: when we are on the List Customers UI we expect to see all the customers recorded in the database. Right now, this UI works like the Query Results UI really.

Manually move to the `http://localhost:8080/customers/query`, and you end up in the Query Customers UI. Paste the remembered phone number in the query field and hit **Search**. Have a look at the following screenshot:



You should end up on the List Customers UI, which tells you that only one record is found and lists the details about this record. In fact, even if there are several customers with the same phone, we know right from the source code that the system will return only one record to us anyway. That's another feature missing for us. Not mentioning that, we allow only one phone number per customer.

Let's end the session now and get out to conclude this chapter.

Summary

We covered a lot of ground in this chapter while delivering just a single feature. We did not bother with the graphic part of the web application, concentrating only on the functionality provided. A full stack of basic Yii technologies was used here, from the bottom with active records and data providers to the top with controllers, widgets, and active forms.

This chapter was expected to be an example of what is being used when developing the application with Yii. We glossed over the important part of the rendering process—the assets—but that's for the chapters to come. Next, we'll see what else Yii has up its sleeve to reduce the coding effort of the developer: the CRUD automatic code generators.

> In short, you must know a lot of concepts beforehand when developing and application using Yii. We hope that after this coding session you got a good grasp of them.

Also note that while we're not really going to follow the Yii conventions for structuring the files in our project, we ended up with a structure surprisingly similar to the basic application template. This came up naturally, and part of the answer to this is in *Chapter 7*, *Modules*, where we'll find out more about the internals of MVC implementation in Yii.

3
Automatically Generating the CRUD Code

If some part of your application just needs to have a user interface to manipulate the database table, Yii has everything to support such a design decision. Using active records that are mapped to database tables in a ratio of 1:1, you can write very succinct code in your controller that will correspond to the standard create, read, update, and delete operations on table records.

Unfortunately, when you have many similar tables to manipulate, you end up with highly repetitive code, which is also quite boring to write. To overcome this issue, Yii has a tool called Gii to automatically generate standard code for you.

In this chapter, we will see how Gii can help you while developing a Yii-based application.

Definition of the model to work with

We'll continue with the example defined in *Chapter 2*, *Making a Custom Application with Yii 2*. Let's pretend we need to manage a list of services in our database. It will just be a lookup table, defining the most important properties of a service we need:

- Name
- Hourly rate

We will create this table via the migration mechanics we covered in *Chapter 2, Making a Custom Application with Yii 2*:

1. First, we generate a new migration by calling the following command from the command line:

   ```
   $ ./yii migrate/create init_services_table
   ```

2. Then, we write the following `up` and `down` methods:

   ```
       public function up()
       {
           $this->createTable(
               'service',
               [
                   'id' => 'pk',
                   'name' => 'string unique',
                   'hourly_rate' => 'integer',
               ]
           );
       }

       public function down()
       {
           $this->dropTable('service');
       }
   ```

3. Finally, we run the migration we created:

   ```
   $ ./yii migrate
   ```

Now we have a table. Our strategy from now on will be to use Gii to perform the following:

- Generate the `Model` class for the `service` table to set up ORM
- Generate the CRUD interface

Using Gii

Before we use Gii, the automatic code generator, we need to install it into our code base.

Installing Gii into the application

Run the following command to fetch the required files to use Gii:

```
$ php composer.phar require --prefer-dist "yiisoft/yii2-gii:*"
```

The command-line parameter `--prefer-dist` is used, so we'll only get the files needed to *use* Gii, not develop it.

Now we need to wire Gii into our application:

1. First, as we're going to use something installed by Composer in our application, we need the autoloader from Composer. It's located as a file called `autoload.php` under the `vendor` subdirectory. The `require` call should be placed inside the entry point of our application in the `web/index.php` file:

   ```
   <?php
   define('YII_DEBUG', true);

   require(__DIR__ . '/../vendor/autoload.php');
   require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

   ini_set('display_errors', true);

   $config = require(__DIR__ . '/../config/web.php');

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

   The line you have to insert is highlighted.

2. Then, we need to add the Gii user interface to our application. This is done by declaring the `gii` module in the application config in the file `config/web.php`:

   ```
       'modules' => [
           'gii' => [
               'class' => 'yii\gii\Module',
               'allowedIPs' => ['*']
           ]
       ],
   ```

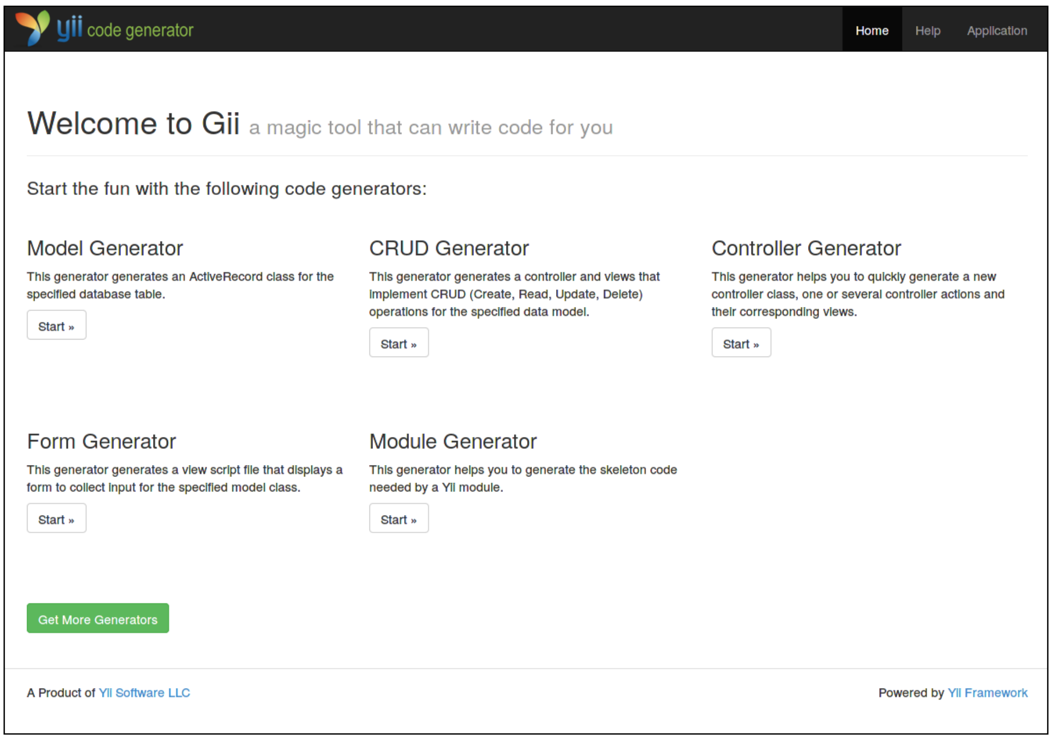
This `modules` section should be at the top level in the configuration array. The `allowedIPs` section is needed if you're working with an application that is deployed remotely (which you expected to do). By default, Gii allows access only from the localhost (`127.0.0.1` and `::1` IP addresses).

Also, it's pretty important that you add another wiring to Yii 2 internals—this time not where they should be, but inside the application config. Put the following line at the end of the config tree in the same `config/web.php` file:

```
'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php')
```

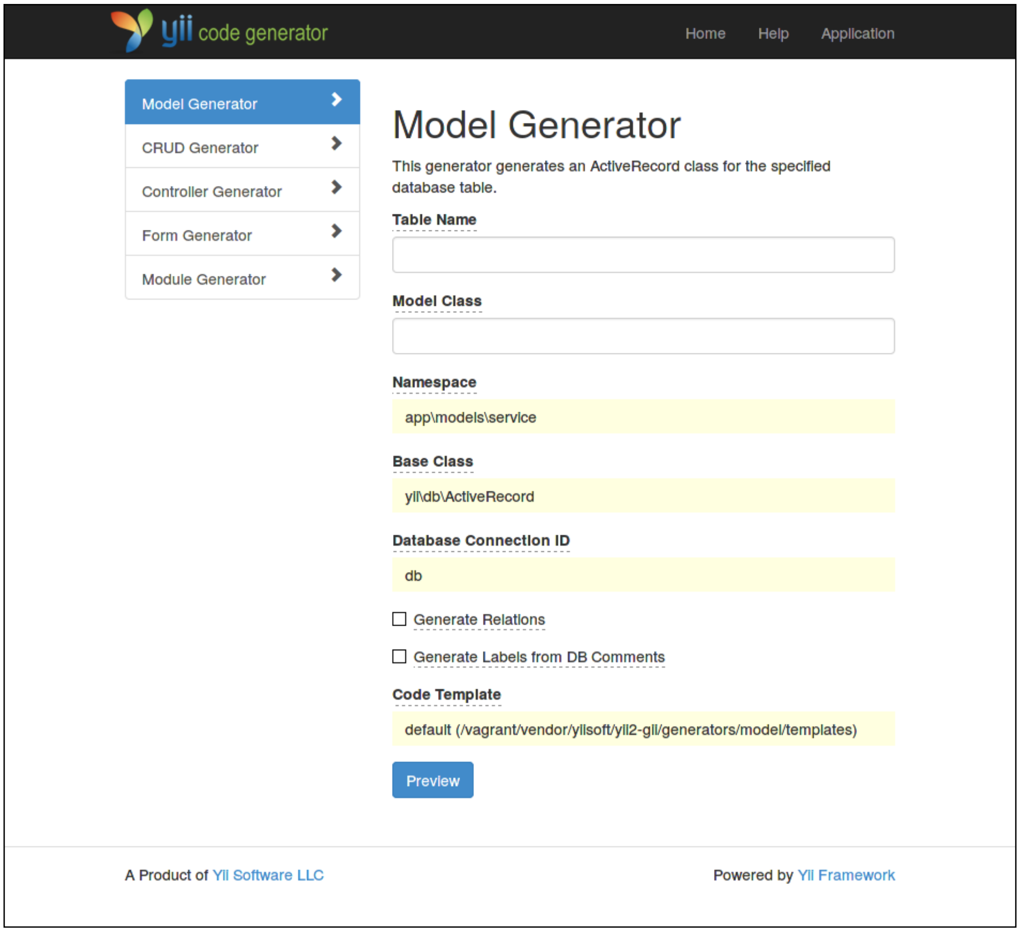
This `extensions` section should be at the top level too. Your real path can be different, but it should ultimately point at the `extensions.php` file under the `yiisoft` subdirectory. We'll explore the real meaning of this file in *Chapter 9*, *Making an Extension*.

With this, you can open the `/gii` route in your application and should finally see the Gii UI, as shown in the following screenshot:



Generating the code for the Model class

From the main menu of Gii, select the **Model** item and end up in **Model Generator**. It'll make the active record definition for you based on the table in the database.

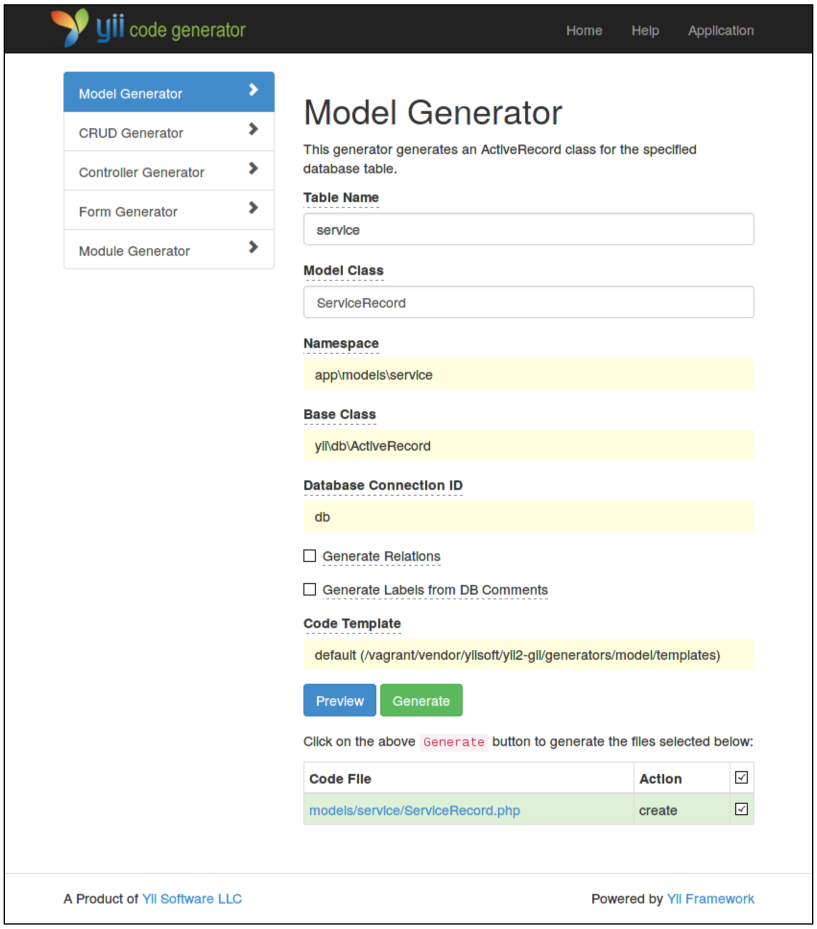


The purpose of this generator is to take the table from the database and make the `ActiveRecord` class configured to support ORM with this table. So, note that all the fields in the generator's UI are quite easy to understand, especially with all the tooltips.

You should fill in the available fields as shown in the following table and hit the **Preview** button:

Field name	Value to enter (verbatim)
Table Name	`service`
Model Class	`ServiceRecord`
Namespace	`app\models\service`

After this, you should get a page exactly like the one presented in the following screenshot:



> We named the model `ServiceRecord` to follow the convention set in the previous chapter. To be totally clear, this is not a standard naming convention (and Yii 2 does not impose any on you). The suffix `Record` is added to explicitly state that the class in question is a descendant of `ActiveRecord` and has nothing to do with the domain model of our application.

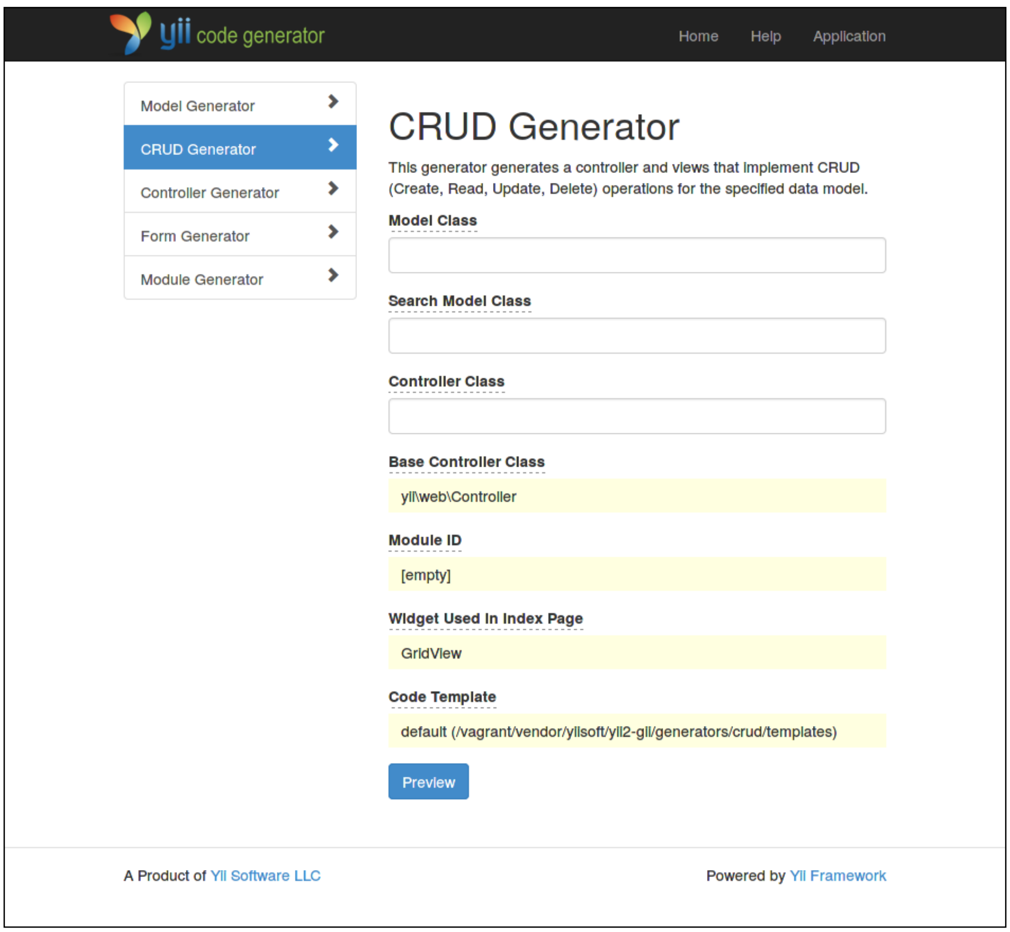
This generator exclusively creates the descendants of the `ActiveRecord` class. We declared the desired namespace for the same reason. The `app\models\service` namespace is mapped to the `models/service` subdirectory in our project.

Once you hit **Preview**, a small area listing the files to be generated will be rendered below the UI, and the **Generate** button will appear. This is a standard two-step behavior of Gii that allows you to see exactly what will be automatically generated and where.

Hit **Generate** and we're done. We have the `ServiceRecord` class definition in the `models/service/ServiceRecord.php` file.

Generating the CRUD code

Now, let's open the **CRUD Generator** item from the main menu. You will see the following page:



The purpose of this generator is to create a controller for you based on the model you specify. The controller being generated will have five actions already prepared for you:

- **index**: This action will list the models recorded in the database. This will correspond to our Services List UI.
- **view**: This action allows us to see the detailed view of a single model. This will correspond to our View Service Details UI.
- **create**: This action allows us to record new models in the database. This will correspond to our Register New Service UI.
- **update**: This action allows us to change a single model's details and update them in the database. This will correspond to our Edit Service Details UI.
- **delete**: This action allows us to remove a single model's record from the database.

In the index route, the controller will render not just a list of records, but the search functionality as well. To support this, Gii generates an additional class called Search Model, which encapsulates the mechanics of searching models by their field values.

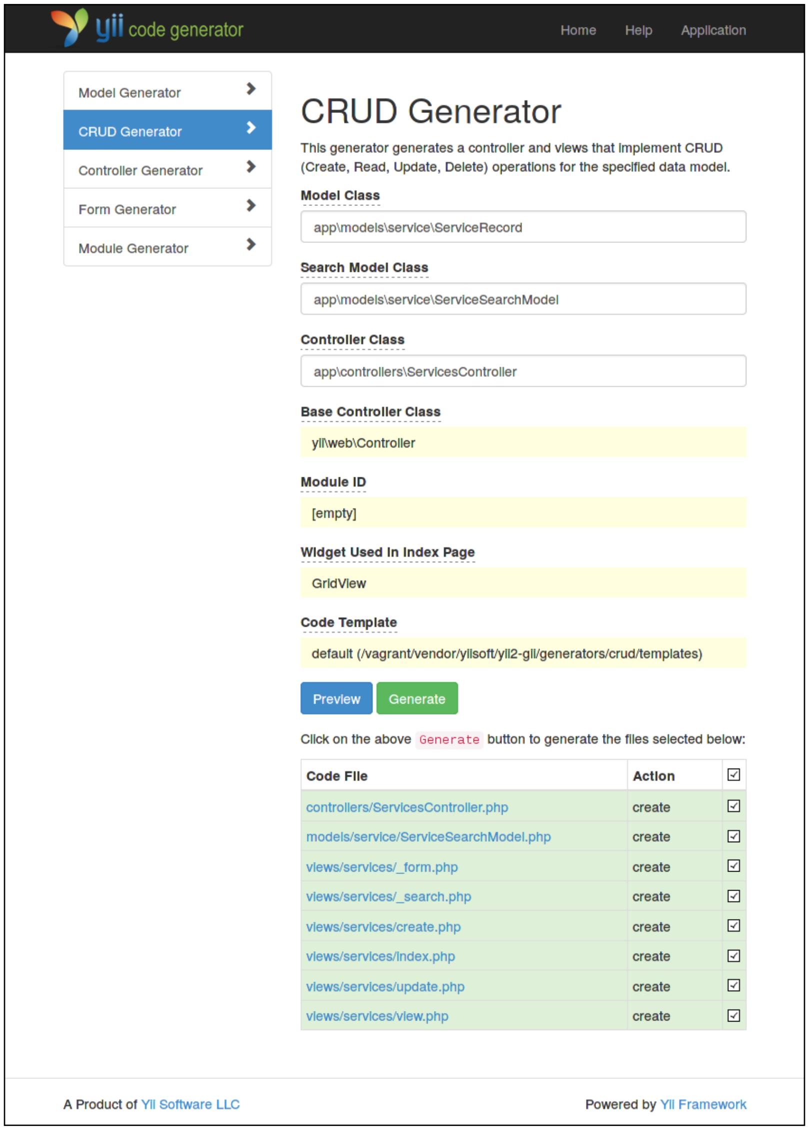
The create and update routes are so similar in their functionality that they even use the same script to generate a web form to fill the field values of a model. The only difference between those actions is that in create, the controller will add a new record to the database, and in update, it will change the existing record.

One special route in here is the delete route because of its potentially dangerous nature. First of all, it's extremely simple and designed to just do its job and redirect the visitor back to the index, making it an ideal target for an AJAX call. Second, Gii generates the controller in such a way that the delete route is accessible only by the POST request, essentially making AJAX the only way to activate this action.

Let's create the CRUD at last. Fill in the fields as described in the following table:

Field name	**Field value**
Model Class	`app\models\service\ServiceRecord`
Search Model Class	`app\models\service\ServiceSearchModel`
Controller Class	`app\controllers\ServicesController`

In the **Model Class** field, there's the name of an existing class generated previously. In other fields, there are names we created that conform to our Yii project's structure. Once you click on the **Preview** button, you should see the result shown in the following screenshot:



Note how much code this generator generates. You are encouraged to read through the code that Gii will generate. You may use a decent IDE that has a feature similar to **Go to definition** to easily navigate between the generated code and base classes in the Yii framework. We will not discuss the default boilerplate code in Yii applications in much detail, or this book will become a code-only book and will probably be three times as large.

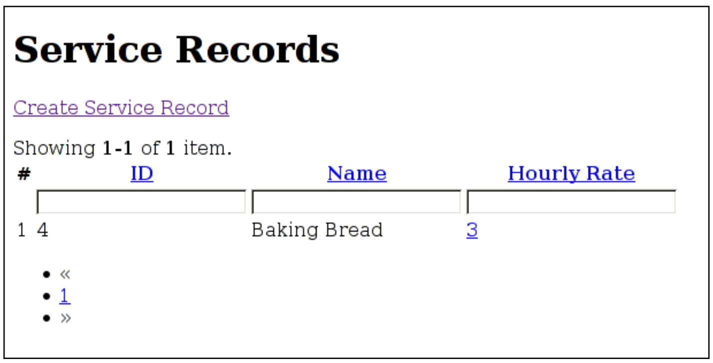
Click on **Generate** and then we'll turn to some finishing touches we need to make.

Finishing touches

You will need to apply some visual tweaks to your UI to make it appear more clean and appealing.

Creating a new layout to support pages generated by Gii

Here is how our UI looks right now, given that one record is already inserted into the `services` table:



Given our spartan visuals from *Chapter 2*, *Making a Custom Application with Yii 2*, you most probably did not expect much from the UI generated by Gii anyway; however, this obviously has some serious layout problems. The issue is that we jumped too far with this step, as we have not bothered with the looks of our application until now. *Chapter 4*, *The Renderer*, talks about rendering a system in Yii 2. However, we can take a glimpse into the future and prepare a bare minimum of the view code to enable the standard Yii 2 project design, as you may have seen in the basic application template already.

Our only issue, really, is the too-limited layout file. Let's recall what our `views/layouts/main.php` looks like:

```
<!DOCTYPE html>
<html>
<head>
    <title>CRM</title>
</head>
<body>
    <?= $content; ?>
</body>
</html>
```

This is not what Gii expects from us. To not delve into much detail, which will be given in *Chapter 4*, *The Renderer*, we'll just replace this layout with the one copied directly from the basic application template:

```
<?php
use yii\helpers\Html;

\yii\bootstrap\BootstrapAsset::register($this);
\yii\web\YiiAsset::register($this);
?>
<?php $this->beginPage() ?>
    <!DOCTYPE html>
    <html lang="<?= Yii::$app->language ?>">
    <head>
        <meta charset="<?= Yii::$app->charset ?>"/>
        <title><?= Html::encode($this->title) ?></title>
        <?php $this->head() ?>
        <?= Html::csrfMetaTags() ?>
    </head>
    <body>
    <?php $this->beginBody() ?>
    <div class="container">
        <?= $content ?>
        <footer class="footer"><?= Yii::powered();?></footer>
    </div>
    <?php $this->endBody() ?>
    </body>
    </html>
<?php $this->endPage() ?>
```

This is the only time we need to do something without using Gii to satisfy our end-to-end tests, and this is a one-time issue because all of the future sections generated by Gii will use the same layout without any changes.

Except various useful calls such as inserting an HTML language attribute, inserting a metatag with a charset, and HTML encoding the title of the page, pay attention to the highlighted lines. They are the framework of the Yii 2 renderer. We will explore it in *Chapter 4*, *The Renderer*.

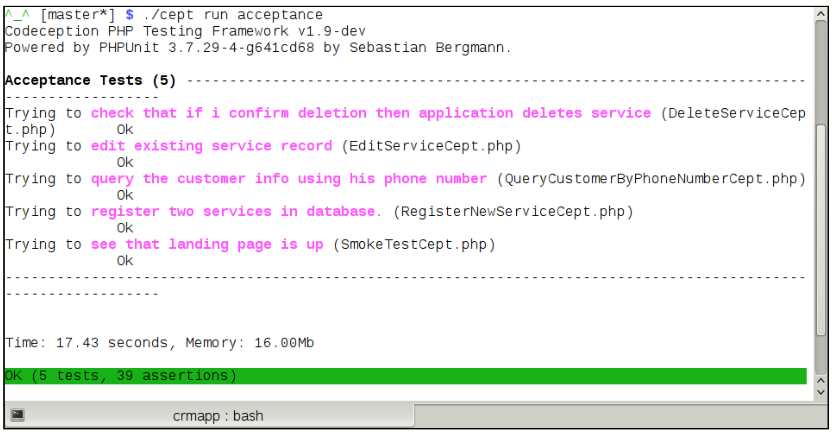
An overview of the generated CRUD UI

Let's see how this UI looks.

Inside the code bundle that comes with this book, you will find the acceptance test definitions for the CRUD UI we have been building in this chapter. This was not included in the text for brevity. Run the following command:

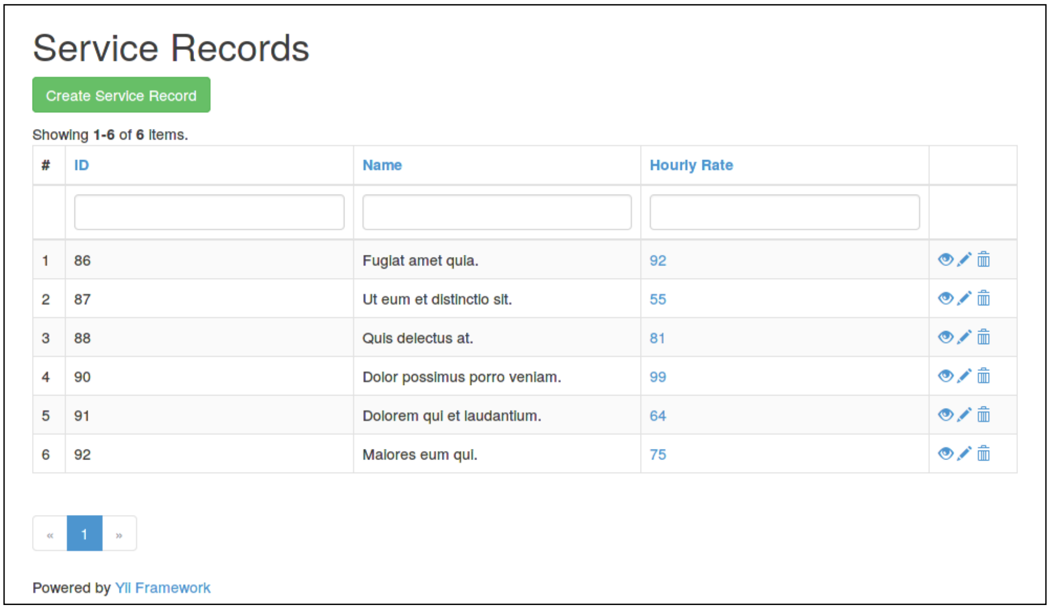
```
$ ./cept run acceptance
```

This command will produce an output as shown in the following screenshot:

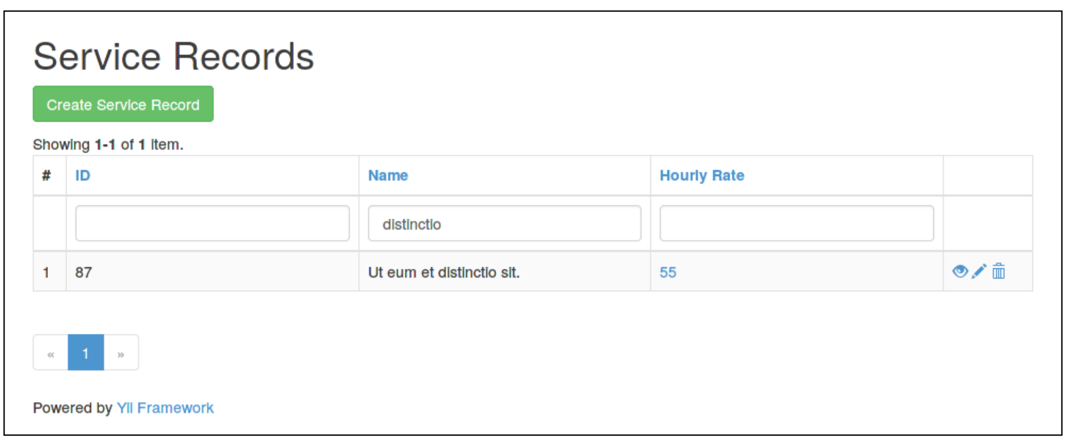


Note that we haven't manually written a single line of production code to make these tests pass (with the exception of the layout). Also, given that all the possible preparations were done already, the next time we need to make such a CRUD UI, we will just need to fill in six fields and click on two buttons, and we'll have it.

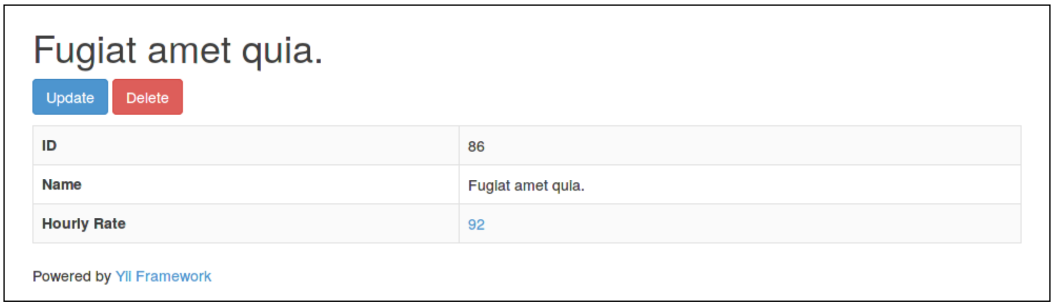
By going to the `/services` route, we get a nice table, filled to the brim with features:



Each record is represented by a row in the table. This tabular UI has pagination with the default page size of 20 lines (this can be a problem if you run acceptance tests too many times without cleaning the database upon deploying or redeploying a target). Also, it has fields at the top of the table to filter rows by the field values, as shown in the following screenshot:

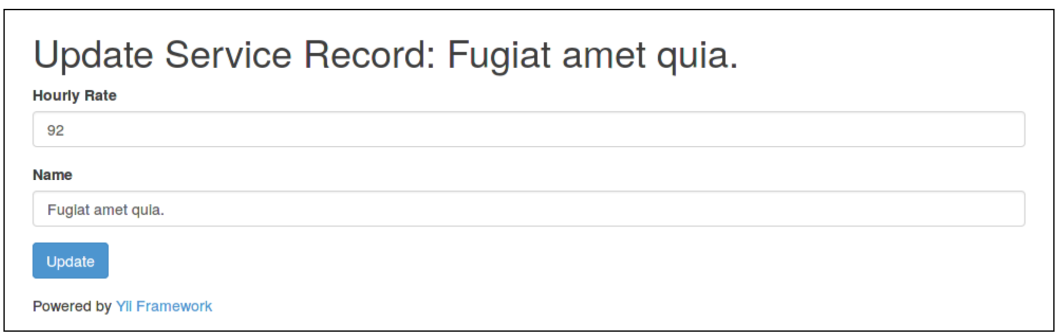


Finally, the last column in each row contains three iconic buttons: one with an eye, one with a pencil, and one with a trash can. They correspond to the view, update, and delete actions, respectively. The view action renders a nice list of fields and their values for the selected record.



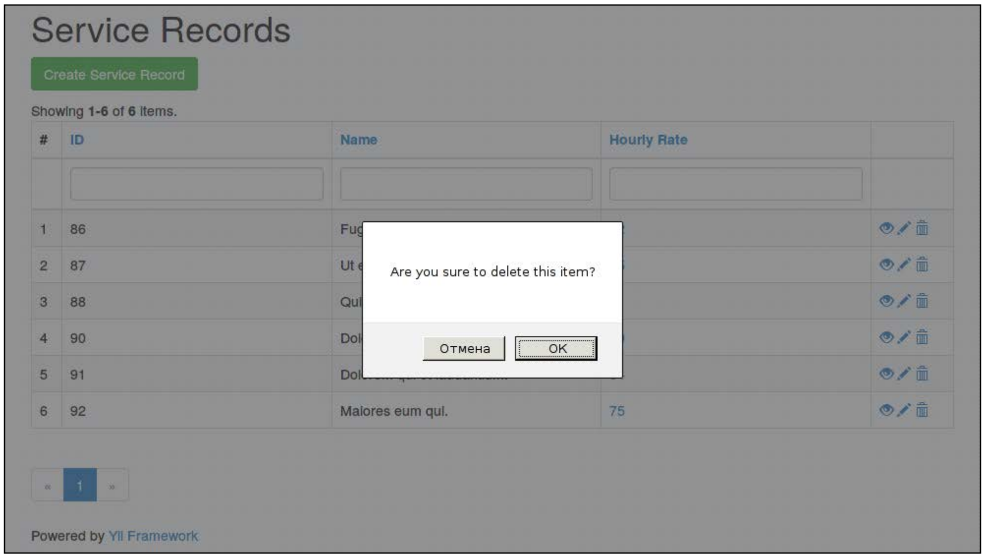
The view route will be `/services/view?id=:id`, where `:id` is the primary key of the `Service` record in the database. In there, you'll see the **Update** and **Delete** buttons too.

If you click on the **Update** button either from the list UI or from the view UI, you get a form to edit the fields of the selected record.



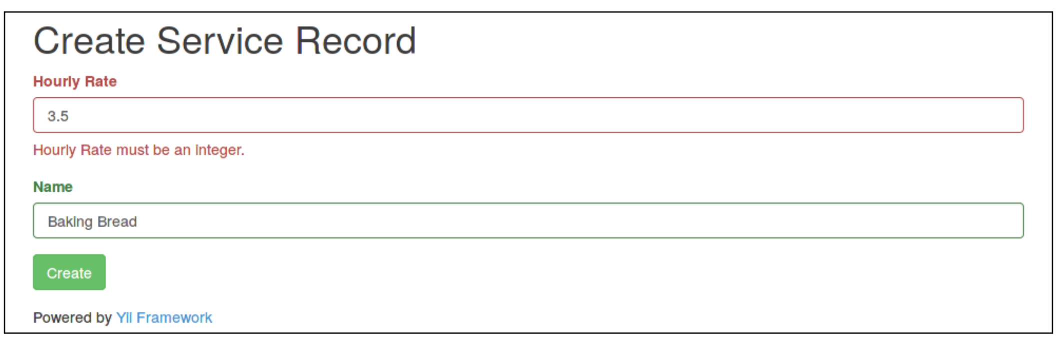
The update route will be `/services/update?id=:id`, with the same rules as that of the view route. This form is absolutely identical to the form where we create a new record, which we will see in the following chapters.

If you click on the **Delete** button either from the list UI or from the view UI, you will get a delete confirmation, which, as we saw earlier in this chapter, is potentially dangerous in nature:



If you accept the deletion, the table will be refilled via AJAX. If you have JavaScript disabled, you'll get no delete behavior at all because `/services/delete?id=:id` is configured to be accessible only by POST requests.

If you click on the big green **Create Service Record** button, you'll get a form to enter data for a new `ServiceRecord` model. The create route will be `/services/create`. This form has built-in client-side validation. It won't allow you to submit the form if there are errors in the input data.



All of this UI is based on the Twitter Bootstrap UI framework plus some Yii-specific JavaScript code.

Pros and cons of generated classes over manually created ones

Like with every automatic generator, Gii should not be treated as an automatic programmer doing all the work for you.

First of all, Gii obscures the mechanics of CRUD. So, when you automatically generate the controller actions, it'll be difficult for you to understand how they work, especially if you are new to the framework. This is a common issue with all automatic code generators; they are useful only for people who already know what code they need to write and just don't want to write it themselves.

Second, it's possible that you will need a set of absolutely standardized uniform CRUD UI pages for many models. In this case, code repetition induced by Gii in the generated controllers will be completely unnecessary. In *Chapter 11*, *The Grid*, we will show you how three controllers generated by Gii can be reduced to one base `Controller` class and three specific controllers trimmed down to virtually *two lines* of custom code each.

To sum up, the automatic generation of a CRUD UI in Gii is especially usable only when you need some generic baseline for your actual work, *and the changes in index, create, edit, and delete actions are expected*. In other cases, maybe it'll be easier to just write controllers from scratch, which is not so hard anyway.

Do note, however, that Gii is not restricted to generating the CRUD UI. Its model generator is virtually priceless, because you certainly do not need to write boilerplate code for your active records, and Gii can infer a lot of information from the table schema for you. Also, it has several other generators, including an Extension generator; we will not be using it in *Chapter 9*, *Making an Extension*, though, because it will hinder your ability to understand how things work inside Yii 2.

Summary

In this chapter, we learned how to integrate Gii into an existing project and wire it to the code base. Also, we discussed the bare minimum of view code that the Yii framework expects from us to have if we are building everything from scratch.

We have used quite a lot of code related to the view part of the MVC ideology in this and the preceding chapters, and the necessity of layouts has been especially obscure so far. In the next chapter, we'll talk about how Yii 2 actually *performs* rendering of a view.

4
The Renderer

Towards the end of the last chapter, we were forced to jump over our heads and touch the view code. This chapter will basically be an explanation of what we really did then.

Despite the name of this chapter, there's no such thing as a separate `Renderer` object in Yii. Being an MVC-based framework, Yii employs an entire set of processes that perform the rendering. These processes are spread through the whole code base.

Anatomy of Yii rendering

When a web application visitor's request is being handled, there are a few processing steps your data goes through before it is sent to the visitor's browser:

1. A controller action is run. It will use the render() method to process a PHP script (sending it some parameters if necessary) and get HTML form from it to send to the client browser. Please note that this action is voluntary, not mandatory. You can have controller actions that do not call the render() method at all.
2. The `render()` method on the View component is called and the `$view` and `$params` arguments are passed to it.
3. The View component figures out the view file to be used with the help of the path alias passed in the `$view` argument.
4. The View component checks whether it has any view renderers associated with the *filename extension* of the view file it has just found.
5. If there's indeed such a view renderer, its `render()` method is called, and the path to the view file, the View component instance, and the original `$params` arguments are passed to it.
6. If there's no such view renderer, the view file is processed using the traditional PHP `require()` method mechanics.

7. If the result of the rendering is not an instance of the `yii\web\Response` class, such an instance is created and the rendering result is passed to it as the `data` attribute.
8. The instance of `Response` looks at the value of its `format` attribute and checks whether it has custom `ResponseFormatterInterface` implementers associated with this value. The value of the `format` attribute can also be one of the built-in formats, which can be handled by the `Response` instance itself.
9. If indeed there is such a response formatter, its `format()` method is called, and the whole `Response` instance is passed to it so that `format()` can properly set the headers and content of the `Response` instance.
10. If the `Response` instance can process the given format itself, it does so by modifying its headers and content.
11. Finally, the headers are sent using the standard PHP `header()` mechanics, and the content is just flushed down the pipe to the client's browser.

The view renderer concept is implemented in Yii 2, unsurprisingly, by the `\yii\base\ViewRenderer` abstract class. The response formatter is implemented by the `\yii\web\ResponseFormatterInterface` interface.

In the most rudimentary case, where you don't have any themes, tweak the view renderers or response formatters, and execute the following in your controller:

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

When the preceding line of code is executed, the following happens:

1. The `index.php` file is checked for its own existence at the following location `<root_directory>/<views_directory>/<controller_id>/`.
2. Inside the View component, the `extract(['dataProvider' => $dataProvider])` action happens. As a result, the value stored in the `dataProvider` key becomes accessible as the `$dataProvider` variable in the current scope.
3. The `require("<root_directory>/<views_directory>/<controller_id>/index.php")` method is run inside the `ob_start()` … `ob_get_clean()` pair, so all of its output is captured as a string.
4. The `Content-Type: text/html; charset=<yii application charset>` header is sent to the client.
5. Finally, the return value of the `render()` method is sent to the client verbatim.

Now, to know the intricate details of how `render()` does its job, we can look into both the documentation and the source code of the `yii\base\Controller.render()` method. What we are really interested in are four simple questions:

- How do we specify the first argument of the `render()` method so that Yii will find it?
- How does Yii determine what layout is to be used and how we can set it explicitly?
- Can we use other template formats as view files?
- Can we send the rendering result to the client in some unusual way? JSON being the most obvious example.

Before we talk about the answers to these questions, an explanation about the concept of Yii application components will be useful.

The Yii application components

Let's take a look at the application initialization happening in the `index.php` entry point file:

```
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
$config = require(__DIR__ . '/../config/web.php');
(new yii\web\Application($config))->run();
```

As per the ideology of Yii, the preceding three lines mean the following:

1. Set up the `Yii` class definition.
2. Create the `yii\web\Application` instance.
3. Create various components declared in the `components` section of `$config` and attach them to the application instance.
4. Next, perform all the other processing of `$config` like setting up the application's own attributes.
5. Attach the loaded `yii\web\Application` instance to the `Yii` class as the `Yii::$app` static variable.

In step 3, the components are created using the following rule: each array inside the `components` section of the application config is transformed into the instance of the class mentioned in the `class` key of the array, and other key/value pairs define the initial values that should be set for the properties of these instances. This rule is recursive. There are some special components for which the application knows the default classes beforehand. These components will be attached to the application as properties whose names are equal to the key/value pairs of the configuration in the `components` section.

For example, consider the following configuration snippet from the `components` section, perfectly fitting our didactic purposes:

```
'log' => [
    'traceLevel' => 3,
    'targets' => [
        [
            'class' => 'yii\log\FileTarget',
            'levels' => ['error', 'warning'],
        ],
    ],
],
```

This configuration means that the following code will be executed during the creation of the application:

```
$log = new yii\log\Logger; // default class name
$log->traceLevel = 3;
$fileTarget = new yii\log\FileTarget;
$fileTarget->levels = ['error', 'warning'];
$log->targets = [$fileTarget];
Yii::$app->log = $log;
```

There is a whole bunch of components that are attached by default to any application. Their IDs, by which they are mentioned in the configuration, are already known, so Yii 2 knows which classes to use for these components even if you don't specify them.

Also, basically, each array that has a key named `class` and a valid class name as the value of this key will be treated as the configuration for the instance of that class.

> These rules are encapsulated in the `Yii::createObject()` invocation. You can always consult the source code for an enthralling adventure into the **Dependency Injection Containers** implementation, which we will skip in this book both because these implementation details are irrelevant to us and to save our sanity overall.

As soon as the `log` component is generated, it is attached to the application as the `log` property. As a result, you'll be able to reach this newly created instance of the `yii\log\Logger` component with the following incantation:

```
Yii::$app->log
```

The logger is not so useful in this case though as it is usually used only indirectly through the `Yii::error()`, `Yii::warning()`, `Yii::info()`, and `Yii::trace()` calls (you probably already know what they do).

Here is a list of the components attached to both `yii\console\Application` and `yii\web\Application` by default, whether you specify the configuration for them or not:

Component ID	**Component class**
`log`	`yii\log\Dispatcher`
`formatter`	`yii\base\Formatter`
`i18n`	`yii\i18n\I18N`
`mailer`	`yii\swiftmailer\Mailer`
`urlManager`	`yii\web\UrlManager`
`view`	`yii\web\View`
`assetManager`	`yii\web\AssetManager`
`security`	`yii\base\Security`

The following components are attached only to the console application:

Component ID	**Component class**
`request`	`yii\console\Request`
`response`	`yii\console\Response`
`errorHandler`	`Yii\console\ErrorHandler`

The following components are attached only to the web application:

Component ID	**Component class**
`request`	`yii\web\Request`
`response`	`yii\web\Response`
`session`	`yii\web\Session`
`user`	`yii\web\User`
`errorHandler`	`yii\web\ErrorHandler`

Note that both the console and web applications have the `response`, `request`, and `errorHandler` components, but they are of different classes.

In case Yii 2 updates itself in some way or another, you can always see the actual list of the built-in components inside the definition of the `Application.coreComponents()` method.

The most important point of this system of components is that nothing prevents you from creating and registering your own component. It is indistinguishable from the built-in components.

The View component

The View component responsible for making the output for the web application clients is one of application components configurable through the application config in the `components.view` key. To learn the complete list of parameters, you can either look at the documentation about the `yii\web\View` class or just take a look at its source code, as the documentation is autogenerated from the comments in the source code anyway.

View, being an application component, allows you to execute the following:

```
Yii::$app->view->render($viewAlias, $params);
```

The `Controller` class has a wrapper method that abstracts the View component a bit:

```
$this->render($viewAlias, $params);
```

However, the controller doesn't just wrap the call to the View component. It also renders the special view file, which in Yii terms is called `layout`, and then passes the view file to be rendered into this `layout` file. Before we talk about layouts, let's first see what the `$viewAlias` argument is.

Algorithm to find the view files

The concept of modules, in some ways being core to Yii as a whole, complicates things a bit. We'll discuss modules later in *Chapter 7, Modules*, where we'll go into more theoretic detail in an attempt to bring you an insight into how Yii is really constructed. Until then, let's pretend we're dealing with a plain application that doesn't have submodules.

An application, as you remember from *Chapter 2, Making a Custom Application with Yii 2*, must have its `basePath` property defined. The base path is the path to the root of the application's code base.

In fact, an application has another property called `viewPath`. It points to the folder in which all of the views of that application should reside. By default, `viewPath` contains a relative path to the folder named `view`. Being relative, this path resolves to the `view` folder under the application's `basePath`.

The `viewPath`, in turn, acts as the base to determine the relative path you mention as the first argument of the `Controller.render($view, $params)` method. This `viewPath` is appended with the ID of the controller we call `render()` on.

That's it! When you call the following `actionIndex()` method (the one we used to search customers when developing the UI), all the settings of the Yii application are set to their default values:

```
class CustomersController
{
    public function actionIndex()
    {
        $this->render('index');
    }
}
```

The following file will be used:

```
Yii::$app->basePath . "/views/customers/index.php"
```

The `customers` part of the path in the preceding code is the controller ID parsed directly from the name of `CustomersController`.

> Yii 2 creates a controller ID in a very simple way. The name of the `Controller` class is stripped off the `Controller` suffix, and then the leftover name in the uppercase is replaced with the dash-separated one.

Previously, we mentioned that the default extension expected from the view file is `php`.

However, fully relative paths are not all that you can use. If you start a path with the "@" symbol, it'll be treated as the Yii path alias.

Path aliases were in Yii 1.x too, but for Yii 2 the syntax has changed greatly. Now, only the first token will be expanded. This works as shown in the following code (no need for a dedicated chapter specifically for path aliases):

```
Yii::setAlias("@token", "some/filesystem/path/to/application");
Yii::$app->view->render("@token/subfolder/view");
```

As a result of the preceding code, the `view.php` file will be searched in the `some/filesystem/path/to/application/subfolder/` location.

Well, of course, the `setAlias()` call is usually done a lot earlier in the application lifetime. Yii 2 is so nice that it defines the five most important alias tokens for you:

- `@webroot` is the absolute path to the directory containing the `index.php` entry point script
- `@web` is the relative path to the directory containing the `index.php` entry point script, relative to the document root defined by the web server
- `@app` points to the directory set in the `basePath` setting of the application instance (this is expected to be the actual application code base root)
- `@runtime` and `@vendor` are set to the `runtime` and `vendor` subfolders of `@app`, respectively

All of these aliases expand *without* the trailing slash.

> To be absolutely precise, `@webroot` is basically `dirname($_SERVER['SCRIPT_FILENAME'])`. On the other hand, `@web` is the path from `$_SERVER['DOCUMENT_ROOT']` till the `index.php` script (Yii also tries very hard to standardize the value of `DOCUMENT_ROOT` between different platforms). As a result, in the default application setup (for example, in the basic application template), `@webroot` equals the document root set in the web server host setup, and `@web` is empty.

You can use aliases when specifying the path to the view file. Almost any setting of the base Yii components, which accepts some sort of path, accepts aliases as well.

Apart from aliases, you can specify the absolute path for the view files. Absolute means not in terms of a filesystem, but in terms of the application code base. If you start the path to the view file with `//` (two forward slashes), Yii will start looking for the view file in the `viewPath` of the application.

When your application grows and you separate it into modules, you can fine-tune the path specifications for view files by starting them with the `/` (single forward slash) symbol. In this case, Yii will start from `viewPath` of the current module. *In case an application lacks any modules, it will work as the only module*, so a single slash is equivalent to two slashes. In fact, in the case of a fully relative view file path, the path expands relative to the current module, and not just from the application's `viewPath`.

The following table is a summary of the view file path specifications:

Specification	Meaning
`$filename`	`Yii::$app->viewPath . '/' . Yii::$app->controller->id . '/' . $filename`
`"@aliasname". $filepath`	`Yii::getAlias("@aliasname") . '/' . $filepath`
`"//".$filepath`	`Yii::$app->viewPath . '/' . $filepath`
`"/".$filepath`	`Yii::$app->controller->module->viewPath . '/' . $filepath`

In all cases mentioned, if you omit the file extension at the end of `$filename` or `$filepath`, `php` will be assumed by default.

Algorithm to search the layout file to be used

The `Controller.render()` method not only renders the view file it was told to render, but places the result of this rendering inside another view file called `Layout`.

An actual implementation of this mechanics is as follows:

```
public function render($view, $params = [])
{
    $output = $this->getView()->render($view, $params, $this);
    $layoutFile = $this->findLayoutFile($this->getView());
    if ($layoutFile !== false) {
        return $this->getView()->renderFile($layoutFile,
['content' => $output], $this);
    } else {
        return $output;
    }
}
```

This code is taken verbatim from the sources of the `yii\base\Controller` class. A method called `getView()` grabs the View component from the current application instance, which is identical to the `Yii::$app->view` incantation. The `View.render()` and `View.renderFile()` methods are virtually identical for our examples (`renderFile()` does not locate the view file explained in detail in the previous section).

From these sources of the `Controller` class, you can infer the following points:

1. `Layout` is just another view file and it's processed as any other view file.
2. `Layout` has the `$content` variable accessible (and no other variable can be passed there, with `$this` being the only exception). It's the text string that contains the result of rendering the view file we really want to render.
3. The controller has to find the `Layout` file in some way. In this way, it *does not depend on the view file alias* that we pass to the `render()` method.

In fact, `Controller` has the `layout` property that holds the path alias to the `Layout` file `render()` has to use. If for some reason you set the layout property to `false`, the layout system will be disabled and the controller will render just the view file requested. You can also achieve these results by directly calling the `\yii\base\View::renderFile()` method. If the layout property is set to `null`, the value of the `layout` property of the parent module will be used. As mentioned in *Chapter 2, Making a Custom Application with Yii 2*, the top-level module, which is a parent to all the other ones, is the application itself.

After the value of the layout property is fetched, and it's neither `null` nor empty nor `false`, the rules summarized in the following table are applied:

Specification	**Meaning**
`"@alias"`	`Yii::getAlias("@alias")`
`"/" + $filepath`	`Yii::$app->layoutPath . '/' . $filepath`
`$filepath`	`$module->layoutPath . '/' . $filepath`

As you can see, there's a `layoutPath` property in each module, which should contain the path to the directory holding the layout files of that module. By default, `layoutPath` contains the folder named `layout` inside the folder pointed by the `viewPath` property. By using path aliases with the `@` symbol, you can point to any folder in the application code base. By starting the `layout` value with `/`, you force Yii to find the layout path specifically inside the `layoutPath` application. In all other cases, as we noted in the preceding table, using the `$module` symbol, the `layoutPath` of the module whose `layout` property we ended up using will be searched.

By default, `Controller.layout` contains `null`. This means that Yii should use the `layout` property of the module containing that controller. Even if you use the system of modules, by default, each module starts with the `layout` property equal to `null` anyway. This means that the application's global `layout` value will be used, which is usually set in the application config. The default value of this property is `main`, which is the `main.php` file under `layoutPath`. If you specify the file extension, the `php` extension will not be appended automatically.

If you don't override any of the default values, don't use modules, and execute the following:

```
class CustomersController
{
    public function actionIndex()
    {
        $this->render('index');
    }
}
```

Then, the following layout file will be used:

```
Yii::$app->basePath . '/views/layouts/main.php'
```

It is an amazingly simple mechanism to have the HTML markup common to many (usually all) pages of your web application.

The internal workings of rendering the view file

While we certainly don't want to go too deep into the internal workings of the View component, Yii forces us to do so by the conventions of structuring the layout files. We will look at the custom renderers in the very next section. For now, let's remember how the typical PHP layout in Yii is structured, as we used it in *Chapter 3*, *Automatically Generating the CRUD Code*. We have used the following methods of the View component there:

Method	Explanation
`beginPage()`	This starts buffering the content that follows it. Everything after this call is buffered until the `endPage()` call is expected to be postprocessed. It also triggers the `View::EVENT_BEGIN_PAGE` event.
`head()`	This marks the location to place the dynamically-registered elements into. These elements include the `<meta>` elements, custom `<link>` elements, and the registered CSS and JavaScript files with the `View::POS_HEAD` positioning.
`beginBody()`	This marks the location to insert JavaScript with the `View::POS_BEGIN` positioning. It also triggers the `View::EVENT_BEGIN_BODY` event.
`endBody()`	This marks the place to insert JavaScript with the `View::POS_END`, `View::POS_READY`, and `View::POS_LOAD` positioning. Asset Bundles register their CSS and JavaScript files here. It also triggers the `View::EVENT_END_BODY` event.
`endPage()`	This triggers the `View::EVENT_END_PAGE` event. It also puts an end to buffering. It *actually* inserts the registered `meta`, `link`, `style`, and `script` elements into the resulting HTML page.

> A call to `View.endPage()` clears everything that was registered in its `View`. So if you're absolutely crazy, you can even start the second sequence of the HTML page rendering afterwards.

The goal of this elaborate structure of the conventional methods is to help you have total control on the semantics of the different `View::POS_*` position specifiers. Moreover, everything around the `beginPage()` ... `endPage()` calls will be left unprocessed by the View component, and due to the cleanup behavior of the `endPage()` call, you can even generate several pages in one run of `Yii::$app->view->render()`.

Custom renderers

With all this, you now probably know the most contrived part of the Yii conventions. Let's now talk about the custom renderers.

Yii uses renderers to process the view file you referenced in the `render()` method, *based on its file extension*. If the View component of Yii is unable to find any renderer for the given view file, it treats this view file as a PHP script, and executes the `renderPhpFile()` method:

```
/**
 * @param string $_file_ the view file.
 * @param array $_params_ the parameters (name-value pairs) that
will be extracted and made available in the view file.
 * @return string the rendering result
 */
public function renderPhpFile($_file_, $_params_ = [])
{
    ob_start();
    ob_implicit_flush(false);
    extract($_params_, EXTR_OVERWRITE);
    require($_file_);
    return ob_get_clean();
}
```

As you can see in the preceding code, Yii will just require the file for buffering the output. An important effect of custom renderers is that you can do anything inside your view files, which will be dangerous if you really *do* anything. You should ideally treat PHP view files as being in some template system's format, only allowing them to paste data passed to them as associative arrays or plain data structures.

Yii 2 does not ship along with any custom renderers. Let's think about how we can utilize this feature for our convenience.

The original design goal of the developers, obviously, was that you'll write the parser for some template system format, and then you'll be able to write view files not as raw PHP, but as these (supposedly, more restricted, and domain-specific) templates. By giving the templates a new format with a distinct extension, you'll force Yii to use your custom parser on them, which will supposedly convert the given template to, say, HTML, which you'll send to the client.

Let's employ a simple solution here, which is using **Markdown** (`http://daringfireball.net/projects/markdown/syntax`) to author the static pages. Let's say we want the set of user-level documentation pages, which will be handcrafted by a skilled tech writer. You probably neither want to force him or her to author pages in HTML nor allow him or her to author them in Word to painfully convert it to HTML yourself. Markdown will be a simple middle ground for this task.

Employing a custom renderer to render static Markdown files is pretty simple. We assume that you have the same CRM application we have been building in the previous two chapters.

First, let's declare what we want:

```
./cept generate:cept acceptance Documentation
```

Then, in the `tests/acceptance/DocumentationCept.php` file just generated, use the following code:

```
$I = new AcceptanceTester\CRMUserSteps($scenario);
$I->wantTo('see whether user documentation is accessible');

$I->amOnPage('/site/docs');
$I->see('Documentation', 'h1');
$I->seeLargeBodyOfText();
```

The `seeLargeBodyOfText()` method will be defined in the `CRMUserSteps` class as follows:

```
    public function seeLargeBodyOfText()
    {
        $I = $this;
        $text = $I->grabTextFrom('p'); // naive selector
        $I->seeContentIsLong($text);
    }
```

We basically assert that we do see a documentation page if there's a heading called **Documentation** and a lengthy body of text below that. It's pretty naïve, of course, but we cannot afford to write an AI code capable of automatically checking whether the given text is indeed a documentation page or not.

We will place the `seeContentIsLong()` method in the `AcceptanceHelper` class inside `tests/_support/AcceptanceHelper.php`:

```
public function seeContentIsLong($content, $trigger_length = 100)
{
    $this->assertGreaterThen($trigger_length, strlen($content));
}
```

We have to do this because we don't have assertions in the `AcceptanceTester` class itself. Don't forget to run `./cept build` afterwards, as we have modified the module of `AcceptanceTester` (yup, `AcceptanceHelper` is technically a Codeception module).

Now, run the test and watch it fail:

```
$ ./cept run tests/acceptance/DocumentationCept.php


1) Failed to see whether user documentation is accessible in
DocumentationCept.php
Sorry, I couldn't see "Documentation","h1":
Failed asserting that any element by 'h1' on page /site/docs
Elements:
+ <h1> Not Found (#404)
contains text 'Documentation'


Scenario Steps:
2. I see "Documentation","h1"
1. I am on page "/site/docs"
```

Of course, we don't have the `/site/docs` route handler yet. Create the `SiteController.actionDocs` method:

```
public function actionDocs()
{
    return $this->render('docindex.md');
}
```

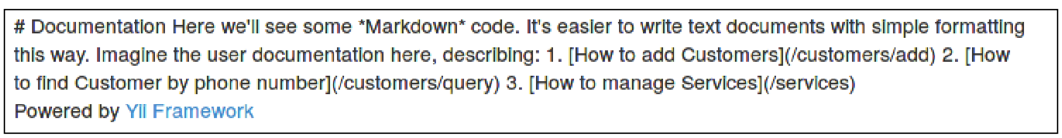
Note the absence of any Markdown-handling code, which is the whole point of using a custom renderer.

Now, the view file at `views/site/docindex.md` should look as follows:

```
# Documentation

Here we'll see some *Markdown* code.
It's easier to write text documents with simple formatting this way.

Imagine the user documentation here, describing:

1. [How to add Customers](/customers/add)
2. [How to find Customer by phone number](/customers/query)
3. [How to manage Services](/services)
```

The preceding code is a perfectly valid page in the Markdown format.

If you access the `/site/docs` page now, you'll see that the text is rendered verbatim, because `docindex.md` is being processed as the PHP script, and as it does not contain the `<?php` processing instruction, it is parsed as HTML text. Here is the screenshot:


Documentation Here we'll see some *Markdown* code. It's easier to write text documents with simple formatting this way. Imagine the user documentation here, describing: 1. [How to add Customers](/customers/add) 2. [How to find Customer by phone number](/customers/query) 3. [How to manage Services](/services)
Powered by Yii Framework

Finally, let's write the custom renderer itself. As this class is a part of the application infrastructure and has nothing to do with the domain model or the route handling, let's create a separate `utilities` subdirectory and namespace for it. So, in a `utilities/MarkdownRenderer.php` file, write the following:

```
<?php
namespace app\utilities;

use yii\helpers\Markdown;
use yii\base\ViewRenderer;

class MarkdownRenderer extends ViewRenderer
{
    public function render($view, $file, $params)
    {
        // TODO
    }
}
```

The `app\utilities` namespace is automatically mapped to the `utilities` directory under the root of the code base, thanks to the Yii 2 autoloader we require in the `index.php` entry script.

You may ask, how do we render the Markdown? We can render the Markdown as follows:

```
public function render($view, $file, $params)
{
    return Markdown::process(file_get_contents($file));
}
```

Yii 2 includes the whole Markdown processor as one of its dependencies.

Be cautious, though, because just calling the raw `file_get_contents()` method is pretty unsafe. We rely on `$file` being safely constructed by the Yii internals here.

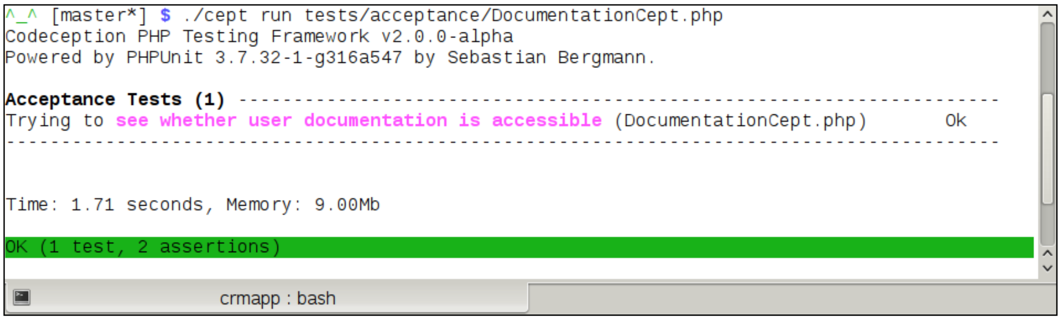
> The `MarkdownRenderer` class does not have the proper unit tests because arguably all we did was wrap two built-in and already extensively tested functions into one call. To simplify the discussion, we omitted the proper test-driving `MarkdownRenderer`. Note, however, that in more complex cases, you must not do this. It is not all the time that you have a parser as simple as the one presented here.

With `MarkdownRenderer` written, we have to wire it to the application. Add the description of our custom renderer to the `components.view.renderers.md` section of the config at `config/web.php`:

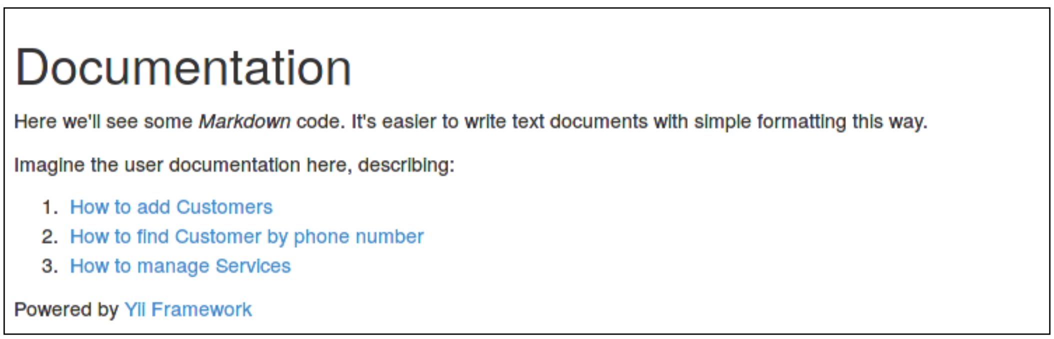
```
'components' => [
    'view' => [
        'renderers' => [
            'md' => [
                'class' => 'app\utilities\MarkdownRenderer'
            ]
        ]
    ]
]
```

The index in the `renderers` array is the filename extension. In our case, it should be `md`. Our renderer does not have any properties, so to properly reference it in the application configuration, we just need to provide the fully qualified name of its class.

Now run the tests. They will be executed successfully, as shown in the following screenshot:



At the `/site/docs` location, you can see the properly formatted HTML page now:



A custom response formatter

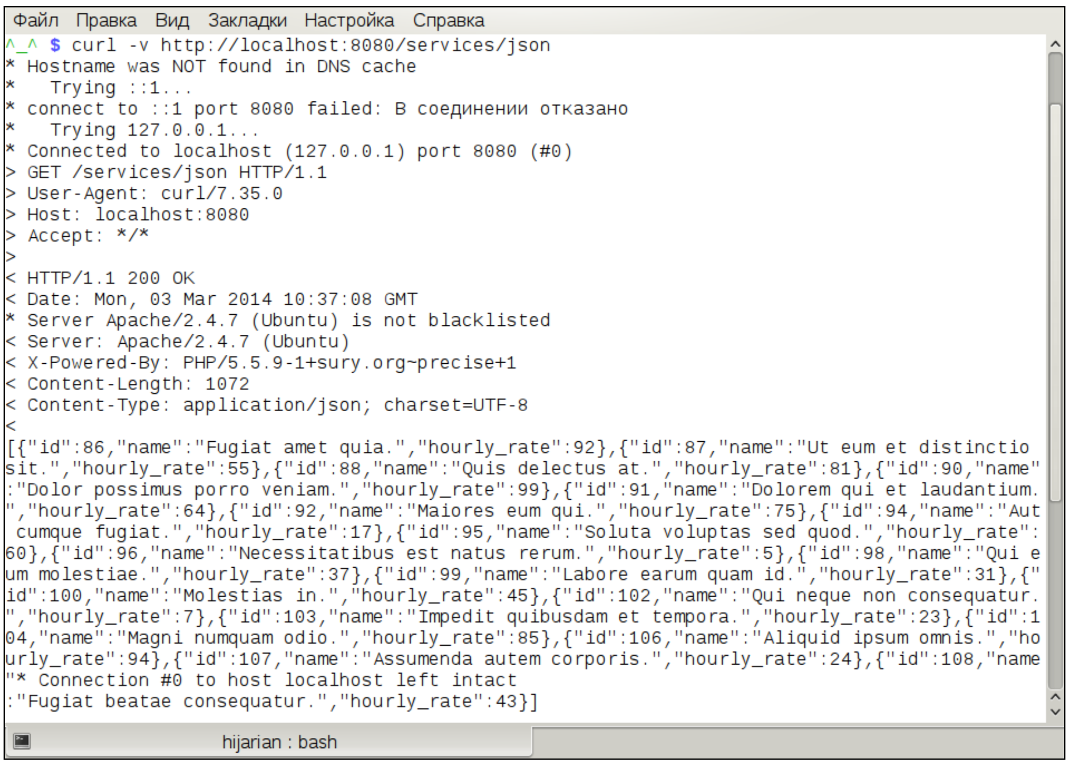
As was mentioned at the beginning of this chapter, the last stage of processing the data before sending it to the client is passing it through the response formatter. Anything that the controller action returned is wrapped into the `Response` object, which decides how to ultimately send the data down the pipe. Let's see how we can utilize a custom response formatter for our purposes.

Probably the most obvious use of the custom response formatter is returning the JSON data for a given route. The following is a throwaway snippet of the exploratory code, which returns the list of attributes of the registered services from the database:

```
public function actionJson()
{
    $models = ServiceRecord::find()->all();
    $data = array_map(function ($model) {return $model-
>attributes;}, $models);
```

```
        $response = Yii::$app->response;
        $response->format = Response::FORMAT_JSON;
        $response->data = $data;

        return $response;
    }
```

In the following screenshot, the result of talking to the `/services/json` route from the command line is presented. Note that the correct content-type HTTP header was set by the server.



First, we have to note that we do not render anything here. The data we get, an associative array, is wrapped into a manually crafted `Response` instance and returned from the controller action and added to the `Response.data` field. When we return the string generated by the `Controller.render()` method, Yii wraps it in the `Response` instance behind the curtains for us. In fact, everything that is not `Response` descendant and is being returned from the controller action will be added to the `Response.data` field automatically.

Secondly, we do not create the `Response` object ourselves, but we get the reference to it as a Yii component instead. This way, we get the default values of its properties already set by Yii at the application initialization step. The `Response` object is used exactly once in an application's lifetime, so there's really no difference whether we'll get it as a Yii component or by calling `__construct()`.

The `format` field tells the `Response` object how to format the outgoing data. At the time of writing, Yii developers have the following types built-in, and hence don't need to re-implement them:

Literal	**Effect**
`Response::FORMAT_HTML`	This is the default literal. `Content-Type` will be set to `text/html`. No processing will be done on the data, except that objects will be serialized using the `__toString()` call.
`Response::FORMAT_RAW`	Data will be returned without any processing, except that objects will be serialized using the `__toString()` call. `Content-Type` will not be set by Yii.
`Response::FORMAT_JSON`	Data will be processed by the `Json::encode()` method shipped with the Yii framework. `Content-Type` will be set to `application/json`.
`Response::FORMAT_JSONP`	Data must be an array with the `callback` string element and the `data` element. The response is the string `callback(data)`, where data is processed by `Json::encode()`. This literal basically implements a strict JSONP recommendation (see `http://json-p.org/`). `Content-Type` will be set to `text/javascript`.
`Response::FORMAT_XML`	Data will be processed by the `XmlResponseFormatter` class. In short, this means that the well-formed XML string has a response, and the `Content-Type` header is set to `application/xml`. Data is expected to be either an associative array or an object with public fields.

In each case when the `Content-Type` header is set, `charset` will be set to the value of the `charset` property of the `Response` instance else, to the value of the `charset` property of the application. The only exception here is JSON, where `charset` will always be UTF-8 by specification.

Let's do something crazy and serialize the data about services not to JSON but to **YAML** (see `http://www.yaml.org/spec/1.2/spec.html`). One of Codeception's dependencies is the YAML library from the **Symfony2** project, so we'll just utilize it instead of writing our own serializer.

Create the `YamlResponseFormatter` class inside the `utilities` directory with the following content:

```
<?php
namespace app\utilities;

use Symfony\Component\Yaml\Yaml;
use yii\web\ResponseFormatterInterface;

class YamlResponseFormatter implements ResponseFormatterInterface
{
    const FORMAT = 'yaml';

    public function format($response)
    {
        $response->headers->set('Content-Type: application/yaml');
        $response->headers->set('Content-Disposition: inline');
        $response->content = Yaml::dump($response->data);
    }
}
```

Note the highlighted parts. We are using the `Yaml` class from the `Symfony` library, and we have to implement the `ResponseFormatterInterface`.

A class constant named `FORMAT` is just convenient so when we set the format of the `Response` instance, we'll be more descriptive.

The implementation of the `format()` method is pretty straightforward, thanks to the intuitive `Yaml::dump()` method. The idea is that we need to set the `headers` and `content` fields of the `Response` instance and not return anything from this method.

The YAML format does not have a **Multipurpose Internet Mail Extensions** (**MIME**) type registered (check here: `http://www.iana.org/assignments/media-types/media-types.xhtml`), so we have arbitrarily decided to use `application/yaml` to stress the fact that YAML format a serialization format intended to be read by some program.

To wire this formatter to the `Response` component, we need to add the formatter's declaration to the `components.response.formatters` item during the application configuration, as follows:

```
    'components' => [
        'response' => [
            'formatters' => [
                'yaml' => [
                    'class' => 'app\utilities\YamlResponseFormatter'
                ]
            ]
```

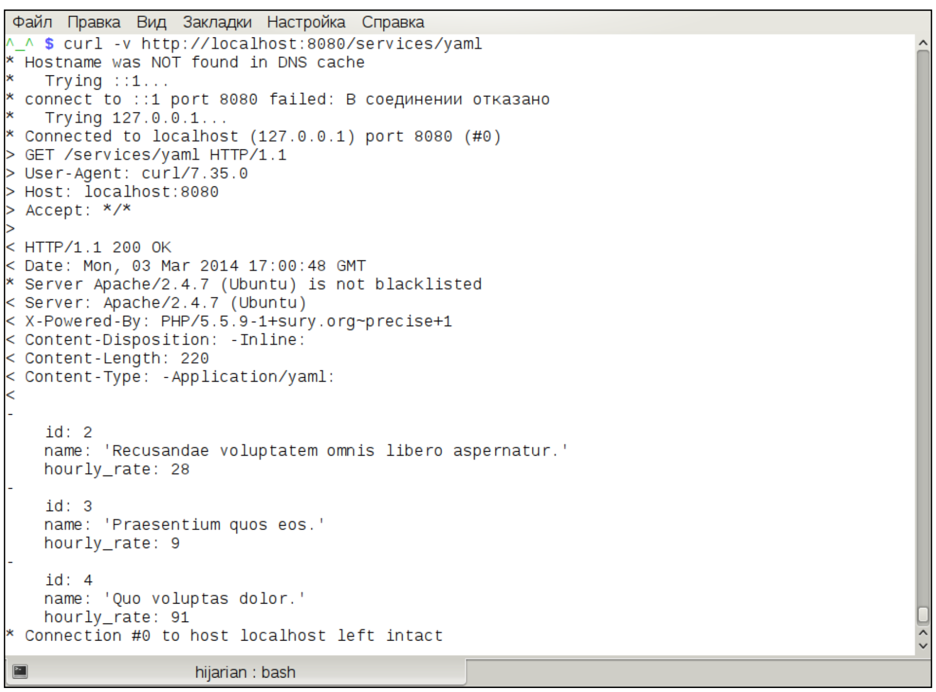
```
            ]
        ]
```

The highlighted part is the declaration of `YamlResponseFormatter` to handle the `yaml` format.

Finally, add the action routine to `ServiceController`:

```
        public function actionYaml()
        {
            $models = ServiceRecord::find()->all();
            $data = array_map(function ($model) {return $model-
    >attributes;}, $models);

            $response = Yii::$app->response;
            $response->format = YamlResponseFormatter::FORMAT;
            $response->data = $data;

            return $response;
        }
```

Note that the code is almost identical to the code that formatted data as JSON. This is the whole point of the custom response formatters. An example result of reaching the `/services/yaml` route using CURL is shown in the following screenshot:



This YAML data can be read back to the PHP data structures using the `Yaml::parse()` method, which mirrors the `Yaml::dump()` method. If you open this route in the browser, it'll open the **Save as...** dialog prompting you to save the file, since the `application/yaml` MIME type will not be opened as plain text in the browser. We included the `Content-Disposition: inline` header to force the browser to display the data if it's capable of doing so.

The asset bundles

We already mentioned and used asset bundles in brief back in *Chapter 3, Automatically Generating the CRUD Code*. Let's discuss them in more detail.

The purpose of an asset bundle is to have a group of related CSS and/or JavaScript files lying somewhere in the code base, and to be able to register them in the HTML page with a single PHP call. More than this, asset bundles can depend on other asset bundles, and in this case, a single call to register one top-level "master" asset bundle can result in registering the whole UI for the application. The CSS and JavaScript files are called "assets" here, hence the name of the concept is asset bundle.

An asset bundle with files from an arbitrary folder

The following is a typical asset bundle, which references files from an arbitrary folder in the code base:

```
class YiiAsset extends AssetBundle
{
    public $sourcePath = '@yii/assets';
    public $js = [
        'yii.js',
    ];
    public $depends = [
        'yii\web\JqueryAsset',
    ];
}
```

It's a built-in asset bundle, which is required for any built-in widget of Yii 2 to work. It extends the `AssetBundle` class, and has the following properties defined:

Property name	Meaning
`sourcePath`	It gives the path alias to the folder, which is prepended to all the paths in JS and the CSS properties in the bundle. This is effectively the only arbitrary folder in the code base containing all of the assets we are going to register.
`js`	This is an array of relative paths to the files that will be registered as the `<script src=""></script>` elements in HTML. They are presumably the JavaScript files.
`css`	This is an array of relative paths to the files that will be registered as the `<link rel="stylesheet" href="" />` elements in the resulting HTML file. These files are presumably the CSS files. `YiiAsset` does not define anything in this property, that is, it does not contain any CSS files.
`depends`	This is an array of fully qualified names of classes that should be treated as asset bundles and registered before this asset bundle.

This is, of course, not the complete list of properties. A complete description of the `AssetBundle` class can be found in the Yii 2 source code, in the `yii2/web/AssetBundle.php` file.

What does it mean to "register" a few CSS or JavaScript files? It means to put the corresponding HTML element into the resulting page and fill its `href` or `src` attribute. However, if we store the asset files in some arbitrary folder in the code base, maybe in a place inaccessible by the web server, what URLs do we use?

Asset publishing

To answer the questions just asked, Yii has the `AssetManager` class and the concept of **asset publishing**. When you use `AssetBundle` with the `sourcePath` property defined, the Yii application utilizes the `AssetManager` component to copy the directory specified by `sourcePath` to some directory specified in the `AssetManager.basePath` parameter. By default, `AssetManager.basePath` is equal to `@webroot/assets`, and so in effect, the `AssetBundle` directory will be copied to some web-accessible location. It will be renamed to a unique, timestamp-dependent hash so that `AssetManager` will not unnecessarily republish the `AssetBundle` folder if it has not changed any content at all.

Asset publishing is a very complex part of the Yii framework and, surely, a very important and useful feature. You can get into details of it by reading the official documentation for Yii 2. We will now concentrate on the practical consequences of publishing assets, which are as follows:

- By default, all the assets are being published in the `assets` subfolder of your web root. Usually, you have no reason to change this default setting.
- You can completely erase the contents of this folder, which holds the assets, at any time without any problem except at the time of copying assets again.
- You must not delete this folder because Yii never checks for its existence and doesn't create it.
- The whole `AssetBundle.sourcePath` folder is copied to assets. This means that you can have, say, CSS files in the `css` subfolder that reference the PNG images stored in the `img` subfolder by using relative paths such as ../img/<imagename>. This is the same case for fonts.
- When developing an application, it's meaningless to change files in the `@webroot/assets` directory. You have to change the original files in `AssetBundle.sourcePath` and then republish the files again. Yii 2 tries to decide whether it needs to republish assets by comparing timestamps, but it can fail. So a foolproof way to republish assets is just to purge the `assets` directory so that Yii will be forced to repopulate it.
- There is this configuration setting called `AssetManager.linkAssets`, which you can set in the application configuration. When it is true, `AssetManager` doesn't copy the `assets` folder but makes a symbolic link for it. This does not work on Windows and may have possible security issues. For example, you have to have `FollowSymlinks` enabled inside the `@webroot/assets` folder if you're using Apache. However, on a system where it's not a problem, symlinking assets instead of copying them removes the problem described previously, as `@webroot/assets` will always have the newest "copy" of the CSS and JavaScript files.

You can publish any folder or file from the code base by making the following call:

```
list($dir, $url) = Yii:$app->assetManager->publish($path)
```

- The `$path` variable is the path to the file or directory inside the code base. It is processed by `Yii::getAlias()`, so it can be any path alias understandable by Yii 2.
- The `$dir` variable will hold the complete absolute path, given by the `$path` variable just published.

- The `$url` variable will hold the complete absolute URL to the path (`$path`) just published, so you'll be able to use this URL in your HTML file and be sure that the browser's request to it will really return the published file or directory.

An asset bundle with files from a web-accessible folder

Here is the example `AssetBundle` definition, which references files that already lie in the web-accessible folder:

```
class MyUiAsset extends yii\web\AssetBundle
{
    public $basePath = '@webroot/ui';
    public $baseUrl = '@web/ui';
    public $css = ['main.css'];
    public $js = ['main.js'];
}
```

Dependencies were omitted for brevity.

Here, we see the `basePath` and `baseUrl` properties. The base path is the path alias to the folder that contains the files relevant to this bundle. The base URL is the absolute URL prefix that should be appended to all the file references so that the browser will be able to properly request it. Note that the alias system can also be applied to URLs.

Usually, the base URL and base path are similar as we utilize the ability of a web server to serve files directly using the path in any filesystem. However, in more complex cases, you may have complicated URL rewriting in place or maybe routes that dynamically generate CSS or JavaScript code on the fly. All of these are pretty rare, of course, but Yii provides the capabilities anyway.

The point is that when you use the `sourcePath` property in your asset bundle, Yii first publishes this source folder and modifies the `basePath` and `baseUrl` properties of the asset bundle in question for you. As a side effect, it's pointless to use all three properties at the same time as `sourcePath` will have precedence anyway.

This usage is meaningful when you can tolerate the folders that pre-exist in your web root directory. However, the whole reason to use the asset bundle is to contain the assets relevant to a single library inside a single folder. If you use a lot of asset bundles based on the base path, you will inevitably fill your web root with a lot of different folders from different libraries. And this can become really ugly really fast.

However, nothing will stop you from using the same base path for all the asset bundles (for example `@webroot`) and just referencing different files in each bundle. This will lead to a tangled mess of interrelated files and is a maintenance nightmare waiting to happen.

Thus, asset bundles based on the source path, in spite of the cumbersome change-republish cycle, look like a more maintainable and idiomatic solution.

Registering CSS and JavaScript files manually

Given the system of asset bundles, you probably will never really need to register assets manually, but in case you do, here's a list of methods you can use for it:

Method call	Effect
`registerCss($css, $options)`	It will place the `<style>` HTML element inside the `<head>` element with the value of `$css` as the content and the attributes listed in the optional `$options` array. Ultimately, it makes the `Html::style($css, $options)` call.
`registerCssFile($url, $depends, $options)`	It will place a `<link>` element pointing to `$url` inside the `<head>` element of the resulting HTML file. If the optional `$depends` attribute(s) are defined, then the additional `AssetBundle` is registered with just `$url` as the sole CSS element and the dependencies listed in `$depends`. The optional `$options` attribute is an array of attributes for the `<link>` element. Ultimately, it makes the `Html::cssFile($url, $options)` call.
`registerJs($js, $position)`	It will place the `<script>` HTML element as specified by the `$position` attribute with the value of `$js` as the content. Ultimately, it makes the `Html::script($js, ['type' => 'text/javascript'])` call.
`registerJsFile($url, $depends, $options)`	It will place a `<script>` HTML element as specified by the position that is determined by the `$options['position']` field this time. If the `$depends` attributes are specified, then the additional asset bundle is registered with just the value specified by the `$url` attribute as the sole JS element and the dependencies listed in `$depends`. Ultimately, this method makes the `Html::jsFile($url, $options)` call (the `'position'` key being removed from `$options`).

> Do note that the CSS and JavaScript files are still better than writing the `<script>` and `<link>` elements in HTML layouts or, worse, view files directly. One of the most obvious reasons why they are better is the ability to include only those assets that are relevant to the route currently being processed (that is, to the page being currently rendered).

All these methods are in the `View` class, so you can use them as `$this->registerCss($css)` inside the view files and as `Yii::$app->view->registerCss($css)` anywhere else.

Note that we did not list exact function declarations, only the parts most useful in practice.

For JavaScript files, there's the concept of *positions*. Here's a list of the possible positions for JavaScript files:

- `View::POS_HEAD` places the JavaScript files inside the `<head>` element
- `View::POS_BEGIN` places the JavaScript files at the beginning of the `<body>` element
- `View::POS_END` places the JavaScript files at the end of the `<body>` element
- `View::POS_READY` means the JavaScript code block will be enclosed within `jQuery(document).ready()` and placed at end of the `<body>` element
- `View:: POS_LOAD` means the JavaScript code block will be enclosed within `jQuery(window).load()` and placed at the end of the `<body>` element

> We discard the possibility that you need to include `<script>` somewhere inside the HTML page. Yii will not help you with this misbehavior.

Placing JavaScript in different positions in the asset bundles

There's a really useful trick with asset bundles related to manually registering the JavaScript and CSS files described in the previous section.

By default, when registering an asset bundle, all JavaScript files mentioned in it will be placed at the bottom of the `body` element. However, there's a method to change this position. Unfortunately, you cannot specify the position for each individual JavaScript file, only for the whole asset bundle.

There's a property called `jsOptions` in the `AssetBundle` class. It holds the parameters that will be passed to the `registerJsFile` call as the `$options` argument when each of the JavaScript files in the asset bundle will be registered. So you can add the following line to your `AssetBundle` definition:

```
public $jsOptions = ['position' => View::POS_HEAD];
```

All JavaScript files mentioned in the `js` property will be placed in the `<head>` element of the resulting HTML page instead of at the end of `<body>`.

There is also the `cssOptions` property that can be used to set options for the `registerCssFile` calls for CSS files. Using these options, you can set the `media` property of the resulting `<link>` tag, for example:

```
public $cssOptions = ['media' => 'print,aural,tty'];
```

Making a custom asset bundle for our application

Let's create our own asset bundle so that we can initialize our styles and client-side behavior with just a single call.

Create a directory called `assets` in the root of the code base. Inside it, create a file called `ApplicationUiAssetBundle.php` with the following definition:

```
namespace app\assets;

use yii\web\AssetBundle;

class ApplicationUiAssetBundle extends AssetBundle
{
    public $sourcePath = '@app/assets/ui';
    public $css = [
        'css/main.css'
    ];
    public $js = [
        'js/main.js'
    ];
    public $depends = [
        'yii\bootstrap\BootstrapAsset',
        'yii\web\YiiAsset'
    ];
}
```

Hopefully, this definition completely makes sense. For it to not be a lie, we need to create two asset files in the code base root, which will be empty for now: `assets/ui/css/main.css` and `assets/ui/js/main.js`.

As we declared that our assets depend on `YiiAsset` and `BootstrapAsset`, we can replace the option of registering them with registering our asset bundle. Inside the main layout file, consider the following line:

```
\yii\bootstrap\BootstrapAsset::register($this);
\yii\web\YiiAsset::register($this);
```

We can replace these lines with the following:

```
app\assets\ApplicationUiAssetBundle::register($this);
```

Now we have the foundation to write custom styles for our example application.

Themes

One of the interesting things built into the rendering system of Yii 2 is the support for **themes**. Together with the amazingly versatile asset bundle system, Yii 2 gives another level of control over the appearance of the web application under maintenance.

The definition of a theme from the official documentation and DocBlock in the comments is probably the best explanation of this concept ever. Here it is:

> *A theme is a directory of view and layout files. Each file of the theme overrides corresponding file of an application when rendered. A single application may use multiple themes and each may provide totally different experience. At any time only one theme can be active.*

The crucial part is that a theme is a separate set of view files that can override the existing view files. The data that will be sent to the view files from the controller will be the same for all view files, but the view file from the theme can do something completely different with it. With the support of the asset bundle concept, you can not only rearrange the stuff in the resulting page but also restyle it.

You can read the exact rules of applying a theme in the documentation; they're not worth long descriptions here. Let's look at some sufficiently small examples instead.

Making a custom snowy theme

Let's restyle the home page of our example CRM application in such a way that it'll have snowflakes in background and an enlarged title "Our CRM" on it. There's no real meaning behind the snowflakes (yet). Right now, our `SiteController.actionIndex()` method looks like this:

```
public function actionIndex()
{
    return 'Our CRM';
}
```

So, it doesn't really render any view file and, as a result, the layout is not applied. Change the `actionIndex()` method to:

```
public function actionIndex()
{
    return $this->render('homepage');
}
```

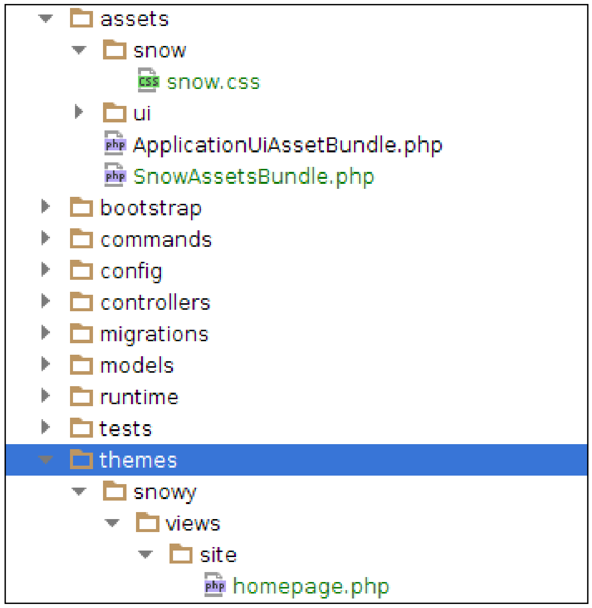
Then, make the `views/site/homepage.php` file with the letters `Our CRM` as the only content inside it. Nothing else is needed here.

All acceptance tests should still pass after this change.

As the theme is a directory with view files, and those view files override the view files that already exist in your code base, let's create a `themes/snowy/views/site/homepage.php` view file.

We'll also use a separate asset bundle for this theme, so create the directory for it, named `assets/snow`. Also, create a `SnowAssetsBundle.php` class file in the `assets` directory and a `snow.css` CSS file in the `assets/snow` directory.

The resulting directory tree should look like the following (files highlighted in green are to be created):



The snow assets bundle will have the obvious declaration:

```
class SnowAssetsBundle extends AssetBundle
{
    public $sourcePath = '@app/assets/snow';
    public $css = ['snow.css'];
    public $depends = ['app\\assets\\ApplicationUiAssetBundle'];
}
```

It depends on the main assets bundle because we want it to override the default styles, that is, be loaded last.

CSS is really simple; we just apply a background to the body and add a class to make some text stand out:

```
body {
    background: #a6e1ec url(snow.jpg) repeat;
}
.inside-snowflakes {
    margin: 10% 15%;
    font-size: 2em;
}
```

The image of the snowflakes is inside the code bundle, which can be downloaded from the Packt Publishing website and is courtesy of Jordan Lloyd. The URL is `http://www.flickr.com/photos/jordanlloyd/5342749399/in/photostream/`.

After this, we apply the new assets bundle and wrap the text inside the newly styled container in `homepage.php`, as follows:

```
<?php app\assets\SnowAssetsBundle::register($this); ?>
<p class="inside-snowflakes">Our CRM</p>
```

So, the preparations are done; there's a theme contained in a folder and a new asset bundle is referenced from this theme. Now to apply the newly created theme to the site, we need to add the following declaration to the `components.view.theme` application configuration key:

```
    'components' => [
    ...
        'view' => [
            ...
            'theme' => [
                'class' => yii\base\Theme::className(),
                'basePath' => '@app/themes/snowy',
            ]
        ...

],
```

Here is the end result that you should get:

It's a bit silly but illustrates the concept well. The real power of themes comes with their dynamic loading based on some condition, which ideally should be done during application loading. The first option is to create a subclass of `yii\web\Application` with the appropriate override of the `init()` method and instantiate it inside the `index.php` entry point script. The second option is to use the event system and add a hook to the `yii\base\Application::EVENT_BEFORE_REQUEST` event. We will discuss more on events in *Chapter 10*, *Events and Behaviors*.

Widgets

In the user interface of your web application, you will often have some parts that should be placed on several pages but cannot be simply placed in the layout file, for example, because they should be filled with the data generated by the route being processed. Yii 2 encapsulates this into the concept of a widget, which is essentially a way to render a separate view file with some arbitrary logic attached to it.

Widgets can be very specific, such as some custom-crafted shorthand widget to render a specific button in our user interface, or very abstract, such as the `\yii\widgets\ListView`, widget which encapsulates the activity of rendering a view file repeatedly given an array of data in a list-like fashion.

While it seems like this chapter is the appropriate place to discuss widgets in detail, it's better to wait until *Chapter 11*, *The Grid*, when we will have a chance to actually *use* a widget and see how it can be configured. So, if you are interested, skip to the *FEATURE – widgets* section of that chapter right now.

Summary

The resulting HTML page is being generated by Yii in three steps.

First, Yii prepares all of the HTML code you provide to it in the view files.

Second, it "registers" the CSS and JavaScript files you tell it to register either by means of asset bundles or with manual calls to the `register*()` methods.

Finally, all the registered assets are being put inside the HTML page to be sent to the client.

The second step that decouples adding assets and rendering the page allows us to do really interesting things such as conditional loading of CSS and/or JavaScript files.

We learned about application components, a core feature of Yii 2, and a lot of features that allow us to control and customize the process of rendering the resulting page, that is, asset bundles, view renderers, response formatters, themes, and the workings of the view itself in general.

However, we skipped several themes here related to the rendering process. The first is the concept of widgets and the list of built-in widgets in particular. We'll use a lot of widgets throughout this book so you'll definitely get an understanding about how they work, and a lot of them will be used in other chapters when the appropriate topic arises. Also, we did not cover the various helper methods of the View component that deal with the caching of content. This will be revealed in *Chapter 8*, *Overall Behavior*.

Now, we'll digress from the fundamental things a bit and go more practical in the next two chapters when we'll add the authentication and authorization mechanics to our application. No more silly simple examples.

5
User Authentication

In the previous chapter, we discussed in detail the precise mechanics of how Yii 2 renders anything for the client browser. In the following chapters, we will look at features we have not used before.

Let's talk about user authentication now. Our CRM application example in its current state is pretty useless. It allows everyone to access it, and more often than not we don't want just anyone to be able to fiddle with the personal data of our customers.

In this chapter, we'll look at what Yii offers us to help identify the user, that is, user authentication. In the next chapter, we'll answer the question of user authorization, that is, deciding whether to allow a user to perform an action in the application.

We'll add the following features to our example CRM application in this chapter:

- A table to record users known to the system and the corresponding user interface to manage it.
- An indicator visible on all pages displaying whether the person currently using the application is registered in the table. However, he/she will need to explicitly declare it to "log in to" the application.

At this point, you already understand what Yii Application Component is, especially, how to configure them and how to access them. If you don't, then it's better that you re-read the previous chapter and the official documentation about this concept, as we'll use more and more built-in Yii Components from now on.

Anatomy of the user login in Yii

You need the following to successfully authenticate the user using Yii:

1. An object implementing the `IdentityInterface` interface.
2. Call `Yii::$app->user->login()` and pass this object there.

The main catch here is the concept of Identity. To Yii 2, this can be any class implementing `IdentityInterface`. You are encouraged to read through the definition of `IdentityInterface` in the `web/IdentityInterface.php` file inside the Yii 2 framework code base, as it's very thoroughly explained there.

When a user is not logged in (no successful call to `yii\web\User.login()` was performed), the property `Yii::$app->user->isGuest` will return `true`. After successful login and as long as the user stays logged in, `Yii::$app->user->isGuest` will always return `false` and `Yii::$app->user->identity` will return the object that was passed to the call to the `login()` method (that is, the user identity).

This basically concludes the high-level usage of the authentication system built-in to Yii 2.

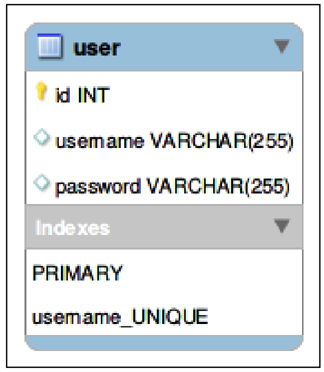
Yii only manages the state of the authentication, that is, it holds the data identifying the user. Any check regarding user authentication should be performed by the application.

While nowadays user authentication by login ID / password pair is slowly becoming old-fashioned, chances are, you'll still need to implement it in your next application. Let's consider the following use case.

Password-based login mechanics in general

Everyone knows the decades-old scheme of password-based login. Records about users known to the application are stored in the database table called `users`. Each record has the `username` and `password` fields (the actual information tokens necessary for authentication) along with other information about the user.

The basic table looks like this in the MySQL Workbench (`http://www.mysql.com/products/workbench/`):



The `username` field is used to actually identify the user among others. Therefore, it has the `UNIQUE` constraint.

The `password` field contains a special string known, by definition, only to the user and the application. To further protect the password, the application does not store it as plain text, but as its *cryptographically secure* hash value, which is generated by an algorithm such as `bcrypt`. In this way, the application has no way to know the original plaintext version of password.

> The most important part of password storage is the *cryptographically secure* part. So, the algorithm of hashing should exhibit the following traits useful for our purposes:
>
> - It should use the *cryptographic hash function*, the main characteristic of which is being *one-way*, that is, you can compute hash from plaintext easily but getting plaintext from hash is very hard.
> - It should ideally use a different hashing function for each given plaintext. This is done by the technique called **salting**.
> - It should be *slow*.

When authenticating, a user provides the username to claim who he is and the password to prove it. As the application knows which algorithm it used to hash the password when the user was registering, it hashes the provided password in the same way and compares the resulting string with what it has in the database for the provided username. If hashes are found identical, then the user is really who he claims to be and we consider him authenticated.

Well, you probably know that already. Let's implement this authentication method in our example CRM application.

> Note that username/password authentication is not the only way to authenticate the user. There are other methods, varying both in complexity of security checks and the user interface. To name a few, there is the OpenID initiative (`http://openid.net/`) and the OAuth protocol (`http://oauth.net/2/`) to authenticate in an application using credentials of another application, and signed and confirmed SSL certificates (`https://developer.mozilla.org/ru/docs/Introduction_to_SSL#_Client_Authentication_`).

We will ultimately perform the following steps:

1. Build the user records management interface.
2. Build the user login form.
3. Introduce the indicator to display user status (logged in/logged out).

Making the user management interface

There is no need for us to make some custom user interface. Also, for simplicity, we will not bother with separating from ORM as we did in *Chapter 2*, *Making a Custom Application with Yii 2*. Thus, we can live with the interface automatically generated by Gii as described back in *Chapter 3*, *Automatically Generating the CRUD Code*. We will not list here the exact same steps as were described already, but will present a short outline before implementing features that are specific for users.

Acceptance tests for the user management interface

Similar to the services back in *Chapter 3*, *Automatically Generating the CRUD Code*, we need to be able to create, update, view, and delete users in the system.

> As the acceptance tests for user management are going to be almost exactly the same as for services management, if you just follow the lessons in this book, you don't need to implement them. However, strictly speaking, in a real-world production environment you do need to implement them anyway. They are included in the code bundle relevant to this chapter.

The easiest way right now is to copy `RegisterNewServiceCept.php`, `EditServiceCept.php`, and `DeleteServiceCept.php` from the `tests/acceptance` directory to the `RegisterNewUserCept.php`, `EditUserCept.php`, and `DeleteUserCept.php` files in the same directory, respectively, and then replace all mentions of the following in method names, variable names, and strings:

- services to users
- Services to Users

By doing this, we get the reference to a nonexistent class `AcceptanceTester\CRMUsersManagementSteps`, which we create by copying the file `tests/acceptance/_steps/CRMServicesManagementSteps.php` to `tests/acceptance/_steps/CRMUsersManagementSteps.php` and replacing all mentions of `Services` by `Users`.

> Of course, it's blatant code duplication and it really needs to be refactored out, but you can take it as the home assignment, because you really can't escape at least some duplication in your highest level feature specifications for manipulating with entities in application, and it's outside the scope of this book to teach proper refactoring techniques. Just remember that you absolutely *MUST* get rid of this duplication.

The only functional difference in the specs for user management will be inside the `CRMUsersManagementSteps` class. After the proposed replacements, it mentions the `UserRecord[name]` string several times now. Our user record will not have the `name` field, but the `username` one. Then, you'll need to change `UserRecord[name]` to `UserRecord[username]`. We'll stick with the `username` field name because it's not the actual *name* of the real-world person behind this `UserRecord`, it's just the identifier it presents to our application.

Also, the `imagineUser()` method in the same class becomes as follows:

```
function imagineUser()
{
    $faker = \Faker\Factory::create();
    return [
        'UserRecord[username]' => $faker->userName,
        'UserRecord[password]' => md5(time())
    ];
}
```

We will not bother with password hashes, as all data here is the user input, not what will be stored in the database. Note the way we generate a random password, although this way we can get only alphanumeric characters. We are doing this just for simplicity, as the password ideally should be a string containing *any possible* characters, including non-printing ones.

The PHP built-in function `md5()` was chosen only as an easy way to get a long stream of pseudorandom alphanumeric characters. You should not use a fast hashing algorithm to get a password hash. See this excellent answer if you're still not convinced: `http://security.stackexchange.com/a/31846`.

Database table to store user records

We need to prepare the table to hold our future user records. Here's the database migration you'll need to implement the schema displayed before:

```
public function up()
{
    $this->createTable(
        'user',
        [
            'id' => 'pk',
            'username' => 'string UNIQUE',
            'password' => 'string'
        ]
    );
}

public function down()
{
    $this->dropTable('user');
}
```

Generating the model and CRUD code by Gii

Perform the same steps discussed in *Chapter 3*, *Automatically Generating the CRUD Code*. We will store the UserRecord model into the separate namespace `app\models\user`.

Field name	**Field value**
Model class	`app\models\user\UserRecord`
Search Model class	`app\models\user\UserSearchModel`
Controller class	`app\models\user\UsersController`

Removing the password field from the autogenerated code

We will handle the `password` field automatically behind the scenes, but Gii doesn't know our intentions. You need to remove mentions of this field manually from the following places. Please note that we don't want to see passwords for any user records.

- `models/user/UserSearchModel.php`: First, remove the `password` field from the `safe` rule inside the `rules()` method. Then, remove the following line from the `search()` method:

  ```
  $query->andFilterWhere(['like', 'password',
    $this->password])
  ```

- `views/user/_search.php`: Remove the following line from the `ActiveForm` widget configuration:

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

- `views/user/index.php`: Remove the `password` field from the `columns` setting inside the `GridView` widget configuration.
- `views/user/view.php`: Remove the `password` field from the `attributes` setting inside the `DetailView` widget configuration.

However, we *do* want to be able to enter and change the password for user records, so the input field for password in the `views/user/_form.php` view file should stay intact.

Now the interesting part begins: password should be hashed upon saving.

Hashing a password upon saving a user record

What do we lack in our current scheme? Obviously, *before saving* a user record in the database, we need to compute the secure hash of the password provided and store the hashed value instead of the plaintext one. More than that, when updating the user record, we don't want to rehash the already hashed value in case we don't change the password at all.

Yii 2 (as well its predecessor, Yii 1.1.x) defines several methods for the instances of the `ActiveRecord` class, which we can override to do stuff at several predefined stages of the active record life. While you can read about them all in more detail in the *Events of \yii\db\BaseActiveRecord* section of *Chapter 10*, *Events and Behaviors*, we will be using one of them right now, the `beforeSave()` method. We will use it right before the active record is to be saved to the database.

It has the following default definition:

```
public function beforeSave($insert) { return true; }
```

The `$insert` argument passed to this method indicates whether the active record in question is a new one that is to be inserted into the database or if it's an already existing one that is to be updated into the database.

This method must return a Boolean value indicating whether this record is allowed to be saved. If not, it will not be saved, without any additional indication to the user or program. It's really important to remember this fact. If you don't return anything from this method, then a `null` will be implicitly returned, which is equal to `false` and you will effectively prohibit the relevant operation completely. We will be overriding this method in the `UserRecord` class for our purposes.

Additionally, Yii 2 has the special `\yii\base\Security` application component that holds, among others, the helper methods for securely computing hashes for passwords. This component is accessible through the `Yii::$app->security` invocation. This is of great help to us, as we become relieved from the task of manually fiddling with the specifics of PHP's `crypt()` method. We are particularly interested in the `Security::generatePasswordHash()` and `Security::validatePassword()` methods.

Functional tests for password hashing

How do we check that a password is stored in the hashed form in the database? After saving a user record, we will simply check that a call to the `Security::validatePassword()` method returns `true` for a given plaintext and hash.

This feature doesn't need to be tested by the end-to-end acceptance test. We will be using the functional testing suite for it, which is already included by the Codeception framework for us since *Chapter 2*, *Making a Custom Application with Yii 2*, but is unused until now.

For functional (or, in different words, integration) testing in Codeception, a special module called `Db` exists, documentation for which can be found at `http://codeception.com/docs/modules/Db`. Its most important feature for us is the fact that before each test run it reverts the database to the state described in the `tests/_data/dump.sql` file, which does not exist yet. This SQL file should contain instructions about how to create the database schema. With the default toolset of MySQL 5 it can be generated as follows, given that your database doesn't require credentials and is named `crmapp`:

```
$ mysqldump -d crmapp > tests/_data/dump.sql
```

The `-d` flag means *no data, only schema* .

We need to configure the functional test suite in the `tests/functional.suite.yml` file like this (lines to be inserted are highlighted):

```
class_name: FunctionalTester
modules:
    enabled: [Db, Filesystem, FunctionalHelper]
    config:
      Db:
        dsn: 'mysql:host=localhost;dbname=crmapp'
        user: 'root'
        password: 'mysqlroot'
        dump: tests/_data/dump.sql
```

Of course, you need to insert your own hostname, database name, username, and password for a database on the deploy target. You also need to rebuild the Codeception suite using the `./cept build` invocation.

So, generate the new functional test, as we will be testing the database:

```
$ ./cept generate:test functional PasswordHashing
```

Here's how we encode this test in the generated `tests/functional/PasswordHashingTest.php` file:

```
    /** @test */
    public function PasswordIsHashedWhenSavingUser()
    {
        $user = $this->imagineUserRecord();

        $plaintext_password = $user->password; //1

        $user->save();

        // Don't care about mutated model now, just fetch new one.
```

```
        $saved_user = UserRecord::findOne($user->id); //2

        $security = new \yii\base\Security();
        $this->assertInstanceOf(get_class($user), $saved_user);
        $this->assertTrue(
            $security->validatePassword( // 3
                $plaintext_password,
                $saved_user->password
            )
        );
    }
```

The entire idea is expressed in highlighted lines of the preceding code:

- At 1, the user is not stored in the database yet, and the `password` field still holds the plaintext value, which we save for future reference.
- At 2, after the user is saved to the database it is assigned an ID. To imitate the temporal gap between creating a new user record and logging in, we fetch this user again from the database.
- At 3, we test whether our initially generated plaintext version of the password can be successfully validated against the now hashed value, which is stored in the `password` field of the `UserRecord` class fetched from the database.

We don't really care here how exactly the password is hashed. All we want to know is whether `validatePassword()` will return `true` for the given plaintext and hash value.

We are imagining the user record the same way as in our acceptance tests for user management:

```
    private function imagineUserRecord()
    {
        $faker = Faker\Factory::create();

        $user = new UserRecord();
        $user->username = $faker->word;
        $user->password = md5(time());
        return $user;
    }
```

This is obviously a case of code duplication here, and it should be refactored too.

This test will not run yet, though. If you run it right now, it will say that it cannot find the UserRecord model. This is because you don't have the appropriate autoloaders prepared in the Codeception harness. Fortunately, this harness includes special `bootstrap` scripts, which are executed before all of the tests in the given suite. So, we will utilize the `tests/functional/_bootstrap.php` file and put the `require` calls to our autoloaders in there:

```
require_once(__DIR__ . '/../../vendor/autoload.php'); require_once(__
DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');

new yii\web\Application(
    require(__DIR__ . '/../../config/web.php')
);
```

Unfortunately, we need to create the full application for Yii autoloader to be initialized. We need it to get to our `UserRecord` class.

After the bootstrap file is in place, configuration is appropriately changed and the Codeception harness is rebuilt, the password hashing test should finally fail with the message `Hash is invalid`. It's time to actually implement password hashing.

Password hashing implementation inside the active record

To really implement this feature, you need the now-obvious method inside the `UserRecord` class:

```
    public function beforeSave($insert)
    {
        $return = parent::beforeSave($insert);

        $this->password = Yii::$app->security-
>generatePasswordHash($this->password);

        return $return;
    }
```

Well, turns out, it's not so obvious really. The highlighted part in the preceding code is the line we need. Lines before and after this line are framework-related code to preserve the original behavior of the `beforeSave()` method on the all the `ActiveRecord` instances. To be short, parent implementation of `beforeSave()` uses the concept of Yii 2 called Event. It triggers the particular event telling to whoever is listening in the `ActiveRecord` component that this record is being saved now. The point is that listeners can deny the event. You can read more about events in *Chapter 10*, *Events and Behaviors*, but here we don't really have any use of this system at all.

> Note that in our functional test, we generate the `Security` class instance manually. But in production code, we reuse the application component. In tests, it was done so because it's generally not acceptable that tests are dependent on some global state, which Yii singleton clearly is. In production code, it was done so because it's simply irrational to not use what is already here. However, if for some reason you have some custom-crafted component in your application named `security`, you will end up changing tests.

We now have a problem that was already mentioned before: when we save the `UserRecord` instance again, the password will be hashed again. Now, we have no way to know how this user will login, because he now needs to enter the hash of his original password in the `password` field in login form!

The problem is expressed in the following test:

```
    /** @test */
    public function
PasswordIsNotRehashedAfterUpdatingWithoutChangingPassword()
    {
        $user = $this->imagineUserRecord();
        $user->save();

        /** @var UserRecord $saved_user */
        $saved_user = UserRecord::findOne($user->id);
        $expected_hash = $saved_user->password;

        $saved_user->username = md5(time());
        $saved_user->save();

        /** @var UserRecord $updated_user */
        $updated_user = UserRecord::findOne($saved_user->id);

        $this->assertEquals($expected_hash, $saved_user->password);
        $this->assertEquals($expected_hash, $updated_user->password);
    }
```

The difference between `$saved_user->password` and `$updated_user->password` in the assertions (at the end of the test) is that `$saved_user` at that point in time is the `UserRecord` instance modified in the memory, and `$updated_user` at that point in time is the `UserRecord` instance newly-fetched from the updated record in the database. We need to be sure we don't have mangled data in both cases.

Stateful programming is *hard*, and now we see it ourselves.

Fortunately for us, Yii 2 has a functionality integrated into its active records, which allows us to check whether any field was changed. You can choose among the following options:

Method of ActiveRecord	Usage
`getOldAttributes()`	This method will get the array of original values of attributes of the current active record since the last `save()` or `find()` call.
`getDirtyAttributes($names = null)`	This method will get the array of all the values of the attributes that were changed since the last `save()` or `find()` call. Basically, it returns those results from `getOldAttributes()` that are different from those returned by `getAttributes()`. You can specify the `$names` parameter mentioning which particular attributes you are interested in.
`isAttributeChanged($name)`	This method will check an attribute whether it was changed since the last `save()` or `find()` call.
`markAttributeDirty($name)`	This method will tell Yii that this particular attribute should be treated as changed irrelevant of whether it really is. This way, you can force the resaving of this attribute to the database.

The idea is that only values of "dirty" (changed) attributes are saved into the database when you call the `save()` method. Obviously, when you create a new record, all attributes are "dirty," so all of them will be saved to the database.

The method named `isAttributeChanged()` is exactly what we need. Just adding one line to the `beforeSave()` handler as follows makes our test pass:

```
public function beforeSave($insert)
{
    ...
    if ($this->isAttributeChanged('password'))
        $this->password = Security::generatePasswordHash($this-
>password);
    ...
}
```

So, we don't unnecessarily rehash the password now. The backend of user management is now complete.

Making a user record into an identity

To be usable in the login/logout functionality we are building, our `UserRecord` class should implement the `yii\web\IdentityInterface` and it doesn't do this yet. Let's start with the addition of `implements yii\web\IdentityInterface` declaration and proceed from that. We need to implement five methods for it.

Two of the methods are directly relevant to the concept of "user identity." The first one is the `getId()` method, which should return a unique identifier of user among others. In our case, it's the database ID of the user record in question, so the implementation is easy:

```
public function getId()
{
    return $this->id;
}
```

The second one is the static `findById()` method, which, given the ID of the user, should return an instance of the corresponding model. In our case, the ID of the user was defined by the `getId()` method as a database ID of the user record, and we need to return an instance of the `UserRecord` class, so the implementation as follows:

```
public static function findIdentity($id)
{
    return static::findOne($id);
}
```

Two other methods are there to support the well-known "remember me" feature. Given that we provide these methods, Yii 2 will handle this feature automatically behind the scenes. We will not bother testing this feature as it wasn't one of our initial goals, but it's not hard at all to implement, so let's just do it.

The first method is `getAuthKey()`, which should return some persistent ID (different from the user ID we return from `getId()` method) uniquely associated with the user identity in question. The hard part is that this **authentication key** must be persistent for Yii to be able to check its validity between the requests.

Let's make up some special attribute on `UserRecord` and return its value as the authentication key:

```
public function getAuthKey()
{
```

```
        return $this->auth_key;
    }
```

We need three steps to actually "make up" this attribute. First, we need a migration to add this column to the table. The actual call in this migration should be as follows:

```
$this->addColumn('user', 'auth_key', 'string UNIQUE');
```

A `UNIQUE` constraint is required as the user record must be identifiable by this field.

> Do not forget to repopulate the `tests/_data/dump.sql` file after the migration.

We'll populate this field at its creation and never change afterwards. Thus, we can add the following lines to the `beforeSave()` handler:

```
            if ($this->isNewRecord)
                $this->auth_key = Yii::$app->security-
>generateRandomKey($length = 255);
```

The meaning of this code should be obvious, but the urge to read the documentation for both `\yii\db\BaseActiveRecord.$isNewRecord` and `\yii\base\Security.generateRandomKey` is encouraged anyway.

Finally, the `auth_key` field should be added to list of required fields in `UserRecord.rules()` as follows:

```
                [['username', 'password', 'auth_key'], 'string',
                  'max' => 255],
```

This concludes our addition of a column to the table.

The second method required for the "remember me" feature is `validateAuthKey()`. It actually checks whether the given authentication key is corresponding to the current user identity. In our case, we can just compare the given key with the value stored in the database for current the user record, as follows:

```
    public function validateAuthKey($authKey)
    {
        return $this->getAuthKey() === $authKey;
    }
```

The final fifth method required to be implemented in the `IdentityInterface` is the `findIdentityByAccessToken()` method. Given some special **access token**, it should return an instance of `UserRecord` (in our case) directly, with some internal checks probably. The nature of this token is not defined, and this method is obviously useful in case of authorization by means of something like OAuth2 or OpenID. We are not going to use such features, so we'll just make the stub:

```
public static function findIdentityByAccessToken(
    $token,
    $type = null
) {
    throw new NotSupportedException(
        'You can only login by username/password pair for now.');
}
```

All of these five methods, `getId()`, `findIdentity($id)`, `getAuthKey()`, `validateAuthKey($authKey)`, and `findIdentityByAccessToken($token)`, are used internally by the Yii 2 mechanics of user login. In most cases, you'll never ever need to call them in your client code.

After the user record becomes the identity, you need to declare it in the `components.user.identityClass` setting in the application config. For our example application, it'll look like the following:

```
        'user' => [
            'identityClass' => 'app\models\user\UserRecord'
        ]
```

Making the login interface

At last, we need to implement the feature we were interested in at the beginning: authenticating the user. First, we need to decide what it will mean for the user to be authenticated.

In the next chapter, which is dedicated to user authorization, we will check whether a user is allowed to do something in the system. But for now, we don't care about authorization, just the authentication, that is, *some indication that the system recognized the user*.

Specifications of user authentication

Let's look at the specifications of user authentication:

- At each page of the application, in the top-right corner is the indicator of user authentication
- When a user is not authenticated, the indicator holds the string **guest** and a link named **login**
- When a user is authenticated, the indicator holds the user's username and a link named **logout**
- A click on **login** leads to the **Login** page
- A click on **logout** de-authenticates the user and redirects him to the homepage

This is pretty hard to comprehensively cover by use cases and the acceptance tests. As our goal is to learn about the features of Yii 2 that help us to implement stuff we want, let's settle with satisfying just the following acceptance test:

```
./cept generate:cept acceptance LoginAndLogout
```

Here are its full contents translated from the preceding high-level description:

```
$I = new AcceptanceTester\CRMUsersManagementSteps($scenario);
$I->wantTo('check that login and logout work');

$I->amGoingTo('Register new User');

$I->amInListUsersUi();
$I->clickOnRegisterNewUserButton();
$I->seeIAmInAddUserUi();
$user = $I->imagineUser();
$I->fillUserDataForm($user);
$I->submitUserDataForm();

$I = new AcceptanceTester\CRMUserSteps($scenario);
$I->amGoingTo('login');

$I->seeLink('login');
$I->click('login');
$I->seeIAmInLoginFormUi();
$I->fillLoginForm($user);
$I->submitLoginForm();

$I->seeIAmAtHomepage();
$I->dontSeeLink('login');
```

```
$I->seeUsername($user);
$I->seeLink('logout');

$I->amGoingTo('logout from arbitrary page');
$I->amInQueryCustomerUi();
$I->click('logout');

$I->seeIAmAtHomepage();
$I->dontSeeUsername($user);
$I->dontSeeLink('logout');
$I->seeLink('login');
```

First, we create a user, then we try to login and see whether the indicator is reflecting the authentication, and then we try to logout and see whether the indicator is back to the state meant for guests.

The highlighted lines are the steps that are not defined yet. They are pretty straightforward.

Checking that we are in the Login Form UI:

```
    public function seeIAmInLoginFormUi()
    {
        $I = $this;
        $I->seeCurrentUrlEquals('/site/login');
    }
```

Filling the login form with the data generated previously:

```
    public function fillLoginForm($user)
    {
        $I = $this;
        $I->fillField('LoginForm[username]',
$user['UserRecord[username]']);
        $I->fillField('LoginForm[password]',
$user['UserRecord[password]']);
    }
```

Note that the function shown in the preceding code is the exact duplicate of the similar "fill form" method in `AcceptanceTester\CRMUsersManagementSteps`. We skipped the refactoring step again.

Submit the login form using the `submitLoginForm()` method:

```
public function submitLoginForm()
{
    $I = $this;
    $I->click('button[type=submit]');
    $I->wait(1);
}
```

As we saw in *Chapter 3*, *Automatically Generating the CRUD Code*, the login form will have client-side validation too, so we are forced to wait.

Check that we are on homepage, that is, on the route `/`:

```
public function seeIAmAtHomepage()
{
    $I = $this;
    $I->seeCurrentUrlEquals('/');
}
```

Check that we have the username from the given fields somewhere on the page:

```
public function seeUsername($user)
{
    $I = $this;
    $I->see($user['UserRecord[username]']);
}
```

Check that we *don't* see the same as above:

```
public function dontSeeUsername($user)
{
    $I = $this;
    $I->dontSee($user['UserRecord[username]']);
}
```

This test fails at the step where we try to click the link named **login**. Let's add it to the layout.

Making the authentication indicator

As specified, the authentication indicator should show the **guest** text and the **login** link for the unauthorized users, and it should show the user's username text and the **logout** link for the authorized users.

We can check whether the user is a guest using the following invocation:

```
Yii::$app->user->isGuest
```

So, the following HTML code in the `views/layouts/main.php` layout file right after the `<div class="container">` opening tag will suffice:

```
<div class="authorization-indicator">
    <?php if (Yii::$app->user->isGuest):?>
        <?= Html::tag('span', 'guest');?>
        <?= Html::a('login', '/site/login');?>
    <?php else:?>
        <?= Html::tag('span', Yii::$app->user->identity-
>username);?>
        <?= Html::a('logout', '/site/logout');?>
    <?php endif;?>
</div>
```

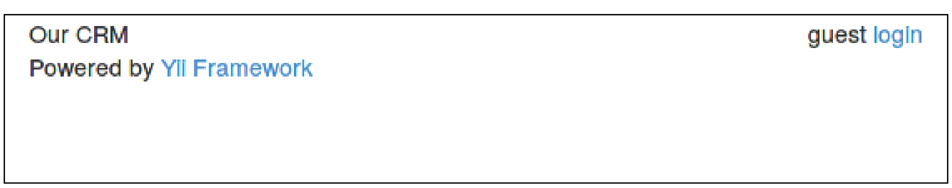
Now, given that we have `ApplicationUiAssetBundle` described in *Chapter 4, The Renderer*, we can add the following bit of CSS to `assets/ui/css/main.css` to make the authorization indicator right-aligned:

```
.authorization-indicator {
    float: right;
    width: 25%;
    text-align: right;
}
```

That's pretty rude, but for now our UI is rudimentary anyway, so it'll suffice.

The login form functionality

Here's how the homepage should look like when opened in a browser (without the theme introduced in *Chapter 4, The Renderer*):



Now we need to make the login form functionality. As we already mentioned at two places in acceptance tests, the route to login is `/site/login`, so we need to provide the `SiteController.actionLogin()` method. Starting from here, we'll make the canonical login form implementation, which can be seen in the advanced application template from Yii 2. It's really hard to make password-based authentication in some other way in Yii, and to do so is unnecessary anyway.

Here's the logic for the traditional `/site/login` route handler:

```
public function actionLogin()
{
    if (!\Yii::$app->user->isGuest)
        return $this->goHome();

    $model = new LoginForm();
    if ($model->load(Yii::$app->request->post()) and $model->login())
        return $this->goBack();

    return $this->render('login', compact('model'));
}
```

If a user is already authenticated, we redirect him back to the homepage using the `\yii\web\Controller.goHome()` helper method that does the 302 redirect to the / root route for us.

If there's some data POSTed to us, we try to check with the `LoginForm.login()` method whether this data is sufficient to authenticate the user. In case of successful authentication, we redirect the user to the last URL visited by him by using the `goBack()` helper method. Otherwise, we render the login form HTML.

The most interesting part is the highlighted lines. We are going to utilize the power of models here. Using the same model, we'll both authenticate the user and manage the HTML form for providing username and password.

Let's start with the login form appearance in the `views/site/login.php` view file. A typical, minimal login form looks like this:

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin(['id' => 'login-form']);
echo $form->field($model, 'username');
echo $form->field($model, 'password')->passwordInput();
echo $form->field($model, 'rememberMe')->checkbox();
echo Html::submitButton(
```

```
        'Login',
        ['class' => 'btn btn-primary', 'name' => 'login-button']
    );
    ActiveForm::end();
```

There is the usual call to the `ActiveForm` widget and three calls to render different kinds of inputs. Note the style in which the fields are rendered. You can consult the documentation and/or source for the `ActiveForm.field()` method for details.

We have an additional checkbox for the "remember me" feature, which we have already prepared in the previous section.

The form described here looks like this (given our application's layout):

You will not see it, although, until we will have the `LoginForm` class in place. It should be a model but not an `ActiveRecord` one, just a `yii\base\Model`.

This model has three fields, as expected in the view file. For our purposes, we need to provide the validation rules and the `login()` method we use in the `SiteController.actionLogin()`.

This is the boilerplate of the `LoginForm` model, placed in `models/user/LoginForm.php`:

```
    <?php
    namespace app\models\user;

    use yii\base\Model;

    class LoginForm extends Model
    {
        public $username;
```

```
    public $password;
    public $rememberMe;
}
```

For validation, we need three rules:

- Both a username and a password are required
- The "remember me" option is a Boolean (set/not set) value
- The password should be valid, that is, if there's a user recorded for the given username, the given password should have the same hash as the one saved for this user (as explained before).

This can be expressed in the following way in the `LoginForm` class:

```
    public function rules()
    {
        return [
            [['username', 'password'], 'required'],
            ['rememberMe', 'boolean'],
            ['password', 'validatePassword']
        ];
    }
```

If the validator to apply is not one of the built-in ones, we need to provide at least a method in this model class with the same name as the validator. The second option is to make a full-fledged custom `Validator` class and register it for the `Application`. For now, we can do with the inline type of validators. The `validatePassword()` inline validator looks like this:

```
    public function validatePassword($attributeName)
    {
        if ($this->hasErrors())
            return;

        $user = $this->getUser($this->username);
        if (!($user and $this->isCorrectHash($this->$attributeName,
$user->password)))
            $this->addError('password', 'Incorrect username or
password.');
    }
```

The guard case at the start is standard: if there are errors already, do not do anything. Then, we try to get `UserRecord` with the username provided. If such a record does not exist, or the provided password does not correspond to the hash stored in this record, we add an error to this model, what a proper validator should really do.

Getting the user record, given its username field value, can be done in any way, but for a nice clean solution, we decided to use lazy loading:

```
/** @var UserRecord */
public $user;

private function getUser($username)
{
    if (!$this->user)
        $this->user = $this->fetchUser($username);

    return $this->user;
}

private function fetchUser($username)
{
    return UserRecord::findOne(compact('username'));
}
```

To check whether the password provided is valid, the following implementation is sufficient:

```
private function isCorrectHash($plaintext, $hash)
{
    return Yii::$app->security->validatePassword($plaintext,
$hash);
}
```

That's the distilled core of our user authentication. In fact, everything until this point was just the support boilerplate code for it. All we really wanted to know is whether `Security::validatePassword()` returns `true` for the given plaintext version of the password and a hash taken from the database.

Then, at last, the `login()` method for `LoginForm` that is used in `SiteController.actionLogin()` concludes the mechanics of the login form:

```
public function login()
{
    if (!$this->validate())
        return false;

    $user = $this->getUser($this->username);
```

```
        if (!$user)
            return false;

        return Yii::$app->user->login(
            $user,
            $this->rememberMe ? 3600 * 24 * 30 : 0
        );
    }
```

If the field values are not valid, obviously, don't allow the user to log in successfully. Finally, call the Yii built-in method for logging a user in and pass the fetched user record to it as identity. The second parameter is the time for which to keep the user session active (in seconds). A zero value means until the closure of the browser window. If "remember me" is set, we keep the user logged in for a month here.

The logout functionality and wrapping things up

Now we only need the logout handler, which can be done by the following almost one-liner method in the `SiteController` class:

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

This is the conclusion to the feature we've built so far. Now run the test:

`./cept run tests/acceptance/LoginAndLogoutCept.php`

It should pass, as shown in the following screenshot:

```
0_0 [master*] $ ./cept run tests/acceptance/LoginAndLogoutCept.php
Codeception PHP Testing Framework v2.0.0-alpha
Powered by PHPUnit 3.7.32-1-g316a547 by Sebastian Bergmann.

Acceptance Tests (1) --------------------------------------------------
Trying to check that login and logout work (LoginAndLogoutCept.php)       Ok
-----------------------------------------------------------------------


Time: 4.78 seconds, Memory: 12.75Mb

OK (1 test, 11 assertions)
```

Summary

Logging in with a username and password is both cumbersome to use and boring to write. A lot of concepts are involved in this form of user authentication. In fact, we basically repeated the authentication already shipped with the advanced Yii 2 application template, probably with different code style convention. However, if you just looked at the source code for advanced template, chances are that you would spend a lot of time trying to understand which pieces, in the form of class methods, variables, classes, and view files, are used in the login functionality and what role does each play in it.

By doing this exercise, we briefly glimpsed over such tricks of Yii 2 and PHP application development in general:

- Writing stuff to the view file depending on whether the user is authenticated or not
- Making the non-active record models in Yii and using them to generate input forms
- The `beforeSave()` method of active records
- The `yii\base\Security` helper class to not bother with securely generating hashes for strings
- Inline validators
- Lazy object loading
- Probably something other which is hard to decouple

Again, *of course* there is authorization using other means: the OAuth2 protocol, OpenID, hardware tokens, SMS codes, you name it. We cannot cover everything, so the most usual method was explained here, which you probably will be asked to implement anyway.

However, we should not forget that we covered only half of the access control. We still need to implement the user authorization after he's authenticated, which will be the topic of the next chapter.

6

User Authorization and Access Control

Authenticating users is just half of the story. Even if you are writing some Web community hub or, God forbid, social network, and identifying users is crucial part of your business rules, you almost always need to control which users can access which part of the application's functionality. It can be said that there's no real point in authentication without access control accompanying it, which is, **user authorization**.

In this chapter, we'll see how Yii 2 can help us in preventing or allowing users to access the functionality of the web application. We will focus on the following four features of Yii:

- Hook methods of the controller
- Exception handling in Yii 2
- Controller action filters
- Role-based access control

As a code example, we'll implement a simple scheme of access control to our example CRM application based on the preceding four features.

Let's start simple.

Access control using the state of user authentication

We already learned about rudimentary access control in the previous chapter, when we wrote the code to render the authentication indicator like the following:

```
if (Yii::$app->user->isGuest)
    // render the indicator for guests
else
    // render the indicator for authenticated users
```

So, we are able to differentiate the content based on whether the user is authenticated or not.

Similarly, we cannot just differentiate the content, but totally prohibit visitors from entering some routes without authentication. For this we need to learn about two features of Yii: exception handling and the hook methods of the controller.

FEATURE – hook methods of the controller

Similar to the `beforeSave()` method we used in the previous chapter, `Controller` classes also have hook methods. They are as follows:

Method name	**What it does**
`beforeAction($action)`	This executes before the actual controller action executes. As with `ActiveRecord.beforeSave()`, it must return a Boolean value indicating whether the given action is allowed to be run.
`afterAction($action, $result)`	This executes after the actual controller action finishes executing, but before the result is sent to `ViewRenderer`. In fact, the `$result` argument is the result of the action, and we can do something with it. Of course, we need to return this argument or else the route will output nothing.

Both of these hooks get the `$action` argument, which represents the controller action being executed. This object is of little use, really, being just the container for the following:

- Action ID, which can be useful for ultra-precise access control based on the action IDs.
- Reference to the controller this action belongs to.
- The `run()` method, which is why this action exists in the first place, as it's the actual route handler represented by the action.

> Please note that in the case where `beforeAction()` returns `false`, the client browser will get a *blank page without any explanations*. This in fact can happen quite often when writing these hooks and forgetting to return a Boolean value from them! The same will happen if you forget to return `$result` from `afterAction()`.

While it's certainly interesting to know what this action really is, right now its brief description is perfectly sufficient. A close inspection of what role an action plays in route handling will be done in *Chapter 12, Route Management*. Most of the time, there's no need to care about the actions.

> All possible hooks, how they work, and how we can use them for our benefit will be discussed in the *Chapter 10, Events and Behaviors*.

Given the definition of the `beforeAction()` method, it is obvious that we can restrict access to some parts of the application in the simplest possible way:

```
public function beforeAction($action)
{
    $parentAllowed = parent::beforeAction($action);
    $meAllowed = !Yii::$app->user->isGuest;
    return $parentAllowed and $meAllowed;
}
```

First we check whether the parent implementation allows the execution of the action in question. We then check whether the current user is unauthenticated by querying the `isGuest` property of the `user` component. If both these conditions are true, then we allow the execution of the action.

> What checks does the `yii\web\Controller.beforeAction()` method do? It protects you from **Cross-Site Request Forgery** (**CSRF**) attacks. In Yii 2, each and every form generated by the `ActiveForm` widget, by default, includes a special token as the additional request parameter. Only if the expected token from the form is equal to the token included in the client request will the request be processed. Of course, this check is unnecessary (and is not being done) for the `GET`, `HEAD`, and `OPTIONS` requests (the last one being extremely rare). You can also disable this check (don't do this) via the `enableCsrfValidation` setting directly in the application configuration.
>
> By the way, in the main layout file we made in *Chapter 3*, *Automatically Generating the CRUD Code*, and discussed in detail in *Chapter 4*, *The Renderer*, there is a call to the `Html::csrfMetaTags()` method that we haven't even mentioned yet. This call is here exactly to support the CSRF protection. More than that, with the `enableCsrfValidation` setting set to `true`, you must call this method or else you will lose the ability to submit any HTML form generated by the `ActiveForm` widget.
>
> Most of the time you wouldn't do any serious harm if you don't call `parent::beforeAction()`, but you better do it, because if you forget to do this in the form-submitting action you'll lose an important line of security.

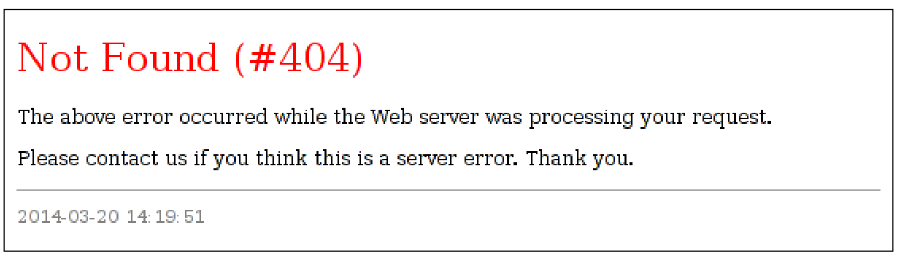
However, this way of blocking is not really interesting. First, it does not show the client any feedback about what really happened. Second, such a usage of `beforeAction()` is already anticipated in Yii 2 in the form of a special concept called **filters**.

Let's postpone the explanation of filters for a while and look at how we can show the client's browser what really happened with its request.

FEATURE – exception handling in Yii

Yii employs its own handler for unhandled exceptions, which is really usable right out of the box without any further improvements. Basically, if you throw an exception of any `Exception` class descendant somewhere in your controller action and not handle it, Yii will show the following error message to the client:

However, here's what it will show if you throw the `yii\web\NotFoundHttpException`:



Note that it correctly shows the status code and the error message according to the specification of HTTP status codes. The actual status code returned will also be 404.

In fact, `yii\web\NotFoundHttpException` is a wrapper around the base `yii\web\HttpException`, which you can throw to emit arbitrary status code and message to the client, as follows:

```
throw new HttpException(406, 'Pretty rare error, usually you should
never see it.');
```

Don't forget to write `use yii\web\HttpException` at the beginning of your script. Here is what will be shown:

Note that the message provided in the exception constructor is shown underneath the main message of the 406 status code, which is autogenerated according to HTTP specification and is not changeable. The status code actually returned from the server will also be 406.

So, just by throwing the various kinds of HTTP exceptions you can block the user request with a descriptive message. Note, however, that nothing prevents someone from getting the following output when using yii\web\HttpException instead of its specialized subclasses:

Well, indeed anyone will think this is a server error. Even while Yii really returns status code 200, the response will be the error page. In case anyone tries to throw the `HttpException` with the status code 302, for example, the status code will be 302 but no redirection will happen. This is just the static page error handler in all cases.

Here is the list of all descendants of `HttpException` that Yii 2 defines for its users' convenience:

Exception	HTTP status code shown
`BadRequestHttpException`	**400 Bad Request**
`ConflictHttpException`	**409 Conflict**
`ForbiddenHttpException`	**403 Forbidden**
`GoneHttpException`	**410 Gone**
`MethodNotAllowedHttpException`	**405 Method Not Allowed**
`NotAcceptableHttpException`	**406 Not Acceptable**
`NotFoundHttpException`	**404 Not Found**
`TooManyRequestsHttpException`	**429 Too Many Requests** (from RFC 6585 additional HTTP status codes, see `http://tools.ietf.org/html/rfc6585`)
`UnauthorizedHttpException`	**401 Unauthorized**
`UnsupportedMediaTypeHttpException`	**415 Unsupported Media Type**

Another good feature of Yii exception handling is its error reporting in case the debugging mode is enabled.

Inside the `index.php` entry point script, before a call to `require()` for Yii library, the constant `YII_DEBUG` will be defined with the value of `true`:

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

> It's vital for this constant to be defined before the Yii library is loaded, as in its absence Yii will define it to be `false`. Note that `YII_DEBUG` also performs other tasks apart from enabling verbose error reporting.

If an unhandled exception like the following one is thrown in the code (this is different from the `HttpException`):

```
throw new \LogicException('I am unhandled exception and I am proud of
it');
```

Then here's what will be shown instead of the generic "Exception" page:

This is a really amazing stacktrace page. It's so long that to fit it here we had to tear the non-unique pieces out of it. It gives several layers of code before the failed line with code highlighting, filenames, line numbers, and *links to the documentation pages for the Yii classes and methods mentioned* (!). It shows the server runtime configuration and also cookies sent by the client.

> This page should not be presented to visitors in the production environment. Always set `YII_DEBUG` to `false` there.

Okay, now here's the guard case that prevents unauthenticated users from using the controller action it is in:

```
if (Yii::$app->user->isGuest)
    throw new ForbiddenHttpException;
```

This is the simplest line of access control available for us in Yii 2. Of course, any check can be used as long as the `HttpException` descendant is being thrown. This is so fundamental that Yii has a special built-in `AccessControl` filter for it. We'll learn about the `AccessControl` filter later. Let's learn about what a filter is in Yii terminology.

FEATURE – controller action filters

An action filter, in short, is the `beforeAction()` and `afterAction()` methods packed into a single class, pluggable to the `Controller` instances through the `Controller.behaviors()` method.

The benefits of this approach are as follows:

- You can define arbitrarily long and complex guard cases and/or post- or pre-process them, given that you now have an entire class dedicated for this purpose.
- You can combine different filters in any combinations and order.

For the class to become the action filter for some controller, it needs to be:

- A descendant of the `ActionFilter` class.
- Referenced in the controller's `behaviors()` method.

The `ActionFilter` itself is the special case of the behaviors that are described in *Chapter 10, Events and Behaviors*. For now, it's sufficient to understand that a behavior is some class containing some methods that can be declared to be an integral part of some other class, extending its feature set. If you are familiar with the concept of "traits" from PHP 5.4, this is it.

We'll not look at how to create our own `ActionFilter`, as it's quite unnecessary given that we already know how the `beforeAction()` and `afterAction()` methods work. Let's briefly look at the built-in Yii 2 filters instead. Detailed information about their usage can be read from their corresponding documentation pages.

Filter class	What it does
`\yii\filters\VerbFilter`	This filter prevents or allows access to controller actions based on the HTTP method used for request. For example, you can allow only POST requests for your log in and log out actions with this filter. This is the only action filter that doesn't inherit the `ActionFilter` class but directly inherits the `Behavior` class instead. It responds with the **405 Method Not Allowed** message if the action is not to be executed. Visit `http://www.yiiframework.com/doc-2.0/yii-filters-verbfilter.html` for more information.
`\yii\filters\PageCache`	This filter caches the rendering result of the given controller's actions. It has a sufficiently complex and flexible configuration to control where, for how long, and which particular routes should be cached. Visit `http://www.yiiframework.com/doc-2.0/yii-filters-pagecache.html` for more information.
`\yii\filters\HttpCache`	This filter is conceptually equivalent to `PageCache`, but it uses the HTTP **Last-Modified** and **ETag** headers to perform caching. In effect, the client's browser will be responsible for keeping and presenting the cached version of the response to the user. Visit `http://www.yiiframework.com/doc-2.0/yii-filters-httpcache.html` for more information.
`\yii\filters\AccessControl`	This filter prevents or allows access to the actions depending on the set of rules in a very flexible and expressive syntax. It is so powerful that often there's no need to implement any other form of access control in the controller at all. This filter can control access using the HTTP method, user authentication status, user role (to be defined later), action and/or controller ID, user IP, or even a custom callback provided. It responds with **403 Forbidden** if the action is not to be executed. Visit `http://www.yiiframework.com/doc-2.0/yii-filters-accesscontrol.html` for more information, but we'll be discussing this filter in this chapter anyway.
`\yii\filters\ContentNegotiator`	This filter is a very handy filter for large applications. It automatically changes the application language and response format based on the HTTP headers and query parameters send by the client browser. Visit `http://www.yiiframework.com/doc-2.0/yii-filters-contentnegotiator.html` for more information.

Filter class	What it does
`\yii\filters\RateLimiter`	This filter prevents the access for users who exceeded their rate limit, that is, amount of page requests in some time range. The user identities (explained in the previous chapter) need to implement the `\yii\filters\RateLimitInterface` to support specifying the allowed rates per user. Visit `http://www.yiiframework.com/doc-2.0/yii-filters-ratelimiter.html` for more information.

As this chapter is about access control, let's look at some examples of the usage of `VerbFilter` and `AccessControl`. The `AccessControl` filter is so important and powerful that we'll look at it later in more detail after we explore the user roles system.

Here's how Gii protects the controller autogenerated by the `VerbFilter`:

```
public function behaviors()
{
    return [
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'delete' => ['post'],
            ],
        ],
    ];
}
```

> Note the `VerbFilter::className()` invocation. It is the Yii 2 idiom allowing us to easily get the fully qualified name of any class extending the base `yii\base\Object`, and the Yii 2 code uses this idiom extensively. In this case, it will always return the `\yii\filters\VerbFilter` string.

The `behaviors()` method must return an array of configurations for the filters to be attached to the controller. In this case, we attach the `VerbFilter` at the arbitrarily chosen `verbs` key. This `VerbFilter` is configured so that the `delete` action can be accessed only by the POST request. All other actions are unguarded, because the update form request accepts the data being POSTed and renders the HTML page for the form.

Here's how we can quite naively protect our `login` and `logout` actions using the `AccessControl` filter:

```
public function behaviors()
{
    return [
        'access' => [ // 1
            'class' => AccessControl::className(), // 2
            'only' => ['login', 'logout'], // 3
            'rules' => [
                [
                    'actions' => ['login'], // 4
                    'roles' => ['?'], // 5
                    'allow' => true, // 6
                ],
                [
                    'actions' => ['logout'], // 7
                    'roles' => ['@'], // 8
                    'allow' => true, // 9
                ]
            ]
        ]
    ];
}
```

In the preceding code, we are using the `behaviors` property of our `Controller` to attach the `\yii\filters\AccessControl` class to it, which will be already configured as specified.

Let's understand the preceding code line-by-line from the top to bottom:

1. An arbitrarily-selected name `access` is used to return back to the calling method from the `behaviors()` method.
2. We register the `AccessControl` filter.
3. The `only` variable denotes that only the `login` and `logout` actions, which translate to `actionLogin()` and `actionLogout()` methods of the controller, are used. The default for access control filter is to deny everything not explicitly allowed, so we need to restrict it here.
4. For the `login` action.
5. For unauthenticated users. The symbol `?` means *guest users*.
6. Allow request.

7. For the `logout` action.
8. For authenticated users. The symbol `@` means *authenticated users*.
9. Allow request.

The specific configuration parameters for each rule can be read in the documentation and/or source of the `yii\filters\AccessRule` class (`http://www.yiiframework.com/doc-2.0/yii-filters-accessrule.html`).

Role-based access control

Yii 2 uses an extremely easy-to-use form of user access control called permissions. When employed, each user can be given permission (exact mechanics are not important right now). When necessary, the application can check whether authenticated user has this permission by calling:

```
Yii::$app->user->can($permission);
```

Here, the `$permission` argument is the string that is the textual label for the permission. That's it.

There are two important additional features provided by the concept of permissions:

- A permission can be declared as the *child* of other permission. However, when some user has the *parent* permission, he is automatically considered granted of all his *child* permissions.
- We can assign some additional parameterized constraint named `rule` to the permission. Technically, a rule is a PHP function that takes parameters as arguments. If this function returns `false`, even if the user is assigned the permission in question, he is considered blocked anyway.

This scheme is certainly lacking a concept of *role* to be called **role-based access control** (**RBAC**). Yii 2 has roles working the same way as the permissions. In fact, they are implemented using the same base class `\yii\rbac\Item`, differing only by a value of one class constant. There is no `Yii::$app->user->is($role)` invocation, so you need to check for a user being assigned some role by the same call to `\yii\web\User.can()`. Roles in RBAC of Yii 2 are meant to be used only as a group of permissions.

The application component handling the RBAC is called the authorization manager and is accessible as the `authManager` component by the `Yii::$app->authManager` property. The previously described `\yii\web\User.can()` method is a shorthand wrapper around the call to `Yii::$app->authManager->checkAccess($user_id, $permission)`. For convenience, there exists a concept of **default role**. The list of default role names can be set in the `components.authManager.defaultRoles` setting in the `Application` configuration, which corresponds to the `\yii\rbac\BaseManager::$defaultRoles` property. A user is assumed to have the default role all of the time, that is, given the value of `defaultRoles` as `["guest"]`, the call to `Yii::$app->authManager->checkAccess($user_id, "guest")` will always return `true`.

In the `AccessFilter` described previously, each rule can be configured to check for the user role. We used the special symbols `?` and `@` back there, which denoted guests and authenticated users, respectively, but in fact, the role names can be used there.

A big challenge for this system to work is to set up the required associations between users and roles. Let's explore how it's done by making a feature in our example CRM application.

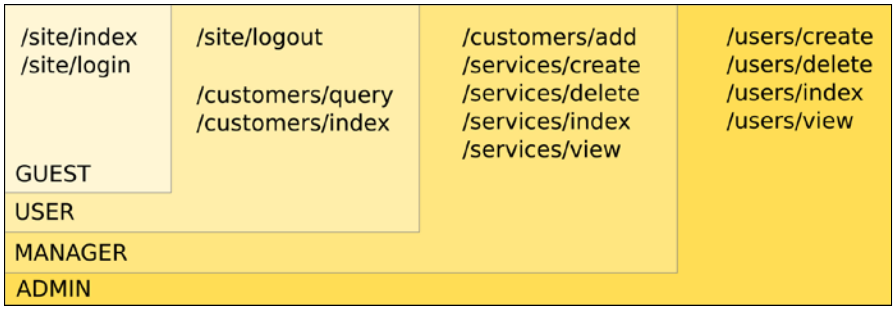
Protecting the CRM management from CRM users

We have differentiated the CRM manager and CRM user roles since the very beginning in our specifications for the application. So far, each acceptance test began with some work on the side of database management and ended with either usage of public interface or checking some assumptions right in the management UI.

Now, it's time to really prohibit the CRM user from accessing the database management UI pages. We are going to implement the following business ruleset:

- Unauthenticated (guest) users should not be able to access anything except the home page and the login form.
- User-level users should be able to access the Query Customer By Phone UI.
- Manager-level users should be able to access everything except the User Management UI.
- Administrator-level users should be able to access everything.

Have a look at the following scheme:



We already have tests for the login and logout functionality without testing for access rights afterwards. However, now we are forcing users of our application to log in first. This would mean that in all our existing acceptance tests, the first step should be the logging in one as some users have enough rights to do what the test needs to do.

It is incredibly hard to do proper end-to-end tests, as it'll require not only using the corresponding admin UI to generate entities for manipulation with which we want to test, but also using the root-level administration UI to create the user and grant it with enough rights to perform such manipulations. Not even saying that we need a working UI to assign access rights (that is, roles) to users, this is too much of a preliminary work to tolerate. However, such a course of actions is preferable for extremely thorough end-to-end tests, ensuring that every part of the application works without missing anything important.

Installing predefined users

In this example, we will trade the time required for preparing the users to hold them in the database right from the start. Instead of creating a user of sufficient caliber for each individual test, we'll predefine one user for each role in the database right from the start, and our acceptance tests will use credentials of those users for logging in before doing anything. Of course, it will not relieve us from the task of actually making the logic to login before each action.

Here are the users that should be created (passwords are arbitrary high-entropy phrases that are easy to remember):

Username	**Password**	**Role name**
No username, default role	No password	`guest`
`JoeUser`	`7 wonder @ American soil`	`user`
`AnnieManager`	`Shiny 3 things hmm, vulnerable`	`manager`
`RobAdmin`	`Imitate #14th syndrome of apathy`	`admin`

> You can get a nice breakdown of the password strength at `http://www.passwordmeter.com/`. Currently, the most important trait of a good password is the length and the types of characters used in it (see for example this note from Microsoft: `http://www.microsoft.com/en-gb/security/online-privacy/passwords-create.aspx`). We deliberately chose simple phrases with uppercase and lowercase letters, numbers, and special symbols in them. Note that we don't have any reason to exclude the space symbol from our passwords.

Create a migration to add them to the database:

```
./yii migrate/create add_predefined_users
```

Users listed in the preceding table can be generated using whatever style seems reasonable, but remember that a user must be created like this:

```
$user = new \app\models\user\UserRecord();
$user->attributes = compact('username', 'password');
$user->save();
```

Our `beforeSave()` hook handles the generation of password hashes and authorization tokens for us.

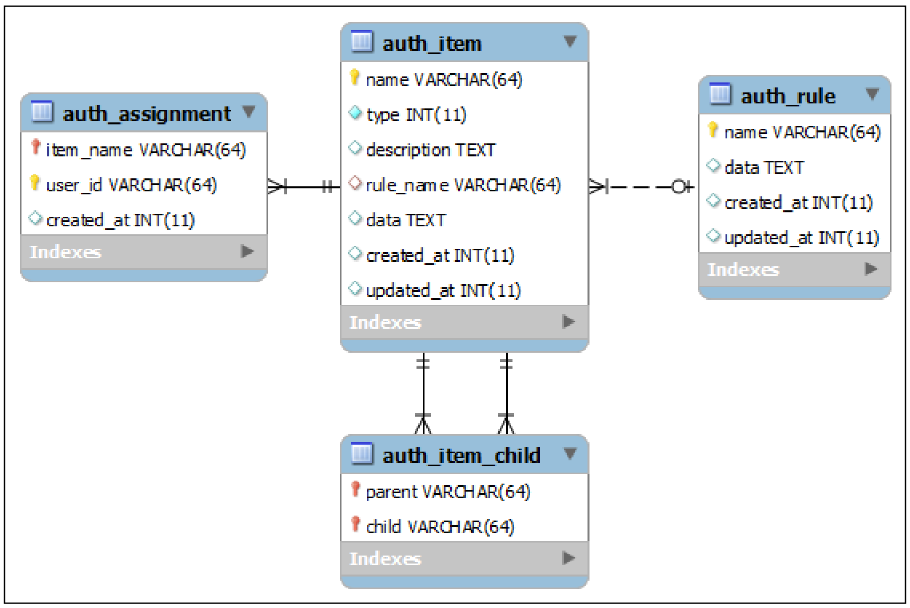
> You can look at the way we generated our users in the `migrations/m140718_063423_add_predefined_users.php` file in the example code base accompanying this book.

Then, we need to enable the RBAC manager role in our application in order to be able to assign roles to the users.

RBAC managers in Yii

Yii has two built-in RBAC managers. One is `\yii\rbac\PhpManager`, which reads the role assignments from the PHP script each time the application is loaded, and the other is `\yii\rbac\DbManager`, which stores assignments in the database. We'll use the `DbManager`, as it'll allow more freedom in manipulation of assignments.

Database-based RBAC manager uses the following schema to store the information about user roles:



In fact, Yii's RBAC manager does not operate in terms of roles at all. It operates in terms of authorization items, which can be of two types. Here's the excerpt from the class definition:

```
namespace yii\rbac;
class Item extends Object
{
    const TYPE_ROLE = 1;
    const TYPE_PERMISSION = 2;
```

The database manager stores the authorization items in the table named `auth_item` by default. As usual in Yii 2, this is configurable.

The problem is that these types are neither enforced nor checked anywhere. `Yii::$app->user->can($itemName)` fits them all. So, for simplicity, it's easier to talk in terms of roles instead of authorization items. However, in case of really elaborated RBAC, this distinction can be useful from the design standpoint.

Authorization items, be it role or permissions stored by the authorization manager, form a directed acyclic graph so they can have *children*, as was mentioned before. Let's elaborate on that, for simplicity, supposing that we are dealing only with roles.

If a user `$user` has a role `$role` and this role has a child role `$child`, then the call to `$user->can($child)` will return `true`, that is, *parent has access to everything that child has*. However, if a user has a role `$child` and this role is a child of `$role`, then the call to `$user->can($role)` will return `false`, that is, the *child role has no access to what the parent role has access to*.

Parent-child relationships between roles are stored by `DbManager` in the table named `auth_item_child` by default.

The actual assignments of roles to users are stored inside the table named `auth_assignment`. Note that in the `user_id` column there is not a foreign key and it is not even of `INT` type in default schema shown before. This is, of course, because the default schema has no knowledge about how you store your user records (they can even be in different databases, after all). Also, the `user_id` column should be filled with IDs that are returned by the `IdentityInterface.getId()` method, which our `UserRecord` class happily implements This is not necessarily the actual primary keys of the records in the `user` table. However, this is indeed exactly as it is in our case.

Yii 2 developers provided us with the migration already prepared to initialize the database schema for `\yii\rbac\DbManager`. You can set up the appropriate tables using the following console command:

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

> You can also use the following hack to fill our database with the tables expected by the database manager. The schema is stored in the set of files with names `schema-*.sql` inside the `rbac` directory under the root folder of Yii 2 installation. Each file corresponds to some specific RDBMS. Under the assumptions of using MySQL and Yii 2 being installed by composer, the required file should be exactly `vendor/yiisoft/yii2/rbac/schema-mysql.sql`. Create the migration as follows:
>
> **`./yii migrate/create create_rbac_tables`**
>
> Then, in the `up()` method inside the generated migration script, write the following hack which loads the raw SQL from the file just discussed:
>
> ```
>         $this->execute(
>             file_get_contents(
>                 Yii::getAlias('@yii/rbac/schema-mysql.
> sql')));
> ```
>
> This trick is really useful in cases when you have an already battle-tested, very old database schema, large enough that rewriting it in Yii idiomatic migrations code will be too cumbersome.

The failing test for our role hierarchy

Before we start to really fill our database with roles, how can we ensure that our access control is actually in place? Let's construct a functional test for our role hierarchy. Create the test:

```
./cept generate:test functional RoleHierarchy
```

> Things became a little tricky with the addition of predefined roles. To have a clean state in our functional tests, we are using the dump from clean database, which is not so easy to get now with the migrations that generate actual records in the table. Even without them, there's still need to regenerate the dump after each migration added. Also, with this approach we can completely wipe out production database if we launch the functional tests on the production environment. All of these issues we'll postpone until the final chapter, *Chapter 13*, *Collaborative Work*. Let's pretend until then that we somehow have the clean state before running any of the functional and acceptance tests provided.
>
> Look at the files `tests/functional.suite.yml`, `tests/functional/_bootstrap.php`, and `config/test.php` in the code base bundled with this book to see how we solved this problem.

Inside this test, we'll write the brute-force implementation of checking the default role, which is a `guest`, as shown in the following code:

```
/** @test */
public function DefaultRoleIsGuest()
{
    // no login at all

    $this->assertFalse($this->user->can('admin'));
    $this->assertFalse($this->user->can('manager'));
    $this->assertFalse($this->user->can('user'));
    $this->assertTrue($this->user->can('guest'));
}
```

Yes, we need to use the `can()` method for checking the assigned roles, which reads really strange.

The class property `$this->user` is set up with the `\yii\web\User` component at the setup stage of the test to act both as shorthand and as a single place for changes:

```
/** @var \yii\web\User */
private $user;

protected function _before()
{
    $this->user = Yii::$app->user;
}
```

We will test the roles of the predefined users using the DataProvider feature of the PHPUnit framework (visit `http://phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.data-providers`, do not mistake it for the `DataProvider` concept from Yii):

```
public function PredefinedUserRoles()
{
    return [
        ['RobAdmin',     ['admin' => true,  'manager' => true,
'user' => true, 'guest' => true]],
        ['AnnieManager', ['admin' => false, 'manager' => true,
'user' => true, 'guest' => true]],
        ['JoeUser',      ['admin' => false, 'manager' => false,
'user' => true, 'guest' => true]],
    ];
}

/**
```

```
     * @test
     * @dataProvider PredefinedUserRoles
     * @param string $username
     * @param array $rbac
     */
    public function PredefinedUsersHasProperRoles($username, $rbac)
    {
        $identity = \app\models\user\UserRecord::findOne(compact('use
rname'));

        $this->user->login($identity);

        foreach ($rbac as $role => $allowed)
            $this->assertEquals($allowed, $this->user->can($role));
    }
```

The `foreach` instruction will partially hinder us in case an error occurs in checking the access. However, increase in readability is an important tradeoff.

For each dataset provided by our data provider, we find the user record in database with the provided username, and then check whether `Yii::$app->user->can()` is behaving according to our role hierarchy for each of these users.

As we're logging into the system on each test, let's log out at the teardown stage:

```
    protected function _after()
    {
        $this->user->logout();
    }
```

This test should obviously fail, because we did not even attach the RBAC Manager component to our application.

Setting up the role hierarchy

We'll fill the database with our roles and role assignments using the migration. This means that we need to attach the RBAC manager both to the console application that will be used when we'll launch the `./yii migrate` script in the root of code base and by the web application that will actually use the RBAC. In addition to being able to write this one migration, we'll be able to write custom console commands which use our RBAC manager, if necessary.

To attach the database-based RBAC manager to our application, we only need to reference it in the `components` section of the configuration:

```
'authManager' => [
    'class' => 'yii\rbac\DbManager',
    'defaultRoles' => ['guest'],
],
```

As explained, this should be either added to both `@app/config/web.php` and `@app/config/console.php` or separated out to an additional config snippet and included in both of these configurations anyway. The paths provided are valid for the example CRM application so far.

So, from now on, we have `Yii::$app->authManager` available, pointing to the instance of `yii\rbac\DbManager`. The default role, as specified in the highlighted part of the preceding code, will be the guest, that is, everyone who is not authenticated or does not have role assigned will be treated as having a guest role.

We are now able to make migrations which install the needed roles. We can use the following methods in the `yii\rbac\DbManager` class:

Method name	Reason to use
`createRole($name)`	This method is the main method to create roles. This returns a configured instance of `\yii\rbac\Role`.
`createPermission($name)`	This method is same as `createRole()`, but creates instances of `\yii\rbac\Permission`. We will not use this method as, for simplicity, we are using only roles in our security schema.
`assign($role, $userId)`	This method binds the instance of `\yii\rbac\Role` to the user with `$userId`. The `$userId` must be the ID that `IdentityInterface.getId()` returns!
`add($item)`	This method registers the given authorization item, be it an instance of `\yii\rbac\Permission` or `\yii\rbac\Role`.
`addChild($parent, $child);`	This method registers one authorization item to be the child to the other.

There are many more methods, but we listed only the most fundamental ones. Others can be found in the reference documentation for `\yii\rbac\ManagerInterface` at `http://www.yiiframework.com/doc-2.0/yii-rbac-managerinterface.html`.

Finally, here's the migration to set up our role hierarchy:

```
    public function up()
    {
        $rbac = \Yii::$app->authManager;

        $guest = $rbac->createRole('guest');
        $guest->description = 'Nobody';
        $rbac->add($guest);

        $user = $rbac->createRole('user');
        $user->description = 'Can use the query UI and nothing else';
        $rbac->add($user);

        $manager = $rbac->createRole('manager');
        $manager->description = 'Can manage entities in database, but
not users';
        $rbac->add($manager);

        $admin = $rbac->createRole('admin');
        $admin->description = 'Can do anything including managing
users';
        $rbac->add($admin);

        $rbac->addChild($admin, $manager);
        $rbac->addChild($manager, $user);
        $rbac->addChild($user, $guest);

        $rbac->assign(
            $user,
            \app\models\user\UserRecord::findOne(['username' =>
'JoeUser'])->id
        );
        $rbac->assign(
            $manager,
            \app\models\user\UserRecord::findOne(['username' =>
'AnnieManager'])->id
        );
        $rbac->assign(
            $admin,
            \app\models\user\UserRecord::findOne(['username' =>
'RobAdmin'])->id
        );    }
```

```
    public function down()
    {
        $manager = \Yii::$app->authManager;
        $manager->removeAll();
    }
```

We really can recover from this migration by using the `yii\rbac\DbManager.removeAll()` method, which is not really useful otherwise.

Now our functional test for roles hierarchy passes. We're ready to implement the real protection for our controllers.

The failing test for access control in controllers

In this section, we return from the implementation details of the RBAC management to the feature we actually want to implement.

How are we going to test the access restrictions described in the *Protecting the CRM management from CRM users* section? It's a really tough question, and it's amazingly hard to do it in the right way. For ease of explanation, let's make a very, very simple implementation that will *just work* for us right now.

Each of the subclasses of `AcceptanceTester` we created for our acceptance tests so far are implied to have some roles from our hierarchy. We will treat each of them as users from the predefined ones in the following manner:

- `AcceptanceTester\CRMOperatorSteps` corresponds to '`AnnieManager`', who is a manager
- `AcceptanceTester\CRMServiceManagementSteps` corresponds to '`AnnieManager`', who is a manager
- `AcceptanceTester\CRMUserSteps` corresponds to '`JoeUser`', who is a user
- `AcceptanceTester\CRMUsersManagementSteps` corresponds to '`RobAdmin`', who is the admin and the only role capable of managing users

We'll write the set of absolutely straightforward acceptance tests to check our assumptions in access rights:

```
./cept generate:cept acceptance AdminAccessRights
./cept generate:cept acceptance ManagerAccessRights
./cept generate:cept acceptance UserAccessRights
./cept generate:cept acceptance GuestAccessRights
```

Here's the content of `tests/acceptance/ManagerAccessRightsCept.php` (other tests are similar):

```
$I = new AcceptanceTester\CRMOperatorSteps($scenario);
$I->wantTo('Check Manager-level access rights');

$I->amOnPage('/customers/query');
$I->dontSee('Forbidden');

$I->amOnPage('/customers/index');
$I->dontSee('Forbidden');

// ... and so on...

$I->amOnPage('/users/create');
$I->see('Forbidden');

$I->amOnPage('/users/index');
$I->see('Forbidden');
```

> The preceding test code is neither maintainable nor thorough. Do not write your tests like this. This style was chosen because it expresses the idea well and is concise enough to present in the book format.

We just go to each of the routes we have on the system and check whether we will get the **403 Forbidden** error page, which has the word **Forbidden** written on it in huge letters. The routes `/services/delete` and `/users/delete` were omitted, because (as explained earlier in this chapter) they're protected by the `VerbFilter` and can be accessed only by POST requests. The routes `/site/index`, `/site/login`, and `/site/logout` were omitted because they will have their own special tests for access rights.

The preceding list of *see* and *don't see* should be repeated for each of the `CRMOperatorSteps`, `CRMServiceManagementSteps`, `CRMUserSteps`, and `CRMUsersManagementSteps` `AcceptanceTester` classes. The test scenario for manager access rights, thus, should have two classes tested.

For guest users, there should be a slightly different expected behavior. While it's possible to change it, by default Yii 2 will redirect guest users to the login form instead of showing them as **403 Forbidden**. This is really a nice addition. So, `GuestAccessRightsCept` should have checks not in the form:

```
$I->see('Forbidden');
```

Checks should be in the following form:

```
$I->seeElement('#login-form');
```

Codeception's `seeElement()` method from the `WebDriver` module (visit `http://codeception.com/docs/modules/WebDriver#seeElement`) can search DOM elements by CSS selectors, and `login-form` is what we specified as ID for this form when configuring it.

We should have the additional `AcceptanceTester` subclass that represents the guest users. A guest user is what `LoginAndLogoutCept` from the previous chapter really requires. Let's create it:

```
./cept generate:stepobject acceptance CRMGuest
```

This subclass will just be empty for now, because it's only used in `GuestAccessRightsCept`.

In the end, using this test strategy, we will end up with the following set of acceptance test scenarios:

- `tests/acceptance/GuestAccessRightsCept.php`: This file checks the access rights for the `guest` role, which is represented by the newly-created `\AcceptanceTester\CRMGuestSteps` class
- `tests/acceptance/UserAccessRightsCept.php`: This file checks the access rights for the `user` role, which is represented by the `\AcceptanceTester\CRMUserSteps` class
- `tests/acceptance/ManagerAccessRightsCept.php`: This file checks the access rights for the `manager` role, which is represented by two classes in our test suite, `\AcceptanceTester\CRMOperatorSteps` and `\AcceptanceTester\CRMServicesManagementSteps`
- `tests/acceptance/AdminAccessRightsCept.php`: This file checks the access rights for the `admin` role, which is represented by the `\AcceptanceTester\CRMUsersManagementSteps` class

Access rights will be checked by combinations of calls to the `see('Forbidden')` and `dontSee('Forbidden')` methods, according to the access rights scheme we decided to use at the beginning of this section.

After that, our tests will run without exceptions, but without success too. How do we make them pass, that is, how do we protect the controllers from access by unauthorized users?

We'll use the access control filter.

FEATURE – access control filter

An access control filter is a special action filter that allows us to specify access rights to actions inside some controller.

It should be attached to the controller in the following way:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'rules' => [
                // rules in the format described above
            ]
        ]
    ];
}
```

So, the predefined `behaviors()` method should return an array that should have the configuration for the `AccessControl` class initialization.

Each element in the `rules` setting is an array with the following fields:

```
 [
    'allow' => true, // or false
    'actions' => ['ids', 'of', 'actions', 'to', 'apply', 'rule',
'to'],
    'controllers' => ['ids', 'of', 'controllers', 'to', 'apply',
'rule', 'to'],
    'roles' => ['roles', 'including', 'symbols', '?', 'and', '@',
'explained', 'before'],
    'ips' => ['IP', 'addresses', 'possibly', 'including', 'wildcards',
'*'],
    'verbs' => ['HTTP', 'Request', 'Methods', 'to', 'apply', 'rule',
'to'],
    'matchCallback' => $callable1, // to determine whether we should
apply rule in arbitrary way
    'denyCallback' => $callable2 // defining what to do when the
access is rejected by this rule
]
```

The highlighted fields correspond to the public attributes of the `\yii\web\AccessRule` class. To save space, we will not discuss the full meaning of every field, especially when the class definition for `\yii\web\AccessRule` holds a complete description for them.

Rules are checked in order of appearance, so they can have overlapping prerequisites. If no rule was fired, the request is blocked. So if the `rules` section of `AccessControl` filter configuration is left empty, everything is banned in this controller for everyone.

The `denyCallback` field mentioned in the rule configuration, as all other options, is optional. If omitted, the similar `denyCallback` property of the `AccessControl` class will be used. By default, it shows the **403 Forbidden** page for authenticated users and the login form page for guests, which is *exactly* the behavior we anticipate in our acceptance tests. Yeah, we cheated again.

Applying access control to the site

First, we need to protect the `UsersController`, so it'll be only accessible by users with the `admin` role:

```
'rules' => [
    [
        'roles' => ['admin'],
        'allow' => true
    ]
```

We presented only the rules, as the `AccessControl` config surrounding them is the same all of the time. We need to add our `AccessControl` config to the `behaviors()` method. Note that the `UserController` class already has this method defined by Gii upon generation.

For `CustomersController` we need the following:

```
'rules' => [
    [
        'actions' => ['add'],
        'roles' => ['manager'],
        'allow' => true
    ],
    [
        'actions' => ['index', 'query'],
        'roles' => ['user'],
        'allow' => true
    ],
]
```

According to our specifications for access rights, any user has read access to the customers management UI, but manager-level access rights requires one to register as a new customer.

For `ServicesController`, we need the following rule, which is similar to the `UsersController` but has a different role:

```
'rules' => [
    [
        'roles' => ['manager'],
        'allow' => true
    ]
]
```

Well, that's all that we needed to do. The most interesting part is the restructuring of our `AcceptanceTester` variants so that they survive in this new world filled with restrictions. All of our tests now basically require a log in first. Allow `CRMGuest` to log in and log out, and then have all other `AcceptanceTester` subclasses extend `CRMGuest` so they'll also be able to do so:

```
function login($username, $password) // 1
{
    $I = $this;
    $I->amOnPage('/site/login');
    $I->fillField('LoginForm[username]', $username);
    $I->fillField('LoginForm[password]', $password);
    $I->click('Login');
    $I->wait(1); // 2
    $I->seeCurrentUrlEquals('/'); // 3
}
```

The highlighted parts show the important pieces of this code. First of all, this method is versatile and accepts the username and password as parameters. Secondly, it's using our login form, which still has JavaScript validation, so we are forced to wait a second after submitting it. Thirdly, to protect us early in case of problems, as the acceptance tests are notoriously slow, we immediately check whether we were redirected to the homepage after logging in. This means the login was successful. The code is as follows:

```
function logout()
{
    $I = $this;
    $I->amOnPage('/');
    // Expecting that this button is presented on the homepage.
    $I->click('logout');
}
```

The `logout` option is a lot simpler here. We just need to move to the homepage, because we can be on some page like **403 Forbidden** (and in fact, we'll end up there *a lot*) where there's no logout link at all. However, we should be careful where we use this test step, as the logout link exists only when the user is authenticated.

As mentioned before, we need to make all kinds of `AcceptanceTester` subclasses into descendants of `CRMGuestSteps` now: `CRMOperatorSteps`, `CRMUsersManagementSteps`, `CRMUserSteps`, and `CRMServicesManagementSteps`.

Having all of that in place, we now have the ability to make all of our `AcceptanceTester` instances login right after birth. We just put the following two properties and a constructor in `CRMGuestSteps`:

```
public $username;
public $password;

public function __construct($scenario)
{
    parent::__construct($scenario);

    if ($this->username and $this->password)
        $this->login($this->username, $this->password);
}
```

So, if the descendant has the username and password defined, it'll login as the very first step in any test. We don't need to change any of our existing tests using this technique.

In `CRMOperatorSteps`, therefore, the first lines should be as follows:

```
class CRMOperatorSteps extends CRMGuestSteps
{
    public $username = 'AnnieManager';
    public $password = 'managerpass';
```

These steps can be the performed in a similar manner for the other `CRM...Steps` classes.

We need to make five changes in our existing tests nevertheless.

The first change is in `LoginAndLogoutCept`. After `AcceptanceTester\CRMUsersManagementSteps` finishes creating a new user, this user is obviously a guest, as we don't have any RBAC management UI. So, the user who will be doing real testing of the login and logout functionality should not be associated with `AcceptanceTester\CRMUserSteps`, but `AcceptanceTester\CRMGuestSteps`. Also, `AcceptanceTester\CRMUsersManagementSteps` should log out after it finishes work.

> Now, it's clear that it's very hard to relate the notion of *step objects* enforced by Codeception with the intuitive understanding of how we should treat the subclass of `AcceptanceTester`. Why did they not name the inheritors of Tester classes a *TesterKinds* or something like that?

We'll not show the full final code, as it's quite easy to implement in `tests/acceptance/LoginAndLogoutCept.php`:

```
$I = new AcceptanceTester\CRMUsersManagementSteps($scenario);
// … steps to create a user ...
$I->logout();
$I = new AcceptanceTester\CRMGuestSteps($scenario);
// … steps to check the login/logout features.
```

The second change is that we have the following lines in `LoginAndLogoutCept.php`:

```
$I->amGoingTo('logout from arbitrary page');
$I->amInQueryCustomerUi();
$I->click('logout');
```

Given our new access control rules, guest users cannot go to a arbitrary page, so this test is meaningless for now. Just remove the preceding code lines.

The third change is in the same `LoginAndLogoutCept.php` file. Here is the line:

```
$I->seeLink('logout');
```

This line is there so we will be able to check whether we logged in successfully. After that, there are another four test steps that are there to check that `logout` actually works. But for that, we need to actually log out! Put the following line after that `seeLink()` invocation:

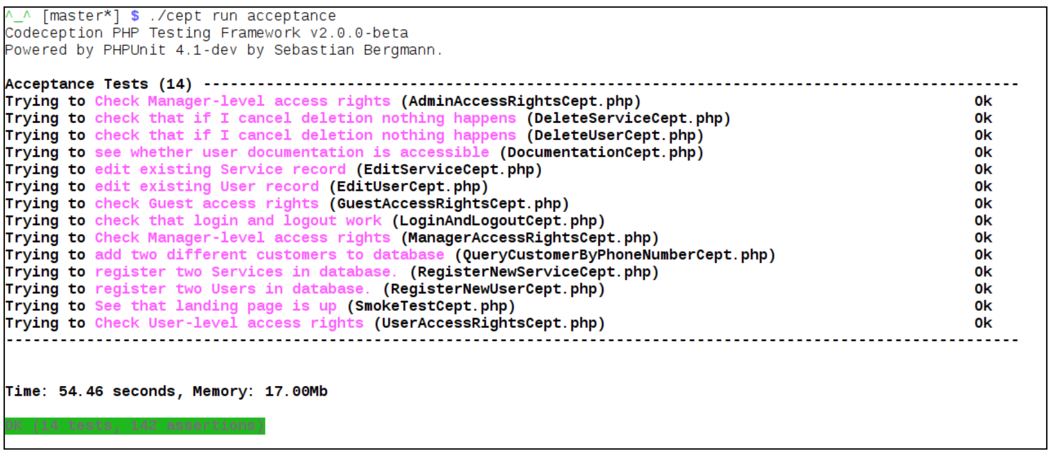
```
$I->logout();
```

The fourth change is that you must move definitions for `seeIAmInLoginFormUi()`, `fillLoginForm()`, `submitLoginForm()`, `seeIAmAtHomepage()`, `seeUsername($user)`, and `dontSeeUsername($user)` to the `AcceptanceTester\CRMGuestSteps` class from `AcceptanceTester\CRMUserSteps`.

The last change we should make is to log out from `QueryCustomerByPhoneNumberCept` after we're done creating a customer:

```
$I = new \AcceptanceTester\CRMOperatorSteps($scenario);
// … steps to create a customer ...
$I->logout();
$I = new \AcceptanceTester\CRMUserSteps($scenario);
// … steps to try the "query by phone number" feature.
```

This is the end of our implementation. Run the tests as shown in the following screenshot:



Success! More than that, we now have a sufficiently reliable automatic test suite which will relieve us from the job of clicking through these login forms manually.

Summary

In this chapter, you learned about a number of features of Yii that help us control the access to the parts of our application. Also, you learned that behind the scenes, Yii silently protects us from the CSRF attacks, which is quite cool actually.

In addition to this, we saw how exception handling is being employed by Yii 2 and looked at the special exceptions based on `HttpException`, which shows the appropriate response with the proper HTTP status code to the user.

You learned about the concept of controller action filters that help us to build combinations of stuff we can do before the controller action fires.

The most important part of this chapter is the building of the access control policy based on user roles. We saw how the RBAC manager's application component works together with the `AccessControl` filter class to provide such functionality. Finally, when implementing the testing harness for access control, we saw the intricacies of the automated authentication in the end-to-end tests and simple methods of overcoming them. There can be entire books written about full-featured solutions.

In the next chapter, we'll finally unfold the fundamental basis of the Yii 2 structure, the Modules system, which are the embodiment of the Model-View-Controller technique.

7
Modules

This chapter will unveil at last the concept of modules, which we mentioned in almost all the previous chapters. Modules impact the working of every feature of Yii, from the level of controllers to the level of views. However, it's possible that in small- to medium-sized web applications you'll never use them (apart from the single exception that we'll describe shortly).

First, we'll explore what the Module concept is and how it's implemented in Yii 2. After that, we'll rearrange things in our example CRM application and add a special section exclusively to report API using the Modules feature.

Let's start.

FEATURE – Yii modules

A **module** is the entity that has its own views, controllers, components, and possibly other modules. So, it's basically the embodiment of the MVC composite pattern.

As you probably guessed, the root Yii application is an example of a module.

> The official documentation is a bit misleading when it states that a module is like an application, but can't be used alone. It's really a clunky explanation. Quite the contrary, an application is a special kind of module that is enhanced with the features that allow it to run, just like the entry point of a Java application is not just any class but the class holding the `main()` method. This is clearly expressed in the `\yii\base\Application` class that extends the `\yii\base\Module` class, adding several methods to it.

What is meant by "has its own views"? It means that a module has the settings to set all the following entities:

Settings	Meaning
`controllerPath`	This is the path to the folder that contains all the controller classes that should be reachable from this module. This property is read only and is being automatically set based on the `controllerNamespace` setting.
`controllerNamespace`	This is the fully qualified name of the namespace in which all controllers from `controllerPath` should belong. This will be used to calculate the `controllerPath` setting according to the PSR-4 specification (visit `https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md`).
`controllerMap`	This is an array that specifies precisely the controllers that should be reachable from this module. Each key-value pair is the controller ID and either the array with controller properties or the controller instance itself.
`basePath`	This is a path alias to the folder that will be the default base for all relative paths inside this module.
`viewPath`	This is a path alias to the folder that will hold all the view files for controllers in this module.
`layoutPath`	This is a path alias to the folder that will hold all the layout files for this module.
`layout`	This is a path alias to the view file that will be treated as the layout for all view files in this module. If relative, it's expanded relative to the `layoutPath` setting. If this setting is not set, the layout setting of parent module will be used. If it ultimately resolves to a `false` value, no layout will be applied (view files will be rendered raw).
`modules`	This is an array of other modules that should be reachable from this module.
`components`	This is an array of components that are available in this module.
`module`	This is the parent module that has the current module in its `modules` array.

This table uses a specific term: reachable. It's used for our own convenience and is related to the routing system employed by Yii. This really helps us to understand how the MVC pattern in Yii influences the system.

The informal concept of reachability

Let's look at the controller action we introduced to show the list of services in YAML format back there in *Chapter 4*, *The Renderer*. It's implemented as the `actionYaml()` method inside the `ServicesController` class, which is in the `app\controllers` namespace, so its fully qualified name is `app\controllers\ServicesController.actionYaml()`. When someone requests the URI `http://yourdomainname.dom/services/yaml`, Yii 2 will call this exact `actionYaml()` method. This is possible for two reasons:

- The application has the namespace `app\controllers` set in the `controllerNamespace` setting (we did not set it explicitly, but it's the default value)
- By convention, any public method of a controller whose name starts with `action` is treated as the controller action (this is called the inline action)

Based on this, we say informally that the `actionYaml()` method is *reachable from* the `ServicesController` class and, subsequently, from the application instance because we can essentially call this method by accessing some known URI.

Note that the application has the `controllerNamespace` setting too, and this is because the application is just another module in the system.

The modules add another step to the previous description of *reachability*. If some controller is inside the namespace mentioned in the `controllerNamespace` setting of some Module class in the system, its actions will be reachable by the URIs only if this module is registered in the `modules` setting of the application or in the `modules` setting of another module.

> Thus, basically, the routing mechanics that we'll inspect in detail in *Chapter 12*, *Route Management*, boils down to the following URI pattern:
>
> `/moduleid/moduleid/.../moduleid/controllerid/actionid`.
>
> Of course, there is a lot of detail, such as the ID conversion rules, non-inline actions, and so on, but its core doesn't change.

Exploring the intricacies of module configuration through simple examples

At the bottom of it all, a module is just a PHP class extending the `yii\base\Module` class. This class has to be registered in the `modules` setting in either the application itself or some other module. Due to the referencing of modules by the fully qualified name, the exact location of the file with the definition of this class can be arbitrary. More than that, the existence of all the preceding configuration variables, such as `basePath`, `viewPath`, and others, allows us to have great freedom in choosing the physical structure of our application.

While we're on the topic of the physical structure of an application, let's make a really, really simple hierarchy of modules. It will be constructed in such a way that `SiteController` and other controllers under the `app\controllers` namespace will be reachable not only by the default URLs but also from the URL containing the chain of calls to modules. This will become clearer in a while.

> Remember that Yii 2 uses a PSR-4-compliant autoloader and the `app\` namespace corresponds to the root directory of the application. So, the `app\controllers` namespace maps to the `controllers` directory under the root of the code base. This, in turn, corresponds to the `@app/controllers` path alias of Yii 2.

We'll utilize the `app\utilities` namespace for our design. We created this namespace in *Chapter 4*, *The Renderer*, to hold the custom view renderer and response formatter. Let's create the module `FirstModule` there with absolutely minimal definition. Here is the full content of the `utilities/FirstModule.php` file we require:

```
namespace app\utilities;
use yii\base\Module;
class FirstModule extends Module { }
```

> It's better if you perform the changes described in this section in a separate branch of your version control system. It's all a throwaway code.

Congratulations, you've just created your first module. It's absolutely basic though, because of the following reasons:

- It thinks that its `basePath` is the folder it is in
- It thinks that views and layouts are in the `views` and `views/layouts` subdirectories of its `basePath`

- It thinks that it has controllers reachable in the `controllers` subnamespace under its own one
- It's attached neither to the application nor to some other module

This module will be attached to the application using its `modules` config setting as follows:

```
'modules' => [
    ...
    'firstlevel' => [
        'class' => 'app\utilities\FirstModule',
    ]
]
```

So, we attached the module with the ID `firstlevel` to our application. This ID is arbitrary and will be used to construct the URLs for us later. The class representing this module is `app\utilities\FirstModule`. Now create the second module in the same way, and name the class as `SecondModule` this time.

This second module will be attached to the `firstlevel` module in the application config. It's possible because the config will be processed recursively. Modify the `modules.firstlevel` configuration setting as follows:

```
'modules' => [
    ...
    'firstlevel' => [
        'class' => 'app\utilities\FirstModule',
        'modules' => [
            'secondlevel' => [
                'class' => 'app\utilities\SecondModule',
            ]
        ]
    ]
],
```

Each module has its own `modules` setting configurable identically.

It'll be too boring to add a third module in the same manner. Let's add it more dynamically using the special `init()` method of the `Module` class, which is called at the end of the constructor. Include the following code in the `SecondModule` class:

```
public function init()
{
    parent::init();
    $this->modules = [
```

```
            'thirdlevel' => [
                'class' => 'app\utilities\ThirdModule',
                'basePath' => '@app'
            ]
        ];
    }
```

When overriding the `init()` methods of the base Yii 2 classes, do not forget to always call `parent::init()`. There's usually a lot of stuff happening behind the curtains for you.

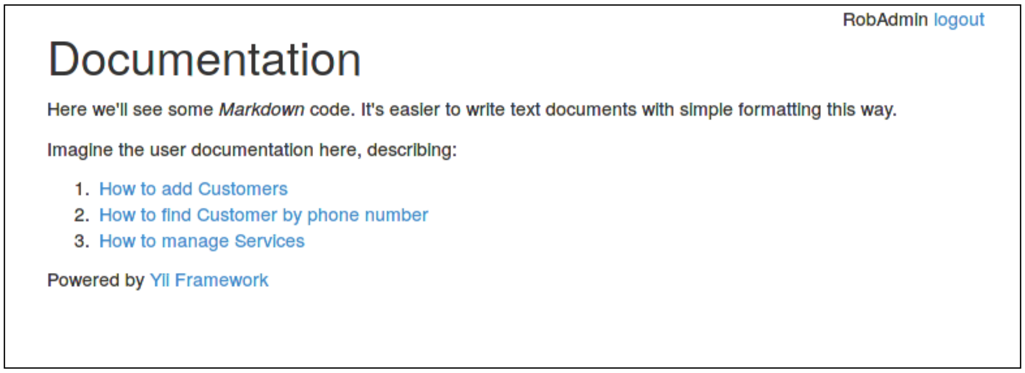
The preceding `init()` method will do essentially the same as putting the configuration array for `ThirdModule` into the application config, but the binding of a module will happen a bit later this time. Of course, you also have to create the `@app/utilities/ThirdModule.php` file with the definition of `ThirdModule`.

We specified the `basePath` of `ThirdModule` to be the root directory of our application. By doing this, we effectively converted this module into the application itself in terms of locating views and controllers, with one exception. To complete the unification, we have to reset the `ThirdModule.controllerNamespace` as follows:

```
public $controllerNamespace = 'app\controllers';
```

Do note that we are using the strings that contain namespace declarations extensively in the Yii application configuration. As namespaces in PHP have to contain backslashes, we should take care that this is the escaping character; if our subnamespace starts with something like `n`, `v`, or `t`, we'll get the escape sequence (`\n`, `\v`, or `\t`, according to our example) in the middle of the namespace declaration, which can lead to nasty untraceable errors. The simplest way to prevent this is to develop a habit to always use single quotes for string literals in PHP that do not expand escape sequences. You can also escape the backslash itself, by writing `"\\"`, if you absolutely must use double quotes in your string literals. The Yii 2 source code uses exactly this method; however, it's way easier to just use single quotes.

Now let's remember that we built a nice documentation viewer in *Chapter 4*, *The Renderer*, at the URL `/site/docs`. Let's open the `/firstlevel/secondlevel/thirdlevel/site/docs` route now:



Except the authentication indicator built in later chapters, it's identical to the `/site/docs` output. In fact, it's the exact same `app\controllers\SiteController.actionDocs()` method, reachable from two URLs now. All other controllers from the `app\controllers` namespace got this trait as well.

As we're using the default values for the `viewPath` setting in our `thirdlevel` module, it is assumed to be relative to the `basePath` of the module `ThirdModule`. If we do not redefine it in the configuration for the `secondlevel` module, we would get the following message when accessing the `/firstlevel/secondlevel/thirdlevel/site/docs` route:

As was said before, the default `basePath` of the module is the folder its class definition is in. In the preceding error message, it can be seen that when we do the following inside the `SiteController`:

```
return $this->render('docindex.md');
```

We are referring to different files depending on what module we are currently in and its `basePath` setting. The root directory of the code base is `/vagrant` in case you set up your deploy remote machine using the **Vagrant** toolset (`http://www.vagrantup.com/`), as described in *Appendix A*, *Deployment Setup with Vagrant*.

Now everything referring to modules in the previous chapters should become clear to you. When we treat the Yii application as a single entity, we just throw away the possibility that controllers and views (and to some extent models too) can be grouped together and everything is reachable by the URIs in the following form:

```
/controllerid/actionid
```

However, even this is not true, as we will learn in *Chapter 12*, *Route Management*.

Breaking up the application into modules using only the default Yii-implied values of `basePath`, `viewPath`, `layoutPath`, and `controllerNamespace` can be either really tricky and unmaintainable or really simple and understandable depending on how you physically structure your modules. You can completely omit the redefinition of these settings if the following prerequisites are met:

- Everything related to a single module is inside a single subdirectory and nothing else is there
- This module is inside the namespace that is mapped to this subdirectory; thus, the file containing its definition is inside this subdirectory as well

We'll rely on this trick in the very next section, when we'll actually use the feature of modules to rearrange stuff in our example CRM application more nicely.

The Debug module

Apart from the Gii automatic code generator, Yii 2 has another built-in mechanism: the Debug module. Like Gii, it is separated out as an extension (more about Yii 2 extensions in *Chapter 9*, *Making an Extension*) and is attached to the application as a module.

The Debug module was already seen as far back as *Chapter 1*, *Getting Started*, when we installed the basic application template. That module provided a nice debug toolbar at the bottom of each page on the site.

As this requires some additional steps, we have not enabled the same toolbar at our example CRM application yet. Now it's time to do it.

> In fact, we are repeating the installation instructions from the `yii2-debug` repository (visit `https://github.com/yiisoft/yii2/blob/master/extensions/debug/README.md`). Our purpose is not only to use this extension but to also understand how it works.

Add the dependency on the `yiisoft\yii2-debug` package to our example CRM application as follows:

```
$ php composer.phar require --prefer-dist yiisoft/yii2-debug "*"
```

Wait a bit until it ends rebuilding. This package provides the special `yii\debug\Module` class that we need to attach to our application as usual:

```
'modules' => [
    ...
    'debug' => [
        'class' => 'yii\debug\Module',
    ]
],
```

That's not all, though. We also need to force Yii 2 to initialize this module at the same time as the application itself, that is, to *bootstrap* it:

```
'bootstrap' => ['debug'],
```

You can check the documentation for the `yii\base\Application.bootstrap` property to see how it works in detail.

> Almost everything in Yii is created "lazily", on demand, usually in the getter method. This has a consequence of initialization code for some components being run too late or at an inappropriate time, resulting in the slowdown of the application. By bootstrapping a component, you prohibit the lazy behavior and force the component to be initialized at the start of the application, moving its initialization at the expected time frame.

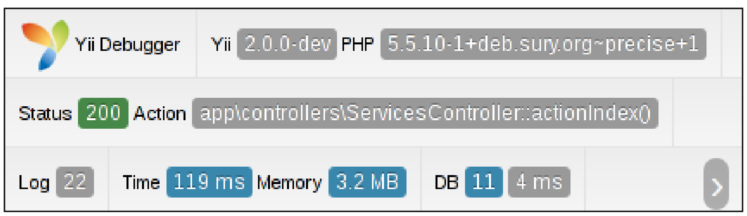
> If you are deploying the application to a separate machine, the Debug module will not be visible to you, as it is protected by the `\yii\debug\Module::$allowedIPs` property, whose default setting is to listen only to localhost. You may want to set an additional configuration for the Debug module, as shown in the following, for example:

```
'debug' => [
    'class' => 'yii\debug\Module',
    'allowedIPs' => ['IP of your development machine']
]
```

In the preceding information block, by "component" we mean not the application component concept explained in the previous chapters but the basic building block of Yii 2, the `yii\base\Component` class, which can be described roughly as the "`stdClass` on steroids". This class is the base for a lot of built-in classes in Yii 2, including the `Module` class. We'll not delve into the details of this class separately, because features of the concept of the component are spread through the whole Yii 2 framework, especially the event handling and behaviors system (which is implemented in the `yii\base\Component` class).

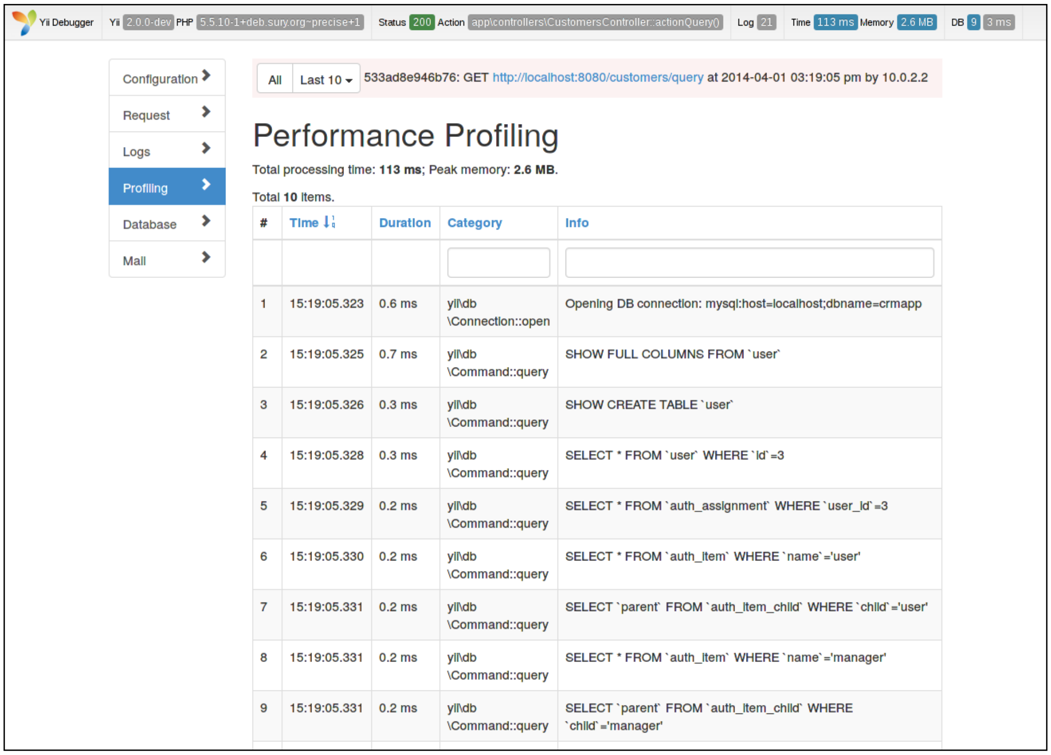
Anyway, let's return to the issue of bootstrapping. Why do we need to bootstrap the Debug module? Isn't the module just the proxy for reaching the controllers?

It turns out that as long as the current module has the `init()` method, it can also be used for other tasks apart from its initialization. The Debug module needs to be bootstrapped to present you with the debug toolbar at the bottom of the page. It does this by listening on the `View::EVENT_END_BODY` event and registering its assets when this event is triggered, which (as you have probably guessed right) is when the view ends rendering the body of the page. This is what the debug toolbar looks like:



In fact, in this case the browser window was intentionally shrunk horizontally, so the panel collapsed into three rows. Normally, it shows all the indicators on the same row. All the indicators presented are the links to the respective sections of the full Debug module interface.

If you do not bootstrap this module, you'll still be able to access its full version at the `/debug` route, but the panel at the bottom of other pages will not be rendered. That's what the performance section of the Debug module looks like:



Do you remember that when we attached the Gii module to our application, we did not bother with anything like bootstrapping? It's precisely because the Gii module is indeed just a proxy for a set of controllers that is packed into a separate folder. It does not do anything invasive with your application (at least until you hit the button, of course). You should understand this trick with `init()` because when you'll develop your own components and modules it can help you make more robust plugins for the applications.

Now let's make some meaningful module in our application ourselves.

Building the API module

We have two controller actions currently in our system, which:

- Are not needed for the user interface
- Provide the API-like representation of the information in the system

All this makes these actions perfect targets for our practice in building modules. Let's make an API module that will contain the actions currently reachable by the `/services/json` and `/services/yaml` routes.

The following use cases should be supported:

- The GET request to the `/api/services/json` route should return the list of attributes of all registered services in the JSON format
- The GET request to the `/api/services/yaml` route should return the list of attributes of all registered services in the YAML format

We define `result in JSON format` as the string that can be converted to the PHP data structures without error using the `yii\helpers\Json::decode()` method.

We define the `result in YAML format` as the string that can be converted to the PHP data structures without error using the `Symfony\Component\Yaml\Yaml::parse()` method.

Building a test suite to support testing the API module

We will not make a REST-compliant API here. We will get just the minimal reader API to provide us with the list of records registered in the database. However, to use the end-to-end acceptance tests as we did all the time until now is an overkill in this case. We need some simple way to perform the following actions:

- Install some known data to the database
- Poll the API endpoint for the data
- Ensure the data returned is indeed a serialized representation of the data from the database

Codeception, which we are using as our test harness, has a limitation here. On one hand, it has the REST module that provides exactly what we need: the `sendGET()`, `canSeeResponseCodeIs()`, and `grabResponse()` methods (their names tell it all). On the other hand, the REST module cannot be used together with the WebDriver module that our acceptance test suite is using.

Let's just create a separate test suite just for testing API endpoints. Run the following autogenerator:

```
$ ./cept generate:suite api ApiTester
```

We have the `tests/api` directory and the `tests/api.suite.yml` configuration now.

To properly configure this suite, we should understand that these tests will make requests to the web server serving the application, but we must have a connection to the database to be able to write the data we control. So, we have to run the test suite from within the deploy target machine, such as the functional tests. However, unlike the functional tests, the API test suite requires that the full application environment together with the web server is up and available, so it's more like the end-to-end tests in this regard. This will be quite tricky to implement on a testing server.

> It is obvious now that we wrapped ourselves into the tight mess of the high-level tests that require the runtime environment to be prepared and the application deployed and running. This setup is slow to run and hard to implement reliably. This is because we rely too much on the Yii machinery in our code, having stopped decoupling from the framework for simplicity in *Chapter 3*, *Automatically Generating the CRUD Code*. As a rude workaround, we could completely drop the requirement of having the Yii application up and running by testing the controller action methods directly, as they return the result of their work, and not (for example) print the result directly to `STDOUT`. However, this will make us unable to see integration problems with Yii in a production environment. In fact, you should have the multilayered test harness, spanning from unit tests of the business rules up to the end-to-end acceptance tests using the UI in an abstract way. To restrict the span of this book, we are showing only the highest-level tests for you here. Fine-grained refactoring was omitted for the same reasons. Although its name has two components, this book is more about Yii 2 and less about the web development in general after all.

Knowing that we'll run the API tests on the deploy target, and therefore call the web server by the loopback interface, we are now able to properly configure the API test suite by means of `tests/api.suite.yml`:

```
class_name: ApiTester
modules:
    enabled:
      - ApiHelper
      - PhpBrowser
      - REST
      - Db
    config:
```

```
        PhpBrowser:
            url: 'http://localhost'
        REST:
            url: 'http://localhost'
```

Let's discuss the highlighted lines from top to bottom:

- The `REST` module requires the `PhpBrowser` module as the transport provider, so it'll be able to actually send requests to the application.
- We need the `Db` module to reset the database after each test run, as we'll autogenerate the data to be present in the database.
- The `PhpBrowser` module forces us to provide the base URL for the application. This will be the loopback interface at the same machine. The web server must be running and be serving our application.
- The `REST` module should have the means to access the application, so we provide the base URL for it. In our case it's clearly a code duplication, which is unavoidable. This is not so in a general case, as the `REST` module can be configured to access some specific subroute defined to be the API endpoint, and thus all requests routes specified in our tests can be shortened by this API endpoint prefix.

As we're using the `Db` module, we need to configure it to be able to access our database. We already did this for the functional test suite, in the `tests/functional.suite.yml` file. So, to avoid code duplication, we can just move the whole `modules.config.Db` section from the `tests/functional.suite.yml` file to the global `codeception.yml` file at the root of the code base. However, if your functional suite is configured to be connected to a database that is different from the one your application is using, then you should *not* do this and instead configure the `Db` module in the API suite individually.

This is not all. We need to instantiate the Yii application for our test cases to be able to utilize the database connection from application classes. We'll do it inside the generated `tests/api/_bootstrap.php` file in the same way it was done for the functional test suite:

```
require_once(__DIR__ . '/../../vendor/autoload.php');
require_once(__DIR__ . '/../../vendor/yiisoft/yii2/Yii.php');
new yii\web\Application(
    require(__DIR__ . '/../../config/web.php')
);
```

We're creating a web application again.

> Do not forget that in the case of API, as well as acceptance suites the instance of `yii\web\Application` and the one that we made requests to are completely separate entities and you cannot hope to share data between them in any way except the filesystem and the database (or maybe some other trick from the domain of crazy professional workarounds). Be very careful when configuring the API suite that the Db module in Codeception and the application itself will be talking with the same database!

Don't forget to generate a tester class for the API test suite by running the `build` command:

```
$ ./cept build
```

Defining the requirements for automatic tests of API modules

Now, the preparations are complete and we can create the actual test script:

```
$ ./cept generate:test api ServicesListApi
```

We're essentially making the API for managing the list of services, after all.

It'll not be the cept-style test, written as a prose, but a normal test case that is written as a PHP class. That's how the requirements can be defined as a Codeception test:

```
/** @test */
public function ReturnsValidJson()
{
    $expectedData = [];
    $expectedData[0] = $this->registerService();
    $expectedData[1] = $this->registerService();

    $this->tester->sendGET('/api/services/json');

    $response = $this->tester->grabResponse();
    $responseData = \yii\helpers\Json::decode($response);

    $this->assertInternalType('array', $responseData);
    $this->assertEquals($expectedData[0], $responseData[0]);
    $this->assertEquals($expectedData[1], $responseData[1]);
}
```

For the sake of simplicity, we are going for relatively large steps here. We register two services in the database to test the ability of our code to deal with nontrivial datasets.

Then, we send the GET request to our predefined API endpoint and check whether we get the JSON response with the data we have just saved to the database.

The test for the YAML endpoint will be exactly the same, except the API endpoint will end in `yaml` instead of `json` and the decoding will be done by a different class. So instead of the line:

```
$this->tester->sendGET('/api/services/json');
```

We will use the following line:

```
$this->tester->sendGET('/api/services/yaml');
```

Also, instead of using the line:

```
$responseData = \yii\helpers\Json::decode($response);
```

We will use the line:

```
$responseData =
  \Symfony\Component\Yaml\Yaml::parse($response);
```

However, before that, let's define the smoke test to check whether we actually have the endpoints we want:

```
/** @test */
public function HasJsonEndpoint()
{
    $this->tester->sendGET('/api/services/json');
    $response = $this->tester->grabResponse();

    $this->tester->canSeeResponseCodeIs(200); //1
    $this->assertNotEquals('', $response); //2
}
```

We're asserting the following points:

- The response code is `HTTP 200 OK`
- The response body is not empty (it should never be empty as the Yii error handler sends quite a long HTML body for error pages)

The same applies for the `HasYamlEndpoint()` test.

Before we make the actual module, let's deal with the `ServicesApiTest.registerService()` helper method we glossed over. The idea behind the reader API is that it returns a plain JavaScript object, a set of key-value pairs. So, we certainly do not want the full serialized `yii\db\ActiveRecord` returned from there, but rather just its attributes. Given all that, we define the following helper method:

```
private function registerService()
{
    $service = $this->imagineService();

    $service->save();

    return $service->attributes;
}
```

So we both have a cake and eat it: a record is saved into the database and we know its attributes that should be returned by the API endpoint.

We imagine a service with the help of the `Faker` library, as usual:

```
private function imagineService()
{
    $faker = \Faker\Factory::create();

    $service = new \app\models\service\ServiceRecord();
    $service->name = $faker->sentence($words = 3);
    $service->hourly_rate = $faker->randomNumber($digits = 2);

    return $service;
}
```

It's the same method as in the acceptance tests for services management. We run the tests and we get the following output:

```
vagrant@precise64:/vagrant$ ./cept run api
Codeception PHP Testing Framework v2.0.0-beta
Powered by PHPUnit 4.1-dev by Sebastian Bergmann.

Api Tests (4) ------------------------------------------------------------------------
Trying to has json endpoint (ServicesListApiTest::HasJsonEndpoint)              [F]Ok
Trying to returns valid json (ServicesListApiTest::ReturnsValidJson)            Error
Trying to has yaml endpoint (ServicesListApiTest::HasYamlEndpoint)              [F]Ok
Trying to returns valid yaml (ServicesListApiTest::ReturnsValidYaml)            Error
--------------------------------------------------------------------------------------


Time: 1.2 seconds, Memory: 16.75Mb

There were 2 errors:

---------
1) ServicesListApiTest::ReturnsValidJson
yii\base\InvalidParamException: Syntax error.

#1  /vagrant/tests/api/ServicesListApiTest.php:30

---------
2) ServicesListApiTest::ReturnsValidYaml
Symfony\Component\Yaml\Exception\ParseException: Unable to parse at line 1 (near "<!DOCTYPE html>").

#1  /vagrant/vendor/symfony/yaml/Symfony/Component/Yaml/Yaml.php:67
#2  /vagrant/tests/api/ServicesListApiTest.php:57

--

There were 2 failures:

---------
1) ServicesListApiTest::HasJsonEndpoint
Failed asserting that 404 matches expected 200.

#1  /vagrant/tests/api/TestGuy.php:1463
#2  /vagrant/tests/api/ServicesListApiTest.php:16

---------
2) ServicesListApiTest::HasYamlEndpoint
Failed asserting that 404 matches expected 200.

#1  /vagrant/tests/api/TestGuy.php:1463
#2  /vagrant/tests/api/ServicesListApiTest.php:43

FAILURES!
Tests: 4, Assertions: 4, Failures: 2, Errors: 2.
```

Of course, four failing tests at once is a huge step in the **test-driven development** (**TDD**) process. However, first of all, you're not writing it yourself and are following a guide, and secondly, all these failures are only because of a simple problem: we don't have anything on the `/api/services/json` and `/api/services/yaml` routes.

Moving the controller actions to a separate module

We have everything we need already implemented; it just needs to be moved at appropriate places. We'll utilize the power of Yii 2 conventions here, starting from the routes we require to be available. Let's reverse-engineer the project structure, so we don't need any special configuration aside from the defaults.

We want the `/api/services/json` route to be available. This means that we need the following three entities:

- A module with ID `api`, that is, some `yii\base\Module` descendant registered in the application config with the `modules.api` key.
- A controller with ID `services`, that is, a descendant of `yii\web\Controller` named `ServicesController` (ID is parsed automatically from the name of the class), reachable from the module with the `api` ID.
- A controller action with the `json` ID, reachable from the controller with the `services` ID. We'll explore some other options in *Chapter 12*, *Route Management*, but in the simplest case it means that we need the method called `actionJson()` defined in the `ServicesController` class.

The default conventions of Yii 2 expect each module to reside in a separate directory. Let's make the `api` directory and place the minimal `ApiModule` definition in there:

```
namespace app\api;
use \yii\base\Module;
class ApiModule extends Module {  }
```

Because we are conforming to PSR-4 in the names of directories and namespaces, we don't need anything special for the `ApiModule` class to be accessible in the code base. The class definition is completely empty, as there's no mandatory configuration for a module.

As our `ApiModule` is in the `app\api` namespace, it will automatically assume that its `controllerNamespace` is in `app\api\controllers`. Let's not disappoint it and create this directory together with the `ServicesController.php` file in there. This file will obviously contain the `ServicesController` class definition, as follows:

```
namespace app\api\controllers;
use yii\web\Controller;
class ServicesController extends Controller
{
    // empty for now
}
```

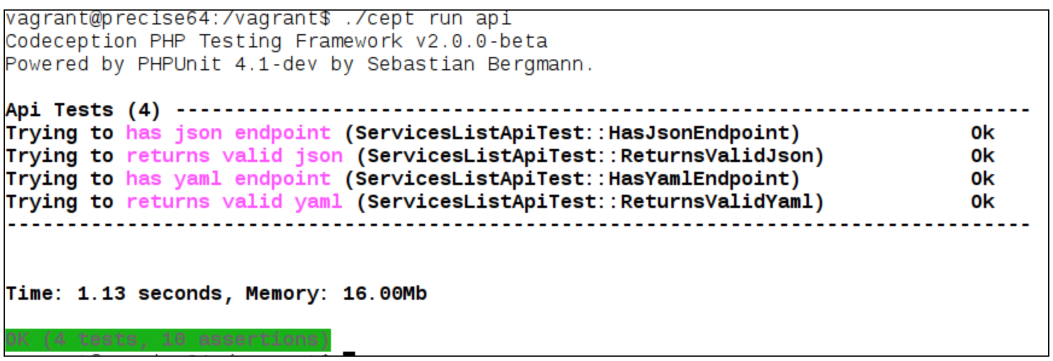
We have two out of three steps. Now all we need is to move the `actionJson()` and `actionYaml()` methods from the `controllers/SiteController.php` file to this new class definition.

> Note that we have two classes with the same names in our code base. Usually, that was a really big problem back in Yii 1.x, but now we have namespaces everywhere. As long as the fully qualified names of classes are different, we're totally OK.

Now we have the complete API module (trust us here). All we need to do now is to tell the Yii application that it exists by placing its declaration in the config inside `modules` as follows:

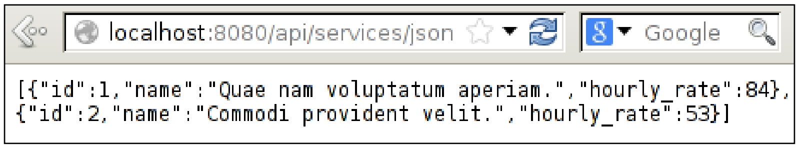
```
'modules' => [
    ...
    'api' => [
        'class' => 'app\api\ApiModule'
    ]
],
```

Now, we run the tests and all four of them pass (provided you really have the actual actions for JSON and YAML serialization implemented according to *Chapter 4, The Renderer*):



> As was said in the beginning of the *Building the test suite to support testing the API module* section, this API module example is not REST-compliant. If you want a REST API, you should check the built-in class `yii\rest\Controller` for the controller specifically tailored for REST API requests and the whole `yii\rest\*` family of classes in general.

If you open the JSON endpoint in the browser (after the API test suite has dutifully added some records to the database), you'll see the following:



There is no simple way to present a screenshot to you from a YAML endpoint, as all browsers will try to download the response from the server instead of presenting it in a textual format on the screen.

Retrospective on the modules mentioned in previous chapters

Let's remember when we mentioned modules previously and hand-waived them for simplicity.

We mentioned modules in *Chapter 2*, *Making a Custom Application with Yii 2*, when we created the configuration file for our application and discussed that the `id` setting is mandatory. You have learned that the application is just a special kind of module and all modules have an ID. For a module, this ID is defined in the `modules` configuration setting when you assign a distinct key for the configuration array of this module. In the API module example that we just implemented, we used the `api` ID. The application itself, of course, is not listed in any `modules` section anywhere so it must get its ID in some other way. To be honest, an application needs an ID for different reasons than a module (it's a *special* kind of module, after all), but it's mandatory anyway.

Later in the same chapter, we noted that we ended up with the project structure surprisingly similar to the basic application template from Yii 2. Now you understand why it is, as we discussed the various configuration settings and conventions in full in this chapter.

In *Chapter 3*, *Automatically Generating the CRUD Code*, we actually *used* the module without knowing what it is. We learned back then that whatever it is, the module is something that is a fully qualified class name you declare in the `modules` section of the configuration. After declaration, you receive a set of additional routes available in your application for browsing. Also, you can note now that we can set values for the properties of the modules right when declaring them in the configuration. This will not help when you need to configure the module at runtime, depending on some data not known at the stage of constructing the application (in this case, your only option is overriding the `init()` method or hooking onto events), but it can really help you when developing some prepackaged, reusable modules, such as Gii itself.

> Please understand that everywhere when we talk about Modules (with the uppercase M at the start of word), we're talking about the specific concept of the Yii 2 framework, implemented as the `yii\base\Module` class. Codeception has a completely different notion of a *module*, which does not help in any way in understanding this important term.

In *Chapter 4*, *The Renderer*, when discussing how Yii 2 finds the view files for the `render()` method calls, we finally came to the point when we have to really pretend that modules do not exist at all to simplify the explanations. Now, we know that the `viewPath` under which the View component searches for view files can be defined individually per module. In fact, this setting *is* defined individually by default. As an application is a module, it has its own `viewPath`; however, when we're executing a controller action that belongs to the controller reached through some module, this module's `viewPath` is used instead of the application's. The same rules apply to the `layoutPath` setting, and the `layout` setting was already discussed in the same table in the same section.

> As a result, if you are rearranging the controllers and especially when rearranging the actions, you have to be careful to move the corresponding view files between modules as well!

Apart from the importance for the view rendering, the system of modules greatly affects route handling. If you paid attention, you probably already noted that you can have arbitrarily deep filesystem-like URLs without any special treatment, just by defining a hierarchy of modules in the application config. However, that's the theme for *Chapter 12*, *Route Management*.

Summary

Yii 2 implements the MVC composite pattern using the Modules system. Each module has a means to reference and access the controllers and views. Modules are attached to the application and additionally you can attach them to other modules, creating the hierarchy.

By utilizing the system of modules, you can have an arbitrary number of arbitrarily deep routes available in your application in the following general form:

```
/moduleid/moduleid/.../moduleid/controllerid/actionid
```

This basic routing system will thus be constructed entirely by means of the PHP classes appropriately wired together by Yii 2 conventions. No additional customizations of the Yii 2 router will be required.

In the next chapter, we'll discuss the issues of security and performance. Yii 2 provides us with some means of housekeeping to make the UI of our application a bit more neat and confirming best practices without any serious work from our side.

8
Overall Behavior

In this chapter, we'll look at some features of the Yii 2 framework that are non-local in some sense. They will not help you implement new features per se, but without these features, your ability to make robust applications both from the developer and user points of view will be severely hindered.

Here is the list of features we will explore shortly:

- Logging messages
- Handling errors
- Caching
- Compiling assets

As usual, in this book, we'll concentrate more on the practical aspects of actually *using* these features, as for the precise technical details of their implementation you can look up either in the excellent Yii documentation or directly in the source code.

Let's start with how message logging is implemented in Yii 2.

FEATURE – message log

More often than not when developing and debugging an application, you want to know about how its internal state changes. In the web-covered, PHP-based world there's usually no habit to use the tools, such as **Xdebug**, to peek into the individual variables, like beard-wearing C programmers do. Most problems can be solved by thoughtfully placed print statements showing the values at the time they are used.

The Yii framework provides the facility to organize this peeking into the internal state and relieves you from writing `var_dump($variable);die()` everywhere. You can record messages from anywhere inside your application to some predefined, controllable, and centralized locations where they can be read without the need to intervene with the rendering process.

The **log message** concept of Yii 2 is an entity consisting of the following:

- A line of text with the message itself
- The severity level
- The category of a message
- The timestamp
- The stack trace at the time of recording a message

We have control only over the first three properties.

Obviously, you can write anything in the message itself, as long as it stays a string.

The severity levels are predefined, and while technically you can force Yii to record the message with a non-standard severity level, it's not guaranteed that it'll be handled like others. Here is the list of constants you can use as a severity level for log messages:

- `yii\log\Logger::LEVEL_TRACE`: This constant is used for messages that are only for debugging
- `yii\log\Logger::LEVEL_INFO`: This constant is used for general-purpose messages
- `yii\log\Logger::LEVEL_WARNING`: This constant is used for warning messages
- `yii\log\Logger::LEVEL_ERROR`: This constant is used for error messages

Except for `LEVEL_TRACE`, which you'll read about shortly, there is no enforced distinction between these levels. Of course, you better assign appropriate severity to the messages you log for the sake of your own sanity.

There are also pseudo-levels `LEVEL_PROFILE`, `LEVEL_PROFILE_BEGIN`, and `LEVEL_PROFILE_END`, but you should never concern yourself with them directly but use the helper methods to perform profiling of the Yii-based application. This will be shown in the next section, shortly.

The category of the log message is an arbitrary string that can be used for group-related messages. Inside its own methods, the Yii framework almost always uses the `__METHOD__` superglobal PHP constant as the category for messages, so it basically proposes that you group messages by the currently executing method's fully qualified name. As you'll see later in this section, this categorization scheme has an additional positive aspect.

The timestamp of the log message is set by the `microtime(true)` call, so it's the automatically calculated time at the moment the message is logged with the precision down to microseconds.

The stack trace of the log message is recorded from whatever the call to the built-in function `debug_backtrace()` returns. The recording of the stack trace is governed by the value of the `Yii::$app->log->traceLevel` property or, identically, the `components.log.traceLevel` setting in the application configuration. By default, the value of `traceLevel` is set to `0`.

> It should be obvious that as the stack trace is associated with *every* log message, setting `traceLevel` to anything above `0` will seriously decrease the performance of the application.

Yii itself logs quite a lot of messages at different stages of its life. You will probably not even need to record any additional messages. It's a lie of course, but we can hope for it to be true anyway, right? That's how you do recording of log messages in Yii 2. Have a look at the following table:

Method name	**Meaning**
`Yii::trace($message, $category)`	This method records the `LEVEL_TRACE` message in the given category. If the `YII_DEBUG` constant is set to `false`, this method won't record anything.
`Yii::warning($message, $category)`	This method records the `LEVEL_WARNING` message in the given category.
`Yii::error($message, $category)`	This method records the `LEVEL_ERROR` message in the given category.
`Yii::info($message, $category)`	This method records the `LEVEL_INFO` message in the given category.

As the `Yii::trace()` call silently does nothing if `YII_DEBUG` is set to `false` (which basically means that we are in a production environment), it is perfectly fit to record the state of some variables at some precise time for debugging purposes.

Actually storing the log messages

The four helper methods described in the preceding table record the log messages and do not persist the messages in any way. All they do is tell the message dispatching mechanism to send these messages to whatever log targets we as developers have configured for the application.

Let's look at an example application configuration that not only tells Yii to store log messages in the database, but to send `LEVEL_ERROR` messages right to the project manager's e-mail:

```
'components' => [
    'log' => [
        'traceLevel' => 3,
        'targets' => [
            'all_messages' => [
                'class' => 'yii\log\DbTarget',
                'levels' => ['info', 'trace', 'warning', 'error']
            ],
            'problems' => [
                'class' => \yii\log\EmailTarget::className(),
                'levels' => \yii\log\Logger::LEVEL_ERROR,
                'message' => [
                    'to' => 'pm@crmapp.us'
                ]
            ]
        ],
    ],
...
],
```

Note the highlighted parts. You will see that the `log` component of the application has the `target` array, and the class of the target defines the log message target at which the log message will be stored. The names of the log targets are arbitrary and provided only as a help for the maintainer.

> You can see two interesting conventions in the preceding configuration snippet.
>
> First, we can either use a string literal to specify the fully qualified name of the class or call `className()` on them, which is a special static method defined by Yii on virtually all of its classes that returns the fully qualified class name for us.
>
> Second, in the `components.log.targets.[].levels` setting, you can specify levels either as the array of literal names of severities or as the binary combination of the `Logger::LEVEL_*` constants as follows: `Logger::LEVEL_ERROR | Logger::LEVEL_WARNING` (binary OR in this case).

We have the following log message targets at our disposal:

- `yii\log\EmailTarget` sends the log messages as e-mail messages.
- `yii\log\FileTarget` writes the log messages to the file, with **log rotation** provided for free
- `yii\log\DbTarget` stores the log messages as records in the database. You must have the table configured there as said in the documentation for the `yii\log\DbTarget.logTable` property.
- `\yii\log\SyslogTarget` sends the log messages to the **syslog** system facility by means of the `openlog()`, `syslog()`, and `closelog()` PHP built-in calls.

The `EmailTarget` is a special beast. First of all, it does not store the messages in the usual sense, as it sends them to the remote e-mail address. Second, it requires the Mail component configured and attached to the application.

Setting up the e-mailing component so that log messages can be mailed

If we need to send e-mails from our application, Yii 2 ships with yet another extension, which brings the power of **SwiftMailer** (see `http://swiftmailer.org/`) for this task. As the other option would be to write the implementation of the`\yii\mail\BaseMailer` descendant ourselves, we better digress and just declare this extension as a requirement for our application:

```
$ php composer.phar require --prefer-dist yiisoft/yii2-swiftmailer "*"
```

This extension will bring us the `\yii\swiftmailer\Mailer` class, which we are able to use as the Mail component for our application. This is the example of a minimal configuration, which will use the `mail()` PHP built-in function as the transport mechanism:

```
'components' => [
    …
    'mail' => [
        'class' => yii\swiftmailer\Mailer::className(),
        'messageConfig' => [
            'charset' => 'UTF-8',
            'from' => 'noreply@crmapp.me'
        ],
        'transport' => [
            'class' => 'Swift_MailTransport',
        ],
    ],
    ...
]
```

The tricky parts of configuration are the highlighted ones.

The `components.mail.messageConfig` setting defines the attributes of all e-mail messages that will be sent by this Mail component. As the **From** field is mandatory in 21st century e-mail correspondence, we have to declare it explicitly. Yii does not help us here in any way. The UTF-8 charset, in the case of some systems, doesn't know that it's the year 20xx now and still uses something other than Unicode.

The `components.mail.transport` setting defines which transport method from the SwiftMailer package should, and will, be used. The most common are probably `Swift_MailTransport`, which uses the `mail()` PHP built-in function, and `Swift_SmtpTransport`, which can connect to remote SMTP servers with authentication and all that stuff. You are encouraged to sift through the SwiftMailer documentation to look at other pretty cool transport methods, such as the load-balanced and failover methods.

Reading the stored log messages

When the message is stored in the configured target, you'll need it someday. Here's how the message stored in the `FileTarget` looks:

```
2014/04/12 01:44:27 [10.0.2.2][3][pccg3bq84sm6b4mnutsa07lk31][trace]
    [yii\base\Controller::runAction] Route to run: site/index
```

The preceding message is split into several lines only to fit the book format. In the logfile, it's a single line of text. It can be seen that we have a timestamp, a series of tokens enclosed in brackets, and some text of the message itself. Here are the tokens being written:

- IP of the user
- User ID of the user or the `-` symbol if the user is not authenticated
- Session ID of the user or the `-` symbol if the user is not authenticated
- Severity level in textual representation
- Category of the log message

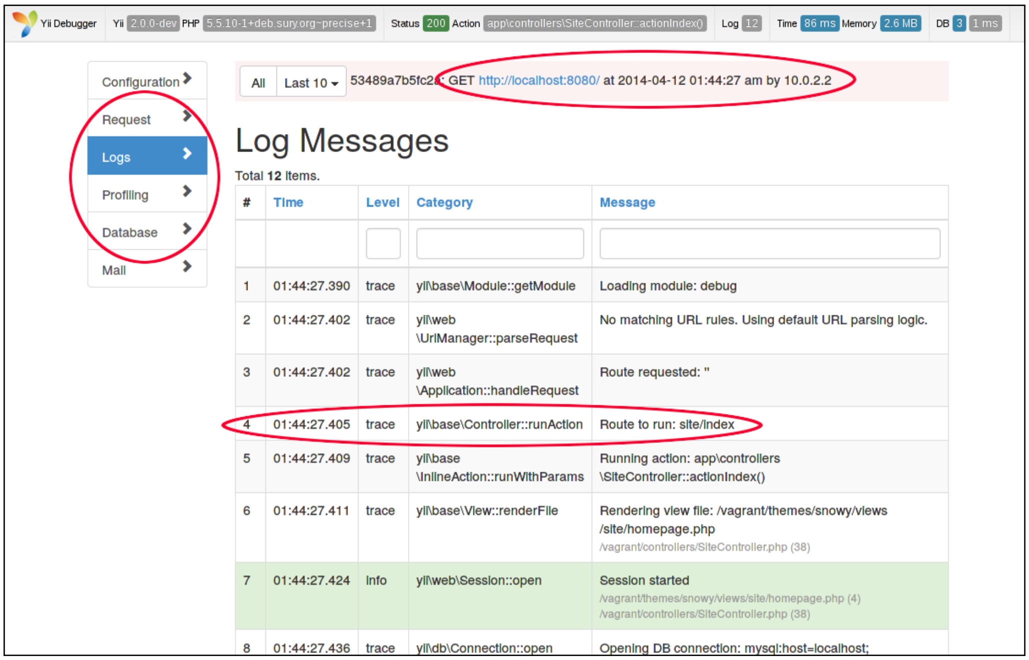
You can see that the category of default system messages is indeed just a fully qualified name of the method inside which this message was logged. Here's how the message from a non-authorized request looks:

```
2014/04/12 01:44:27 [10.0.2.2][-][-][error]
  [yii\web\HttpException:404]
    exception 'yii\base\InvalidRouteException'
    with message 'Unable to resolve the request "favicon.ico".'
    in /vagrant/vendor/yiisoft/yii2/base/Module.php:440
```

Again, it's all in one line in the log. Note that after the error-level messages, there's a stack trace printed (if the `components.log.traceLevel` was configured), in human-readable format. You should understand this so that your script parsing the Yii log output will be prepared to either skip all lines not starting with a timestamp or have some kind of elaborate stack-trace fetcher.

> You can greatly ease the parsing of logs, though, if you use `DbTarget`, which stores each token, including the text of the stack trace, separately in the database table's fields.

The Debug module extension (which we introduced in our project back in *Chapter 7, Modules*) uses its own secret `\yii\debug\LogTarget` class to store the logged messages independently from your logging configuration. In the case of composer installation of the Debug module, this class is inside the `vendor/yiisoft/yii2-debug/LogTarget.php` file. Here's how the same previous messages look in the Debug module:



There's a lot of structure in the log messages presented by the Debug module, because in fact, Yii does not store them strictly one by one.

The Yii logger will buffer the log messages until the end of request (normally) or until the `components.log.flushInterval` is reached, which is by default set to `1000`. Usually you'll have (a lot) less than 1000 log messages, so you can relatively safely assume that you have one batch of messages per request.

At the end of the batch, at the moment of actually writing the logged messages to the target specified, the logger grabs the values of some global variables, that is, the context of the request, and executes `var_export()` on them, writing the result as the ending log message. This is configurable, and you can specify which exact global variables will be written in the `logVars` setting of each individual log target. By default, it's the `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION`, and `$_SERVER` variables describing the request in full, but you can list any variable name as long as it is accessible through the `$GLOBALS` superglobal.

So, knowing all that, the Debug module reflects this behavior by grouping messages by batch and doing several other things to present the full runtime information of the application under debug.

> Note that you do not have any control over that special log target defined by the Debug module. It mainly uses the defaults, which are quite all-encompassing.

You are encouraged to try the Debug module yourself, because it's simple enough that explaining its workings in text will be more cumbersome than just clicking through the user interface.

Even if the Debug module removes the control over its own log messages from you, you still have control over the log messages recorded by your own application configuration. You can format them differently using two methods, one provided by Yii developers and the other being the invasive method.

First, you have control over the tokens with IP, user ID, and session ID. You can replace them with anything you want by specifying the `prefix` setting of the individual log target. It should be callable, receiving `$message` as the only argument, and the structure of `$message` can be sniffed from the documentation of the `\yii\log\Logger::$messages` property. Have a look at the following code:

```
[
    [0] => message (mixed, can be a string or some complex data,
such as an exception object)
    [1] => level (integer)
    [2] => category (string)
    [3] => timestamp (float, obtained by microtime(true))
    [4] => traces (array, debug backtrace, contains the
application code call stacks)
]
```

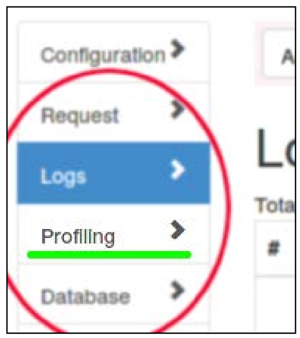
By default, `\yii\log\Target::getMessagePrefix()` is a private method of the base class for all log targets, which indeed formats the IP, user ID, and session ID in brackets and does not even use the provided `$message` argument. You can write anything you want.

The second, invasive, choice is to write your own log target, extending `\yii\log\Target`, and override the `formatMessage()` method there. This way, you acquire total control over the way in which `$message` should be serialized.

Insert a switch that will replace each log message with Lorem Ipsum during April 1, and enrage/scare your system administrators. In fact, it's not a good idea to do anything with your log messages apart from *increasing the information density* there, as often it's the only way to grasp the situation on a production system.

FEATURE – profiling

You probably noticed the **Profiling** menu item in the previous screenshot:



In fact, profiling is not a separate feature in the Yii architecture but a part of logging mechanics, which we glossed over so as not to overcomplicate things.

There are actually three more severity levels of log messages:

- `yii\log\Logger::LEVEL_PROFILE` indicates the message is for profiling purposes in general
- `yii\log\Logger::LEVEL_PROFILE_BEGIN` marks the beginning of a profiling block
- `yii\log\Logger::LEVEL_PROFILE_END` marks the end of a profiling block

You cannot specify `LEVEL_PROFILE_BEGIN` and `LEVEL_PROFILE_END` in the `levels` setting of a target, and they are in general not meant to be used manually. The profiling mechanism in Yii 2 has the following simple API:

Method	Usage
`\yii\BaseYii::beginProfile( $token, $category = 'application' )`	Begin the profiling block by inserting the log message with the `LEVEL_PROFILE_BEGIN` severity level. The token (`$token`) is the unique name of the block. Specifying `$category` can be used to filter the various profiling blocks later in the `getProfiling()` call.
`\yii\BaseYii::endProfile( $token, $category = 'application')`	Ends the profiling block by inserting the log message with the `LEVEL_PROFILE_END` severity level. The token (`$token`) must be the same as in the corresponding `beginProfile()` call, or the block will be skipped. The `$category` is irrelevant here (when filtering in the `getProfiling()` call, only the category recorded by the `beginProfile()` call will be considered), so it's only for categorizing the message itself.
`\yii\log\Logger::getProfiling( $categories, $excludeCategories )`	Walks through all of the collected log messages at the moment of the call and calculates the time between corresponding pairs of `LEVEL_PROFILE_BEGIN`-level and `LEVEL_PROFILE_END`-level messages. You can specify the categories and which messages to be considered in the report.
`\yii\log\Logger::getDbProfiling()`	This is a helper method to get the statistics about database usage. It returns an array with the number of SQL commands executed and the total time spent talking with the database.
`\yii\log\Logger::getElapsedTime()`	This is a microscopic method that returns the time with microseconds since the very beginning of the Yii framework's launch. Beware, though, as it subtracts the float values involved.

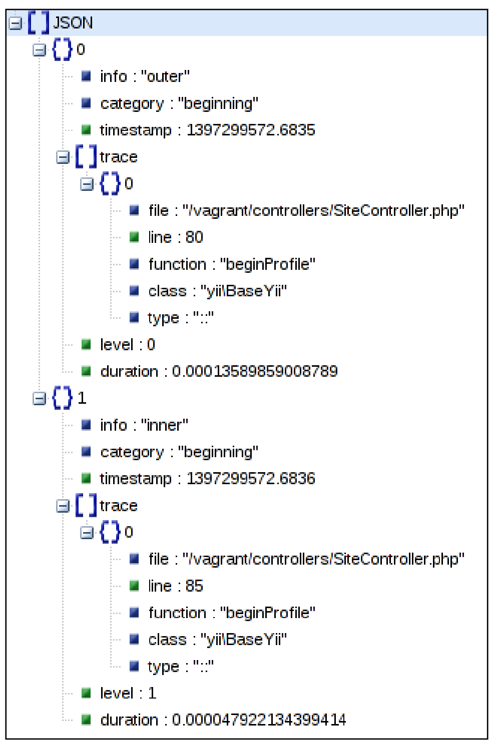
The preceding method signatures are not very helpful to understand how to reach the methods in an already-running Yii application. Here's an example:

```
public function actionProfile()
{
    Yii::beginProfile('outer', 'beginning');
    Yii::getLogger()->log('first', Logger::LEVEL_PROFILE);
    Yii::trace('second');
```

```
        Yii::info('third');

        Yii::beginProfile('inner', 'beginning');
        Yii::warning('fourth', 'nonapplication'); // note the category
        Yii::error('fifth');
        Yii::endProfile('inner', 'ending');

        Yii::endProfile('outer', 'ending');

        $result = Yii::$app->response;
        $result->data = Yii::getLogger()->getProfiling(
            ['beginning', 'application', 'ending']);
        $result->format = Response::FORMAT_JSON;

        return $result;
    }
```

As it was said before, the default category for the log messages recorded manually is `application`, so we specify in the call to `getProfiling()` that we're interested only in this particular category. Note that we can nest the profiling blocks.

Here is the result rendered by the action mentioned in the preceding code:

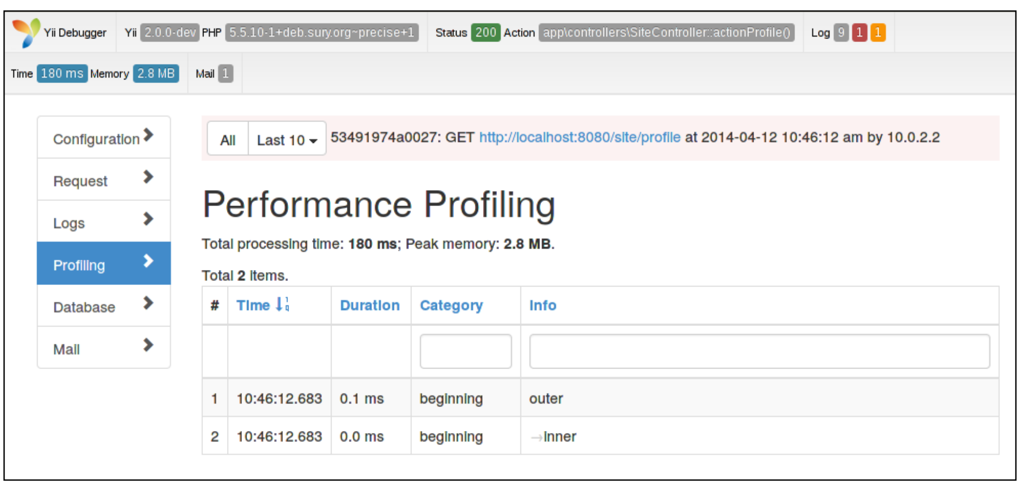


Note that there are no in-between messages but only the `beginProfile` ones.

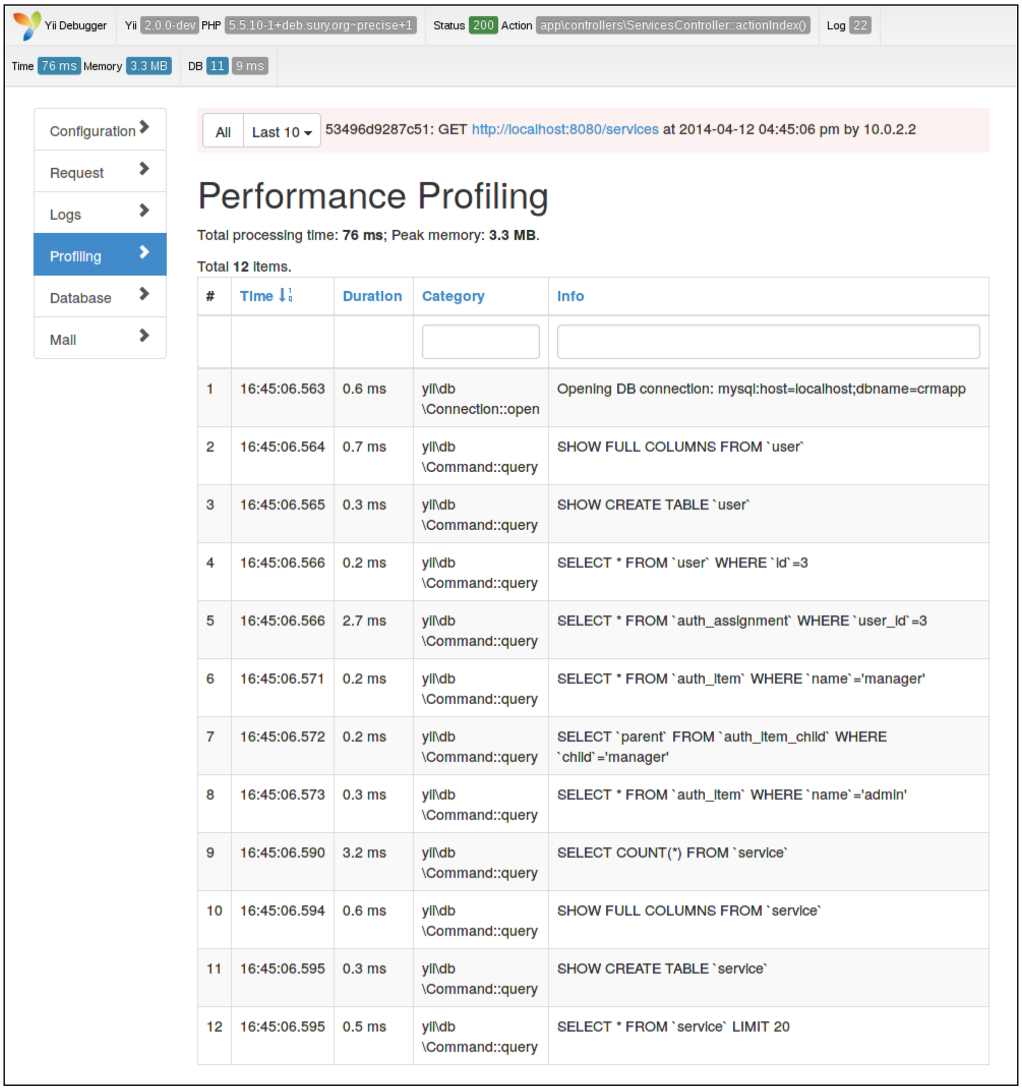
The profiler calculates the time between corresponding pairs of `LEVEL_PROFILE_BEGIN` and `LEVEL_PROFILE_END` messages, which appear on the same level of nesting. The result of profiling each level is returned as the array with the following fields:

- `info`: This is the `$token` specified in the call to `beginProfile()` as the first argument.
- `category`: This is the category to which the current profiling process belongs. It's snatched from the `beginProfile()` call.
- `timestamp`: This is the timestamp of the `beginProfile()` call.
- `trace`: This is the stack trace information for the profiling message (to know in which method it was recorded).
- `level`: This gives an indication of how deep down the stack of profiling blocks we currently are, starting from `0`.
- `duration-`: This is the time in seconds (!) from `beginProfile()` until the `endProfile()` call. This value is calculated by subtracting results of the `microtime($as_float = true)` calls, so it's a float value representing duration in seconds with some precision after the comma.

There is no real-world example of the usage of low-level API to view profiling results, for the simple reason that the Debug module already has the way to visualize them. Here is the same request being viewed in the **Profiling** tab:



Of course, there's not so much to see, because everything we did is marked at two places in code. Yii 2 by default marks every call to the DBMS for profiling, so a request that makes a roundtrip to the database is a lot more interesting target to profile:



#	Time	Duration	Category	Info
1	16:45:06.563	0.6 ms	yii\db \Connection::open	Opening DB connection: mysql:host=localhost;dbname=crmapp
2	16:45:06.564	0.7 ms	yii\db \Command::query	SHOW FULL COLUMNS FROM `user`
3	16:45:06.565	0.3 ms	yii\db \Command::query	SHOW CREATE TABLE `user`
4	16:45:06.566	0.2 ms	yii\db \Command::query	SELECT * FROM `user` WHERE `id`=3
5	16:45:06.566	2.7 ms	yii\db \Command::query	SELECT * FROM `auth_assignment` WHERE `user_id`=3
6	16:45:06.571	0.2 ms	yii\db \Command::query	SELECT * FROM `auth_item` WHERE `name`='manager'
7	16:45:06.572	0.2 ms	yii\db \Command::query	SELECT `parent` FROM `auth_item_child` WHERE `child`='manager'
8	16:45:06.573	0.3 ms	yii\db \Command::query	SELECT * FROM `auth_item` WHERE `name`='admin'
9	16:45:06.590	3.2 ms	yii\db \Command::query	SELECT COUNT(*) FROM `service`
10	16:45:06.594	0.6 ms	yii\db \Command::query	SHOW FULL COLUMNS FROM `service`
11	16:45:06.595	0.3 ms	yii\db \Command::query	SHOW CREATE TABLE `service`
12	16:45:06.595	0.5 ms	yii\db \Command::query	SELECT * FROM `service` LIMIT 20

We did nothing to get this very useful information; Yii profiles it by default. This is the most valuable source of information about performance bottlenecks in your application. However, it's only the database queries that are covered by profiling. If there is anything else, you should mark it manually.

Error handling details

We covered error (exceptions) handling in great detail back in *Chapter 6*, *User Authorization and Access Control*. We glossed over, however, the exceptions not related to HTTP status codes, and we just barely mentioned the way in which you can actually control how Yii handles errors.

> The important point is that you will probably never need to change the way Yii handles errors, because it's really a well thought out algorithm covering all kinds of exceptional situations possible in a PHP application, so maybe you will never need to add anything to it.

The story about how Yii 2 handles errors is pretty simple. A Yii application has a special component attached by default, called the `\yii\base\ErrorHandler`. To be precise, for a web application, an extension of the base error handler is attached, that is, `\yii\web\ErrorHandler`. This component holds three methods, which are registered in PHP as exception handler, error handler, and a fatal error handler. All these methods convert all three types of reported problems into an exception of some kind and pass this exception to a method called `renderException`, thus uniformly handling any possible error that can happen in the PHP code and displaying information about it to the user.

Here is the exact code that hooks all error handling to the Yii framework at the time of this writing:

```
public function register()
{
    ini_set('display_errors', false);
    set_exception_handler([$this, 'handleException']);
    set_error_handler([$this, 'handleError']);
    if ($this->memoryReserveSize > 0) {
        $this->_memoryReserve = str_repeat('x', $this->memoryReserveSize);
    }
    register_shutdown_function([$this, 'handleFatalError']);
}
```

The `$this` variable is `\yii\base\ErrorHandler`. As was said earlier, all three methods, `handleException()`, `handleError()`, and `handleFatalError()`, ultimately resolve into the `\yii\base\ErrorHandler::renderException(\Exception $exception)`, which is just great, because we have only one place of change to override.

Let's look in detail at what exactly will be presented to the client depending on various conditions. Given that the PHP errors and fatal errors already were converted into exceptions, we have the following options:

Conditions	What should be shown
The global `Response` component has the `format` field set to something other than the `FORMAT_HTML` value (which is the default value).	The exception will be converted into an array according to `\yii\web\ErrorHandler::convertExceptionToArray`, and this array will then be rendered according to whatever `Yii::$app->response->format` is set to.
`YII_ENV_TEST` is `true`.	The exception will be converted into a string according to the `\yii\base\ErrorHandler::convertExceptionToString` method, and this string will be rendered HTML-encoded inside the `<pre>` element as the only response from server.
The request is an AJAX one, checked by the `$_SERVER['HTTP_X_REQUESTED_WITH']` value.	
`YII_DEBUG` is `true` and the exception is not a descendant of `\yii\base\UserException`.	The exception will be rendered using the view file referenced in `\yii\web\ErrorHandler::$exceptionView`, which is by default the full exception report we showed back in *Chapter 6*, *User Authorization and Access Control*.
The `ErrorHandler` component has the `errorAction` setting set (it should be some route in the application, such as `site/error`).	The controller action reachable by this route will be run. It is expected to check the value of `Yii::$app->errorHandler->exception` and render it in some way. Traditionally, this action is `\yii\web\ErrorAction`.
All other cases.	The exception will be rendered using the view file referenced in `\yii\web\ErrorHandler::$errorView`, which is by default the short exception report meant for end users, which we showed back in *Chapter 6*, *User Authorization and Access Control*, just before the full exception report.

These options exclude each other from top to bottom, that is, the first one that applies is considered true.

> Note that the exceptions descending from `UserException` are always rendered by the error view, which is short and non-informative. This type of exception is meant for end users, that is, it's an exceptional situation resulting from the actions of the user. So, even in production, we'll show not just the internal server error occurred, but the precise exception message.

In all cases, if the exception thrown is a descendant of `HttpException`, then the proper HTTP status code from this exception will be set. Otherwise, the status code will always be `500`.

The preceding table describes the algorithm to render the exception employed by the built-in class from Yii 2, encoded in `\yii\web\ErrorHandler::renderException`. If you need to employ a different logic for rendering exceptions, you are free to override `\yii\web\ErrorHandler` and attach your custom class to the `components.errorHandler` setting. If you need to employ a different logic for handling exceptions, errors, or PHP fatal errors, you can even override `\yii\base\ErrorHandler` or write an analogous class. It's hard to imagine a use case in which you will need to do this, though.

FEATURE – error handling controller action

Yii 2 has one built-in place where you can use the composition instead of inheritance to control how the exceptions will be rendered in your application. It's listed as the second-to-last row in the table in the previous section.

We can set the `components.errorHandler.errorAction` setting to a route on our application, which will be used to render the errors. This setting corresponds to the `\yii\web\ErrorHandler::$errorAction` property, obviously.

> This will work only if all previous options were discarded!

While we can reference any route in this setting, Yii 2 has a built-in controller action that does the rendering for us called `\yii\web\ErrorAction`. As the result, you can set up a custom error page that your application will show to your users with the following two steps.

First, you tell the error handler in the application configuration that you want to use your own error handling route, for example, `site/error` as follows:

```
'errorHandler' => [
    'errorAction' => 'site/error',
]
```

Second, you go to `SiteController` and declare the `error` action using the `actions()` method as follows:

```
public function actions()
{
    return [
        'error' => ['class' => 'yii\web\ErrorAction'],
    ];
}
```

You're done. Now any exception that the visitors of your web application will encounter will be shown using the view file in the `views/site/error.php` file.

You can control the exact location of the view file using the `view` setting of `ErrorAction`. This view file will receive three variables (apart from the `$this` variable, bound to the `View` instance) as follows:

- `$name`, which is the exception name calculated by `ErrorAction` for you that you can safely show to visitors
- `$message`, which is the exception message calculated by `ErrorAction` for you that you can safely show to visitors
- `$exception`, which is the original exception for you to mess with

> Note that `ErrorAction` tries to be generic too and will handle AJAX requests to itself by returning just the `$name: $message` PHP string, verbatim.

And yes, there are no real-world examples here either.

List of built-in exceptions

In *Chapter 6*, *User Authorization and Access Control*, we listed all the `HttpException`-based exceptions built into Yii 2. But `HttpException` descendants are not all exceptions available for you in Yii 2. Here is the list of all other exceptions for you to use. In the **Purpose** column of the following table, we explain how and where Yii 2 itself uses the corresponding exception class:

Exception class	**Purpose**
`\yii\base\ErrorException`	This class represents the exceptions caused by PHP errors and fatal errors. You'd better not raise them manually, for they have a lot of logic inside.
`\yii\base\Exception`	This class is used in the case of generic exceptions for all purposes.
`\yii\base\ExitException`	This class represents an exception caused by the exiting of an application, such as using the *Ctrl* + *C* key pair in the POSIX-compatible terminal. The first argument of the constructor of this class is the return code, which is useful for console commands.
`\yii\base\InvalidCallException`	This class represents an exception caused by calling a method in the wrong way.
`\yii\base\InvalidConfigException`	This class represents an exception caused by incorrect object configuration. You'll get *a lot* of these exceptions when experimenting with application configuration or widgets.
`\yii\base\InvalidParamException`	This class represents an exception caused by invalid parameters passed to a method.
`\yii\base\InvalidRouteException`	This class represents an exception caused by an invalid route. This exception will be re-thrown as `NotFoundHttpException` by `\yii\web\Application`.
`\yii\base\NotSupportedException`	This class represents an exception caused by accessing features that are not supported.
`\yii\base\UnknownClassException`	This class represents an exception caused by using an unknown class.
`\yii\base\UnknownMethodException`	This class represents an exception caused by accessing an unknown object method.
`\yii\base\UnknownPropertyException`	This class represents an exception caused by accessing unknown object properties.
`\yii\base\UserException`	This class represents the exception we talked about earlier. You are expected to throw it when you want to point at the user's own mistake.
`\yii\console\Exception`	This class represents the exception that is identical in purpose to the `\yii\base\UserException` but is intended to be used in a console application.

Exception class	Purpose
`\yii\db\Exception`	This class represents the problems reported by the database manipulation mechanics. When Yii throws it, it fills its additional `errorInfo` property with the result of the `PDO::errorInfo()` call.
`\yii\db\StaleObjectException`	This class represents an exception that is thrown when the object being referenced in the database (by `UPDATE` or `DELETE` operations) is not there anymore, which is possible in parallel applications.

Caching

As long as PHP programs continue being scripts to launch, output the results, and then die, the idea of **caching** results of computations is used. The mathematical concept of memoization, implemented at the level of a language runtime environment, is used extensively in the PHP ecosystem, and the Yii framework supports a wide range of caching techniques to speed up your program.

There are four levels of caching in Yii 2, which are as follows:

- The database queries cache
- The HTML page fragments cache
- The whole request cache
- The HTML response cache with the help of some HTTP headers

Before we look at all of these features in turn, we need to understand that there's a cache component inside the Yii 2 framework that is the heart of the caching functionality.

FEATURE – cache component

Three out of four caching levels presented in the preceding section use the centralized caching component, which is configurable at the `components.cache` setting in application configuration. This component is a representation of some key-value storage, capable of saving data marked by some key and returning this data when given its key.

Of course, while the caching in Yii 2 uses this component implicitly, you as a developer can use it manually too by accessing the `Yii::$app->cache` object. Without many other details of the Cache API, you can reliably depend on the following calls:

Method	Meaning
`add($key, $value, $duration = 0, $dependency = null)`	Add `$value` marked by `$key` for `$duration` in the seconds specified. If something is already stored in `$key`, do nothing. If `$dependency` changes, `$value` will be considered invalid regardless of `$duration`.
`set($key, $value, $duration = 0, $dependency = null)`	Set `$value` provided in `$key` and set the new `$duration`. If nothing is stored in `$key`, this method is identical to `add()`. If `$dependency` changes, `$value` will be considered invalid regardless of `$duration`.
`get($key)`	Get the object stored in `$key`.
`delete($key)`	Delete the object stored in `$key`.
`exists($key)`	Check whether something exists in `$key`.
`flush()`	Clean the cache completely, removing all the stored values.

> When stored, `$value` will be serialized either by the `serialize()` PHP built-in method or by the callable method specified in the `components.cache.serializer` setting. The same applies to retrieving `$value`, which will use the `unserialize()` PHP built-in method if there is no custom serializer. Check the documentation on that setting for details.

When you register an object in the cache, you can specify two ways in which the object can be **invalidated**, that is, becomes unavailable under the key where it was stored. One way is through the `$duration` parameter, essentially restricting the time to live for this cache object. Another way is a lot more interesting, because you can also set arbitrary `$dependency` for this cache object. The `$dependency` argument for the `add()` and `set()` methods is an object of some subclass of `\yii\caching\Dependency`. To better understand what this all is about, here is the list of all the built-in kinds of caching dependencies:

- `\yii\caching\DbDependency`: The cached object will depend on the result of a SQL query to a database (probably different from the main one).
- `\yii\caching\ExpressionDependency`: The cached object will depend on the result of some arbitrary `eval()` PHP expression.
- `\yii\caching\FileDependency`: The cached object will depend on the last modification time of a file in the filesystem.
- `\yii\caching\TagDependency`: The cached object will be marked by a tag by this dependency. Later, at some point, the application can invalidate all cached objects with the same tag by calling `TagDependency::invalidate($cache, $group)` and passing the Cache component and the desired tag to it.
- `\yii\caching\ChainedDependency`: This is a special kind of dependency that groups other dependencies together, allowing you to make complex conditions to invalidate cached items.

The main issue with caching, of course, is that sometimes, the actual content refreshes faster than the cached objects become invalidated. This results in obsolete data being shown to visitors. Dependencies are a means to solve these problems, because you can basically force the content to be re-cached based on a condition, such as a changed file, database query result, or some arbitrary PHP expression.

> We'll not delve into the details of fine-tuning your application with this elaborate means. It's better to look into the official documentation for them.

The Cache component in Yii 2 is a class that extends `\yii\caching\Cache`, and the framework provides eight caching solutions for you out of the box:

- `\yii\caching\ApcCache`: This uses the **APC** PHP extension (see `http://php.net/manual/book.apc.php`).
- `\yii\caching\DbCache`: This uses the table in the database (you can even set a different connection from the application's main one).

- `\yii\caching\DummyCache`: This provides no caching but provides the mandatory API, so you can use this component to satisfy the requirements of some other facility.
- `\yii\caching\FileCache`: This stores objects in files in a directory, thus relying on the capabilities of the filesystem. Note that depending on the filesystem and the data storage type, you can get *lower* performance with this type of caching.
- `\yii\caching\MemCache`: This uses the **memcache** solution (see `http://php.net/manual/intro.memcache.php`). You can switch to memcache by means of a single Boolean setting (see documentation).
- `\yii\caching\WinCache`: This uses **Windows Cache** by means of WinCache PHP extension (see `http://www.iis.net/expand/wincacheforphp`).
- `\yii\caching\XCache`: This uses **XCache** (see `http://xcache.lighttpd.net/`).
- `\yii\caching\ZendDataCache`: This uses the **Zend Data Cache** (see `http://www.zend.com/en/products/server/`) PHP extension.

As usual, you are encouraged to read the corresponding documentation to learn the details about the correct configuration of these components.

In addition to the built-in ones, there is the Redis Yii extension, installable by using the following command:

```
$ php composer.phar require --prefer-dist yiisoft/yii2-redis "*"
```

After installation, you'll have the `\yii\redis\Cache` component, which provides caching using the **Redis** server (see `http://redis.io/` for details).

Here is the example configuration snippet required to enable the cache component with the Redis provider:

```
'components' => [
    'cache' => [
        'class' => 'yii\redis\Cache',
        'redis' => [
            'hostname' => 'localhost',
            'port' => 6379,
            'database' => 0,
        ]
    ],
],
```

FEATURE – database queries caching

Yii 2 has a set of properties on the DB connection component that control caching for DB queries. You can set them in the following application settings:

Setting	Meaning
`enableSchemaCache`	Whether to enable schema caching at all.
`schemaCacheDuration`	The duration in seconds to cache the schema data for. By default, it is `3600` seconds (1 hour).
`schemaCacheExclude`	Tables whose schema should *not* be cached as a list of strings.
`schemaCache`	Which cache component to use for caching. By default, it is the system's component configurable at the `components.cache` key.
`enableQueryCache`	Whether to enable query-result caching at all.
`queryCacheDuration`	The duration in seconds to cache the query results for. By default it is 3,600 seconds (1 hour).
`queryCacheDependency`	The dependency that will additionally control the invalidation of cached data.
`queryCache`	Which cache component to use for caching. By default, it is the system's component configurable at the `components.cache` key.

Normally, everything you need to *enormously* speed up interaction with the database (given that you're in a production environment and have thousands of records in tables) is the following configuration for the Database Connection component:

```
'components' => [
    'db' => [
        // …all usual stuff with 'class', 'dsn', 'username' and
'password'…
        'enableSchemaCache' => true,
        'enableQueryCache' => true,
    ]
]
```

Assuming that you have configured the Cache component you want, you're done. Both schema and query results will be cached for one hour by default.

FEATURE – page fragment caching

At a level higher than the caching of database results, you can cache pieces of view files. You do this in the following fashion in the view file:

```
$this->beginCache(uniqid());
… this content will be fetched from cache if present, and re-
calculated and put into cache again otherwise...
$this->endCache();
```

The widget called `\yii\widgets\FragmentCache` encapsulates this feature. The methods shown, `\yii\base\View::beginCache()` and `\yii\base\View::endCache()`, are helpers to ease the usage of this widget.

The `beginCache()` method accepts two arguments: `$id` and `$properties`. The value of `$id` has to be a unique ID, and the `$properties` argument consists of properties for the `FragmentCache` widget. Look at the documentation of this widget for details.

This level of caching is perfect for things like dynamic advertising banners, the content of which has to be rendered based on the information from the database.

FEATURE – whole page caching

Then, on the next level of caching, you're able to cache the whole page rendered in response to the request. Remember that the previous two levels of caching will still speed up things, decreasing the overall time required to render the whole view file starting from the layout.

You use the `\yii\web\PageCache` action filter for this task, and if you remember, we already mentioned this back in *Chapter 6, User Authorization and Access Control*. When you want a controller to cache the whole content of a response, you put the following setup in its `behaviors()` method:

```
public function behaviors()
{
     return [
         'pageCache' => [
             'class' => \yii\web\PageCache::className(),
             'only' => ['list', 'of', 'actionIDs', 'to', 'be',
'cached'],
             'duration' => 60, // seconds
             'dependency' => [config for the Cache Dependency],
         ]
     ]
}
```

The absolutely required properties are the name of the behavior (can be arbitrary) and the class name of the `PageCache` action filter. If you omit the *only* setting, every action of this controller will be cached, which most probably is not what you really want.

> Please note that if you blindly throw `PageCache` on the example CRM application we're experimenting with in this book, you will most probably *make the acceptance tests fail*. For example, if you just register `PageCache` for the whole array of actions in `ServicesController`, the system will not be able to even save new `ServiceRecord` instances because of caching. Even if you restrict `PageCache` only until `actionIndex()`, you still get the problems with acceptance tests, because you basically will have to invalidate the cache after each new service is added, changed, or deleted. This requires quite an elaborate cache dependency declared in the `dependency` setting of `PageCache`. You are free to experiment with implementing the whole page caching in `ServicesController`, but generally it's just not a good idea to employ such caching to highly volatile parts of web applications, such as CRUD.

Page-level caching by the `PageCache` filter is in fact the same as calling the `beginCache()` method before rendering the view file and calling the `endCache()` method after rendering. In fact, `PageCache` does exactly this, as can be seen from the source code of this class. Using this behavior will save you from repeating of the same code over and over in all places where you need the caching.

FEATURE – caching the request by HTTP headers

This is the least reliable, last line of caching, which is done (almost) exclusively on the browser side. By utilizing the **Last-Modified** and **ETag** HTTP headers, the server can tell the browser whether the page was modified since the last request and whether it will be useful to redownload it. By correctly utilizing these headers, we can save both the time to download the HTML and/or all of its referenced assets, and the time and assets to render it at the server.

> Look at the **RFC 2616** section 13.3 at `http://tools.ietf.org/html/rfc2616#section-13.3` for an authoritative explanation about how these headers work.

This kind of caching is being done by the `\yii\web\HttpCache` action filter. Its configuration is a bit more elaborate, as we have to provide two callbacks: first, one that will generate the value for the Last-Modified header, and second that will generate the value for the ETag header.

Let's say we add the `last_updated` field to the table of customers in the database, which will auto-update when any other field will change in this table. So, we can add an additional caching level to the `CustomersController.actionIndex()` method by setting the Last-Modified header to the timestamp of the customer's record in this table with the maximum value of this timestamp. To properly version the resulting page by ETag, we can use the same function as follows:

```
public function behaviors()
{
    return [
        'httpCache' => [
            'class' => \yii\web\HttpCache::className(),
            'only' => ['index'],
            'lastModified' => [$this, 'getMaxCustomerTimestamp'],
            'etagSeed' => [$this, 'getMaxCustomerTimestamp'],
        ],
    ];
}
public function getMaxCustomerTimestamp($action, $params)
{
    return strtotime((new Query())->from('customers')->max('last_
updated'));
}
```

Minimizing the assets

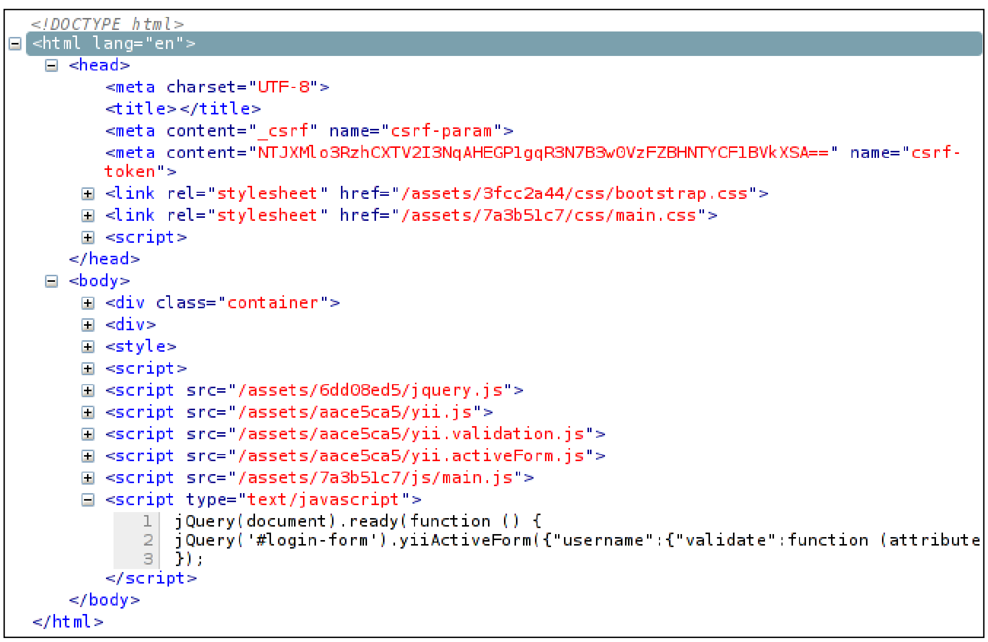
Obviously, in complicated setups, you'll have a lot of CSS and JavaScript files registered to pages, and this is a major drawback. The current best practice is to compress and combine all CSS files into a single file and do the same with JavaScript files. As a result, we serve just two files instead of the whole bunch. You probably already know all of this. It would be really cool to have the asset system in Yii do it for you *implicitly*, though currently, there is no such provision. Yii has, however, got the helper console command that you can use to simplify the process of compressing your assets and wiring them to your application. The most beautiful part is that you will not need to change any of your view files; all existing `*Asset::register()` calls can stay the same.

Although the documentation describes the full process of introducing combined assets to the Yii application, we repeat it here to bring a lot more detail to this really complex process.

The overall idea behind asset compiling in Yii is that you prepare a special asset bundle with just two files: one containing all of the compiled CSS and the other containing all of the compiled JavaScript. Then, you override the configuration for asset bundles using the `components.assetManager.bundles` setting (read about it in the documentation) telling Yii to use this special, newly-created asset bundle instead of all of the usual bundles.

The `\yii\console\controllers\AssetController` class, which provides you with the `./yii asset` command, encapsulates and automates the procedure of asset compilation. It even deals with the issue of files referenced from the CSS code for you, which is really amazing.

Let's see what we need to do to accomplish this. First, we remember the HTML structure we have when accessing a route at our application, for example `/site/login`. The following picture is a screenshot from Firebug in which we see the number of elements our UI consists of:



You can see both Twitter Bootstraps and our own stylesheets inside the `head` element. At the bottom of the `body` element, there's a whole bunch of script elements with various pieces of JavaScript UI. Our goal is to replace all CSS references with just one reference and all these `script` tags with just a single one.

We need the actual executables that will do the job of compressing the assets. Yii 2 uses the **YUI compressor** for CSS compression and the **Google Closure compiler** for JavaScript compression, available at `http://yui.github.io/yuicompressor/` and `https://developers.google.com/closure/compiler/`, respectively.

Let's create a directory named `compression` under the `assets` subdirectory and put everything related to the task at hand there. You have to download two JAR files from the preceding links to this folder. The YUI compressor should be named `yuicompressor.jar`, and the Google Closure compiler should be named `compiler.jar`.

After that, we need to create a special configuration file for the compressor. Yii has the console command to help you make a boilerplate template for this configuration. Run the following at the root of code base:

```
$ ./yii asset/template assets/compression/config.php
```

If you open the template generated by the preceding command, you should see the following code:

```
<?php
return [
    // Adjust command/callback for JavaScript files compressing:
    'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_
file {to}',
    // Adjust command/callback for CSS files compressing:
    'cssCompressor' => 'java -jar yuicompressor.jar --type css {from}
-o {to}',
    // The list of asset bundles to compress:
    'bundles' => [ // 1
        // 'yii\web\YiiAsset',
        // 'yii\web\JqueryAsset',
    ],
    // Asset bundle for compression output:
    'targets' => [ // 2
        'app\assets\AllAsset' => [
            'basePath' => 'path/to/web',
            'baseUrl' => '',
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
```

```
            ],
        ],
        // Asset manager configuration:
        'assetManager' => [ // 3
            'basePath' => __DIR__,
            'baseUrl' => '',
        ],
    ];
```

It's quite long to be included verbatim in a book, but a lot of important but not so obvious pieces of code are mentioned here.

The first part of the code is quite simple: you should define the list of asset bundles you want to compress together. Note that they are named by their fully qualified names.

> Please specify the fully qualified names of asset bundles without the leading slashes; otherwise, you will break the mechanics.

By definition, the contents of the asset bundles listed under the `bundles` setting will be registered on each and every page of your web application. If you want some asset bundles to be registered based on a condition, such as `SnowAssetsBundle` from *Chapter 4*, *The Renderer*, you better leave them off as they can contain overrides for standard styles.

It would be nice to combine all CSS and JavaScript files from our main asset bundle and all its dependencies. For this, we need only the following value for the `bundles` setting:

```
        'bundles' => [
            'app\assets\ApplicationUiAssetBundle',    ]
```

> Do not worry about the files referenced in the CSS files, such as the images and fonts. All asset bundles will be published as usual anyway, so all their supplementary files will still be available in the web-accessible directories. The `./yii asset` command does this so that it is able to rewrite all the `url('')` annotations in a compressed CSS file to point to their actual URLs. This will be as if the corresponding asset bundles were registered as usual, without any compression.

The second part of the code is a lot trickier. In the `targets` setting, you should list the asset bundles that will receive the compressed files. It's really a generic setting, but for our cause, when we need a single, all-encompassing asset bundle containing just two compressed files, the code offered to us by the template is completely sufficient. It says that there'll be the `app\assets\AllAsset` asset bundle that will contain one JavaScript file and one CSS file. We need only to correct the paths in this definition.

However, the `targets` setting lies with us. There's no `app\assets\AllAsset` class definition anywhere, and Yii will not generate it for us. You must put the implementation for this class into the appropriate file `assets/AllAsset.php`. Have a look at the following code:

```
namespace app\assets;
use yii\web\AssetBundle;
class AllAsset extends AssetBundle { }
```

As for paths, we need a plan for it. Our plan will be simple: the two compressed files we are interested in will be placed in `@webroot` to be accessible without publishing into the `compiled-assets` subdirectory.

So, `AllAsset` should have the `basePath` setting equal to `@webroot` and the `baseUrl` setting equal to `/`. The problem is, at the point when `assets/compressed/config.php` will be evaluated, there'll be no `@webroot` alias defined, so we are left with paths shown in the following code:

```
'targets' => [
    'app\\assets\\AllAsset' => [
        'basePath' => realpath(__DIR__ . '/../../web'),
        'baseUrl' => '/',
        'js' => 'compiled-assets/all-{hash}.js',
        'css' => 'compiled-assets/all-{hash}.css',
    ],
],
```

This funny `{hash}` token will be replaced by a unique hash for the corresponding file.

The third part of the code shows the asset manager configuration in which we have to repeat the configuration for the Asset Manager component attached to the main application. We have to specify the working paths there, because as we said earlier, there's no `@webroot` alias defined at the point when the `./yii asset` command works.

The rationale behind `assetManager.basePath` and `assetManager.baseUrl` is for the Asset Manager component to know where to put the published assets. We already decided at the stage of constructing the application that the assets will be published to the `web/assets` directory and so will be accessible by the `/assets/*` routes (as `@webroot` is the directory we published through the web server). Given all that, we should write the following into the `assetManager` setting in the configuration for the `./yii asset` command:

```
'assetManager' => [
    'basePath' => realpath(__DIR__ . '/../../web/assets'),
    'baseUrl'  => '/assets',
],
```

The last part is the configuration for paths to the compressors at the top of the configuration template. By default, the `./yii asset` command expects the JAR files for compressors to be in the root of the code base. But in our case, they're in the `assets/compression` directory, so as not to have stuff lying around in no particular order. The `yii\console\controllers\AssetController`, whose properties we are setting in this supplementary configuration file, has two special settings for us, however, they are quite cumbersome to override:

```
    public $jsCompressor = 'java -jar compiler.jar --js {from} --js_
output_file {to}';
    public $cssCompressor = 'java -jar yuicompressor.jar --type css
{from} -o {to}';
```

As you see, full console commands are required to launch the JavaScript and CSS compressors, and they indeed assume JAR files to be in the root of the code base (or at least in the folder in which we call the `./yii asset` command). As we will always use this command from the root of the code base, we will override these settings in the `assets/compression/config.php` file in the following way:

```
    'cssCompressor' => 'java -jar assets/compression/yuicompressor.jar
--type css {from} -o {to}',
    'jsCompressor' => 'java -jar assets/compression/compiler.jar --js
{from} --js_output_file {to}',
```

With this, you get the complete, proper configuration for the compressor. Before you run it, though, you have to prepare the `web/compiled-assets/` directory you promised to `AssetController` in the `targets.app\\assets\\allAsset.js` and `targets.app\\assets\\allAsset.css` settings. Don't forget to put this directory under the version control system you use.

Now, run the command at last:

```
$ ./yii asset assets/compression/config.php config
  /assets_compressed.php
```

The first argument to the `./yii asset` command is the path to the supplementary configuration file we worked so hard to forge until now. The second argument is the path to the file that will hold the resulting configuration snippet we'll need to attach to our application. We decided to put it to the `assets_compressed.php` file near our other main configuration files.

The configuration snippet generated will hold the definitions of all asset bundles listed in `bundles` with `js` and `css` settings cleaned out. All these asset bundles will be made dependent on the newly-created asset bundle containing the compressed CSS and JavaScript files. We need to use the content of this configuration snippet as the value of the `components.assetManager.bundles` setting in the application configuration, so in our case, we just paste the following into the `components` section:

```
'assetManager' => [
    'bundles' => (require __DIR__ . '/assets_compressed.php')
],
```

Finally, all this would be meaningless if we did not change the insertion of asset bundles in our layout. Currently the relevant code looks like the following:

```
app\assets\ApplicationUiAssetBundle::register($this);
```

Now, we need to use the `AllAsset` class we have created so far:

```
app\assets\AllAsset::register($this);
```

Now, you can open the same route `/site/login` we were using for the previous screenshot, and you'll see the following:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title></title>
    <meta content="_csrf" name="csrf-param">
    <meta content="Tmo1NHNHbks5BVdwCgNDcwpeXV4DN1oIfCRWURgpLQAoOENzLCk.Ow==" name="csrf-token">
    <link rel="stylesheet" href="/compiled-assets/all-1397731806.css">
    <script>
  </head>
  <body>
    <div class="container">
    <div>
    <style>
    <script>
    <script src="/compiled-assets/all-1397731806.js">
    <script src="/assets/aace5ca5/yii.validation.js">
    <script src="/assets/aace5ca5/yii.activeForm.js">
    <script type="text/javascript">
      1  jQuery(document).ready(function () {
      2  jQuery('#login-form').yiiActiveForm({"username":{"validate":function (attribute, valu
      3  });
    </script>
  </body>
</html>
```

As you can see, we removed the `main.css`, `main.js`, `bootstrap.css`, `yii.js`, and `jquery.js` scripts from the assets. However, there are still scripts for validation and the `ActiveForm` widget. We can compress them (we are using these features anyway through the whole application) by including them in the `bundles` setting:

```
'bundles' => [
    'app\assets\ApplicationUiAssetBundle',
    'yii\widgets\ActiveFormAsset',
    'yii\grid\GridViewAsset',
    'yii\validators\ValidationAsset',
],
```

Well, `GridViewAsset` was added just because we have two `GridView` widgets in our application.

After you recompress assets with the configuration shown in the preceding code, you should not see the assets from the validator and active form:

```
<!DOCTYPE html>
<html lang="en-US">
    <head>
        <meta charset="UTF-8">
        <title> </title>
        <link rel="stylesheet" href="/compiled-assets/all-4d8400dcee8e562bfa9b73a62b2fc9f9.css">
        <meta content="_csrf" name="csrf-param">
        <meta content="UjNlLXNHbHYBdVceCwYbHAtYIH0XBRtOGEBdSj0AIk8bWVxLCixfOA==" name="csrf-token">
        <script>
    </head>
    <body>
        <div class="container">
        <div>
        <style>
        <script>
        <script src="/compiled-assets/all-dcf8c375a5b81e46e0b8fb301717d654.js">
        <script type="text/javascript">
            jQuery(document).ready(function () {
            jQuery('#login-form').yiiActiveForm({"username":{"validate":function (attribute, valu
            });
        </script>
    </body>
</html>
```

Now, we come to the timings. Here is the table without implementing compression:

URL	Status	Domain	Size	Remote IP	Timeline
GET login	200 OK	localhost:8888	3,4 KB	127.0.0.1:8888	210ms
GET bootstrap.css	200 OK	localhost:8888	19,2 KB	127.0.0.1:8888	12ms
GET main.css	200 OK	localhost:8888	90 B	127.0.0.1:8888	7ms
GET jquery.js	200 OK	localhost:8888	71,5 KB	127.0.0.1:8888	47ms
GET yii.js	200 OK	localhost:8888	2,9 KB	127.0.0.1:8888	28ms
GET yii.validation.js	200 OK	localhost:8888	1,5 KB	127.0.0.1:8888	29ms
GET yii.activeForm.js	200 OK	localhost:8888	3,7 KB	127.0.0.1:8888	31ms
GET main.js	200 OK	localhost:8888	0 B	127.0.0.1:8888	28ms
GET toolbar?tag=53d		localhost:8888	0 B		
GET toolbar?tag=53d	200 OK	localhost:8888	5,9 KB	127.0.0.1:8888	45ms
10 requests			108,3 KB		686ms (onload: 652ms)

This is after compression:

URL	Status	Domain	Size	Remote IP	Timeline
GET login	200 OK	localhost:8888	3,4 KB	127.0.0.1:8888	199ms
GET all-4d8400dcee8	200 OK	localhost:8888	18,0 KB	127.0.0.1:8888	7ms
GET all-dcf8c375a5b8	200 OK	localhost:8888	33,5 KB	127.0.0.1:8888	9ms
GET toolbar?tag=53d	200 OK	localhost:8888	5,9 KB	127.0.0.1:8888	57ms
4 requests			60,7 KB		471ms (onload: 443ms)

Note that we saved six requests out of 10, and our timing is 31 percent better. Of course, any benchmark is a lie regarding the times, but what is most important is reducing the number of asset files requested. We got these screenshots of Firebug after cleaning the cache and rebuilding the assets in Yii, so no in-browser caching should be involved in the timings presented.

Summary

We looked at two aspects of overall behavior of the application based on Yii 2, namely introspection, which is provided by logging and error reporting facilities, and response speed, which is provided by the four-level caching facility and asset compressor.

In fact, logging will impact performance too. Enabling database query logging usually cripples the processing speed for real, so you probably will never be using it in production.

Nothing we discussed here really adds any features visible for the end user, except for the custom error page probably. It's of doubtful value nevertheless, because the default Yii 2 error page meant for end users meets almost all best practices for error pages. However, in a serious application of any reasonable size, you will need these best practices without any doubt.

In the next short chapter, we'll look at how we can create our own Yii 2 extension. We have used two extensions already and one briefly mentioned. We'll discuss the concepts of the framework prepared for us to package the extensions.

9
Making an Extension

We have used quite a bunch of pre-built Yii 2 extensions, shipped as composer-installable libraries separately from the main framework. In this chapter, we'll learn to make our own extension using the same simple way of installation.

There is a process we have to follow, though some preparation will be needed to wire up your classes to the Yii application. The whole chapter will be devoted to this process.

Extension idea

So, how are we going to extend the Yii 2 framework as an example for this chapter? Let's become vile this time and make a *malicious* extension, which will provide a sort of **phishing backdoor** for us.

> Never do exactly the thing we'll describe in this chapter! It'll not give you instant access to the attacked website anyway, but a skilled **black hat** hacker can easily get enough information to achieve total control over your application.

The idea is this: our extension will provide a special route (a controller with a single action inside), which will dump the complete application configuration to the web page. Let's say it'll be reachable from the route `/app-info/configuration`.

We cannot, however, just get the contents of the configuration file itself and that too reliably. At the point where we can attach ourselves to the application instance, the original configuration array is inaccessible, and even if it were accessible, we can't be sure about where it came from anyway. So, we'll inspect the runtime status of the application and return the most important pieces of information we can fetch at the stage of the controller action resolution. That's the exact payload we want to introduce.

```
public function actionConfiguration()
{
    $app = \Yii::$app;
    $config = [
        'components' => $app->components,
        'basePath' => $app->basePath,
        'params' => $app->params,
        'aliases' => \Yii::$aliases
    ];
    return \yii\helpers\Json::encode($config);
}
```

The preceding code is the core of the extension and is assumed in the following sections.

In fact, if you know the value of the `basePath` setting of the application, a list of its aliases, settings for the components (among which the DB connection may reside), and all custom parameters that developers set manually, you can map the target application quite reliably. Given that you know all the credentials this way, you have an enormous amount of highly valuable information about the application now. All you need to do now is make the user install this extension.

Creating the extension contents

Our plan is as follows:

We will develop our extension in a folder, which is different from our example CRM application.

This extension will be named `yii2-malicious`, to be consistent with the naming of other Yii 2 extensions.

Given the kind of payload we saw earlier, our extension will consist of a single controller and some special wiring code (which we haven't learned about yet) to automatically attach this controller to the application.

Finally, to consider this subproject a true Yii 2 extension and not just some random library, we want it to be installable in the same way as other Yii 2 extensions.

Preparing the boilerplate code for the extension

Let's make a separate directory, initialize the Git repository there, and add the `AppInfoController` to it. In the bash command line, it can be achieved by the following commands:

```
$ mkdir yii2-malicious && cd $_
$ git init
$ > AppInfoController.php
```

Inside the `AppInfoController.php` file, we'll write the usual boilerplate code for the Yii 2 controller as follows:

```
namespace malicious;
use yii\web\Controller;
class AppInfoController extends Controller
{
    // Action here
}
```

Put the action defined in the preceding code snippet inside this controller and we're done with it. Note the namespace: it is not the same as the folder this controller is in, and this is not according to our usual auto-loading rules. We will explore later in this chapter that this is not an issue because of how Yii 2 treats the auto-loading of classes from extensions.

Now this controller needs to be wired to the application somehow. We already know that the application has a special property called `controllerMap`, in which we can manually attach controller classes. However, how do we do this automatically, better yet, right at the application startup time? Yii 2 has a special feature called **bootstrapping** to support exactly this: to attach some activity at the beginning of the application lifetime, though not at the very beginning but before handling the request for sure. This feature is tightly related to the extensions concept in Yii 2, so it's a perfect time to explain it.

FEATURE – bootstrapping

To explain the bootstrapping concept in short, you can declare some components of the application in the `\yii\base\Application::$bootstrap` property. They'll be properly instantiated at the start of the application. If any of these components implement the `BootstrapInterface` interface, its `bootstrap()` method will be called, so you'll get the application initialization enhancement for free. Let's elaborate on this.

The `\yii\base\Application::$bootstrap` property holds the array of generic values that you tell the framework to initialize beforehand. It's basically an improvement over the preload concept from Yii 1.x. You can specify four kinds of values to initialize as follows:

- The ID of an application component
- The ID of some module
- A class name
- A configuration array

If it's the ID of a component, this component is fully initialized. If it's the ID of a module, this module is fully initialized. It matters greatly because Yii 2 has lazy loading employed on the components and modules system, and they are usually initialized only when explicitly referenced. Being *bootstrapped* means to them that their initialization, regardless of whether it's slow or resource-consuming, always happens, and happens always at the start of the application.

> If you have a component and a module with identical IDs, then the component will be initialized and the module will not be initialized!

If the value being mentioned in the bootstrap property is a class name or configuration array, then the instance of the class in question is created using the `\yii\BaseYii::createObject()` facility. The instance created will be thrown away immediately if it doesn't implement the `\yii\base\BootstrapInterface` interface. If it does, its `bootstrap()` method will be called. Then, the object will be thrown away.

So, what's the effect of this bootstrapping feature? We already used this feature while installing the debug extension. We had to bootstrap the debug module using its ID, for it to be able to attach the event handler so that we would get the debug toolbar at the bottom of each page of our web application. This feature is indispensable if you need to be sure that some activity will always take place at the start of the application lifetime.

The `BootstrapInterface` interface is basically the incarnation of a command pattern. By implementing this interface, we gain the ability to attach any activity, not necessarily bound to the component or module, to the application initialization.

FEATURE – extension registering

The bootstrapping feature is repeated in the handling of the `\yii\base\Application::$extensions` property. This property is the only place where the concept of extension can be seen in the Yii framework. Extensions in this property are described as a list of arrays, and each of them should have the following fields:

- `name`: This field will be with the name of the extension.
- `version`: This field will be with the extension's version (nothing will really check it, so it's only for reference).
- `bootstrap`: This field will be with the data for this extension's Bootstrap. This field is filled with the same elements as that of `Yii::$app->bootstrap` described previously and has the same semantics.
- `alias`: This field will be with the mapping from Yii 2 path aliases to real directory paths.

When the application registers the extension, it does two things in the following order:

1. It registers the aliases from the extension, using the `Yii::setAlias()` method.
2. It initializes the *thing* mentioned in the bootstrap of the extension in exactly the same way we described in the previous section.

> Note that the extensions' bootstraps are processed before the application's bootstraps.

Registering aliases is crucial to the whole concept of *extension* in Yii 2. It's because of the Yii 2 PSR-4 compatible autoloader, which we have already discussed briefly in *Chapter 2, Making a Custom Application with Yii 2*.

Here is the quote from the documentation block for the `\yii\BaseYii::autoload()` method:

> *If the class is namespaced (e.g. yii\base\Component), it will attempt to include the file associated with the corresponding path alias (e.g. @yii/base/Component.php).*
>
> *This autoloader allows loading classes that follow the PSR-4 standard and have its top-level namespace or sub-namespaces defined as path aliases.*

The PSR-4 standard is available online at `http://www.php-fig.org/psr/psr-4/`.

Given that behavior, the `alias` setting of the extension is basically a way to tell the autoloader the name of the top-level namespace of the classes in your extension code base. Let's say you have the following value of the `alias` setting of your extension:

```
"alias" => [
    "@companyname/extensionname" => "/some/absolute/path"
]
```

If you have the `/some/absolute/path/subdirectory/ClassName.php` file, and, according to PSR-4 rules, it contains the class whose fully qualified name is `\companyname\extensionname\subdirectory\ClassName`, Yii 2 will be able to autoload this class without problems.

Making the bootstrap for our extension – hideous attachment of a controller

We have a controller already prepared in our extension. Now we want this controller to be automatically attached to the application under attack when the extension is processed. This is achievable using the bootstrapping feature we just learned. Let's create the `\malicious\Bootstrap` class for this cause inside the code base of our extension, with the following boilerplate code:

```
<?php

namespace malicious;

use \yii\base\BootstrapInterface;

class Bootstrap implements BootstrapInterface
{
    /** @param \yii\web\Application $app */
    public function bootstrap($app)
    {
        // Controller addition will be here.
    }
}
```

With this preparation, the `bootstrap()` method will be called at the start of the application, provided we wire everything up correctly. But first, we should consider how we manipulate the application to make use of our controller. This is easy, really, because there's the `\yii\web\Application::$controllerMap` property (don't forget that it's inherited from `\yii\base\Module`, though).

We'll just do the following inside the `bootstrap()` method:

```
$app->controllerMap['app-info'] = '\malicious\AppInfoController';
```

We will rely on the composer and Yii 2 autoloaders to actually find `\malicious\AppInfoController`.

> Just imagine that you can do anything inside the bootstrap. For example, you can open the CURL connection with some botnet and send the accumulated application information there. Never believe random extensions on the Web.

This actually concludes what we need to do to complete our extension.

All that's left now is to make our extension installable in the same way as other Yii 2 extensions we were using up until now. If you need to attach this malicious extension to your application manually, and you have a folder that holds the code base of the extension at the path `/some/filesystem/path`, then all you need to do is to write the following code inside the application configuration:

```
'extensions' => array_merge(
    (require __DIR__ . '/../vendor/yiisoft/extensions.php'),
    [
        'malicious\app-info' => [
            'name' => 'Application Information Dumper',
            'version' => '1.0.0',
            'bootstrap' => '\malicious\Bootstrap',
            'alias' => ['@malicious' =>
                '/some/filesystem/path']
                // that's the path to extension
        ]
    ]
)
```

> This type of extension attaching is illustrated in the *extension-manual-loading* branch of the Git repository in the code bundle provided with this book. It is accessible via the following console command:
>
> **`$ git checkout "extension-manual-loading"`**

Please note the exact way of specifying the `extensions` setting. We're merging the contents of the `extensions.php` file supplied by the Yii 2 distribution from composer and our own manual definition of the extension. This `extensions.php` file is what allows Yiisoft to distribute the extensions in such a way that you are able to install them by a simple, single invocation of a `require` composer command . Let's learn now what we need to do to repeat this feature.

Making the extension installable as... erm, extension

First, to make it clear, we are talking here only about the situation when Yii 2 is installed by composer, and we want our extension to be installable through the composer as well.

This gives us the baseline under all of our assumptions. Let's remember what extensions we have installed so far:

- Gii the code generator from *Chapter 3*, *Automatically Generating the CRUD Code*
- The Twitter Bootstrap extension from *Chapter 4*, *The Renderer*
- The Debug extension from *Chapter 7*, *Modules*
- The SwiftMailer extension from *Chapter 8*, *Overall Behavior*

We have installed all of these extensions using composer. We can remember now that we introduced the `extensions.php` file reference when we installed the Gii extension, and then we did not touch this part of the application configuration until now. Have a look at the following code:

```
'extensions' => (require __DIR__ .
  '/../vendor/yiisoft/extensions.php')
```

If we open the `vendor/yiisoft/extensions.php` file (given that all extensions from the previous chapters were installed) and look at its contents, we'll see the following code (note that in your installation, it can be different):

```
<?php

$vendorDir = dirname(__DIR__);

return array (
  'yiisoft/yii2-bootstrap' =>
  array (
    'name' => 'yiisoft/yii2-bootstrap',
    'version' => '9999999-dev',
    'alias' =>
    array (
      '@yii/bootstrap' => $vendorDir . '/yiisoft/yii2-bootstrap',
    ),
  ),
  'yiisoft/yii2-swiftmailer' =>
  array (
```

```
        'name' => 'yiisoft/yii2-swiftmailer',
        'version' => '9999999-dev',
        'alias' =>
        array (
          '@yii/swiftmailer' => $vendorDir . '
            /yiisoft/yii2-swiftmailer',
        ),
      ),
      'yiisoft/yii2-debug' =>
      array (
        'name' => 'yiisoft/yii2-debug',
        'version' => '9999999-dev',
        'alias' =>
        array (
          '@yii/debug' => $vendorDir . '/yiisoft/yii2-debug',
        ),
      ),
      'yiisoft/yii2-gii' =>
      array (
        'name' => 'yiisoft/yii2-gii',
        'version' => '9999999-dev',
        'alias' =>
        array (
          '@yii/gii' => $vendorDir . '/yiisoft/yii2-gii',
        ),
      ),
    );
```

One extension was highlighted to stand out from the others. So, what does all this mean to us?

- First, it means that Yii 2 somehow generates the required configuration snippet automatically when you install the extension's composer package
- Second, it means that each extension provided by the Yii 2 framework distribution will ultimately be registered in the `extensions` setting of the application
- Third, all the classes in the extensions are made available in the main application code base by the carefully crafted `alias` settings inside the extension configuration
- Fourth, ultimately, easy installation of Yii 2 extensions is made possible by some integration between the Yii framework and the composer distribution system

The magic is hidden inside the `composer.json` manifest of the extensions built into Yii 2. The details about the structure of this manifest are written in the documentation of composer, which is available at `https://getcomposer.org/doc/04-schema.md`. We'll need only one field, though, and that is `type`.

Yii 2 employs a special type of composer package, named `yii2-extension`. If you check the manifests of `yii2-debug`, `yii2-swiftmail` and other extensions, you'll see that they all have the following line inside:

```
"type": "yii2-extension",
```

Normally composer will not understand that this type of package is to be installed. But the main `yii2` package, containing the framework itself, depends on the special auxiliary `yii2-composer` package:

```
"require": {
    … other requirements ...
    "yiisoft/yii2-composer": "*",
```

This package provides Composer Custom Installer (read about it at `https://getcomposer.org/doc/articles/custom-installers.md`), which enables this package type.

The whole point in the `yii2-extension` package type is to automatically update the `extensions.php` file with the information from the extension's manifest file. Basically, all we need to do now is to craft the correct `composer.json` manifest file inside the extension's code base. Let's write it step by step.

Preparing the correct composer.json manifest

We first need a block with an identity. Have a look at the following lines of code:

```
"name": "malicious/app-info",
"version": "1.0.0",
"description": "Example extension which reveals important
  information about the application",
"keywords": ["yii2", "application-info", "example-extension"],
"license": "CC-0",
```

Technically, we must provide only `name`. Even `version` can be omitted if our package meets two prerequisites:

- It is distributed from some version control system repository, such as the Git repository
- It has **tags** in this repository, correctly identifying the versions in the commit history

And we do not want to bother with it right now.

Next, we need to depend on the Yii 2 framework just in case. Normally, users will install the extension after the framework is already in place, but in the case of the extension already being listed in the `require` section of `composer.json`, among other things, we cannot be sure about the exact ordering of the `require` statements, so it's better (and easier) to just declare dependency explicitly as follows:

```
"require": {
    "yiisoft/yii2": "*"
},
```

Then, we must provide the type as follows:

```
"type": "yii2-extension",
```

After this, for the Yii 2 extension installer, we have to provide two additional blocks; `autoload` will be used to correctly fill the `alias` section of the extension configuration. Have a look at the following code:

```
"autoload": {
    "psr-4": {
        "malicious\\": ""
    }
},
```

What we basically mean is that our classes are laid out according to PSR-4 rules in such a way that the classes in the `malicious` namespace are placed right inside the root folder.

The second block is `extra`, in which we tell the installer that we want to declare a `bootstrap` section for the extension configuration:

```
"extra": {
    "bootstrap": "malicious\\Bootstrap"
},
```

Our manifest file is complete now. Commit everything to the version control system:

```
$ git commit -a -m "Added the Composer manifest file to repo"
```

Now, we'll add the tag at last, corresponding to the `version` we declared as follows:

```
$ git tag 1.0.0
```

We already mentioned earlier the purpose for which we're doing this. All that's left is to tell the composer from where to fetch the extension contents.

Configuring the repositories

We need to configure some kind of repository for the extension now so that it is installable.

The easiest way is to use the Packagist service, available at `https://packagist.org/`, which has seamless integration with composer. It has the following pro and con:

- Pro: You don't need to declare anything additional in the `composer.json` file of the application you want to attach the extension to
- Con: You must have a public VCS repository (either Git, SVN, or Mercurial) where your extension is published

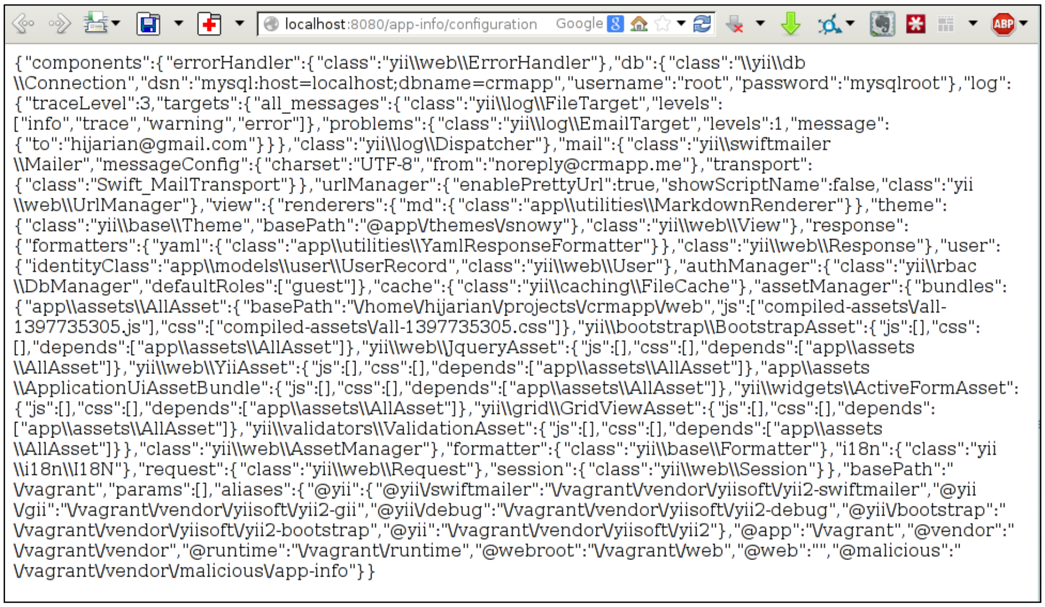
In our case, where we are just in fact learning about how to install things using composer, we certainly do not want to make our extension public.

> Do not use Packagist for the extension example we are building in this chapter.

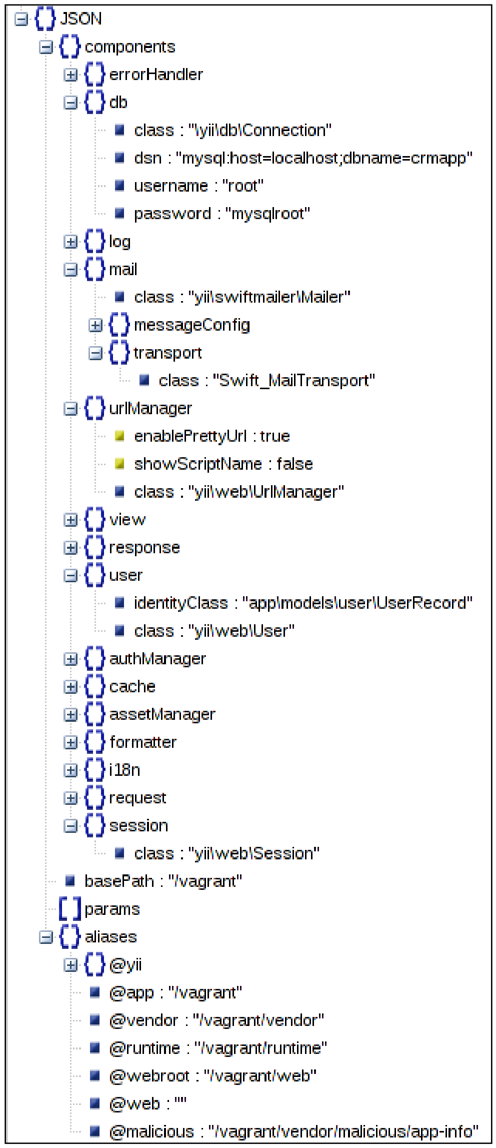
Let's recall our goal. Our goal is to be able to install our extension by calling the following command at the root of the code base of some Yii 2 application:

```
$ php composer.phar require "malicious/app-info:*"
```

After that, we should see something like the following screenshot after requesting the `/app-info/configuration` route:



This corresponds to the following structure (the screenshot is from the `http://jsonviewer.stack.hu/` web service):



Put the extension to some public repository, for example, GitHub, and register a package at Packagist. This command will then work without any preparation in the `composer.json` manifest file of the target application.

But in our case, we will not make this extension public, and so we have two options left for us.

The first option, which is perfectly suited to our learning cause, is to use the archived package directly. For this, you have to add the `repositories` section to `composer.json` in the code base of the application you want to add the extension to:

```
"repositories": [
    // definitions of repositories for the packages required by this
      application
]
```

To specify the repository for the package that should be installed from the ZIP archive, you have to grab the entire contents of the `composer.json` manifest file of this package (in our case, our `malicious/app-info` extension) and put them as an element of the `repositories` section, verbatim. This is the most complex way to set up the composer package requirement, but this way, you can depend on absolutely any folder with files (packaged into an archive).

Of course, the contents of `composer.json` of the extension do not specify the actual location of the extension's files. You have to add this to `repositories` manually. In the end, you should have the following additional section inside the `composer.json` manifest file of the target application:

```
"repositories": [
    {
        "type": "package",
        "package": {
// … skipping whatever were copied verbatim from the composer.json
  of extension...
            "dist": {
                "url": "/home/vagrant/malicious.zip", // example file
                  location
                "type": "zip"
            }
        }
    }
]
```

> JSON does not have comments, so remove them if you are going to the copy and paste code directly from the book.

This way, we specify the location of the package in the filesystem of the same machine and tell the composer that this package is a ZIP archive. Now, you should just zip the contents of the `yii2-malicious` folder we have created for the extension, put them somewhere at the target machine, and provide the correct URL. The URL we specified in the previous example is for the Vagrant setup described in *Appendix A, Deployment Setup with Vagrant*.

> Please note that it's necessary to archive only the contents of the extension and not the folder itself.

After this, you run composer on the machine that really has this URL accessible (you can use `http://` type of URLs, of course, too), and then you get the following response from composer:

```
vagrant@precise64:/vagrant$ php composer.phar require "malicious/app-info:1.0.0"
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing malicious/app-info (1.0.0)
    Downloading: 100%

Writing lock file
Generating autoload files
```

To check that Yii 2 really installed the extension, you can open the file `vendor/yiisoft/extensions.php` and check whether it contains the following block now:

```
'malicious/app-info' =>
array (
  'name' => 'malicious/app-info',
  'version' => '1.0.0.0',
  'alias' =>
  array (
    '@malicious' => $vendorDir . '/malicious/app-info',
  ),
  'bootstrap' => 'malicious\\Bootstrap',
),
```

(The indentation was preserved as is from the actual file.) If this block is indeed there, then all you need to do is open the `/app-info/configuration` route and see whether it reports JSON to you. It should.

The pros and cons of the file-based installation are as follows:

Pros	Cons
You can specify any file as long as it is reachable by some URL. The ZIP archive management capabilities exist on virtually any kind of platform today.	There is too much work in the `composer.json` manifest file of the target application. The requirement to copy the entire manifest to the `repositories` section is overwhelming and leads to code duplication.
You don't need to set up any version control system repository. It's of dubious benefit though.	The manifest from the extension package will not be processed at all. This means that you cannot just strip the entry in repositories, leaving only the `dist` and `name` sections there, because the Yii 2 installer will not be able to get to the `autoloader` and `extra` sections.

The last method is to use the local version control system repository. We already have everything committed to the Git repository, and we have the correct tag placed here, corresponding to the version we declared in the manifest. This is everything we need to prepare inside the extension itself. Now, we need to modify the target application's manifest to add the repositories section in the same way we did previously, but this time we will introduce a lot less code there:

```
"repositories": [
    {
        "type": "git",
        "url": "/home/vagrant/yii2-malicious/" // put your own URL
          here
    }
]
```

All that's needed from you is to specify the correct URL to the Git repository of the extension we were preparing at the beginning of this chapter. After you specify this repository in the target application's composer manifest, you can just issue the desired command:

```
$ php composer.phar require "malicious/app-info:1.0.0"
```

Everything will be installed as usual. Confirm the successful installation again by having a look at the contents of `vendor/yiisoft/extensions.php` and by accessing the `/app-info/configuration` route in the application.

The pros and con of the repository-based installation are as follows:

- Pro: Relatively little code to write in the application's manifest.
- Pro: You don't need to really publish your extension (or the package in general). In some settings, it's really useful, for closed-source software, for example.
- Con: You still have to meddle with the manifest of the application itself, which can be out of your control and in this case, you'll have to guide your users about how to install your extension, which is not good for PR.

In short, the following pieces inside the `composer.json` manifest turn the arbitrary composer package into the Yii 2 extension:

- First, we tell composer to use the special Yii 2 installer for packages as follows:

  ```
  "type": "yii2-extension"
  ```

- Then, we tell the Yii 2 extension installer where the bootstrap for the extension (if any) is as follows:

  ```
  "extra": {"bootstrap": "<Fully qualified name>"}
  ```

- Next, we tell the Yii 2 extension installer how to prepare aliases for your extension so that classes can be autoloaded as follows:

  ```
  "autoloader": {"psr-4": { "namespace": "<folder path>"}}
  ```

- Finally, we add the explicit requirement of the Yii 2 framework itself in the following code, so we'll be sure that the Yii 2 extension installer will be installed at all:

  ```
  "require": {"yiisoft/yii2": "*"}
  ```

Everything else is the details of the installation of any other composer package, which you can read in the official composer documentation.

> In the code accompanying this book, for all of the previous three options (except publishing with Packagist), there are three separate branches in the repository. So check out the `git branch` output, and choose what to look at. This extension is contained only in these branches; you'll never see it in the master branch anymore.

Summary

In this chapter, we looked at how Yii 2 implements its extensions so that they're easily installable by a single composer invocation and can be automatically attached to the application afterwards. We learned that this required some level of integration between these two systems, Yii 2 and composer, and in turn this requires some additional preparation from you as a developer of the extension.

We used a really silly, even a bit dangerous, example for extension. It was for three reasons:

- The extension was fun to make (we hope)
- We showed that using bootstrap mechanics, we can basically automatically wire up the pieces of the extension to the target application without any need for elaborate manual installation instructions
- We showed the potential danger in installing random extensions from the Web, as an extension can run absolutely arbitrary code right at the application initialization and more than that, at each request made to the application

We have discussed three methods of distribution of composer packages, which also apply to the Yii 2 extensions. The general rule of thumb is this: if you want your extension to be publicly available, just use the Packagist service. In any other case, use the local repositories, as you can use both local filesystem paths and web URLs. We looked at the option to attach the extension completely manually, not using the composer installation at all.

In the next chapter, *Chapter 10*, *Events and Behaviors*, we'll dive from the heights of the Yii 2 extensions, floating at the topmost point of the framework architecture, to the dark depths of the Yii 2 event system, spanning down to the framework's lowest levels.

10
Events and Behaviors

In this chapter, we'll return to our decoupled design introduced in *Chapter 2*, *Making a Custom Application with Yii 2*. We left the customer domain model as it was and explored the various separate helpful features of the Yii 2 framework. Now, let's go back to the customer model and implement some of its interesting and useful behaviors, which will undoubtedly be needed for any business application. While implementing this behavior, we'll learn about two related concepts from the Yii 2 framework: **events** and **behaviors**.

Automatically marking database records with the timestamp and user ID

Let's implement one useful feature, which you'll undoubtedly be asked to use one day by the client on a real enterprise-grade application project. It is described like this, in the words of some unknown Mr. Arbitrary Stakeholder:

> "*When a new Customer is recorded, a date and time of creation and the user who did it, should be recorded. Upon any update of the Customer in the database, a date and time of the update and the user doing it should be recorded.*"

We, as developers, will translate it to the following specification:

1. Add four fields to the `customer` table in the database, which are as follows:
    - `created_at`: This field is of the `integer` type, holding a Unix timestamp
    - `created_by`: This field is of the `integer` type, and is a foreign key to the `user.id` field

    - `updated_at`: This field is of the `integer` type, holding a Unix timestamp
    - `updated_by`: This field is of the `integer` type, and is a foreign key to the `user.id` field

2. When a new customer record is added to the database, automatically fill the `created_at` field with the current timestamp and the `created_by` field with the ID of the user currently logged in. Automatically fill the `updated_at` field with the same value as the `created_at` field, and the `updated_by` field with the same value as the `created_by` field.
3. When some customer record is updated in the database, automatically fill the `updated_at` field with the current timestamp and the `updated_by` field with the ID of the user currently logged in.

To keep things simple, we will use the Unix timestamps instead of proper `DATETIME` fields.

Test case for customer creation

We will be blunt and encode the given specifications as the integration test, checking the actual state of the database. The resulting test case will be quite long, so instead of writing it completely as a single listing, we'll describe its construction step by step. We'll start from the case of the new customer record creation.

The following are the steps for this:

1. First, we generate the test as follows:

   ```
   $ ./cept generate:test functional CustomerAudit
   ```

2. Then we clean up the generated test case in the `tests/functional/CustomerAuditTest.php` file until it looks like the following:

   ```
   class CustomerAuditTest extends \Codeception\TestCase\Test
   {
       /**
       * @var \FunctionalTester
       */
       protected $tester;
       /** @test */
       public function NewCustomerHasAuditInfo()
       {
           // Test scenario here...
       }
   }
   ```

We will not need `before()` or `after()` methods here because it's a single scenario we're testing here. All of the following will apply to the contents of the `NewCustomerHasAuditInfo()` test method.

3. After this, we declare the dependencies for the scenario. For simplicity, we will work with the database using active records directly. Also, as we need to store the ID of the currently logged-in user, we need the `UserRecord` class instance of some user known to us beforehand and a system for handling authentication. We have not decoupled from the Yii in the authentication mechanics, so we'll just use `yii\web\User Component` directly. This is not a good style, but in functional tests, we have a Yii singleton lying around anyway. Have a look at the following code:

```
// Dependencies
$identity = UserRecord::findOne(['username' =>
  'RobAdmin']);
$user = Yii::$app->user;
```

We will use the administrator account for our test.

4. As a preparation step, we log in, imagine a customer record, and store it in the database as follows:

```
// Given
$user->login($identity);
$customer = $this-> imagineCustomerRecord();
$before = time();
$customer->save();
$after = time();
```

We imagine a customer record in the following way:

```
private function imagineCustomerRecord()
{
    $faker = \Faker\Factory::create();
    $record = new CustomerRecord();
    $record->name = $faker->name;
    return $record;
}
```

For simplicity, we only set the required attribute of the customer, which is his/her name.

We expect a timestamp to be saved, so to be able to test its correctness, we remember the timestamps before and after the saving of the record.

5. Then, we do nothing more than fetch the saved customer from the database immediately:

```
// When
$saved = CustomerRecord::findOne($customer->id);
```

We use the customer record ID which was generated with Yii 2 when we saved the `$customer` object to the database. This ID is fed to the `findOne()` query method to construct a new active record instance, so we can be sure that we use the data actually stored in the database and not an in-memory record.

6. Finally, we check our assumptions about the customer we fetched as follows:

```
// Then
$this->assertInstanceOf
  ('app\models\customer\CustomerRecord', $saved);
$this->assertBetween
  ($before, $saved->created_at, $after);
$this->assertEquals($user->id, $saved->created_by);
$this->assertEquals
  ($saved->created_at, $saved->updated_at);
$this->assertEquals
  ($saved->created_by, $saved->updated_by);
```

We'll check the correctness of the `created_at` timestamp by a custom assertion called `assertBetween()`, which should check whether its three arguments are in order of increasing value. We are yet to create it. We'll check the correctness of the `created_by` field by comparing its value with the value of `Yii::$app->id` because we have not logged out of the system after creating the customer record. Therefore, it should still hold the same ID of the administrator account.

As stated in the specification, values of `updated_at` and `updated_by` should be equal to `created_at` and `created_by`, correspondingly.

> We basically do not need to even touch the UI to implement the feature in question and be sure that it works. This test case and the one following it will cover everything needed. To test it manually through the UI, we would need to make changes to the currently existing UI.

7. The custom assertion we're using to check the correctness of the timestamp is implemented extremely simply and generically:

```
private function assertBetween($before, $value, $after)
{
```

```
            $this->assertLessThanOrEqual($before, $value);
            $this->assertGreaterThanOrEqual($after, $value);
        }
```

We're comparing the integers in our case, but due to the nature of built-in assertions used, we can compare anything that is comparable in PHP with this assertion, such as strings.

Now, as we cheat and know beforehand what the correct production code satisfying this test case is, we will write the test for updating a customer record next.

Test case for updating customer updates

Let's make the second test method in the same `\CustomerAuditTest` test class as follows:

```
        /** @test */
        public function CustomerRecordRemembersUpdateDatetimeAndUser()
        {
            // Test scenario here...
        }
    }
```

Let's follow the same steps as in the previous section, which were "Dependencies", "Given", "When", "Then".

1. We still depend on the `yii\web\User` class here, and we will use two user identities, because we really want to check changes in the `updated_by` attribute:

   ```
   // Dependencies
   $first_identity = UserRecord::findOne
     (['username' => 'RobAdmin']);
   $second_identity = UserRecord::findOne
     (['username' => 'AnnieManager']);
   $user = Yii::$app->user;
   ```

2. As a preparation, we log in as the first identity, save the customer record, and remember the initial values of the `updated_at` and `updated_by` fields:

   ```
   // Given
   $user->login($first_identity);
   $record = new CustomerRecord;
   $record-> name = 'John';
   ```

```
$record->save();

$initial_updated_at = $record->updated_at;
$initial_updated_by = $record->updated_by;
```

3. In the most interesting part of the test, we log out from the first user account, log in to the second user account, and resave the record:

   ```
   // When
   $user->logout();
   sleep(1);
   $user->login($second_identity);
   $record-> name = 'Bill';
   $record->save();
   ```

   Note the call to `sleep()`. Unix timestamps have precision in seconds, and our test most possibly will run much faster than that. For `updated_at` to change by any amount, we need to wait for at least a second.

   > We *must* change some field in the record as if nothing were changed in the `ActiveRecord` class. The call to `save()` will not bother with actually going to the database. It's a really nice feature, by the way.

4. Assertions in this case are a lot simpler, because we just need to check that the current values of `updated_at` and `updated_by` fields differ from the initial ones:

   ```
   // Then
   $this->assertGreaterThan
     ($initial_updated_at, $record->updated_at);
   $this->assertNotEquals
     ($initial_updated_by, $record->updated_by);
   $this->assertEquals($user->id, $record->updated_by);
   ```

We do not bother checking the accuracy of the timestamp in our case, but of course, we do check that the user ID is actual.

We log in and log out of the system extensively during these tests. We need to be sure that we are logged out after each test run, to preserve a clean state for all other tests. Here is the appropriate teardown method to achieve that:

```
public function _after()
{
    Yii::$app->user->logout();
}
```

Now, we run the tests and watch them fail:

```
$ ./cept run functional
```

The expected outcome will be as follows:

```
Time: 3.29 seconds, Memory: 17.00Mb

There were 2 errors:

---------
1) CustomerAuditTest::NewCustomerHasAuditInfo
yii\base\UnknownPropertyException: Getting unknown property: app\models\customer
\CustomerRecord::created_at

#1  /vagrant/vendor/yiisoft/yii2/db/BaseActiveRecord.php:246
#2  /vagrant/tests/functional/CustomerAuditTest.php:39

---------
2) CustomerAuditTest::CustomerRecordRemembersUpdateDatetimeAndUser
yii\base\UnknownPropertyException: Setting unknown property: app\models\customer
\CustomerRecord::first_name

#1  /vagrant/vendor/yiisoft/yii2/db/BaseActiveRecord.php:266
#2  /vagrant/tests/functional/CustomerAuditTest.php:56

FAILURES!
Tests: 8, Assertions: 21, Errors: 2.
```

Let's start with the fields required in the database records.

Preparing the database fields

As usual, we make a migration as follows:

```
$ ./yii migrate/create add_audit_fields_to_customer
```

Let's write the following code as a migration script:

```
$this->addColumn('customer', 'created_at', 'integer');
$this->addColumn('customer', 'created_by', 'integer');
$this->addColumn('customer', 'updated_at', 'integer');
$this->addColumn('customer', 'updated_by', 'integer');

$this->addForeignKey('customer_created_by', 'customer',
  'created_by', 'user', 'id');
$this->addForeignKey('customer_updated_by', 'customer',
  'updated_by', 'user', 'id');
```

Foreign keys are not really required by our feature, but for the sake of completeness, we include them, because they will be needed anyway in a real-world setting. Run the migration:

```
$ ./yii migrate
```

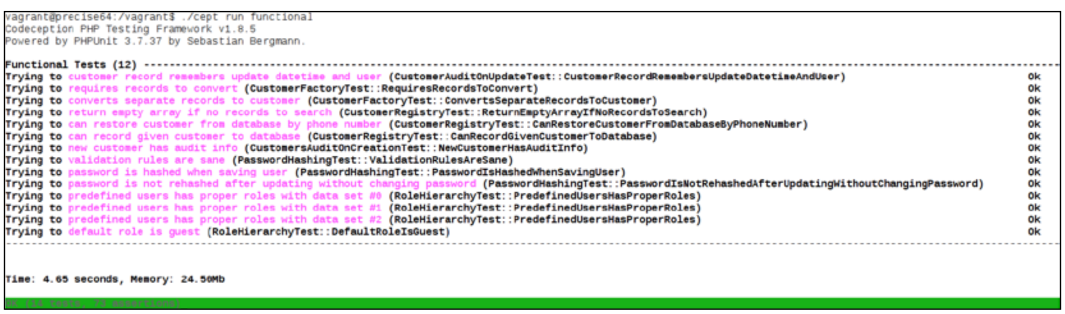
That's all. The `CustomerRecord` class will automatically pick these fields. What code do we need to write next to finally satisfy our tests?

Using the timestamp and blameable behaviors

We need to write the following code into our `CustomerRecord` class:

```
public function behaviors()
{
    return [
        'timestamp' =>
          \yii\behaviors\TimestampBehavior::className(),
        'blame' =>
          \yii\behaviors\BlameableBehavior::className()
    ];
}
```

As a result, our tests magically pass, which can be seen in the following screenshot:



What did we actually do here?

In high-level terms, the `behaviors()` method on `yii\base\ActiveRecord` exists to declare some classes that will provide additional functionality to the active record in question. This functionality can be of two forms:

- New methods available to use
- Event handlers attached

Actually, everything that is an inheritor of `\yii\base\Component` has the `behaviors()` method, so you can attach something to the controllers, modules and application components too.

With the `CustomerRecord` class, we attached two behaviors, `\yii\behaviors\TimestampBehavior` and `\yii\behaviors\BlameableBehavior`, which are built in to the Yii 2 framework.

We used the shorthand syntax to specify the behaviors to attach. The full version is to assign not just the fully-qualified name of the class, but the associative array with the values of properties for this class as follows:

```
public function behaviors()
{
    return [
        'timestamp' => [
            'class' =>
                \yii\behaviors\TimestampBehavior::className(),
            // other settings here ...
        ]
        'blame' => [
            'class' =>
                \yii\behaviors\BlameableBehavior::className()
            // other settings here ...
        ]
    ];
}
```

In fact, this is the same syntax that is parsable by `\yii\BaseYii::createObject`, which we discussed several chapters ago, accepting arrays, class names, or callable instances.

```
Both TimestampBehavior and BlameableBehavior extend the more generic
\yii\behaviors\AttributeBehavior. The following is the example from
the official documentation:
public function behaviors()
{
    return [
        'attributeStamp' => [
            'class' => AttributeBehavior::className(),
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT =>
                    ['attribute1', 'attribute2'],
                ActiveRecord::EVENT_BEFORE_UPDATE => 'attribute2',
```

```
            ],
            'value' => function ($event) {
                return 'some value';
            },
        ],
    ];
}
```

There are two settings in AttributeBehavior:

- The attributes setting: This is an array that maps event names from the ActiveRecord class to the names of the attributes. You can specify several attributes for a single event using arrays.
- The value setting: This is the callable instance receiving the event being reacted at by AttributeBehavior. This callable instance should return the value to be assigned to the attribute associated with this event. Instead of a callable instance, you can specify a fixed value. Of course, you should take care about the type of value being something that ActiveRecord can process; it cannot magically save object or array instances to the database.

The timestamp behavior is a special case of AttributeBehavior. It sets the attribute setting according to the following code:

```
BaseActiveRecord::EVENT_BEFORE_INSERT =>
  ['created_at', 'updated_at'],
BaseActiveRecord::EVENT_BEFORE_UPDATE => 'updated_at',
```

This setup exhibits exactly the behavior we specified at the start of this section, filling up the created_at and updated_at fields at the appropriate steps of the ActiveRecord lifetime. This was our first cheating, when we named the fields according to default values of TimestampBehavior.

By default, it uses the value of the call to time() as the value setting. You can do the same with setting value to the following in the AttributeBehavior:

```
function ($event) {
    return time();
},
```

Our second cheating was when we used timestamps as a value of the created_at and updated_at fields in the database records.

Blameable behavior is also a special case of `AttributeBehavior`. It sets the `attribute` setting in almost the same way as `TimestampBehavior`, but the names of the fields are different:

```
BaseActiveRecord::EVENT_BEFORE_INSERT => ['created_by', 'updated_by'],
BaseActiveRecord::EVENT_BEFORE_UPDATE => 'updated_by',
```

So, we cheated a third time here, again with the choice for the field names.

The default value that blameable behavior sets for these attributes is the current user ID, fetched simply from `Yii::$app->user->id`. So, with the traditional authentication mechanics that we implemented in *Chapter 5*, *User Authentication*, we had everything in place for us without any additional setup.

It should be mentioned that all of the previous cheating was needed only for the awesomeness of making two functional tests pass by adding a method with four lines in it (two of them being the array's literal syntax). Overall, the concept stays the same: in Yii 2 (and, to be honest, back in Yii 1.x, too), you can attach some behavior defined in separate classes to objects. Let's look at the concept of behaviors in more detail.

FEATURE – behaviors

The three behavior classes, `TimestampBehavior`, `BlameableBehavior`, and `AttributeBehavior`, described earlier are the only behaviors built into the Yii 2 framework. The behavior concept is implemented by the `\yii\base\Behavior` class, so nothing stops you from implementing your own behaviors by extending it.

The basic idea behind behaviors was explained before: you can attach behavior to some other object, and this object will get the methods and properties defined on that behavior. What this means is that behavior is the same as the concept of trait, introduced in PHP 5.4 (read about traits at `http://www.php.net/manual/en/language.oop5.traits.php`). In fact, behaviors were introduced to Yii 1.x to use the features of traits before they were implemented in PHP natively.

But behaviors in Yii 2 have another feature loaded: they also attach event handlers to the object they're being bound to. This is exactly what enabled the functionality of `AttributeBehavior` earlier.

Let's look at the four distinctive pieces a behavior class is composed of:

- First, a behavior has an owner. Inside the methods of behavior, you can rely on the `$this->owner` object to be the object we are currently being attached to.
- Second, a behavior has a special method named `events`. This method can (and should) be overridden in the subclasses, and it must return an array that maps event names to callable instances.
- Lastly, a behavior has `attach($component)` and `detach($component)` methods, which you probably will never need to override. Using them, you can attach a behavior to a `$component` object, and it will attach all event handlers to it. Note that it will not provide methods and properties of a behavior to the owner! So, while these methods are there for you to use, they're basically useless in reality.

Event names specified in `events()` should be events of the owner, for reasons that will be described later. This is extremely important, but at the same time it's quite handy, as the events which the owner does not fire will just silently be ignored.

You can assign any kind of callable instance allowed in PHP as an event handler, and additionally, you can specify just a string, which will be interpreted as the name of the method in the behavior itself (so instead of writing `[$this, 'handlerName']` you can just write `'handlerName'`). The event handler will receive the `\yii\base\Event` object as a sole argument.

To properly attach and detach behaviors, you need to use the four special methods of `\yii\base\Component`:

Method name	**Meaning**
`attachBehavior($name, $behavior)`	It means attach a `$behavior` behavior to this component under `$name`.
`attachBehaviors($behaviors)`	It means attach an entire array of behaviors to this component. The array should map names to behaviors, like arguments, to `attachBehavior()`.
`detachBehavior($name)`	It means detach a named behavior from this component.
`detachBehaviors()`	Here, `detachBehavior()` on all behaviors of the component will effectively remove all behaviors.

By *attaching* a behavior to a component, you make event handlers and methods declared in the behavior available to the component. By *detaching* a behavior, you make these event handlers and methods unavailable.

As an example, we can take `TimestampBehavior`, which declares the `touch($attribute)` helper method. We know that we don't have this method on the `UserRecord` model, but we can do the following:

```
$user = UserRecord::findOne($id);
$behavior = new TimestampBehavior();
$behavior->value = function () { return date('Y-m-d'); };
$user->attachBehavior('ts', $behavior);
```

After that, let's imagine we have the `lastLoggedDatetime` attribute on `UserRecord`, and then we can perform the following action:

```
$user->touch('lastLoggedDatetime');
```

The `lastLoggedDatetime` attribute will then hold the value of `date('Y-m-d')` at that moment.

To relieve you from the hassle of attaching behaviors manually, each component in Yii 2 has a special `behaviors()` method defined. By overriding this method, you can specify the behaviors to be attached to this component at the moment of creating its instance. You already know that almost anything in Yii 2 is a descendant of `\yii\base\Component`. An example of attaching behaviors using the `behaviors()` method was provided earlier with the customer model.

> It's important to remember that the properties of the behavior will be available to the owner object, too. One should understand that in Yii 2, if you have a set of `getSomething()` and `setSomething()`, then you have a "something" property even if you don't have `$something` defined in the class at all. The source code for the `__call()`, `__get()` and `__set()` magic methods inside the `\yii\base\Component` class definition contains the exactly the description of the mechanics behind this trait-like behavior of behaviors.

FEATURE – events

The concept of events in itself is almost an observer pattern incarnate (see `http://c2.com/cgi/wiki?ObserverPattern`). An object at some point in time can realize it is going to meet some event. If this object has any event handlers associated with this event, it executes these handlers. Then the object continues to do whatever it did before. So, there are no explicit observers, just the separate, distinct functions attached to observable as event handlers.

This concept is implemented by three methods and one class in Yii 2 as follows:

- The `\yii\base\Component::on($name, $handler, $data = null, $append = true)` method assigns the `$handler` event handler to the event named `$name`. Depending on `$append`, this `$handler` will be either appended or prepended to the possible list of handlers already assigned to this event. By specifying `$data`, you can pass some arbitrary data to `$handler` (more about this later).
- The `\yii\base\Component::off($name, $handler = null)` method removes the `$handler` event handler from the list of handlers for the event named `$name`. There is some magic here: if you do not provide `$handler` to this method, all handlers will be removed. If, on the other hand, you do provide `$handler`, the handler which is equal to `$handler` will be removed. A comparison is done using the simple `==` operator, so you cannot detach the handler specified by an anonymous function, for example.
- The `\yii\base\Component::trigger($name, $event = null)` method triggers the event named `$name`, which simply means that it calls all of the event handlers associated with this event. The `$event` argument, if not empty, must be a `\yii\base\Event` descendant. If this argument is left empty, `trigger()` will create an instance of `\yii\base\Event` by itself.

How can you use this system? Just as with the observer pattern, an events system like this is useful when you need to perform some actions in response to something that happened in a different layer of the application. Another important benefit is that you can dynamically change the behavior that should emerge in response to the events, which can be useful, too. And of course, we can attach any number of event handlers to the single event.

Let's look at the problem of handling user-generated content, specifically the uploading of photos. We assume that you have a separate image component attached to the application, which is accessible through the `Yii::$app->images` call. This component has one method available, which is named `handleUpload`; so when you upload an image, you ultimately call `Yii::$app->images->handleUpload($_FILES)`.

Let's say, you already have the business rule that all uploaded photos are to be downscaled to, at the most, 2,000 pixels. So, in this photo-handling routine, you have an appropriate call to the background task (however, you have implemented the background tasks in your application), which calls some separate library to perform the conversion.

Imagine that at some point one of the stakeholders tells you that we need to additionally copy this uploaded image (unchanged) to a separate subsystem of the application, where the manager will manually review it for copyright violations. Of course, this will add just a call to another function into your photo-handling routine, but there's a very high possibility that feature requests like this will continue to come.

Instead of this, the `handleUpload()` routine can do just one thing: put the image to a known temporary location where it will not be cleaned up by the underlying OS (like the uploaded images in PHP usually end up). After that it will run the following line of code:

```
$this->trigger(self::IMAGE_UPLOADED, $event);
```

Thus, it will announce to all interested parties that the image is indeed uploaded. The `$event` parameter is the new instance of `\yii\base\Event` loaded with information about the filesystem path to the uploaded image.

Given this setup, we can declare in an appropriate place in the application that we want to downscale the uploaded image and put the result to the database as follows:

```
Yii::$app->images->on(Images::IMAGE_UPLOADED,
  $downscale_and_save);
```

Here, the `$downscale_and_save` parameter is the callable instance in the following form:

```
function ($event) {
    // getting the path to file from $event parameter…
    // downscaling the image
    // saving it to the database
}
```

At another appropriate place (for example, in the `init()` method of the controller, which will render the image uploading UI), you place another declaration of the event handler for the `IMAGE_UPLOADED` event. This handler will publish the image to some special part of the administrative UI for a copyright violations review.

It's hard to justify the value of the `off()` method in PHP and where the application starts, works, and dies for each separate HTTP request. It's most useful in systems that silently wait for user input and respond to events happening all of the time. With the `off()` method able to remove event handlers, a system can virtually reconfigure itself when needed, depending on some factors. When your script lives only for the span of a single HTTP request, you usually don't need this level of polymorphism. However, your mileage may vary.

The `\yii\base\Event` class represents information about the event that occurred. It has four pieces of information associated with it:

- `$name`: This is the name of this event object. When you call `trigger()`, you need to pass an event name to it as a first argument. That name is what will be the value of this `$name` attribute. You have no control over this attribute, even if you pass the manually constructed `$event` in the call to `trigger()`.
- `$sender`: This is the object that triggered the event. If it is not set manually, `trigger()` sets it to `$this`, which means that by default, you can rely on the value of `$sender`, it being the object that called `trigger()`.
- `$data`: This is the additional data associated with the handler when you set it by the `on()` method. So, at the time of attaching an event handler, you can specify some data to be available inside the handler via `$event->data`. We need to stress that it's the data calculated not at the time of the `trigger()` call, but at the time of the `on()` call.
- `$handled`: This is the special flag (initially set to `false`) that, when set to `true`, stops handling this event. So, if there were any additional handlers specified for this event, they will not fire. This is why the `$append` argument to the `on() call:` order of event handlers attached does matter.

The `on()`, `off()`, and `trigger()` methods given earlier are specified in terms of some class instance, that is, a specific object will trigger the event handlers attached to it exclusively. But sometimes, it's useful not to have object-bound but class-bound event handlers that will fire regardless of the exact instance triggering the event. There are three methods in the `\yii\base\Event` class that provide this functionality. They are as follows:

- The `\yii\base\Event::on($class, $name, $handler, $data = null, $append = true)` method attaches the `$handler` to the event named `$name` firing on any instance of `$class`. The `$class` is a fully-qualified class name. The semantics of the `$data` and `$append` parameters are the same as in the `Component.on()` method.
- The `\yii\base\Event::off($class, $name, $handler = null)` method removes the `$handler` associated with the event named `$name` from the list of handlers for `$class`. `$class` is a fully-qualified class name and works in the same manner as the `Component.off()` method.
- The `\yii\base\Event::trigger($class, $name, $event = null)` method, similar to `\yii\base\Component::trigger($name, $event = null)`, triggers the event handlers for the event named `$name` defined for `$class` passing `$event` to them. `$class` is either a fully-qualified class name or an object. If it is an object, then `trigger()` will extract the class name from it. If `$class` is a class name and not an object, then the `$sender` property of the event object passed to the handlers will be null.

For example, let's say your business need is to send an e-mail each time a new user is registered in your application. Why should the `createUser()` method, whose only responsibility is to create a new user record in the database, also call the `mail()` function? This will bloat it and violate the **Single Responsibility Principle** (look, for example, `http://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html`). It will be a cleaner solution to use the class-based event propagation instead.

The main idea would be this: we set up the handler for the event of inserting the new `UserRecord` (or whatever its name is in your code base) inside some initialization code on an appropriate layer. For example, in the `init()` method of the whole application, we may put the following line:

```
Event::on(
    '\app\models\user\UserRecord',
    \yii\db\ActiveRecord::EVENT_AFTER_INSERT,
    $send_email
);
```

The `$send_email` variable is the callable instance in the following form:

```
function ($event) {
    // extracting the data about UserRecord from $event parameter…
    // sending the email
}
```

It's quite important to know that when you trigger an event on the object, you trigger the event with the same name on the class of this object. This means, triggering an object-bound event triggers a class-bound event. Or, simply put, `\yii\base\Component::trigger()` calls `\yii\base\Event::trigger()` as the last action, passing its own arguments.

> If the event object passed to the object-bound `trigger()` has the `$handled` flag set to `true` and its handling stops, it will still be passed to the class-bound `trigger()` and will get the `$handled` flag reset to `false` again. You cannot prevent the class-bound events from occurring by manipulating the `$handled` flag.

In the preceding section, we briefly mentioned the `events()` method on the behavior class meant to be overridden to specify the event handlers for the owner of this `Behavior`. Now it should be quite obvious what the behavior does internally when it gets attached to its owner using the `attach()` method: it calls `$owner->on()` on each event-handler pair defined in the `events()` method. This is exactly why the behavior should better declare only the events expected from its future owner, because it's the owner who calls `trigger()`, and there's only a limited choice of events it is triggering. Any other event handler declared in the `Behavior.events()` method has the chance of never being called.

Now, have a look at the following code:

```
$behavior->attach($owner);
```

Then, have a look at this code:

```
$owner->attachBehavior($behavior);
```

We can clearly see the distinction between the preceding two codes. The `$behavior->attach()` form will only call `$owner->on()` for all events listed inside its `events()` method. The `$owner->attachBehavior()` form will do the same and also register the `$behavior` inside the `$owner` component, so it'll be able to use properties and methods defined on `$behavior`.

The events system in Yii 2 allows you to write your application at least partially in the event-based paradigm, which is quite cool by itself. But what makes it really useful right from the start is that a lot of built-in components in the framework already have some events triggering for you. So, you can hook onto various stages of the lifetime of the components, such as active records, controllers, and modules (by the way, it's exactly why we could use the `AttributeBehavior` descendants previously). If `ActiveRecord` would not `trigger()` `EVENT_BEFORE_INSERT` and `EVENT_BEFORE_UPDATE`, then all of the previous manipulations would be meaningless. In the next section, we'll take a look at the events that are already there for you to use.

Built-in events

Here, we'll save you from a full-text search through the quite extensive code base of Yii 2 and will list everything starting with `EVENT_` defined in the framework. Note that you should provide a string as a `$name` argument in the `on()` method. All built-in events define these strings as class constants, and you should always use these class constants when attaching event handlers instead of its actual values.

Events of \yii\base\Application

Both the web and the console applications shipped with Yii 2 will trigger the following events. Nothing will be passed as `$event` to `trigger()`, so expect a default event object. Then, we can proceed to your handlers. Have a look at the following table:

Event name	**When is it triggered**
`EVENT_BEFORE_REQUEST`	It is triggered right before starting the request handling, basically at the beginning of the application lifetime. This is the earliest event taking place in the application that you can hook on to.
`EVENT_AFTER_REQUEST`	It is triggered right after successful request handling or when you call `Yii::$app->end()`. This forcibly shuts the application down. This is not the last event occurring in the application; `\yii\web\Response::EVENT_AFTER_SEND` is the last one.

Events of \yii\base\Controller

Both the web and the console controllers will trigger the following events. Note that the instance of `\yii\base\ActionEvent` will be passed to the handlers, so check the documentation for this class at `http://www.yiiframework.com/doc-2.0/yii-base-controller.html`. It has some nice additional features. Have a look at the following table:

Event name	**When is it triggered**
`EVENT_BEFORE_ACTION`	It is triggered right before running any controller action. If the controller belongs to a module, then this module's `EVENT_BEFORE_ACTION` is triggered first. As was said before, an instance of `ActionEvent` will be passed to the handlers of this event. You may set its `isValid` attribute to `false`, and this will prevent the action in question from happening (the application will shut down without rendering anything, only further events will be triggered).
`EVENT_AFTER_ACTION`	It is triggered right after the execution of the controller action. Note that the response object generated by this action will be available in the `result` field of the `ActionEvent` passed to the handler, so you can post-process it in some way.

Note that both of these events are triggered inside the `beforeAction()` and `afterAction()` methods. With the Yii 1.1.x framework, it was quite traditional to override these methods in the child classes to achieve some pre- or post-processing effects. While you can do the same in Yii 2, you should always call `parent::beforeAction()` and `parent::afterAction()`, respectively, otherwise, you will prevent the triggering of the earlier events.

It's better to use proper event handlers instead of overriding the `beforeAction()` and `afterAction()` methods.

Events of \yii\base\Module

The semantics of the `\yii\base\Module` events are exactly the same as those of `\yii\base\Controller`. But they are called *around* the respective controller events. Have a look at the following table:

Event name	**When is it triggered**
`EVENT_BEFORE_ACTION`	This event is triggered before running any controller action and before the same event on the controller is triggered. An instance of `ActionEvent` will be passed to the handlers of this event. You may set its `isValid` attribute to `false` and this will prevent the action in question from occurring (the application will shut down without rendering anything, only further events will be triggered).
`EVENT_AFTER_ACTION`	This event is triggered after the controller action has finished working and after the same event on the controller is triggered. Though we have to note here that this event *always* fires, even if the action was disabled by setting the `isValid` attribute to `false` in the `EVENT_BEFORE_ACTION` event.

Modules have their `beforeAction()` and `afterAction()` methods, too. Look at the *Events of \yii\base\Controller* section for an explanation about complications raised by this.

Events of \yii\base\View

Please note that the view component used in web pages is `\yii\web\View`, and it defines two additional events you can hook on to. This particular class defines only the basic events in the rendering process. Have a look at the following table:

Event name	When is it triggered
`EVENT_BEGIN_PAGE`	The call to `$this->beginPage()` inside the view file triggers this event. This method is described in *Chapter 4*, *The Renderer*, in great detail and is used in the layout framework.
`EVENT_END_PAGE`	The call to `$this->endPage()` inside the view file triggers this event. This method is described in *Chapter 4*, *The Renderer*, in great detail and is used in the layout framework.
`EVENT_BEFORE_RENDER`	When rendering a view file, this will trigger before actually rendering it. The handlers will receive the instance of `\yii\base\ViewEvent`, and you can set its `isValid` attribute to `false` to prevent rendering from occurring. The actual call to `trigger()` for this event is wrapped into the method named `beforeRender()`, and so if you override it, you should call the `parent::beforeRender()` or lose the event handlers for this event. The base `beforeRender()` returns exactly the value of `ViewEvent.isValid`.
`EVENT_AFTER_RENDER`	When rendering a view file, this will trigger after the view file is rendered. The handlers will receive the instance of `\yii\base\ViewEvent`, and you can use its `output` attribute to post-process the output in some way. The actual call to `trigger()` for this event is wrapped into the method named `afterRender()`, and so if you override it, you should call `parent::afterRender()` or lose the event handlers for this event. The base `afterRender()` has `$output` of rendering as a sole output argument.

Events of \yii\web\View

Let's look at the additional events declared specifically for the view component used in the web application. Have a look at the following table:

Event name	When is it triggered
`EVENT_BEGIN_BODY`	The call to `$this->beginBody()` inside the view file triggers this event. This method is described in *Chapter 4*, *The Renderer*, in great detail and is used in the layout framework.
`EVENT_END_BODY`	The call to `$this->endBody()` inside the view file triggers this event. This method is described in *Chapter 4*, *The Renderer*, in great detail and is used in the layout framework. This is a good place to work with `AssetBundles`, as right after triggering this event, the assets are registered in the view.

Events of \yii\base\Model

The base model class is the foundation of active records. Only the most basic events are defined here, but they are applicable to the `ActiveRecord` lifetime too. Have a look at the following table:

Event name	When is it triggered
`EVENT_BEFORE_VALIDATE`	It is triggered right before validating attributes by the `validate()` method (remember that `save()` also calls `validate()` if you do not suppress this behavior). An instance of `\yii\base\ModelEvent` is passed to the handler of this event. You can use the `isValid` attribute of this event's subclass to control whether the model should be considered validated. If `isValid` is `false`, then the model fails the validation even without the validators firing, so be wary of this feature, because this way you will not get any automatic feedback about the reasons for failure.
`EVENT_AFTER_VALIDATE`	It is triggered right after all validators finish their work. Nothing special is being passed to the handlers of this event. This event is there for you to do some post-processing of the attributes of the model in question.

Note that similar to the `beforeAction()` and `afterAction()` methods of the controller, the triggering of these events is wrapped into the `beforeValidate()` and `afterValidate()` methods, with all warnings about overriding them applicable.

Remember that you can access the model that is being validated by the `$event->sender` field. It's the only hassle you should overcome when using proper event handlers instead of overriding `beforeValidate()` and `afterValidate()`.

Events of \yii\db\BaseActiveRecord

`BaseActiveRecord` is the base class for the `ActiveRecord` you'll be using most of the time. Active records, being the subclasses of `\yii\base\Model`, trigger its events too.

To not repeat the same warning again and again, we will give it once right now. All of the events listed here are triggered inside the special methods of the `BaseActiveRecord` class. In Yii 1.1.x, it was customary to override these methods in the model classes, but if you continue this practice in Yii 2, then you should always call the parent implementation first, or you will lose the `trigger()` call. Have a look at the following table:

Event name	When is it triggered
`EVENT_INIT`	It is triggered at the end of the `init()` method of the `BaseActiveRecord` class, which is called right after the constructor.
`EVENT_AFTER_FIND`	It is triggered after the `ActiveRecord` is found by some query to the database. The triggering of this event is wrapped into the `afterFind()` method of `BaseActiveRecord`.
`EVENT_BEFORE_INSERT`	It is triggered before the actual saving of the new active record to database. The triggering of this event is wrapped into the `beforeSave()` method of `BaseActiveRecord`. This event passes the instance of `\yii\base\ModelEvent` to its handler. If you set its `isValid` attribute to `false` in the handler, then saving will be silently canceled.
`EVENT_AFTER_INSERT`	It is triggered after the actual saving of the new active record to the database. The triggering of this event is wrapped into the `afterSave()` method of `BaseActiveRecord`.
`EVENT_BEFORE_UPDATE`	It is triggered before the actual updating of the active record to the database. The triggering of this event is wrapped into the `beforeSave()` method of `BaseActiveRecord`. This event passes the instance of `\yii\base\ModelEvent` to its handler. If you set its `isValid` attribute to `false` in the handler, then saving will be silently canceled.
`EVENT_AFTER_UPDATE`	It is triggered after the actual updating of the active record to the database. The triggering of this event is wrapped into the `afterSave()` method of `BaseActiveRecord`.
`EVENT_BEFORE_DELETE`	It is triggered before the deleting of the active record from the database. The triggering of this event is wrapped into the `beforeDelete()` method of `BaseActiveRecord`. This event passes the instance of `\yii\base\ModelEvent` to its handler. If you set its `isValid` attribute to `false` in the handler, then the deletion will be silently canceled.

Event name	When is it triggered
`EVENT_AFTER_DELETE`	It is triggered after the deleting of the active record from the database. The triggering of this event is wrapped into the `afterDelete()` method of `BaseActiveRecord`.

Two events are being triggered in a single `beforeSave()` method: `EVENT_BEFORE_INSERT` and `EVENT_BEFORE_UPDATE`. Similarly, two events are being triggered in a single `afterSave()` method: `EVENT_AFTER_INSERT` and `EVENT_AFTER_UPDATE`. Please consult the implementation of these methods in the source code of the Yii 2 framework to learn how they work. They are constructed to selectively trigger events depending on whether the saving inserts a new record or updates the existing one.

For example, let's look at the succession of events when you create and save a model based on the `ActiveRecord` class: `EVENT_INIT` → `EVENT_BEFORE_VALIDATE` → `EVENT_AFTER_VALIDATE` → `EVENT_BEFORE_INSERT` → `EVENT_AFTER_INSERT`.

Events of \yii\db\Connection

The instance of `\yii\db\Connection` is the component that you have attached to the application which is accessible by the `db` ID. When you do `Yii::$app->db`, you access the connection. This class triggers just a single event. Have a look at the following table:

Event name	When is it triggered
`EVENT_AFTER_OPEN`	It is triggered after all of the work to set up the connection to the database driver is done. This event does not pass anything special to its handlers.
`EVENT_BEGIN_TRANSACTION`	It is triggered right before the database transaction actually starts using the underlying PDO library. Note that it will be quite deep inside the call to `Yii::$app->db->beginTransaction()`.
`EVENT_COMMIT_TRANSACTION`	It is triggered right after the database transaction is actually committed using the underlying PDO library.
`EVENT_ROLLBACK_TRANSACTION`	It is triggered right after the database transaction is rolled back because of some error.

Events of \yii\web\Response

All events of the response take place inside the `\yii\web\Response::send()` method. They mark three steps of the response lifetime. Have a look at the following table:

Event name	When is it triggered
`EVENT_BEFORE_SEND`	It is triggered at the very beginning of the response sending.
`EVENT_AFTER_PREPARE`	It is triggered after the response is formatted by appropriate formatters in the `\yii\web\Response::prepare()` method.
`EVENT_AFTER_SEND`	It is triggered after the response is sent.

Events of \yii\web\User

The user component tells the world only about the login and logout events. All event handlers will receive an instance of `\yii\web\UserEvent`. Apart from other properties about which you can read in the documentation, it has the `isValid` attribute. It has the same semantics that we have seen already in the event instances for classes `Response`, `View`, `ActiveRecord`, and `Controller`. Have a look at the following table:

Event name	When is it triggered
`EVENT_BEFORE_LOGIN`	It is triggered at the start of the procedure to sign in, located inside the `login()` method. If you set the `isValid` attribute of `UserEvent` to `false` inside the event handler, `login()` will silently fail.
`EVENT_AFTER_LOGIN`	It is triggered after the user successfully logs in, at the end of the `login()` method.
`EVENT_BEFORE_LOGOUT`	It is triggered at the start of the procedure to sign out, located inside the `logout()` method. If you set the `isValid` attribute of `UserEvent` to `false` inside the event handler, `logout()` will silently fail.
`EVENT_AFTER_LOGOUT`	It is triggered after the user successfully logs out, at the end of the `logout()` method.

Note the quite surprising feature to prevent the user from *logging out* of the application by manipulating the event object. It's really hard to imagine a situation in which you would need to force users to stay authenticated.

Events of \yii\mail\BaseMailer

`BaseMailer` is an abstract class that is meant to be overridden by some concrete implementation of the actual e-mail sender. We saw the example in *Chapter 8, Overall Behavior*, when we set up the sending of log messages through SwiftMailer. It publishes only two events, and both of them pass an instance of `\yii\base\MailEvent` to the event handler. Have a look at the following table:

Event name	When is it triggered
`EVENT_BEFORE_SEND`	It is triggered at the beginning of the `send()` method. The triggering of this event is wrapped in the `beforeSend()` method, so you need to call `parent::beforeSend()` if you override it. The `MailEvent` object passed to the handler of this event has the `isValid` attribute. If you set it to `false`, the sending will be silently canceled.
`EVENT_AFTER_SEND`	It is triggered at the end of the `send()` method. The triggering of this event is wrapped in the `afterSend()` method, so you need to call `parent::afterSend()` if you override it. The `MailEvent` object passed to the handler of this event has the `isSuccessful` attribute, which tells whether the sending was successful.

Note that the value of `isSuccessful` is just a blind copy of the return value of the `sendMessage()` method, which concrete classes should implement.

Summary

Honestly, it's hard to justify using proper event-based programming in PHP, where the program starts, handles the request, and then dies. However, when coupled with the concept of behaviors, events provide places to extend the existing functionality, which can lead to very simple solutions like the one presented at the beginning of this chapter.

That's the power given to you by the events and behaviors in the Yii 2 framework.

In this chapter, we first scratched the surface of the events system using the behaviors prepackaged in the Yii framework. Then, we investigated in great detail just what exactly the event and behavior concepts in Yii 2 are and how they are implemented. Finally, we completely scrutinized the types of events that already occur inside the Yii 2 application and to which you can hook on without special preparation.

The next chapter will mean a lot of work for us. First, we'll enhance our customer model with all of the missing pieces we left over and did not implemented in *Chapter 2, Making a Custom Application with Yii 2*. After that, we'll use the most complex widget from the Yii framework: the Grid view.

11
The Grid

Back in *Chapter 2*, *Making a Custom Application with Yii 2*, we cut some corners and implemented only part of the functionality of our already rudimentary example CRM application. This chapter will conclude the example.

Our goal here will be to finish the CRUD for the customer model and to have in the end an efficient user interface to filter the recorded customers.

We'll learn to use arguably the most complex widget among the ones shipped together with the Yii framework, the GridView widget. But the GridView widget is there only to present records from the database; we will also make the form to actually enter the customer models into the database, utilizing the `ActiveForm` widget.

In this chapter, we'll discuss the following topics:

- The widgets concept in general
- The structure of the GridView widget, especially its concept of columns
- The built-in features of the GridView widget
- Making custom columns via the GridView widget in two ways
- Implementing sorting and filtering inside the GridView widget for custom columns

Dismissing of the domain layer

In *Chapter 2*, *Making a Custom Application with Yii 2*, we tried pretty hard to decouple ourselves from the framework to continue being able to test and extend our application later, when we get some real business rules apart from the mindless CRUD interface. Of course, this is a lot of additional work, as we're essentially placing an additional level of abstraction over the active records.

This is not idiomatic Yii code. Yii provides you with a really robust and useful API for a lot of areas but, at the same time, it tries really hard to couple you to both its data and presentation layer, which is trouble all the way down the road.

Most probably, you'll see such a code out in the wild a lot of times when maintaining Yii-based applications, and, hence, we decided to show you how the web application development is done the *Yii way*, as opposed to *using the Yii way* shown in *Chapter 2*, *Making a Custom Application with Yii 2*. We'll call the active records directly from the controllers and pass them directly to our views. Most of our code will be autogenerated and left untouched. We basically throw away the domain and application layers from our application. Let's see how it'll look and feel this way.

As we already have the active records in our data layer, such a change will cost us nothing, because we would be generating them anyway. We will also not write any automatic tests, relying only on looking visually at our pages being developed. This way, we'll get a working application too, and we will get it even faster than before.

Designing for the customers' index

It's assumed that you do not start from scratch but continue with the example we've been working on until now. However, if you actually do want to start from scratch, just use the migration that creates the `customers` table and autogenerate the customer model and the associated CRUD using Gii. You will also need the changes described in *Chapter 10*, *Events and Behaviors*, as we'll utilize the *audit* fields introduced there.

We will be working on the `/customers/index` route in our example application. We want this route to show us the table with a list of all customers registered in our database. Additionally, this table should provide filtering capabilities, allowing us to filter customers by their first and last names, phone number, country, city, and date of birth. We also want to sort the list by the same fields.

We need to implement the Address model for this to work, and we need to greatly enhance the Create Customer form to accommodate assigning addresses to customers. The same is applicable to the Email model. This chapter is devoted to GridView only, so you can read about changes in the Customer Edit and Create forms in *Appendix B*, *The Active Form Primer*.

We need to take care, as the customer has too many pieces of information now. If we show each field from all the records contained in the Customer model as a column in a table, this table will be too wide to show on screen. Our main problem with GridView will be to show several active record fields in the same table column.

The following is a sketch of the target GridView for the Customer model:

Name	Birth Date	Addresses	Emails	Phones	
John Doe	1973.12.10	some first address some second address etc	a@sbdy.dom b@sbdy.dom etc	666-66-66	View Edit Delete
...	...	...	...	+79133334343 +43432224410	View Edit Delete
...	...	...	...	...	View Edit Delete

Making address, e-mail, and phone active records

We need to prepare tables and `ActiveRecord` definitions for all of our submodels, Address, Email, and Phone.

Let's create a table for `AddressRecord` using the following migration script:

```
$this->createTable(
    'address',
    [
        'id' => 'pk',
        'purpose' => 'string',
        'country' => 'string',
        'state' => 'string',
        'city' => 'string',
        'street' => 'string',
        'building' => 'string',
        'apartment' => 'string',
        'receiver_name' => 'string',
        'postal_code' => 'string',
        'customer_id' => 'int not null'
    ]
);
$this->addForeignKey('customer_address', 'address',
    'customer_id', 'customer', 'id');
```

To keep things simple, we make all parts of the address just strings. We need to explicitly create a foreign key for reasons we will describe shortly.

> The `\yii\db\Migration.addForeignKey()` function is a pretty horrible function, given its five positional arguments, and probably your only option to remember how it works is to have an IDE with the ability to go to the function's definition. One way to remember how to specify it correctly is by utilizing the following sentence featuring all the arguments in the correct order and with sort-of memorable names: add foreign key with *name* on *table column* referencing *table column*.

After this migration is applied, we open Gii and generate the `AddressRecord` model using the following settings in the model generator:

- **Table Name**: `address`
- **Model Class**: `AddressRecord`
- **Namespace**: `app\models\customer`
- **Generate Relations**: Keep this checked

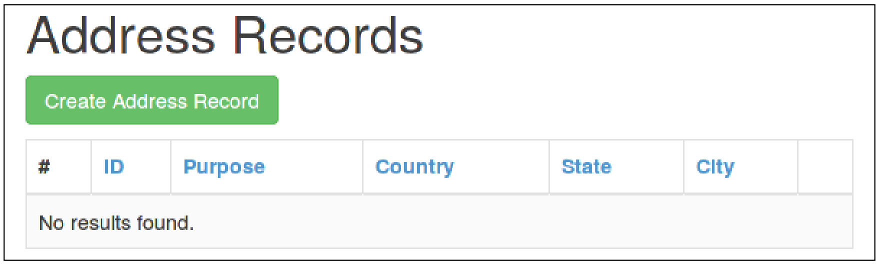
The `ActiveRecord` class in Yii has special means to facilitate relations between DB tables bound by foreign keys. The **Generate Relations** checkbox will enable their generation in `AddressRecord`.

Then, open the CRUD generator and create the CRUD for `AddressRecord` using the following settings:

- **Model Class**: `app\models\customer\AddressRecord`
- **Search Model Class**: Leave this field empty
- **Controller Class**: `app\controllers\AddressesController`

We leave the Search model, because its purpose in the autogenerated CRUD is to provide filtering inside the GridView widget, and we don't need it for the `AddressRecord` model. We don't need it, because it's not our intent to see all models of `AddressRecord`, without the context of a customer.

The /addresses route should show you the following:



Next, we will create the email table with the following migration:

```
$this->createTable(
    'email',
    [
        'id' => 'pk',
        'purpose' => 'string',
        'address' => 'string',
        'customer_id' => 'int not null'
    ]
);
$this->addForeignKey('customer_email', 'email',
    'customer_id', 'customer', 'id');
```

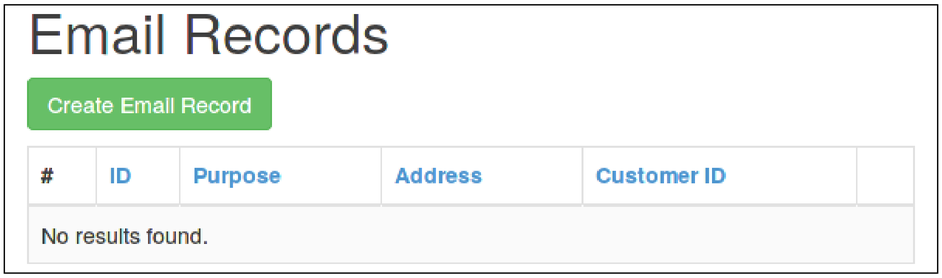
Create the EmailRecord class with the same model generator and with the following settings:

- **Table Name**: email
- **Model Class**: EmailRecord
- **Namespace**: app\models\customer
- **Generate Relations**: Keep this checked

Then generate the CRUD with the following settings:

- **Model Class**: app\models\customer\EmailRecord
- **Search Model Class**: Leave this field empty
- **Controller Class**: app\controllers\EmailsController

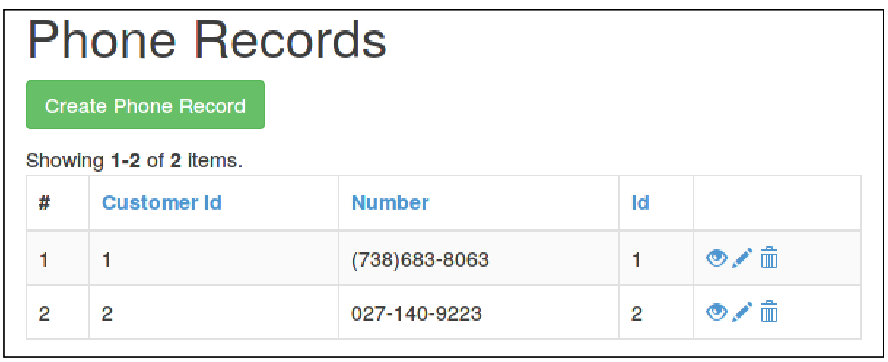
The /emails route should show you the following:



We already have the PhoneRecord model. Generate the CRUD with the following settings:

- **Model Class**: `app\models\customer\PhoneRecord`
- **Search Model Class**: Leave this field empty
- **Controller Class**: `app\controllers\PhonesController`

Our newly created CRUD at /phones should look like the following screenshot (given that we have two records in the database already):



In all three cases, we encounter the GridView widget; we'll toy with this shortly. But first we will carry out some cleaning.

Making the common base controller for submodels

We have three absolutely identical controllers now, except for several `ActiveRecord` class names there. It's distasteful to leave such a duplication in the code, so let's see how we can unify it.

First, pick one out of `AddressesController.php`, `EmailsController.php`, or `PhonesController.php`; copy it to the `@app/utilities` folder; and rename it to `SubmodelController.php`. Change the namespace declaration inside this file to `app\utilities` and the class name to `SubmodelController`. Now, we are ready to generalize the controllers for submodels.

We need the following instance variable to hold the piece of configuration that is changeable:

```
/** @var string Name of the class to be manipulated */
public $recordClass;
```

This is the reason we generalize the controller's definition. The controllers created by the CRUD generator are almost identical except for the name of the class of the active record being manipulated. We will not add any special behavior to these controllers, as we are perfectly happy with the default one, so we can just pass the name of the class as a configuration setting.

Then, go to the `actionIndex()` method and delete it completely. We will never look at the list of submodels separately from the root model.

Next, go to the `actionCreate()` method and the first line will be as follows:

```
$model = new AddressRecord;
```

Replace this line with the following lines:

```
/** @var ActiveRecord $model */
$model = new $this->recordClass;
```

The first line is the hint to both your IDE, if any, and the people who'll read your code, that you are creating an instance of at least `ActiveRecord` here.

At the second line, we see the application of a really high-order feature of the PHP, when we create a class given its name as a string stored inside a variable (string literals will not work, though).

Take a look at the following `if-else` condition:

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

Increase the readability of the previous `if-else` condition by replacing it with the following command with guard case:

```
if ($model->load(Yii::$app->request->post()) &&
    $model->save())
    return $this->redirect(['view', 'id' => $model->id]);

return $this->render('create', compact('model'));
```

Perform the same simplification in `actionUpdate()`:

```
if ($model->load(Yii::$app->request->post()) &&
    $model->save())
    return $this->redirect(['view', 'id' => $model->id]);

return $this->render('update', compact('model'));
```

Note the highlighted part. That's the only line that is different from the same block in `actionCreate()`. But to remove this duplication, we'll need to rethink our approach towards the *Create* and *Update* concepts, so we'll just let it be, especially when it's not so important in this particular chapter.

Next, go to the `findModel()` method. Take a look at the following incantation inside the condition of the `if` clause:

```
$model = CustomerRecord::findOne($id)
```

Replace this incantation with the following code (in the same place):

```
$model = call_user_func([$this->recordClass, 'findOne'], $id)
```

This is an obvious generalization, given that `call_user_func` is able to call static methods of classes.

Then, let's increase the readability of the whole `findModel()` method. You should have the following code in the beginning:

```
if (($model = call_user_func([$this->recordClass,
    'findOne'], $id)) !== null) {
    return $model;
} else {
    throw new NotFoundHttpException
        ('The requested page does not exist.');
}
```

And you should end with the following code:

```
$model = call_user_func([$this->recordClass, 'findOne'],
    $id);
if (!$model)
    throw new NotFoundHttpException
        ('The requested page does not exist.');

return $model;
```

> It's completely redundant to check whether the query results are exactly null. We usually don't care whether it's an empty array, empty string, zero, false, or null, as long as it's not an instance of the `ActiveRecord` that we need. Not to mention that `ActiveRecord::findOne()`, by definition, cannot return any other false-y value.

We now have a generic, base controller class doing the same as the controller generated automatically by Gii, except for the mercilessly removed `actionIndex()` method. Using it, we can reduce `AddressesController` to the following code:

```
class AddressesController extends SubmodelController
{
    public $recordClass = 'app\models\customer\AddressRecord';
}
```

The `EmailsController` is reduced to the following:

```
class EmailsController extends SubmodelController
{
    public $recordClass = 'app\models\customer\EmailRecord';
}
```

And `PhonesController` is reduced to the following:

```
class PhonesController extends SubmodelController
{
    public $recordClass = 'app\models\customer\PhoneRecord';
}
```

Note that we need to use the fully qualified names for the `ActiveRecord` classes in the world of namespaced PHP. Do not forget that you need the corresponding `use app\utilities\SubmodelController` clause if you use the previous code verbatim.

Making relations from customer to address, e-mail address, and phone

In the design of the Edit Customer form, we envisioned three tables that list the phone numbers, e-mail addresses, and postal addresses related to the customer in question. It would be useful if we could somehow get this information in an easy way. After all, we have just removed `actionIndex()` from all of the CRUD controllers for our submodels, and now we need a way to list them somehow.

For this task, Yii 2 has a nice feature on the active records: you can have virtual read-only properties that will return active records related by the foreign keys in the underlying tables.

The following method adds the relation to `Phone` models in `CustomerRecord`:

```
    public function getPhones()
    {
        return $this->hasMany(PhoneRecord::className(),
            ['customer_id' => 'id']);
    }
```

This is the *relation* concept in the Yii 2 framework. It has the following practical effects for us:

- Calling `$customer->phones` with `$customer` being the `CustomerRecord` instance will return an array of `PhoneRecord` instances that have `customer_id` that is the same as `$customer->id`
- Calling `$customer->getPhones()` will return a `\yii\db\ActiveQuery` instance, which we can feed to the constructor of a `DataProvider` to pass as a data source for widgets

This is made possible by the magic of the overridden `__get()` magic method.

We also need the relations to addresses and e-mails, which are done in exactly the same manner:

```
public function getAddresses()
{
    return $this->hasMany(AddressRecord::className(),
        ['customer_id' => 'id']);
}

public function getEmails()
{
    return $this->hasMany(EmailRecord::className(),
        ['customer_id' => 'id']);
}
```

Apart from `hasMany()`, `ActiveRecord` also has the `hasOne()` method, which accepts the same arguments, but the meaning of the second argument is changed: in `hasMany()` we *point* from the related table to *this* table, but in `hasOne()`, we point from *this* table to the related one.

Note that we don't technically need the `FOREIGN KEY` defined in the database tables for this relation to work, as we specify the relation manually at the application level. The foreign keys definition is there for two reasons:

- It's a documentation in code: anyone reading the database schema will be able to understand that there is a logical relation between tables.
- Gii can automatically generate the relation methods based on the foreign keys defined in tables. It generates relation methods from both sides of the relation; that is, for the `EmailRecord` class, it will generate the `getCustomer()` method, too.

> We can skip the manual construction of these relation methods altogether. The model generator in Gii can overwrite the already existing model classes, and you can even look at the differences it will introduce before that. So, given all these foreign keys in the tables related to the *customer* one, if we run the model generator to generate the `CustomerRecord` model again, Gii will regenerate the class with all the relation methods necessary. You will lose the behaviors attached in *Chapter 10*, *Events and Behaviors*, though, and you will need to add them manually again.

Before we actually start working with the GridView widget, we need to decide on how we will insert the database records to see that our user interface is actually working. We have two options:

- Insert a couple of handcrafted records manually into the database using the console interface of MySQL. This is pretty simple, given that we carefully ensure that the customer ID is correctly inserted to all appropriate foreign key fields. Instead of the console interface, we can use the CRUD we generated. Actually, removing the `actionIndex()` method is an unnecessary hindrance to us in this case.
- Introduce the user interface that allows us to generate the customer record along with all related records. This involves a lot of work with the `ActiveForm` widget, and so as not to bloat this chapter too much with not-so-related things, we decided to separate this section as *Appendix B, The Active Form Primer*. If manually constructed database records aren't your style, you are encouraged to turn to this appendix right now and implement the described user interface.

Please understand that all of the following discussion assumes you have at least one record in the `customer` table with several related records in the `address`, `phone`, and `email` tables. This is a direct consequence of working and relying only on manual testing.

FEATURE – widgets

We have seen two styles of using the Yii 2 widgets so far. The first was as follows:

```
$form = ActiveForm::begin();
// … lots of other code ...
ActiveForm::end();
```

When Gii generates the Create Record and Update Record user interfaces, it uses this format in the `_form.php` partial view files.

The second was as follows:

```
GridView::widget([
// … lots of configuration …
]);
```

When Gii generates the List Records user interface, it uses the previous format in the `index.php` view files.

Now we can talk about what a *widget* concept is in the Yii 2 ideology.

To simplify things greatly, a widget is something that allows us to render a view file inside another view file that is already being rendered. Unlike the `$this->render($viewFile)` incantation inside usual view files, widgets allow us to apply any amount of logic to rendering this subview file, which helps to simplify the top-level view files and opens the possibility of reusing the same pieces of content in several places on the web application. And this is surely useful.

> Of course, as view files in Yii 2 are just plain PHP scripts, you can theoretically have any amount of logic in there, and so you don't need an abstraction of widgets. But if you use your presentation layer in such a way, you surely deserve the consequences you will inevitably meet.

To be more specific, a widget in Yii 2 is a subclass of `\yii\base\Widget`, which has several auxiliary methods defined to help you utilize the concept.

Most importantly, it has the `render()` method defined in itself, so it's capable of rendering the output using the `View` application component.

The main logic of any widget is supposed to be hidden inside the `init()` and `run()` methods. Any subclass of `\yii\base\Widget` is expected to override these methods.

When you call `WidgetClassName::widget($configuration)`, it calls `init()` and `run()` in succession. This way all rendering is hidden inside the widget; thus, if you want the widget to render any output, you have to send it in `$configuration`.

When you call `WidgetClassName::begin($configuration)`, it calls its `init()` method and buffers all output until you call `WidgetClassName::end()`. At that point, it calls its `run()` method, collects whatever was buffered, appends the result of the `run()` method to it, and then returns the resulting string. This way you can make reusable widgets that will decorate whatever you output between the `begin()` and `run()` calls.

Note that both the `widget()` and `end()` calls return the rendering result as a string in the same way `Controller.render()` does. Nothing will be actually printed on the page if you don't explicitly echo it.

The base widget class has a special facility for you to be able to nest widget invocations, so you'll be able to use `begin()` to begin several widgets in succession and then use `end()` to end them one by one later; this will just work.

All of the configuration we do to the widgets by passing configuration arrays to the `widget()` and `begin()` calls is possible by the mighty `Yii::createObject()` method. All configuration settings in these arrays correspond to the public properties of the widget classes being prepared. This greatly helps in reading code by other developers even if you don't know the widget being configured: if you encounter some setting you don't understand, you can just go to the definition of the widget and search for the public property with the same name. In the case of Yii 2 built-in widgets, it's usually a lot of help, because its code base has outstanding documentation.

Creating the index page for customers

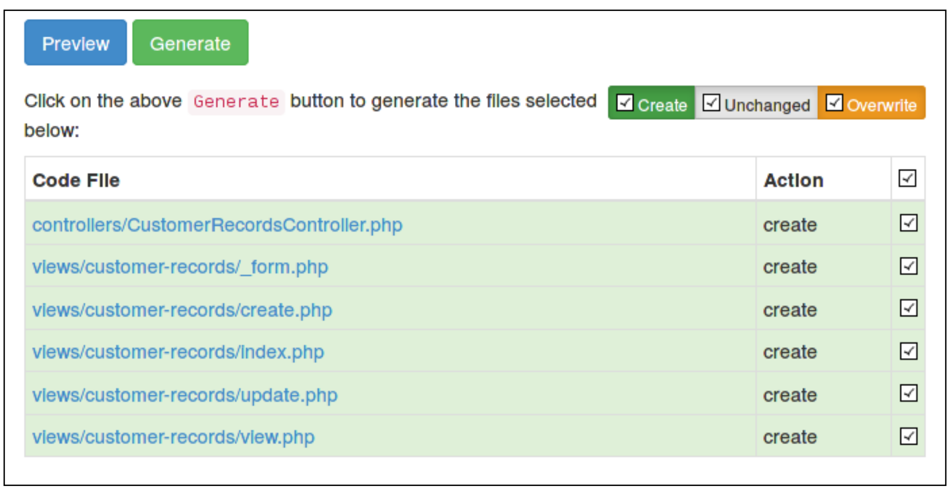
We are now ready to create the interface we spoke about at the start of this chapter.

We will leave our UI from *Chapter 2, Making a Custom Application with Yii 2*, and generate the CRUD for `CustomerRecord` directly this time.

Use the following configuration for the CRUD Generator:

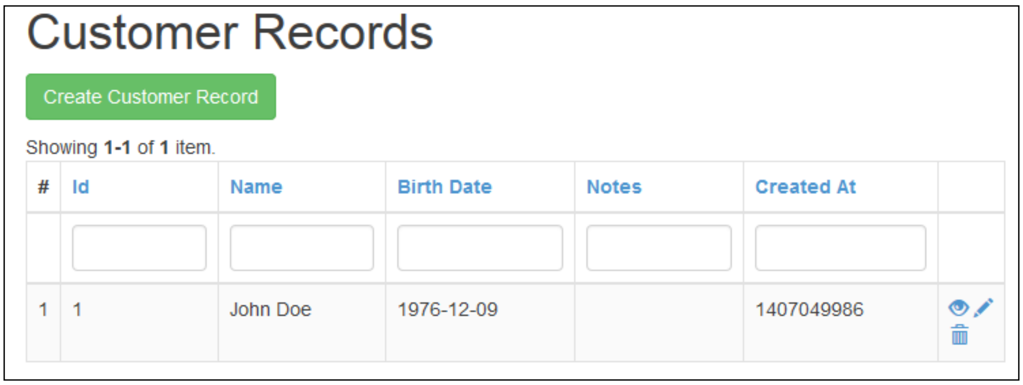
- **Model Class**: `app\models\customer\CustomerRecord`
- **Search Model Class**: `app\models\customer\CustomerRecordSearch`
- **Controller Class**: `app\controllers\CustomerRecordsController`

Note the list of files it will generate for us:



The controller named `CustomerRecordsController` has a folder inside `views` named `customer-records`. Just to remind ourselves: the string `customer-records` is an ID for this controller, which is recognizable by the Yii 2 engine. So actions inside the `CustomerRecordsController` folder will be reachable by the `/customer-records` route. We discussed this in previous chapters.

Click on the **Generate** button and then open the `/customer-records` route. Here is what it renders with a single customer record in the database:



This is quite a simple implementation, and that's what we will be working with through the rest of this chapter.

Making a base GridView for customers

Let's decide which columns we want in the `customers` table:

- `ID`
- `Name`
- `Birthday`
- `Phones`
- `Emails`
- `Addresses`

This list is naturally sorted in order of increased complexity. Let's look at the current autogenerated code for the customers' GridView inside the `views/customer-records/index.php`:

```
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],

        'id',
        'name',
        'birth_date',
        'notes:ntext',
        'created_at',
        // 'created_by',
        // 'updated_at',
        // 'updated_by',

        ['class' => 'yii\grid\ActionColumn'],
    ],
]); ?>
```

Let's note several points the Gii made to us.

First, we have `\yii\grid\SerialColumn` as the first column of a table. This is just an auxiliary column that gives us row numbers. In paginated grids, this can be quite useful, as the row number is displayed not relative to the page currently being shown, but relative to the whole dataset from `dataProvider`.

Secondly, we have all the fields from `CustomerRecord` listed in the column definitions, but all except the first five from them are commented out, so as not to impact the table being rendered.

Next, note the special syntax for the `notes` field: `notes:ntext`. We'll discuss this syntax a bit later but, in short, Yii 2 allows us to specify columns as strings in the `attribute:type:label` format. This is the shorthand syntax for the full configuration array with those three settings. It has made the `ActionColumn` for us already, with default settings.

And finally, we see the settings we did not see previously: the `filterModel` property, to which the instance of `\app\models\customer\CustomerRecordSearch` is being passed. We specified this class name when generating the CRUD in Gii, in the `Search Model Class` field. We'll talk about it later, because this setting enables filters in the table being rendered. You can see that this particular table has one element that was absent on the GridView views we made for phones, e-mails, and addresses in the previous section: the row with text inputs. This row is a working filter for the table, as it was shown in *Chapter 3, Automatically Generating the CRUD Code*. We'll discuss filtering later in this chapter. Now let's do something simpler: put the pieces of information we want into this little space we have.

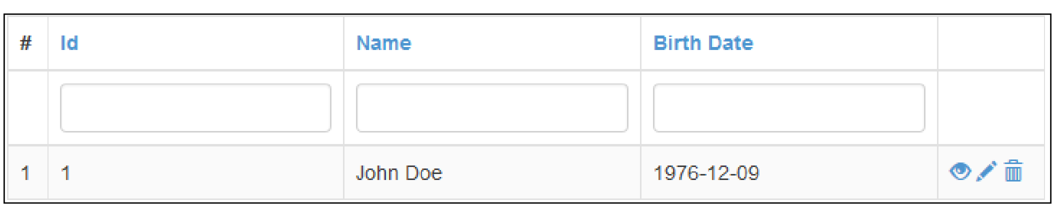
Changing the format of the column content

We can quite easily clean up our customer list by leaving only simple fields among those we want to see in the table: `id`, `name`, and `birth_date`. The following is the columns definition for this:

```
'columns' => [
    ['class' => 'yii\grid\SerialColumn'],
    'id',
    'name',
    'birth_date',
    ['class' => 'yii\grid\ActionColumn'],
],
```

> In all the following code snippets, we'll omit the lines for `SerialColumn` and `ActionColumn` for brevity. We will not touch them in this chapter at all. But it does not mean that we will remove those lines from the code, as the columns still should be there in the table.

The following screenshot is our starting point being rendered on the page. Note that the columns already have automatically generated headers with proper capitalization:



#	Id	Name	Birth Date	
1	1	John Doe	1976-12-09	

The `Id` column looks identical to the serial number column; let's move it to the right-hand side edge of the table. Also, the ISO format for dates looks really strict. We can make it a little more attractive using the following configuration:

```
'columns' => [
    'name',
    [
        'attribute' => 'birth_date',
        'format' => ['date', 'jS M, Y'],
    ],
    'id',
],
```

The `DataColumn` class has a property called `format`, using which we can declare how to represent the data inside the cell. In the case of the `date` format, we can specify the format string itself as an additional parameter. To do it, we provide an array value to the `format` setting. Many other formats are not parameterized, such as `ntext` mentioned previously for the `notes` field. In this case, we can provide either a one-element array or string.

> By default, the list of all of the formats available to you can be inferred from the methods of `\yii\base\Formatter`, whose names start with the prefix `as`. For example, the `ntext` format corresponds to the `asNtext($value)` method, and `date` corresponds to the `asDate($value, $format)` method.

Before looking at the table rendered, we can come up with a trick to increase the usefulness of the information presented in the ID column. Let's stuff the audit data there, using the Popover feature from Twitter Bootstrap (read about it at `http://getbootstrap.com/javascript/#popovers`). But beware, we will need a lot of code to make it work, so we will wrap this into the new column type, named `app\utilities\AuditColumn`.

FEATURE – formatter

The Yii application has a special component attached, named `\yii\base\Formatter`. This component encapsulates the feature of formatting the input data according to specified rules.

The main purpose of the formatter is to provide the following method:

```
public function format($value, $format)
```

> A lot of other components depend on this method of the formatter. So if you override this component, be sure that you still have the `format()` method defined, as it'll break a lot of Yii 2 otherwise.

As a `$format` argument for the `format()` method, you can pass either a string or an array. In the case of an array, its first element should be a string. In both cases, the string given will define the *format* to which `$value` should be converted. To perform the actual conversion, the formatter will try to find and use its method, named as the format given but with the prefix `as`. Thus, the following incantation will ultimately call `\yii\base\Formatter.asRaw($string)`:

```
Yii::$app->formatter->format($string, 'raw');
```

The following incantation will ultimately call `\yii\base\Formatter.asDatetime($datetime, 'Y-m-d H:i:s')`:

```
Yii::$app->formatter->format(
    $datetime,
    ['datetime', 'Y-m-d H:i:s']
);
```

As has already been shown, this feature is used extensively in places where you can specify a format for something to be rendered, such as the GridView column declarations.

At the time of this writing, the following formats were defined at the formatter:

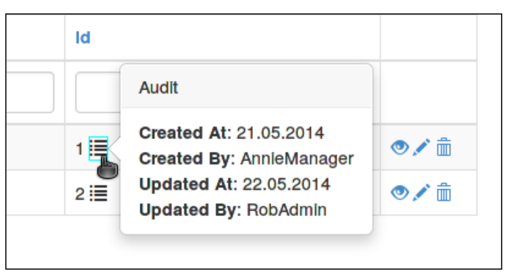
Name	**Meaning**
`raw`	This means that processing will be done to the value. Note that this is quite useful in `DataColumn`, as the default format in it is `text`, which turns HTML markup to visible text.
`text`	This means that the value will be processed by the `Html::encode()` call to reduce HTML tags to plain text.
`ntext`	This is the same as `text`, but the result will be additionally processed by the `nl2br()` method, a PHP built-in function, replacing line breaks with the `br` HTML elements.
`paragraphs`	This is the same as `text`, but the blocks of text separated by two or more consecutive empty lines will be wrapped in the `p` HTML elements.
`html`	This accepts the additional `$config` parameter. The value will be processed by the `HtmlPurifier::process($value, $config)` call. Thus, instead of making the HTML markup into visible text, we strip it completely.

Name	Meaning
`email`	Here, the given `$value` will be processed by the `Html::mailto(Html::encode($value), $value)` call; you will get the `mailto:` link with the same visible label as the e-mail itself.
`image`	Here, the given value will be used as an `src` attribute of the `img` element declaration, which will be returned. This is useful when you have a URL to the image and want to automatically create the tag from it on the resulting HTML page. Note, however, that you don't have any control over other properties of the resulting `img` element (at the moment of writing, at least).
`url`	Here, the given value will be used as a `href` attribute of the `a` element declaration, which will be returned. The omitted `http:// prefix` will be inserted if needed. This is useful to convert URLs to clickable links but note that you don't have any control over other properties of the resulting `a` element (the same as with `image` formats).
`boolean`	If given the `false`-y value (anything that evaluates to `false`), it will return the string `No`, localized to the current application locale. Otherwise, it will return the localized string `Yes`. You can change the `booleanFormat` property of `Formatter` to your own definition of `["No", "Yes"]`. It should be a two-element array.
`date`	This converts the given value to the date format given as a second argument. If no format is explicitly given, the `dateFormat` property of `Formatter` will be used. This method accepts `strtotime()`-parsable strings, integer Unix timestamps, and `DateTime` objects.
`time`	This is the same as `date`, but the `timeFormat` property of `Formatter` will be used by default.
`datetime`	This is the same as `date`, but the `datetimeFormat` property of `Formatter` will be used by default.
`integer`	This converts the given value to a string containing its integer representation. Note, however, that it works differently from a simple typecast to the `int` type. Check the definition of `\yii\base\Formatter::asInteger` to understand exactly how it performs the conversion.
`double`	This converts the given value to the string representation of a floating-point real number. You can specify the number of digits after the decimal separator; if you don't do that, the default of two digits will be used. The default decimal separator is the dot (`.`) symbol, but you can specify another symbol in the `decimalSeparator` property of `Formatter`.

Name	Meaning
`number`	This processes the given value with the `number_format()` PHP built-in. You can pass the number of digits after the decimal separator as an argument, with the default as `0`. The `decimalSeparator` property of `Formatter` will be used to set the symbol for the decimal separator, with the default being the `.` symbol. The `thousandSeparator` property will be used to set the symbol for thousands, with the default as the `,` symbol.
`size`	This treats the given value as the number of bytes and tries hard to describe this number in larger byte sizes, such as kilobytes or megabytes. The additional argument to this format is a flag declaring whether we should use short names of sizes (such as B, KB, MB, and TB) or full names (such as bytes, kilobytes, megabytes, and so on). This defaults to false, which means short names. Look at the definition of the `sizeFormat` property of `Formatter` to see other options to control this format.
`relativeTime`	Here, given value is treated as the date/time definition, and the additional argument is treated as the other date/time definition, with the default as of now. This returns the textual description of the time interval between those definitions. You can pass timestamps, strings parsable by the `strtotime()` PHP built-in, and `DateTime` objects as both value and additional argument. Additionally, you can pass the instance of `DateInterval` or string acceptable by the constructor of this type as the value to be processed. In this way the additional argument will be ignored. Look at the definition of `\yii\base\Formatter::asRelativeTime` to see which localized strings this formatter uses.

Making the custom GridView column for the customer audit info

We imagine very simple-looking cell content for the new column type. It will show the ID of the record, and then it will show the special icon. By clicking on that icon, one can see the popover with the information shown in the following screenshot:



We will override the `DataColumn` class. Let's create the `app\utilities\AuditColumn` class inside the `AuditColumn.php` file at `utilities`:

```
namespace app\utilities;

use yii\grid\DataColumn;

class AuditColumn extends DataColumn
{
    // future code here
}
```

In Yii 1.x and PHP 5.3, it was quite tedious to make a custom column class, because it usually involved overriding the `renderDataCellContent` method, carefully preserving the default supplementary code it included.

In Yii 2, the column class has a powerful property called `content` that accepts arbitrary objects of the callable type and passes the following arguments to it, in the order given:

1. The *data object* returned from the `DataProvider` for this row. In the case of `ActiveDataProvider`, it'll be the `ActiveRecord` instance; in the case of `ArrayDataProvider`, it'll be the associative array.
2. The *key* associated with this row by `DataProvider`. In the case of `ActiveDataProvider`, it'll usually be the values of the field declared as the primary key for the corresponding database table.
3. The *zero-based number* of this row among all returned by `DataProvider`.
4. Finally, the whole instance of the column itself.

To have visual identification, the manual setting of the `content` of the column looks like the following:

```
$column->content = function ($model, $key, $index, $column) {
    return "contents for the table cell as string";
}
```

This property provides an opportunity to make very clean and simple solutions for us.

> Of course, if you want to customize the contents of the header or filter cells in some `DataColumn`, you're out of luck and have to override the `\yii\grid\DataColumn::renderHeaderCellContent` and `\yii\grid\DataColumn::renderFilterCellContent` methods, correspondingly. They have a lot of convoluted logic inside, so it'll not be a joyful ride. It's quite rarely needed, though.

We will not show the full step-by-step refactoring process that leads us to the resulting code, as it would be too long; only the end result will be presented.

The `init()` method in `AuditColumn` would be the most suitable to initialize the `content` property. Let's write the following simple code to it:

```
public function init()
{
    $this->content = [$this, 'makeAuditCellContent'];
}
```

The `makeAuditCellContent` method will be the place where we'll construct our cell internals. This method should be at least of `protected` visibility, or else our column will not be able to call it (the one performing the call_`user_func` call will be the parent `DataColumn` class).

This method looks as follows:

```
protected function makeAuditCellContent($model)
{
    $id = $this->formatID($model);
    $audit = $this->makeAuditPopoverElement
        ($this->getAuditValues($model));

    return sprintf('%s %s', $id, $audit);
}
```

We don't need anything special in there. The cell will be composed of the prettified ID and the symbol to call the popover with audit info.

By *prettifying* the ID, we mean just making it a seven-digit number padded with zeros:

```
protected function formatID($model)
{
    return sprintf("%07d", $model->id);
}
```

From the documentation about popovers in Twitter Bootstrap (`http://getbootstrap.com/javascript/#popovers`) and after taking a quick look at the icons bundled in there (`http://getbootstrap.com/components/#glyphicons-glyphs`), we infer that we need the following HTML code for the icon to reveal the audit info:

```
<span class="audit-toggler glyphicon glyphicon-list"
      data-toggle="popover"
      data-html="true"
```

```
        data-title="Audit"
        data-content="...">
</span>
```

This can be converted to the following function, creating an HTML code for popover toggler:

```
protected function makeAuditPopoverElement($values)
{
    return Html::tag(
        'span',
        '',
        [
            'class' => 'audit-toggler glyphicon
                glyphicon-list',
            'data-toggle' => 'popover',
            'data-html' => 'true',
            'data-title' => 'Audit',
            'data-content' =>
                $this->makePopoverContent($values)
        ]
    );
}
```

Don't forget to insert the `use yii\helpers\Html` clause at the beginning of the file. The *contents* that the `makePopoverContent()` method makes would be as follows:

- `Created At` (in bold): This is the date of creation in the PHP's `Y-m-d` date format
- `Created By` (in bold): This is the username of the user who was blamed for the creation
- `Updated At` (in bold): This is the date of the last update in PHP's `Y-m-d` date format
- `Updated By` (in bold): This is the username of the user who was blamed for the last update

This is a lot of repetition, so let's become functional and write the following code creating the contents of the popover:

```
protected function makePopoverContent($values)
{
    $formatter = function ($pair) {
        return sprintf(
            "<div><strong>%s:</strong> %s</div>",
            $pair[0],
```

```
                $pair[1]
            );
        };

        $appender = function ($accumulator, $value) {
            return $accumulator . $value;
        };

        return array_reduce(array_map($formatter, $values),
          $appender, "");
    }
```

This is a lot more high-order than anything else written in this book, but the intent should be obvious: we make a single line of audit information in the previous format by the `$formatter` function and then append all the lines into a single string by `array_reduce`. The definition of `$formatter` suggests that `$values`, which is converted into the text contents of the popover, is an array of arrays. The inner arrays are two-element arrays, and we call them `$pair`. Each such *pair* consists of a label (that should be presented in bold) and a value (presented after the colon). Here is how we construct this value list:

```
    protected function getAuditValues($model)
    {
        return [
            [
                $model->getAttributeLabel('created_at'),
                date('d.m.Y', $model->created_at)
            ],
            [
                $model->getAttributeLabel('created_by'),
                UserRecord::findOne($model->created_by)->username
            ],
            [
                $model->getAttributeLabel('updated_at'),
                date('d.m.Y', $model->updated_at)
            ],
            [
                $model->getAttributeLabel('updated_by'),
                UserRecord::findOne($model->updated_by)->username
            ]
        ];
    }
```

This concludes the generation of the HTML contents for our new audit column.

In the documentation for popovers in Bootstrap, it is said that we have to enable this JavaScript plugin manually. We have the asset named `BootstrapAsset` as the dependency for our `\app\assets\ApplicationUiAssetBundle`, but it holds just the CSS file for the bootstrap. To also include the JavaScript part of Twitter Bootstrap, we have to depend on another asset bundle, named `yii\bootstrap\BootstrapPluginAsset`.

Let's just create a small asset bundle for our audit column. Create the `assets/AuditColumnAssetsBundle.php` file with the following class definition:

```
namespace app\assets;

use yii\web\AssetBundle;

class AuditColumnAssetsBundle extends AssetBundle
{
    public $sourcePath = '@app/assets/audit-column';
    public $css = [
        'styles.css'
    ];
    public $js = [
        'scripts.js'
    ];
    public $depends = [
        'yii\bootstrap\BootstrapPluginAsset',
    ];
}
```

Inside the `assets/audit-column/scripts.js` file, we write the following:

```
$(".audit-toggler").popover();
```

This will enable our popovers, as they have to be enabled manually. Inside the `assets/audit-column/styles.css` file, we write the following code:

```
.audit-toggler {
    cursor: pointer;
}
.audit-toggler:hover {
    outline: 1px solid cyan;
}
```

This will bring a bit of style to the button.

Let's also make `ApplicationUiAssetsBundle` depend on the new assets bundle as follows:

```
class ApplicationUiAssetBundle extends AssetBundle
{
// … other declarations...
    public $depends = [
        'yii\bootstrap\BootstrapAsset',
        'yii\web\YiiAsset',
        'app\assets\AuditColumnAssetsBundle',
    ];
}
```

After you have prepared the asset bundle, register it at the end of the `AuditColumn.init()` method as follows:

```
app\assets\AuditColumnAssetsBundle::register
  ($this->grid->view);
```

Do not forget to rerun the assets minifier if you have set it up in *Chapter 8, Overall Behavior*.

> In fact, we could just place two commands at the end of the `init()` method instead of registering the whole asset bundle:
>
> ```
> $this->grid->view->registerJs(
>     '$(".audit-toggler").popover();'
> );
> $this->grid->view->registerCss(
>     '.audit-toggler { cursor: pointer; }
>      .audit-toggler:hover { outline: 1px solid cyan; }'
> );
> ```
>
> Such lines can drive any maintainer crazy. You can just imagine more than five such custom widgets through the web application having 50 different view files and who knows how many controller actions. When they start conflicting with each other or, even worse, other code becomes dependent on their styles or scripts, the only option your colleagues will present to you at some point is to rewrite the UI from scratch.
>
> Do not just blindly initiate the `registerCss` and `registerJs` calls in random places like this. In any real-world project, you should always care about separation of concerns, and your assets should be encapsulated somewhere else.

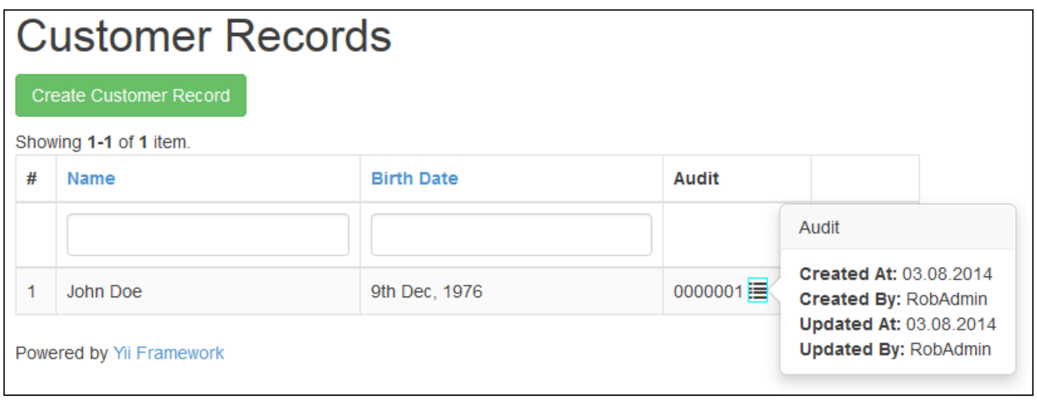
Let's also put the `Audit` label in the header cell by adding the following line at the beginning of the `init()` method:

```
$this->label = $this->label ?: 'Audit';
```

With the `AuditColumn` class defined, we can reference it among the columns inside `views/customer-records/index.php`:

```
<?= GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        'first_name',
        'last_name',
        [
            'attribute' => 'birth_date',
            'format' => ['date', 'jS M, Y'],
        ],
        ['class' => 'app\utilities\AuditColumn'],
        ['class' => 'yii\grid\ActionColumn'],
    ],
]); ?>
```

Finally, we open the `/customer-records` route, and we should see the following screenshot:



Note that this column does not have either filtering or sorting capabilities. We'll address this problem later.

Compressing submodels related to customers into single columns

The most problematic part of the customer information is the availability of multiple addresses, phones, and e-mail IDs. One way to simplify the representation for them is to write, for example, all phones in a single table cell. The same is the case with addresses and e-mail IDs.

We'll solve this issue in a more simple way than making the custom column class. The following is the full code to meditate on:

```
[
    'label' => 'Addresses',
    'format' => 'paragraphs',
    'value' => function ($model) {
        $result = '';
        foreach ($model->addresses as $address) {
            $result .= $address->fullAddress . "\n\n";
        }
        return $result;
    }
],
```

We are utilizing the `paragraphs` format described before. It will treat all blocks of text delimited by two newlines as `p` elements and, as a value, we'll produce the string with combined addresses separated by two newlines. The only thing we don't have yet is the `fullAddress` property on the `AddressRecord` model. The following is the definition of the `AddressRecord.getFullAddress()` method we need:

```
public function getFullAddress()
{
    return implode(', ',
        array_filter(
            $this->getAttributes(
                ['country', 'state', 'city', 'street',
                  'building', 'apartment'])));
}
```

The magic getter in `ActiveRecords` will call the `getFullAddress()` method when we try to read the nonexistent `fullAddress` property.

In the same way, we can create the **Emails** and **Phones** column, except that we don't need a virtual aggregating field and can directly use `address` and `number` fields, respectively.

FEATURE – GridView columns

So, GridView is composed of column entities. This concept is implemented by the `\yii\grid\Column` class. Each column conceptually consists of the following four elements:

- The header row, in which the label for the column is written. Yii 2 places the header properly inside the `thead` HTML element. The header is rendered by the `\yii\grid\Column::renderHeaderCell` method.
- The filter row, in which the input fields are presented for you to specify the filtering condition on the dataset. It's inside the second row of the same `thead` element. The filter is being rendered by the `\yii\grid\Column::renderFilterCell` method.
- The body rows, which hold the values to be rendered in the `tbody` element of the GridView in question. The body rows are being rendered by the `\yii\grid\Column::renderDataCell` method.
- The footer row, which is being rendered in the `tfoot` element. The footer row is rendered by the `\yii\grid\Column::renderFooterCell` method.

We have already shown that you can just use the `content` property of the `\yii\grid\Column` class to do some quite heavy processing on the data cells of the column. But if you have to do the same amount of custom rendering inside the other three parts of a column, you have to override those methods (and read their default implementation beforehand, of course).

When GridView is rendering itself, it goes through the list of columns in its `columns` property and initializes everything listed there. If the column is specified as an array, it is being send to `Yii::createObject()` directly. Note that the default class assumed is `\yii\grid\DataColumn`, not plain column.

If the column is specified as a string, this string is treated as follows, in the given order:

- `attribute:format:label`
- `attribute:format`
- `attribute`
- `InvalidConfigException`

The `InvalidConfigException` is, of course, not a value, but what you'll get in the event that there's no such attribute on the model being rendered.

The label is what will be put into the header cell. The format is the identifier of the format for the formatter component. The attribute is either the name of an attribute of `ActiveRecord` in the dataset returned by `DataProvider` or, in the case of associative arrays, the key inside the array.

`DataColumn` differs from a bare column by relying on the `ActiveRecord` mechanics. Unlike column, it has the `renderFilterCell` method fully defined, and it is able to automatically generate the header cell. It even places a link to perform the sorting request inside the header cell, if applicable.

The following is the list of all column types built into Yii 2 (there are not many of them):

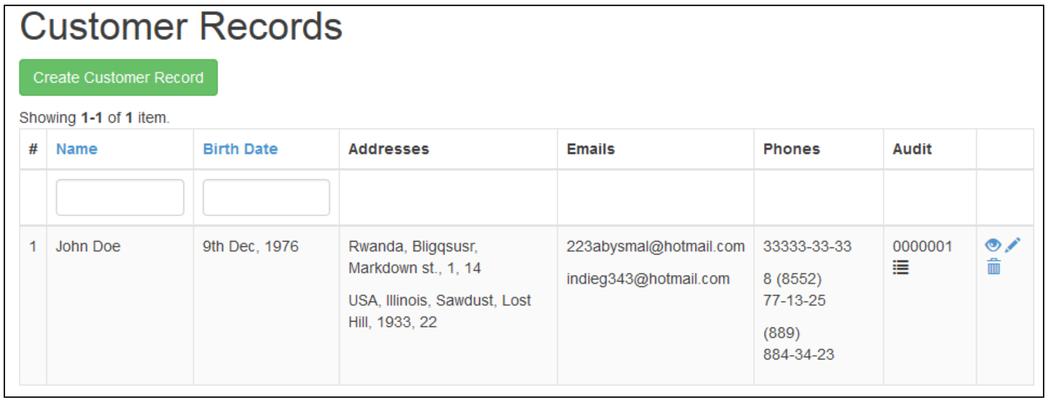
- The `\yii\grid\CheckboxColumn` column type renders the checkbox as a data cell. This is useful to enhance the JavaScript-backed rich UI, because you can get the row keys for the rows with checked checkboxes.
- The `\yii\grid\ActionColumn` column type is an enormously useful column type that allows you to put a column inside the table, which will hold arbitrary buttons. By default, it renders three buttons *view*, *update*, and *delete*, which you have seen already. You define the buttons available to render in this column using the `buttons` property of this class, and you define how they should be placed relative to the column contents using the `template` property.
- The `\yii\grid\DataColumn` column type has already been discussed in great detail, and we'll explore its filtering and sorting feature in the next sections. It allows you to output the attribute values of the objects from the dataset returned by `DataProvider` and passed to GridView.
- The `\yii\grid\SerialColumn` column type just provides you with a column with a counter displaying a row number among the rows in the whole dataset, which is useful in many domains. This counter is not related to the primary key of the `ActiveRecord` being rendered in any way, as we have already said.

And you can always just use the base column class with its universal `content` property.

You are encouraged to read through the excellent Yii 2 documentation about the grid columns and learn about how to use the action column yourself (it's the most useful column right after `DataColumn`), or just read the DocBlocks inside the source code.

Implementing filtering inside GridView of customers

By now, after implementing the **Emails** and **Phones** columns, you should have the following table:



You can see that some columns have the filter row (**Name** and **Birth Date**) and some do not. We'll make two changes to this table in this section: first, we'll add filtering by ID in the **Audit** column (which is easy), and second, we'll add filtering by country in the **Addresses** column (which is a lot harder).

If you try the filtering feature, you'll see that it works as follows:

1. Enter some text in the input field in the filter row.
2. Hit *Enter*, or make the input field lose focus any other way.
3. The whole page gets reloaded.
4. In the reloaded page, there are only rows corresponding to the filtering condition.

Let's see what URL we end up with after filling the previous table with the string `Kasey` in the filter for the **Name** column:

```
/customer-records
    ?CustomerRecordSearch[name]=Kasey
    &CustomerRecordSearch[birth_date]=
```

You probably thought right then: "Hey, but that's just a GET request, and all this filter thing should be a plain HTML form with some JavaScript in place." Note that we are passing not `$_GET["CustomerRecord"]`, but `$_GET["CustomerRecordSearch"]`; not the `ActiveRecord` we have generated in the model generator, but the search model generated by the CRUD generator. Let's look at the implementation of `CustomerRecordsController.actionIndex()` now:

```
public function actionIndex()
{
    $searchModel = new CustomerRecordSearch;
    $dataProvider = $searchModel->search
      (Yii::$app->request->getQueryParams());

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

Indeed, the `index` controller action is not just the list of all `CustomerRecords` to show; it has querying capabilities built-in.

The idea is simple. In the view file for the `index` action, we will render GridView, which requires a `DataProvider` class instance. Obviously, we'll create this `DataProvider` in the controller action, before rendering the view.

The special *search* model will perform the generation of the `DataProvider` class instance required. As we have seen before, you can create `DataProvider` based on `ActiveQuery`. So, as is stated on the second line of `actionIndex`, we just grab all of the GET parameters and pass them to the search model; it will construct the appropriate `ActiveQuery` to load into `DataProvider` and return the provider to us. No parameters means unrestricted `DataProvider`, returning all records from the database.

We pass the search model used to the view file (and subsequently, to `GridView` inside it), so the filter row will display the filtering conditions to us. Also, the very presence of the search model serves as a trigger to show the filter row at all. No search model passed, no filter row rendered. If the `filterModel` property of the `GridView` instance will not be set to some search model, we'll get no filter row.

Let's open the `CustomerRecordSearch.search()` method:

```
public function search($params)
{
    $query = CustomerRecord::find();

    $dataProvider = new ActiveDataProvider([
        'query' => $query,
```

```
        ]);
        if (!($this->load($params) && $this->validate())) {
            return $dataProvider;
        }

        $query->andFilterWhere([
            'id' => $this->id,
            // … same lines for birth_date, created_at,
              created_by, updated_at and updated_by ...
        ]);

        $query->andFilterWhere(['like', 'name', $this->name])
            ->andFilterWhere(['like', 'notes', $this->notes]);
        return $dataProvider;
    }
```

This is the canonical preparation to filter the GridView in question.

Note the highlighted line. This is an important guard case, as it states that, if no data for the `CustomerRecordSearch` model was passed as an input argument or if it was malformed, we return the `DataProvider` class instance with a blank `ActiveQuery`, filtering nothing.

The next lines are somewhat sneaky in the sense that we modify the object already passed as an input argument to another object. This is quite dangerous in general.

First we add the *where* conditions for comparing the non-textual fields, using the special `andFilterWhere()` method. Then we add the *where* conditions for non-strict comparing of textual fields, using the same method but with a different syntax for input arguments.

This is the real power of the `search()` method. We have all of the functionality of the `ActiveQuery` class, and the whole class of the corresponding search model to use. This allows us to make arbitrarily complex filters. More than that, we are not restricted to only GridView views here, as we can pass the resulting `DataProvider` instance to anything that accepts them.

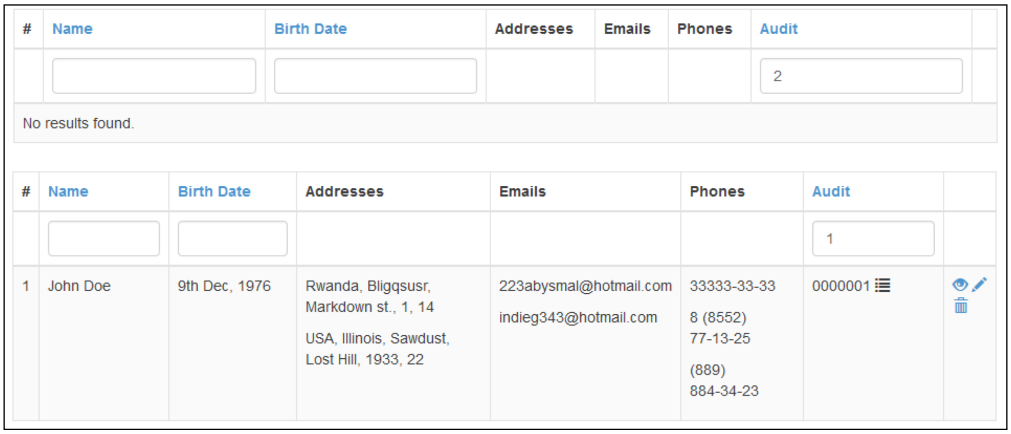
To enable filtering by the `id` attribute inside our new audit column, we need a laughably small change in the code base. All we need is to tell this column class which attribute it corresponds to, that is, fill its `attribute` property. We can pass the `id` value when declaring the audit column in the `columns` property of the `GridView` instance, as follows:

```
[
    'class' => 'app\utilities\AuditColumn',
    'attribute' => 'id'
],
```

We can also be more obstinate and initialize this property at the start of the `AuditColumn.init()` method as follows:

```
$this->attribute = $this->attribute ?: 'id';
```

We'll not be as obstinate as we could be and will allow overriding of the `attribute` property in the column's declaration after all. Now we are able to filter by ID:



Now let's implement a more interesting filtering condition: by country name in the **Addresses** column.

First, we'll need to define a virtual attribute of the `CustomerRecordSearch` model, that is, an attribute that does not exist among the fields of the customer table but will be treated as such by the filtering mechanics.

To define a virtual attribute, we need to do two things. First, we need to declare it as public on the `CustomerRecordSearch` model:

```
public $country;
```

Second, we need to mention it among any validation rules, except the `safe => false` one. The point is, you need to declare this attribute *safe* for use by the user interface. Let's use the *safe* rule that is already there in the model class:

```
public function rules()
{
    return [
        // … not interesting lines omitted...
        [[ /* … other fields ... */, 'birth_date', 'notes',
          'country'], 'safe'],
    ];
}
```

OK, as we need to filter by the value of the field in the related table, we obviously need to join some tables in the SQL query under `ActiveQuery`. Fortunately, we have exactly the right method to use, named `joinWith()`, that tells `ActiveQuery` to take into account the relation, defined in the base `ActiveRecord`. Read the documentation for the `joinWith()` method for intricate details. As we have defined the `getAddresses()`method for this inside the `CustomerRecord` class, our relation is named `addresses`, so, let's call `joinWith()` at the end of the `search()` routine (before the return clause, of course):

```
$query->joinWith('addresses');
```

And after that, as we have the `address` table joined by `LEFT JOIN`, we can just add the same search condition we have seen for other textual attributes as follows:

```
$query->andWhere(['like', 'address.country',
    $this->country]);
```

> If the `JOIN` clauses in SQL queries make you shudder because of a possible performance hit, you can add the simple guard case around this whole condition:
>
> ```
> if ($this->country)
> {
>         $query->joinWith('addresses');
>         $query->andWhere(['like', 'address.country',
>           $this->country]);
> }
> ```
>
> But, be aware that you should always check such changes in the profiler. For example, exactly this change (joining only when the country name is passed to the query) actually increased the number of queries made by Yii 2 ORM to the database when authors checked it in the *Database Queries* section of the *Debug Panel Extension*.

We also should correct the filtering condition on the `id` field in the same `CustomerRecordSearch.search()` method, because of the same `LEFT JOIN`. We need to be more specific so that the `ActiveQuery` backend will be able to understand precisely which `id` you want to compare. Have a look at the following code:

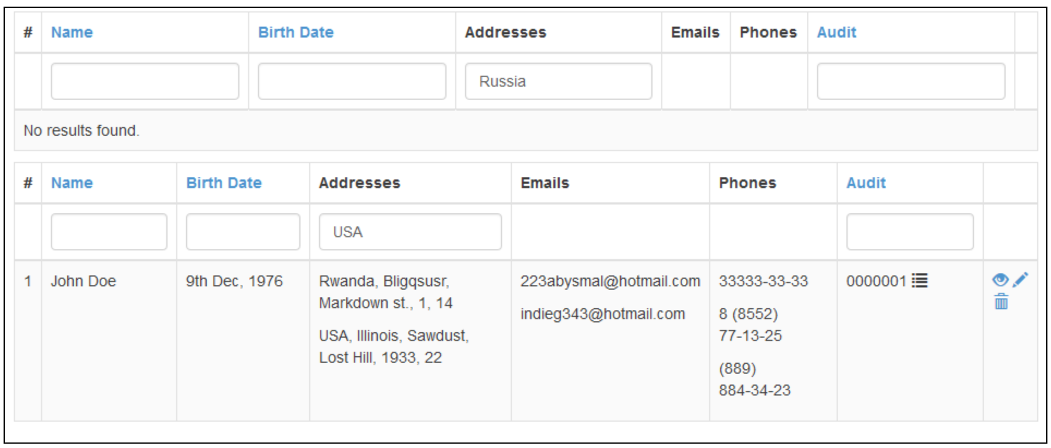
```
$query->andFilterWhere([
    'customer.id' => $this->id,
    'birth_date' => $this->birth_date,
    // … other declarations ...
]);
```

The highlighted part is to be inserted.

With the `CustomerRecordSearch` model improved like this, we can now declare that the **Addresses** column actually corresponds to the `country` attribute:

```
[
    'attribute' => 'country',
    'label' => 'Addresses',
    'format' => 'paragraphs',
    // … long irrelevant definition of the value
        skipped ...
],
```

And we get the filtering by country name in there:



You can note that `LEFT JOIN` with the one-to-many relationship actually produces partially duplicated rows in the resulting dataset. However, as you can see in the GridView after the described changes, ORM in Yii 2 automatically takes care of it, so you are able to see only the distinct top-level records in the table rows and are able to use all of the related records, as you wish.

Implementing sorting inside GridView of customers

It's time now to uncover the truth behind the color coding of the labels in the header of GridView. The blue headers are links. By clicking on them, you trigger sorting of the table: the page is being reloaded and the rows are reordered according to the column targeted. The first click on the header performs the sorting by this column in ascending order, the second click sorts in descending order, and all successive clicks toggle the order. Clicking on any other header sorts by the corresponding column again in ascending order.

Let's look again at the URL we end up with after the click on the **Name** column header:

```
/customer-records/index?sort=name
```

The second click on the same header moves us to the following route:

```
/customer-records/index?sort=-name
```

If a filter is active, we'll get something like the following code:

```
/customer-records/index
    ?CustomerRecordSearch[name]=
    &CustomerRecordSearch[birth_date]=
    &CustomerRecordSearch[country]=
    &CustomerRecordSearch[id]=1
    &sort=- name
```

It can be clearly seen that sorting and filtering are integral parts of the GridView UI, and they can work together. But we have already seen the complete implementation of the `actionIndex()` method, and there was nothing that accepted the *sort* parameter, right? In fact, GridView, with some support from `DataProvider`, cheats on us here.

The sorting activity is encapsulated in the `\yii\data\Sort` class in Yii 2. The configured instance of this class can be attached to the `DataProvider` instance to tell it how it should order the resulting dataset. If the `Sort` instance is not defined specifically, `ActiveDataProvider` will configure sorting by all attributes available in the `ActiveRecord` instance in question. All other data providers will just use whatever order the underlying database used.

While `Sort` objects have quite a lot of properties, we are mostly interested in `sortParam` and `attributes`. If you read the source code of the `Sort` class, you will see that the default value of `sortParam` is `sort`, exactly what GridView is passing to `actionIndex()` in the first URL example presented four paragraphs earlier. The `attributes` property holds the description of how the `DataProvider` class instance can potentially sort the dataset.

The idea is this: we configure the `Sort` object attached to the `DataProvider` class instance, describing to it the various options to sort the dataset. Afterwards, we can tell this `Sort` object that we need to sort in some way; if this way was configured previously, the `Sort` object will instruct its parent, the `DataProvider` class instance, about the ordering of the resulting dataset. That's quite an inverted flow of control, but it allows Yii 2 to hide the cheating inside the `Sort` object where it should belong and not leak it to the `DataProvider` component.

According to these explanations of the process, you should perform the sorting in the following way:

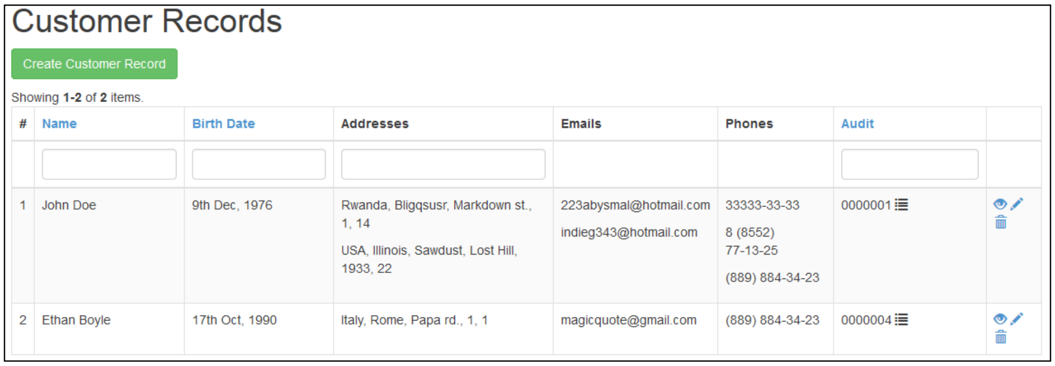
```
$query = $this->makeSomeQueryForActiveRecords();
$sort = new Sort;
$sort->sortParam = 'sort'; // only for clarity, it's default value
anyway
$sort->attributes = $this->makeDescriptionsOfKindsOfSorting();
$dataProvider = new ActiveDataProvider(compact('query', 'sort'));
// … some time later, maybe ...
$dataProvider->sort->params = ['sort' => 'one-of-kinds-defined-in-
  attributes-property'];
$models = $dataProvider->getModels();
```

After that, `$models` are the iterable collections of the `ActiveRecord` objects sorted by the attribute passed as `$dataProvider->sort->params['sort']`.

The cheating here is this: as GridView cannot go so deep inside the `DataProvider` class instance, and to relieve developers of the burden of fiddling with this low-level detail in the controller actions, just as with filtering, `Sort`, when it's time to decide on the ordering, just goes to the Request Application component and asks for the GET parameters passed by the client. This is an amazing violation of the borders between abstractions, of course, but it's what allows us to use the convenient UI to sort in GridView. At the time of this writing, it can be seen how exactly it's done inside the `\yii\data\Sort::getAttributeOrders` method, which is ultimately called by `\yii\data\ActiveDataProvider::prepareModels`, required for the call to `\yii\data\BaseDataProvider::getModels`.

So, let's solve two exercise problems, again one easy and one not so easy. As an easy task, let's implement sorting by our **Audit** column; as a hard task, let's implement sorting by the **Addresses** column and the **Emails** column.

Let's see what we have right now:



Wait, what? The **Audit** column header is a blue link, and it sorts the table already!

That's right. When we wanted to enable the filtering by the `id` attribute, we declared the `attribute` setting for the **Audit** column. This was enough to also enable sorting by the same attribute.

As we said earlier, if there's no specific configuration for the *sort* setting of `ActiveDataProvider`, it creates an empty `Sort` instance and then actually goes to the model, fetches all its declared attributes, corresponding to the database table columns, and installs them one by one to the `attributes` setting of the `Sort` object.

> This has a nice side-effect: you can sort GridView by fields not visible as a table column, but existing in the underlying database table. For example, open the following file:
>
> ```
> /customer-records/index?sort=-created_at
> ```
>
> This will get the table reordered. The rows should be in decreasing order of the IDs in the **Audit** column, as the `customer.id` field is automatically incremented and later records obviously have a later creation timestamp.

That was too easy, really. However, with our setup, we can implement sorting by country name quite easily, too.

From all the explanations so far, it's pretty clear that we need to configure the data provider to enable custom sorting. So, the best place to place our changes will be the `CustomerRecordSearch.search()` method again.

> It's quite important to understand at this point that the actual sorting done by the data provider will be performed by carefully constructing the SQL query on the side of the database. The data provider itself will not reorder records in the fetched dataset by means of PHP!

After a quick peek at the documentation for the `\yii\data\Sort::$attributes` property to learn the precise configuration syntax, let's append the definition of sorting by country name to the `search()` method:

```
$dataProvider->sort->attributes['country'] = [
    'asc' => ['address.country' => SORT_ASC],
    'desc' => ['address.country' => SORT_DESC]
];
```

We are appending to the already-filled `attributes` property; otherwise, we will need to rewrite definitions of sorting by other attributes again. It's absolutely crucial that you put this configuration command before the guard case shown in the previous section:

```
if (!($this->load($params) && $this->validate())) {
    return $dataProvider;
}
```

Otherwise, you'll end in a silly situation when you will have sorting enabled only when the table is filtered in some way. Also, the `$query->joinWith('addresses')` invocation should be moved up there, too, so it doesn't just work when there's some filtering condition.

This line is all we need to enable sorting by country. If you reload the page with GridView on it, you should note that the **Addresses** label turned into a blue link and really reorders the table after a click. The resulting order can look quite strange and non-obvious, though, because the database sorts the table made by `LEFT JOIN`, and Yii 2 then collapses this table behind the scenes.

We can add sorting by the `emails` attribute by doing exactly the same, except we need to tell `ActiveQuery` to join another table to the party:

```
$query->joinWith('emails');
$dataProvider->sort->attributes['email'] = [
    'asc' => ['email.address' => SORT_ASC],
    'desc' => ['email.address' => SORT_DESC],
];
```

A similar code for countries worked without other additions, because we already declared the **Addresses** column to correspond to the virtual `country` attribute while implementing filtering. The **Emails** column doesn't have the `attribute` declaration yet, and this is the only thing left to do:

```
[
    'attribute' => 'email',
    'label' => 'Emails',
    // … collapsing declared here ...
],
```

After inserting the highlighted line of code into the view file and reloading the page, you'll get sorting by the **Emails** column too. Note that here we do not need to declare the virtual `email` attribute on the `CustomerRecordSearch` class as we have to do when implementing filtering. This is because just the side-effect of declaring this setting is enough to turn the header label into a link for sorting, which is all we need here. If we need filtering, making a virtual attribute will become mandatory.

> In this chapter, we have not written a single line of testing code. In part, it was because there was no business logic at all. Partially, it was because the only way we could test the work we have done is through end-to-end acceptance tests. The GridView widget presents a feature set so large that the end-to-end test suite needed to cover all of it would be absolutely tremendous. So, all the code we have produced in this chapter is a legacy you put into your application, destined to be tested manually.
>
> This is the lesson we wanted to teach you in this chapter while dropping the ATDD approach: you can construct an amazing UI really quickly by blindly using the Yii 2 capabilities everywhere. But you should understand that you willingly trade the confidence that all of your code works as expected for the speed of developing a UI. If you are still unsure or unclear about the meaning of this tradeoff, then imagine what you will need to do several years later in the application lifetime, when Yii 2 becomes Yii 3 or Twitter Bootstrap should be suddenly replaced by something like Foundation (see `http://foundation.zurb.com/`).

Summary

Thus, our adventure into the world of setting up tabular interfaces has come to an end. At this point, you probably want to turn to *Appendix B*, *The Active Form Primer*, which will continue from there and show the development of the UI to update our `CustomerRecord` model along with all its submodels.

The GridView widget is an enormously complex component of the Yii 2 framework. It's not a widget anymore in the usual sense of *widget*; it's a complete user interface for managing datasets in a tabular format. A lot of moving parts are involved in its functioning.

We covered a lot of ground in this chapter, that is, we performed the following:

- Aggressively autogenerated the CRUD interface for a set of database tables related to each other by foreign keys
- Shamelessly used the bootstrap styles for our UI
- Quickly snapped together the whole custom `GridView` column to be able to click on the icon and be proud of the tooltip that appeared
- Made major customizations blazingly fast to the appearance of the data in GridView by writing custom callbacks to render cell content
- Were amazed at how easy it is to implement simple and not-so-simple kinds of filtering and sorting inside GridView

And a couple of not-so-great achievements were also performed.

At the same time, we glossed over a lot of mechanics we used in this chapter. Among them are:

- Other types of GridView filters, such as dropdowns. We are able to put custom HTML there!
- JavaScript bindings to the events happening in the table. The concept of row keys and how we can use them in our rich UI.

We have to have some kind of scope, and this book is not an all-encompassing reference anyway.

The next chapter, the last-but-one chapter of this book, will be dedicated to the routing system in the Yii 2 framework. Finally, here will be two real-world features, that we'll implement in our example CRM application.

12

Route Management

In this chapter, we will explore how the Yii 2 routing system works, that is, how the framework responds to different URL routes requested from it.

We'll start with the description of the process that the Yii application performs to determine the controller action to execute in response to client requests. Then, we will implement a small feature, which will demonstrate how we can control our routes using just the application configuration.

And lastly, we'll implement one particularly interesting feature that will require our own custom rule class and show the considerations while doing so.

Yii 2 routing 102

We have seen Yii 2 routing 101 in *Chapter 2*, *Making a Custom Application with Yii 2*, and this section is an extension of it.

As we already know, everything in Yii 2 starts with the entry-point script, which in our example application is `web/index.php`. This script should be the only PHP script in the directory published by the web server.

As everything comes to this script, all routes we used in the 10 chapters (excluding *Chapter 1*, *Getting Started*, discussing the installation of the framework) look as follows:

```
protocol://domainname/path/to/
  index.php?r=module/controller/action&param=value&etc
```

So, any request is being processed by accessing the `index.php` file, and passing it the further route inside the application as the GET parameter named `r`. This name is configurable in the `\yii\web\UrlManager::$routeParam` property, so the following configuration snippet will set the name of the route parameter to `icecream`:

```
[
    'components' => [
        'urlManager' => [
            'routeParam' => 'icecream'
        ]
    ]
]
```

Such fiddling is irrelevant in most cases, though, because we have two other properties, `UrlManager`: `enablePrettyUrl` and `showScriptName`, the usage of which we discussed in the *Routing 101* section in *Chapter 2*, *Making a Custom Application with Yii 2*:

- The `enablePrettyUrl` property effectively removes the `?r=` part from the URLs acceptable by the application. What was accessible by `index.php?r=module/controller/action` will be accessible by just `index.php/module/controller/action`. But in addition to that, it enables the support to define custom rules of parsing and generating routes. Basically, it completely changes the way the Yii application handles requests.
- The `showScriptName` property, when set to false, will prohibit the URL manager from adding the script name to the URLs it generates. The actual name of the entry-point script is irrelevant; it's inferred from the current `$_SERVER` settings automatically.

Practically speaking, you will almost always use the following combinations of the parameters:

```
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
        ]
```

The only downside of this approach is that you have to configure your web server to send all requests to your entry point, like the following Apache rewrite directive:

```
RewriteRule . index.php
```

Note that the `enablePrettyUrl` modes are incompatible. If you are using pretty URLs, you cannot request your application using the `?r=:route` notation and vice versa.

Getting the actual route from the request string is just the first step in the Yii 2 routing system.

As discussed in the *The informal concept of reachability* section in *Chapter 7*, *Modules*, the route concept boils down to the following string:

```
/moduleid/moduleid/.../moduleid/controllerid/actionid
```

This string allows the framework to guess which controller action it should execute. If there is no `moduleid`, then the application itself is considered as a target module.

When we have `enablePrettyUrl` set to true, though, we are able to define special URL rules to parse client requests to the application. In fact, we are able to use arbitrary strings as routes in our system.

> There are three ways to control routes, which are as follows:
>
> - Names of modules, controllers, and actions
> - Custom rule definitions inside the `components.urlManager.rules` setting of the Yii application configuration
> - Custom rule classes referenced inside the same setting

FEATURE – routing using names of modules, controllers, and actions

Unless you make really convoluted URL rules using the other two options (custom URL rules and custom URL rule classes, see the end of the preceding section), you can rely on the basic path format *Module ID – Controller ID – Action ID* with arguments to the action being inside the request parameters. This already gives you a lot of power and is suitable in most cases. By carefully and thoughtfully choosing the names of modules, controllers, and actions, you probably will never need to define any other routing rules.

Inside the module, you have the `controllerMap` property, which allows you to assign controller IDs to specific controller classes manually.

Otherwise, `controllerNamespace` works as a fallback mechanism. The controller ID will be used to infer the expected controller class name, and the namespace in `controllerNamespace` will be used to find the physical location of the file holding its definition.

> The controller ID is converted to the controller class name using exactly the following conversion:
>
> ```
> $className = str_replace(' ', '', ucwords(str_replace
>     ('-', ' ', $className))) . 'Controller';
> ```
>
> That is, the *dash-separated* format of the ID becomes replaced by the uppercase format of the class name, and additionally Yii 2 expects that the controller class name will end with the string `Controller`.

Inside the controller, you have similar mechanics. Using the `actions` property (never before discussed in this book), you can assign action IDs to specific controller action classes, which are descendants of `\yii\base\Action`.

If there are no actions declared this way, the controller will use its public methods, the names of which start with the string `action`, as actions. This is called inline action. In fact, when you request the `user/view` route and `UserController` does not have anything assigned to the `view` ID in its `actions` property, it actually creates the instance `\yii\base\InlineAction`, instructs it to refer to the `actionView` method, and then uses this inline action as any other `\yii\base\Action` action (see the documentation and definition of `\yii\base\Controller::runAction`, which is available at `http://www.yiiframework.com/doc-2.0/yii-base-controller.html#runAction%28%29-detail`, for the exact reference).

Maybe it's quite ironic, but these two fallback mechanisms of the module and controller are what you usually use when developing web applications with Yii 2.

Fundamental rules of URL management in Yii 2

It is quite important to understand some fundamental concepts behind the Yii 2 routing system; it will make everything easier to understand:

1. Any request you pass to the Yii application eventually will be resolved as the name of the module, name of the controller, name of the action, and parameters to this action.

2. The universal format, which allows Yii 2 routing mechanics to understand which controller action to call, is as follows:

```
[
    "/moduleid/.../moduleid/controllerid/actionid",
    [
        "param1" => "value1",
        …,
        "paramN" => "valueN"
    ]
]
```

3. The URL manager component parses the incoming request to the format described earlier using the URL rule entities.
4. The URL manager component converts the route from this format to the string to be placed as the URL into the HTML page being rendered, using the same URL rule entities.
5. The URL rules define the parsing of URLs and the creating of URLs as completely separate activities. In fact, by using custom URL rules, you'll be able to create URLs from one route definition, which will be parsed as a completely different route definition.

FEATURE – creating URLs in Yii

Before we delve into the details of how the URLs are parsed by Yii 2 and converted into calls to controller actions, let's mention the enormously important function that actually creates the URLs to be displayed to the client. This function is the `\yii\web\UrlManager.createUrl($params)` function. You can access this method by the following incantation from anywhere in your application:

```
Yii::$app->urlManager->createUrl($params);
```

This method receives one argument, which has the parameters to create a text representation of the URL in the universal format presented in the previous section. This method is so widely and often used that there is a helper method to simplify its usage, the `\yii\helpers\BaseUrl::toRoute()` method, which you have to call as follows:

```
URL::toRoute($params);
```

Custom routes using a configuration

Look at the following path we go to when opening the View Customer Record page:

```
/customer-records/view?id=1
```

This is quite verbose. Why not just use the following:

```
/customer/1
```

We can implement it quite easily using the following declaration inside the `components.urlManager.rules` setting of the Yii application configuration:

```
'components' => [
    'urlManager' => [
        'rules' => [
            'customer/<id:\d+>' => 'customer-records/view',
        ]
    ]
]
```

This declaration is being read and understood as follows:

1. If the request begins with `customer/`.
2. After this there are only digits.
3. Store these digits as an argument named `id`.
4. Process the request as the `customer-records/view` route.
5. Pass the stored `id` argument to the resulting controller action.

FEATURE – URL rules

The `components.urlManager.rules` setting of the application, which corresponds to the `\yii\web\UrlManager::$rules` property, can be filled with objects in two notations. The full notation is the array notation to be consumed by the `Yii::createObject()` method. You define the instances of the `\yii\web\UrlRule` class here. The following table has the most crucial properties of the URL rules:

Property	Meaning
`pattern`	This means what the incoming request should look like for this rule to take effect.
`verb`	This means which HTTP method should be used in the request for this rule to take effect.
`route`	This is the route to the controller action in the `moduleid/controllerid/actionid` format, to which the `pattern` should resolve to.

Property	Meaning
`suffix`	This means what string to append when creating URLs and what string must be present at the end of the pattern when parsing the request for this rule to take effect.
`mode`	If set to `0`, the rule will be used both to create URLs and parse them. Otherwise, you can set it to `\yii\web\UrlRule::PARSING_ONLY` or `\yii\web\UrlRule::CREATION_ONLY`.

There are some other properties, which have more specific usages, such as `host` and `defaults`. You are encouraged to read the documentation (`http://www.yiiframework.com/doc-2.0/yii-web-urlrule.html`) and/or source code of the `\yii\web\UrlRule` class to get the exact details.

The most important and useful property here is the `pattern` property. It defines what should be presented to Yii as a request string (without the request parameters), with the addition of the named parameters. The named parameter is the construct of the form given as follows:

```
<Name:RegularExpression>
```

Angle brackets and colons are the mandatory syntax. Named parameters can be referenced in the `route` property later, increasing the genericity of the rule. If they are not referenced in the route, they become stored as a `$_GET` parameter with the specified name (overriding existing values, if any).

The shorthand notation for defining the URL rules is as follows:

```
"[verb ]pattern" => "route"
```

This will initialize the URL rule object with just the `pattern`, `verb`, and `route` properties specified, leaving everything else blank, with `verb` being optional. The Yii 2 documentation for `\yii\web\UrlManager::$rules` presents a great self-describing example of a URL manager configured for REST-compliant requests:

```
'rules' => [
    'dashboard' => 'site/index',

    'POST <controller:\w+>s' => '<controller>/create',

    '<controller:\w+>s' => '<controller>/index',

    'PUT <controller:\w+>/<id:\d+>' => '<controller>/update',

    'DELETE <controller:\w+>/<id:\d+>' => '<controller>/delete',

    '<controller:\w+>/<id:\d+>' => '<controller>/view',
];
```

It actually reads quite easily given that you always remember that the line noise between the colon and the closing angular bracket is just a regular expression. As you can see, the named `id` parameters from the pattern are not mentioned in the route part of the rule, which means that they will be passed to the corresponding controller action as arguments with the same names. The parameter named `controller` makes this ruleset generic, applicable to all controllers in the system.

The rules declared in the rules setting are checked in the order of appearance. The first rule, which corresponds to the current request, is applied, and everything else is ignored after that. If no rule was fired, then the URL manager falls back on the parsing of the base `/moduleid/controllerid/actionid` format.

Apart from manipulating the properties of the base `\yii\web\UrlRule` class, we can reference any other class as the URL rule inside the rules setting, given that this class implements `\yii\web\UrlRuleInterface`. This interface declares two fundamental activities of the URL rule:

Method	**Meaning**
`parseRequest( $manager, $request )`	The `$manager` argument is an instance of the `UrlManager` class. The `$request` argument is an instance of a request class, which you can use to get the request string and parameters to parse. This method must return either a route in the canonical format shown in the *Fundamental rules of URL management in Yii 2* section or exactly `false`, which means that this rule cannot be applied to the current request.
`createUrl( $manager, $route, $params )`	The `$manager` argument is an instance of the `UrlManager` class. The `$route` argument is a string denoting the route in the base `/moduleid/controllerid/actionid` format. This can have variations, such as omitted parts. Be careful about it. The `$params` argument is an array of key-value pairs specifying the query parameters for this route. This method is expected to return a string, which should be a relative URL, ideally parsable back to route/parameters by the `parseRequest` call of the same `UrlRule` class. If the route/parameters do not correspond to this rule, this method should return exactly the `false` value.

Custom routes using custom URL rule classes

As a harder task, let's implement a more convoluted feature. Given our migration script creating default users in the database, the following URL brings us to the View page for the user who has the username `AnnieManager`:

```
/users/view?id=2
```

A real-world feature request from the higher-ups can be the following: make the same page accessible by the following URL:

```
/AnnieManager
```

This rule cannot be represented as the pattern-route pair, because we should match the given username stored inside the database, with the ID expected by the `actionView()` controller action.

> To understand custom URL rules, you should understand this concept very well: in the end, a particular action of the controller will be called, and we should pass to it the arguments it expects. What we are talking about in this chapter is the abstraction, which is one level above that, when we hide this base routing under a different vocabulary, with the goals to be more SEO-friendly, more regular like REST, or just overall more pleasant to see. But this level is ultimately resolved to the base route/parameters pair.

We will solve this task by introducing the custom URL rule class. Declare it in the ruleset as follows:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'showScriptName' => false,
    'rules' => [
        'customer/<id:\d+>' => 'customer-records/view',
            // our rule from previous task
        [
            'class' => 'app\utilities\UsernameUrlRule'
        ]
    ]
],
```

This time, we started with the declaration instead of wiring up the already existing class. Let's create the class, then, inside the `@app/utilities/UsernameUrlRule.php` file as declared:

```
namespace app\utilities;

use yii\web\UrlRuleInterface;

class UsernameUrlRule implements UrlRuleInterface
{
    public function parseRequest($manager, $request)
    {
        // parse request here ...
    }
    public function createUrl($manager, $route, $params)
    {
        // make the URL from route/params pair here...
    }
}
```

First, parse by the `parseRequest()` method. We will check whether the rule applies to us as follows:

```
$maybeUsername = $request->pathInfo;

$user = UserRecord::findOne(['username' => $maybeUsername]);
if (!$user)
    return false;
```

There is a very simple logic here. If the string between hostname and the `?` symbol denoting the start of request parameters is not a username of `UserRecord` in our database, we tell `UrlManager` that we can't do anything here.

> It is a serious security breach here, which you must solve in some way if you really end up implementing such a feature on your web application. If users can register themselves and choose their username, then nothing stops them from choosing names, such as `site`, or `users`, or `customer-records`, which correspond to the controller IDs. As URL rules are checked first, the existence of such a user will damage access to the referenced controller. For example, the `/customer-records` URL, which normally is identical to `/customer-records/index`, will be resolved to the View page of the user with the username `customer-records`.
>
> One way, which is quite cumbersome to maintain, is to check whether `$maybeUsername` is among some set of prohibited keywords, and this set can be automatically generated from the controller IDs used in your application.

On the other hand, if `$user` is really found, we construct the required route/parameters pair and return it:

```
$route = 'users/view';
$params = ['id' => $user->id];
return [$route, $params];
```

Second, we should be able to create URLs in the same format through our application by the `createUrl()` method. One place where we can find URLs to the View User pages is `/users/index`, with `GridView` listing the `UserRecord` models.

We should first check whether the route and parameters given to us are really relevant to us:

```
if ($route !== 'users/view' || !array_key_exists
    ('id', $params))
    return false;
```

We don't care about routes other than `users/view`, and we require that `id` to be given to us.

Also, we must be sure that the `UserRecord` really exists in the database, or else we will not be able to fetch the username:

```
$user = UserRecord::findOne($params['id']);
if (!$user)
    return false;
```

If everything's all right, all we have to do is return the username:

```
return "{$user->username}";
```

Note the absence of a leading slash, made explicit by the string interpolation notation. Yii 2 automatically inserts this slash for us, so `createUrl()` should return only the remaining string.

Now open the `/users` page, and hover on any button with the icon of an eye. It should have the URL equal to the corresponding username. After clicking on this button, you should end on the View page for the corresponding `UserRecord` object.

Summary

This chapter summarizes the routing mechanics that we dealt with through the whole book. We discussed the layer of custom routes that can be resolved back to basic controller actions.

We looked at two practical examples from real life that showed how we can utilize these mechanics. Everything else can be read in the documentation for `UrlManager` and `UrlRule` and additionally, in the `\yii\helpers\Url helper` class.

The next chapter will be the final one in our adventure. We'll deal with the infrastructure problems of developing a Yii 2 application, especially the problem of sharing the code base between developers and between development and production environments.

13
Collaborative Work

This is the last chapter of the book, except the appendices (which have useful information too; don't skip them).

We will conclude the adventure we started 12 chapters ago with the features of Yii 2, which are not related to the business value of the application being constructed but are related to its infrastructure.

When working with the code base together with a team of other developers and routinely deploying code between test and production servers, you will inevitably face some problems you need to solve to continue effectively delivering new features.

First, we'll learn some nice tricks to manage the application configuration to accommodate different deploy targets.

Secondly, we'll talk about the database migrations we were using the entire time until now. Hopefully, we have enough practical knowledge about using migrations. Here, we'll talk about the reasons to use them and some nice tricks to maintain them more effectively.

Finally, we'll look at the Yii console applications, the side we implicitly used several times in this book but never explicitly talked about.

Configuration construction

We all know the following problem:

Imagine we are developing a web application, and we do it at our local workstation. The application uses the locally-installed MySQL instance with some particular database name, username, and a password. When we deploy this application, it'll use the MySQL instance installed at the deploy target, over which we can or cannot have control in choosing names and passwords. Even if we have total control over the deploy target and can use the same connection settings, it's highly impractical to enforce our own usernames and passwords on other people in the team, who are working on the same web application on their own workstations.

As the Yii application configuration is just a PHP script returning an associative array, we have a simple way of solving this problem: all pieces of configuration that should be specified for each deploy target individually can be moved to separate files. The main configuration will just include their contents using standard `require()` calls.

In fact, we already have such a setup in our example CRM application. The following are the lines of code that include other pieces of configuration into our main configuration file, which is ultimately passed to the constructor of `\yii\web\Application`:

```
return [
    // … global settings skipped here ...
    'components' => [
        'db' => require(__DIR__ . '/db.php'),
        // … a lot of other components skipped here …
        'assetManager' => [
            'bundles' => (require __DIR__ .
                '/assets_compressed.php')
        ],
    ],
    'extensions' => (require __DIR__ .
      '/../vendor/yiisoft/extensions.php')
];
```

We have already used separate configuration for database, compressed assets (from *Chapter 8*, *Overall Behavior*), and extensions.

> It's probably the only place where an alternate syntax for the `require()` PHP built-in method looks useful. The `(require PATH)` notation explicitly expresses that we fetch something from `PATH` and put it right inside the brackets, which is arguably easier to read than the usual function-call notation.

Using the same technique, you can separate the declarations of the custom parameters in the `params` setting of the application. The Yii basic-application template uses this trick (see `https://github.com/yiisoft/yii2/tree/master/apps/basic/config`).

Configuration overriding can be raised to the next level if you take into account the `array_merge_recursive()` PHP built-in method and the `\yii\helpers\ArrayHelper::merge()` method from the Yii framework. The `ArrayHelper::merge()` static function is especially useful. Instead of combining values with identical keys, `array_merge_recursive()` overrides the initial value with the new one.

How can it be used? Obviously, we can have the default application configuration in one file and deploy target-specific overrides in another file. Then the resulting configuration to be fed to the Yii web application instance will be constructed by `ArrayHelper::merge()`, as follows:

```
// config/web.php, as we always had
return \yii\helpers\ArrayHelper::merge(
    (require "default.php"),
    (require "local.php")
);
```

Inside the `local.php` file, you maintain exactly the same structure as the `default.php` file, which is a lot easier to do than constantly remembering which part of the configuration lies in what configuration snippet. Of course, some files, such as the `extensions.php` file will be left as is. In case of the `extensions.php` file, we don't have control over its contents anyway.

Adding local overrides to the configuration

In *Chapter 2*, *Making a Custom Application with Yii 2*, we introduced a separate `db.php` configuration snippet to conform to the Yii basic application template, without really explaining exactly why we needed to separate this configuration at all. In fact, such a separation is a rudimentary step to making a code base more portable between developers and deploy targets. On each of them you can have different `db.php` scripts with credentials and the like. To achieve that, you cannot just commit this snippet to the version control system. However, such an approach has a serious drawback: each instance of the application should have a database configuration written manually from scratch. You need to communicate any common settings between developers in some other way than the code committed into repository, which means you cannot have any common settings committed to the repository, like maybe the custom database connection class or caching settings.

Let's apply the technique explained earlier to our code base. To do this, we need to make some really small changes:

1. First, create a subdirectory named `overrides` inside the `config` directory. It's more correct to name it `layers`, though, but `overrides` implies the meaning that the snippets will override each other, while `layers` require additional understanding about the technique we are employing.
2. Inside `overrides` create a file named `base.php`, that will hold the basic configuration for both console and web applications on any deploy target.
3. We will move a really small number of configuration settings to `config/overrides/base.php`. They are as follows:

```
return [
    'basePath' => realpath(__DIR__ . '/../../'),
    'components' => [
        'db' => [
            'class' => '\yii\db\Connection'
        ],
    ]
];
```

In the Yii advanced application template, there are separate subdirectories for the console and two web applications, so `basePath` for them will be different. In our case, both applications share the `basePath` setting.

We have initialized the database connection component with just the name of the class. The credentials will be provided in later overrides. We need the database connection in the console application as well as in the web application because of migrations.

Here are the steps to perform it:

1. Move the `web.php` configuration file to the `config/overrides/web_base.php` file. This name is chosen because we will recreate the `config/web.php` file later, and it's generally not so good to have two files in the code base sharing a name, even if they are in different folders.
2. Inside `web_base.php`, we need to remove parts of the configuration already defined in the base configuration, which are just the `basePath` and `components.db` settings.

3. Apart from that, it's important to change the paths to the `extensions.php` and `assets_compressed.php` configuration snippets inside `web_base.php`, as we are one folder deeper now. It's up to you whether to move `assets_compressed.php` somewhere, but it'll be a violation of the overrides concept to move this snippet to the `overrides` directory.
4. Next, move the `console.php` configuration file to the `config/overrides/console_base.php` file. Similarly, remove the `basePath` and `components.db` settings from there.
5. Now we arrive at the most interesting part. Create the `config/overrides/local.php` config file, with just the database connection settings in there as follows:

```
return [
    'components' => [
        'db' => [
            'dsn' => 'mysql:host=localhost;dbname=crmapp',
            'username' => 'root',
            'password' => 'mysqlroot'
        ]
    ]
];
```

   Regardless of how specific the local overrides, you need to have exactly the same structure of the configuration file as in the main Yii configuration. The whole idea of overrides is based on this sharing of structure.

6. With this local override in place, we can finally reconstruct `config/web.php` and `config/console.php` expected by our entry points. Here is how `config/web.php` will look:

```
return \yii\helpers\ArrayHelper::merge(
    (require __DIR__ . '/overrides/base.php'),
    (require __DIR__ . '/overrides/web_base.php'),
    (require __DIR__ . '/overrides/local.php')
);
```

We just merge the three configurations, and the order is crucial. Now, if you open the website in a browser, nothing should change, which is exactly what we need. All tests should still pass as well.

In the same manner, construct `config/console.php`, except instead of the `web_base.php`, we'll use `console_base.php`.

These changes lead to important consequences. The main benefit is that we can now add to the version control system only the base and web base configuration overrides. The local override will be constructed on each deployment. To help developers and operation engineers, we can commit to the repository a specially prepared copy of `local.php`, with all exact credentials replaced by some meaningful placeholders. Such configuration templates are usually named after the file they represent, with the `-example` suffix appended, for example, `local-example.php`. Here is how we can format such an example in our CRM application:

```
<?php
/**
 * This is the example for local configuration overrides.
 * You must declare at least the database connection parameters.
 */

return [
    'components' => [
        'db' => [
            'dsn' => 'mysql:host=localhost;dbname=DB_NAME',
            'username' => 'DB_USERNAME',
            'password' => 'DB_PASSWORD'
        ]
    ]
];
```

> So as not to commit the real `local.php` configuration snippet to the repository, in the Git version control system, you can add the `.gitignore` rule to it.

The advanced application template from the Yii bundle already uses a similar trick with overrides but only for custom application parameters (see, for example, `https://github.com/yiisoft/yii2/blob/master/apps/advanced/frontend/config/main.php`). To look at a real-world example of elaborated multilevel configuration building, you can turn to the YiiBoilerplate project from Clevertech at `https://github.com/clevertech/YiiBoilerplate`. It's for Yii 1.x, but the concept is the same, albeit a lot more extended.

Console application

In *Chapter 2*, *Making a Custom Application with Yii 2*, we set up the entity we called console runner in the form of the PHP script called `yii` in the root of the code base and skipped the explanations about what it really is. Starting then, we extensively used two particular command-line invocations, which used the scripts `./yii migrate/create` and `./yii migrate`. Also, we used the `./yii asset` invocations in *Chapter 8*, *Overall Behavior*, to prepare our compiled assets. Now it's time to explain the value and possibilities of this entity.

Apart from the `\yii\web\Application` class, which represents the web application we were building in the previous 11 chapters (*Chapter 1*, *Getting Started*, was excluded), the Yii 2 framework includes the `\yii\console\Application` class, which represents the console application. This kind of application conceptually is exactly the same as a web application in the sense that it is a module supporting MVC, too. Specifics of the console application lie in the fact that it must render the results of its work to the console, and it needs to receive input parameters from the command line. Apart from that, the console application uses the same concept of controllers (which should be descendants of the `\yii\console\Controller`, though) and is configured and created in the same way, which is obvious when you compare the contents of the `yii` and `web/index.php` scripts.

In essence, to reach the controller action using the console application, you make the following command-line invocation:

```
./yii controllerid/actionid param1value param2value
    --property1=value1
```

The details should be read from the framework documentation (`http://www.yiiframework.com/doc-2.0/guide-tutorial-console.html`). Note that the controller action parameters are passed as unnamed command-line arguments, and the values for controller properties are passed as named ones.

Unless you are exactly making the console application using the Yii framework, it's hard to justify the need to group the console controllers using modules. However, as the console application is still a module and uses the same routing mechanics as a web application, you can use the `/moduleid/.../moduleid/controllerid/actionid` routes as well.

As talking about console controllers is somewhat cumbersome, we'll refer to them as console commands, as Yii 1.x did, but what we really mean is the controller classes inheriting `\yii\console\Controller`. You get the set of console commands built into the framework, like `\yii\console\controllers\MigrateController`. They are listed in the `\yii\console\Application::coreCommands()` method, and you are encouraged to read through their classes' documentation. These commands are always accessible from the console application unless you override this method (and why should you do it at all?). We will not discuss them in detail as apart from the migrate and asset commands, all other are somewhat too specific.

You can declare and use custom properties on the console controllers. How to pass values to them at the time of calling a controller action will be discussed in the next section. How to set up values for them in the console application configuration will be explained in the next major section, which will deal with the database migrations specifically.

Custom console commands

Let's make some custom console command. We'll do a little hacking for developers to be able to custom-craft the user records directly in the database.

As you remember from *Chapter 5*, *User Authentication*, we store user passwords hashed using the Yii framework built-in helper classes. This hinders the ability of developers to make the user records manually using direct access to the database. As we can imagine and use some plaintext password easily, we need to store it in database hashed, and this hash needs to be computed somehow, which is basically impossible to do with pen and paper. More than that, we don't want to know exactly which methods of hashing were used at all.

So, let's make the console command which, given some string, computes and shows its hashed value in the same way our `\app\models\user\UserRecord::beforeSave()` method does.

As console controllers are controller classes, the Yii framework requires them all to be in the same directory. We arbitrarily decided that it was going to be the `commands` subdirectory under the root of the code base. According to PSR-4 rules and Yii framework conventions, this directory represents the `app\commands` namespace; so let's wire this namespace to our console application right now. Open the `config/overrides/console_base.php` file, and insert the `controllerNamespace` setting there:

```
return [
    'id' => 'crmapp-console',
    'controllerNamespace' => 'app\commands',
];
```

Let our new command be named hash, so it will be invocable by the `./yii hash` command. We need the `HashController` class, the `Controller` suffix being mandatory, inside the `commands` directory, with the following boilerplate code in it:

```
namespace app\commands;
use yii\console\Controller;
class HashController extends Controller
{
    public function actionIndex()
    {
        // don't know what to do yet
    }
}
```

You should already know how to declare new controllers, so the namespace and use clauses are most important pieces here. It's really important that we remember that we create the console controller and not a web one, as they run actions differently. The default action is `index`, as in web controllers.

Instead of using the `echo` PHP built-in to output to the console, we'll utilize the `\yii\helpers\Console` helper class from the Yii 2 framework. It has the `output()` method that will handle the newlines automatically, as appending them each time always was a hassle. Here is what we need to insert in order to satisfy our needs:

```
    public function actionIndex($string)
    {
        \yii\helpers\Console::output(\Yii::$app->security->
            generatePasswordHash($string));    }
```

The call to `Security::generatePasswordHash()` is exactly how we generate hash for the passwords inside our user records. Here is what you should get after calling `./yii hash 1234`, to see the hash of a really dumb user password:

```
vagrant@precise64:/vagrant$ ./yii hash 1234
$2y$13$GnGFWlYaecpoKoJTIUlPcuzqI7qyaODej8dQ6ZBMZHtmxgApSpiRq
```

You can also call this command `./yii hash/index 1234`. The hash will be different, of course, but it will be the same action being run nevertheless.

Note that the arguments of the controller action are passed as unnamed command-line arguments. This means, among other things, that you should pass strings containing special characters enclosed in quotes to the console commands of Yii. To help visualize what we are really hashing, let's output the string to be hashed as well. However, it's pretty nice that our initial command conforms to the Unix way of doing things of our initial command (that is, the fact that it emits results directly to standard output without any additional information). We'll utilize the built-in flag of the console controller to check whether we really need to output this additional data. Add the following two lines at the beginning of the `actionIndex()` method:

```
if ($this->interactive)
    Console::output(sprintf('Input string was: %s',
        $string));
```

The `$this->interactive` property being referenced is the `\yii\console\Controller::$interactive` one, available in all console commands. As was said before, you can set the values of such properties using named parameters. In our case, this means we need to pass `--interactive=0` to the `./yii hash 1234` call to suppress the debug output. Otherwise, we'll get a helpful reminder:

```
vagrant@precise64:/vagrant$ ./yii hash some string with spaces
Input string was: some
$2y$13$.C40HrK8Muj5SN3Fw2fpp.JJvdm7WPQlYnwDKS59kWGzWTJ2IqegS
vagrant@precise64:/vagrant$ ./yii hash "some string with spaces"
Input string was: some string with spaces
$2y$13$MNOHw.n0NEkt4wJvr.EjOuc.hFtwqG/qFq0hV5R5PkHibk9OHME.y
vagrant@precise64:/vagrant$ ./yii hash "some string with spaces" --interactive=0
$2y$13$2npFbKFaX3n7hDoiOn1rieNsvYp7vXU.QlJM.kzNpRPOel4wRrztu
```

This will help to resolve silly mistakes when handcrafting reasonably complex passwords, as otherwise you would not know what you are really hashing. Note which string was passed to our controller action in the first case, when we haven't escaped the input string.

To pass `false` to the controller properties, use any string that parses to the `false`-like value, like empty string or zero. The string `null` or `false` will be treated as a non-empty string and as such will be treated as a true Boolean value. So `--interactive=false` will not work as intended, but `--interactive=` will.

Database migrations

Dependency on the database is a hard one. Ideally, your application will not depend on the separate **relational database management system** (**RDBMS**) at all. But, most of the time, clients need the applications, which are conceptually just the elaborate CRUD interfaces over the database, so this dependency is inevitable.

The troubles caused with collaborative development are well known. If the application expects the database to be of some structure, and in the process of development a change is made to the code base, which changes some of these expectations, we need to correct the database schema on every machine this code base is currently deployed to. Of course, we should also have some sort of database schema initialization script, and this change must be introduced in this script too.

The *migrations* trick employed by Yii (and described in the documentation at `http://www.yiiframework.com/doc-2.0/guide-db-migrations.html`) has roots in the Ruby on Rails migrations concept (described in their alien documentation at `http://guides.rubyonrails.org/migrations.html`). You already saw that the Yii 2 database migration is essentially a small program that can use the methods of the `\yii\db\Migration` class to perform changes in the relational database schema in a uniform, vendor-independent way. Using them has obvious benefits over running the raw SQL scripts:

- You are independent of the database vendor. In fact, you can even swap the underlying database over the application lifetime without any change in the existing migrations collected over the time.
- You are at the application level when inside a migration. Arbitrary checks can be done before messing with the database. More than that, you actually can run the arbitrary code in the migration, not necessarily database-related at all.

There is an additional non-obvious feature: if you want, you can define both upgrade and downgrade routines. As a result, you will be able to move back and forth between versions of the database. However, this is only if you are very careful with your changes in the database, as some changes can be irreversible. This feature was presented to you in every migration template so far:

```
class m140318_173202_add_auth_key_to_user extends
    \yii\db\Migration
{
    public function up()
    {
        $this->addColumn('user', 'auth_key', 'string UNIQUE');
    }
```

```
    public function down()
    {
        $this->dropColumn('user', 'auth_key');
    }
}
```

While the `up()` method is what you usually want to make the change you need, the `down()` method is there to rollback changes. As it was said, there exist irreversible changes. For example, you can decide at some point to rehash user passwords using a newer crypt algorithm, and this is really a one-way operation, as you certainly don't want to store the old dataset just to have downgrading support. For such irreversible operations, you can just omit the `down()` method, and when downgrading, this migration will be silently skipped.

If the `up()` method returns a false Boolean value, this migration is considered not applied and all of the possible future migrations are canceled. This is useful if you make the changes that require possible manual preparations from the operator. The conditional expression in the migration will check whether the preparations are needed and halt the migration if necessary. Then, after the problem is resolved, the migration will proceed as usual.

The `down()` method has the same feature. If it returns a false Boolean value, the current downgrade along with all possible next downgrades will be canceled. This can be used in case you have some really irreversible changes in the `up()` method, after which there's nowhere to revert to. The default template for migrations in Yii 2 prepares exactly this kind of `down()` method for you.

While downgrading is especially useful for you to be able to easily correct possible mistakes in the `up()` method, it can be used to revert the state of the application to some specific point of the development history. This means that if someone encounters the bug in the database-dependent code in version x of your application, and the most recent version is, say, x + 5, then you can peek at the commit history in your version control system, notice the latest migration script present there, and roll back the migrations to that point. After that, you revert the code base five versions back using the version control system and will get the state of the application exactly at version x. This is less invasive than rolling the code base back to the version x, destroying the current database, and recreating it from scratch using only the `up()` methods, but you need pretty serious discipline to always make reversible or harmless irreversible changes in migrations.

Apart from this complication with the downgrade feature, there's absolutely nothing complex in the migrations concept itself. Using them, on the other hand, is a different topic.

We have used migrations extensively through the course of this book already, so you should be accustomed to the `./yii migrate` command itself (which is a shorthand to `./yii migrate/up`, and in the next section we'll explain why). In addition to this, running the `./yii help migrate` command will bring you the list of all possible invocations of `./yii migrate`.

To know which migrations are applied to the database and which are not (yet), Yii 2 creates and manages a special table in the database behind your back, the name of which is configured at the `\yii\console\controllers\MigrateController::$migrationTable` property. Each record in this table holds the name of the migration class used and the time of its usage. The default name for this table is `migration`, and if you happen to use this name for one of your own tables, you need to change this setting of `MigrateController`.

For such a fundamental change, it'll be cumbersome to pass the `--migrationTable` named parameter to each invocation of `./yii migrate`, so it's better to use the application configuration instead. As this is a controller and not a component, you need to use the `controllerMap` setting of the console application for this, as follows:

```
'controllerMap' => [
    'migrate' => [
        'class' => 'yii\console\controllers\MigrateController',
        'migrationTable' => 'my_custom_migrate_table',
    ],
]
```

In the same way, you can override any other property of the console controllers. This technique is applicable to web controllers as well.

> Note that this parameter overriding has an important side effect: we essentially declare the ID for the specific controller with preset parameters. Nothing stops us from declaring the same console controller class several times under different IDs and with different settings. For example, with `HashController` discussed before, we can do the following:
>
> ```
> 'controllerMap' => [
>     'silentHash' => [
>         'class' => 'app\commands\HashController',
>         'interactive' => false
>     ]
> ]
> ```
>
> This enables the `./yii silentHash` invocation for us without the need of additional class definition.

Another important property is `migrationPath`, which is the directory where `MigrationController` will search for the migration classes. By default, its `migrations` subdirectory is under the root of code base, and this is exactly what we decided on in *Chapter 2*, *Making a Custom Application with Yii 2* (miraculously). We used this property in *Chapter 6*, *User Authorization and Access Control*, when we set up the initial schema for RBAC in the database.

You can also use the `db` property, which should be either the `\yii\db\Connection` instance or the string ID of the application component, that is, the `\yii\db\Connection` instance. Using this property, you can run migrations on different databases. By manipulating the `migrationPath` and `db` properties inside the `controllerMap` setting of the console application, you can even manage several different databases inside the same application. By default, it has the value `db`, which is the ID of the default database connection component in the Yii application.

Finally, there is the `templateFile` setting, which we'll play with in the next section.

> You can read in the documentation for `MigrateController` that there are two complementary methods named `safeUp()` and `safeDown()`, which do what `up()` and `down()` do, correspondingly, except they do it in transactions. There is an important catch though. In MySQL, for versions up to 5.5 (at least), you may as well forget about them, as any command from the data definition language will automatically be committed. Thus, the following code will create the `first` and `second` tables, even while the transaction should fail because of an exception:
>
> ```
> public function safeUp()
> {
>     $this->createTable('first',
>         ['id' => 'pk', 'name' => 'string']);
>     $this->createTable('second',
>         ['id' => 'pk', 'value' => 'int']);
>     throw new \LogicException;
>     $this->createTable('third',
>         ['id' => 'pk', 'value' => 'date']);
> }
> ```
>
> This is documented behavior, but the importance of this point is maybe not stressed enough. The net result is that you should just not use the safe methods with MySQL at all. More than that, you should never override both the `up()` and `safeUp()` methods in the same migration class, as the parent implementation of `up()` is what calls `safeUp()` internally. The same is the case for `down()` and `safeDown()`.

Making custom templates for database migrations

Here is what a default template for database migrations looks like in the Yii 2 framework at the time of writing this chapter:

```
use yii\db\Schema;

class <?= $className ?> extends \yii\db\Migration
{
    public function up()
    {

    }

    public function down()
    {
```

```
        echo "<?= $className ?> cannot be reverted.\n";

        return false;
    }
}
```

Let's pretend that we are a careful and disciplined team of developers, which has excellent documentation on all source code files. We use the capable RDBMS, so transactions is a norm for us (let's not call any names, as flame war is not our intention), and we are pretty strict in our changes to the database schema, so our changes are almost always reversible. In this case, the following template will be more suitable for us:

```
/**
 * TODO: Migration explanation.
 */
class <?= $className ?> extends \yii\db\Migration
{
    public function safeUp()
    {
        // TODO: migration routine contents.
    }

    public function safeDown()
    {
        // TODO: migration rollback contents.
    }
}
```

The changes are as follows:

- Introduced the DocBlock for the explanation for migration
- Used the transactional versions of the `up()` and `down()` methods
- Clearly marked the places to be filled with actual code
- Removed `return false` from rollback routine, as our migrations will be reversible more often than irreversible

To not bloat the code base with another subdirectory, let's place this template in the `views/layouts/migration.php` file, as it is the most logical place to use. Do not forget that this file is a PHP script that will be processed by PHP runtime and outputted as any other PHP script. So, in addition to the previous code, the `views/layouts/migration.php` file should contain the following lines at the top:

```
<?php
/**
 * Template for migrations.
 * Property named `MigrateController.templateView` controls what
template to use.
 */
echo "<?php\n";
?>
```

It's good style to provide explanations for any source code file overall, and we must output the `<?php` processing directive at the beginning of the resulting file.

Now, it's really simple to wire this template to our application, so all future migrations will be written based on this template. We need to use the `controllerMap.migrate.templateFile` setting in the console application configuration:

```
    'controllerMap' => [
        'migrate' => [
            'class' =>
              'yii\console\controllers\MigrateController',
            'templateFile' => '@app/views/layouts/migration.php'
        ]
    ]
```

Unfortunately, even if Yii automatically enables `MigrateController` to be available, we still must specify the `class` setting for the `migrate` controller ID. Not so hard, though. Note that the `templateFile` setting can accept the path aliases, which is really useful.

With this preparation in place, all future migrations created will look as described before, quite differently from the standard view. As we already explored before, by defining several different controller IDs in the `controllerMap` setting, we can configure several different invocations of the `MigrateController` using different templates, if needed.

Summary

That's all, folks.

This chapter, the last one in the book, has dealt with the furthest layer of the Yii 2 framework: maintenance of the code base itself. We covered three topics here:

- How to construct the application configuration dynamically so that developers will be able to easily deploy it to any target machine
- How to make custom console commands to provide nice tricks that help in development
- How to customize the default look of the autogenerated migration scripts

We also learned how the database migrations and console commands in general are implemented in Yii. And let's not forget that we know about two helper classes now: `ArrayHelper` and `Console`.

To be honest, we glossed over a lot of topics, and a lot of information was simply not present here, because it would be a verbatim copy from the already existing Yii documentation. To become even more fluent in the Yii 2 framework, it would be quite useful to learn about the following advanced features in real depth instead of a quick mention in this book:

- At the data layer: Validators, active query, and `\yii\db\Command`
- At the presentation layer: Built-in widgets, the entire `yii\rest` namespace, support for files uploading, methods of the active form widget rendering the form fields, and also, surprisingly, the `\yii\captcha\Captcha` class, which is a complete captcha solution implemented
- Middleware components: The dependency injection container implementation inside the `yii\di` namespace, mutexes inside the `yii\mutex` namespace, helper classes, and the support to internationalize the application

The Yii framework documentation is excellent and all-encompassing, and the documentation blocks inside the source code are more like the definitive reference. In this book, we tried to cover the themes which are either not obvious from these sources of information or for some reason, were not exposed from the underlying source code.

A
Deployment Setup with Vagrant

As it's pretty important that you should be able to develop the application locally, let's use the Vagrant project for setting up the local deploy target.

To be brutally short, Vagrant (see `http://www.vagrantup.com/`) is a toolset to manage virtual machines from the command line. The end result of your setup would be this:

1. You boot your own workstation.
2. You go to the root of the code base.
3. You issue `vagrant up` in the command line.
4. You wait and then open the `http://localhost:8888/` URL in your browser.
5. Your web application responds to you from there.
6. You issue `vagrant halt` in the command line.
7. Your web application is not accessible now and does not waste any resources anymore.

All dependencies for your application, including the web server and database, are inside the virtual machine image managed implicitly by the Vagrant toolset. Nothing leaks to your host system.

Vagrant supports several virtual machine vendors, and we should have no trouble if we decide to use the VirtualBox project (see `https://www.virtualbox.org/`) as both technologies are open source and free to use.

Planning

As we'll configure the virtual machine, we need to plan our LAMP stack in detail right from the start.

We need to set up four things:

1. PHP 5.4.
2. Apache 2.4 (it's modern standard anyway).
3. MySQL 5.5 (we need a database because of our application domain, which is data management).
4. A website configured to be accessible from inside the virtual machine.

More than that, we'll need a setup script that will automate the necessary preparations of the base system, including installing all the things mentioned for setup.

Deployment will be greatly simplified by Vagrant, because it just makes the directory holding the code base (where you call `vagrant up`, specifically) shared inside the virtual machine as the directory named `/vagrant`. That slash is important as it's the `vagrant` folder at the root of the directory tree. As a result, the code base will always be synchronized between the deploy target and the development workstation, so you can use whatever toolchain you use to write and maintain the code base, and it will transparently and constantly be sent to the deployment target.

Vagrant hides the management of the virtual machine behind the concept of box. The box is a specially prepared virtual machine image in any format recognizable by Vagrant. These boxes have to be downloadable from somewhere, and a website located at `http://www.vagrantbox.es/` holds references to a lot of such prepared images for Vagrant.

However, Vagrant recognizes, and is able to, install some boxes right out of the box. One of them is called precise64, which is Ubuntu 12.04. For simplicity, we'll use this box as it requires essentially no additional setup outside the box.

Initial setup

Before we go into explanations, let's create the required configuration for Vagrant. Create a file named `Vagrantfile` at the root of your project code base, and write the following code into it:

```
Vagrant.configure("2") do |config|

# Which box we'll be using as base
```

```
  config.vm.box = "hashicorp/precise64"

# Following is only for reference, as Vagrant knows
# where to get the "precise64" box right from the start.
#  config.vm.box_url = "http://files.vagrantup.com/precise64.box"

# What to do with the base box as initial setup
  config.vm.provision :shell, :path => "bootstrap/01-prepare-
precise64.sh"
  config.vm.provision :shell, :path => "bootstrap/02-configure-app-
for-precise64.sh"
  config.vm.provision :shell, :path => "bootstrap/03-prepare-
application.sh"

# How to expose the web application inside the box:
# publish port 80 at the virtual machine as port 8888 at the host
machine.
  config.vm.network "forwarded_port", guest: 80, host: 8888

end
```

This file is Ruby code. If you want, you can write arbitrary expressions here as long as you do `Vagrant.configure` in it as well.

With this file in place, you can do the magic of `vagrant up`. When you issue this command for the first time in an unprepared machine and code base, Vagrant will do the following:

1. Download the box from `box_url`. In our case, Vagrant knows already where the box named `hashicorp/precise64` is.
2. Unpack the box and register it as the virtual machine image in the VirtualBox suite. The unpacked box will be stored in the home directory of the current user and will be reused for all later invocations of `vagrant up`.
3. Launch the virtual machine using the usual VirtualBox means. If you wish, you can even open the VirtualBox management UI and see the Vagrant-managed virtual machine listed among other machines, if any.
4. Run the provisioning scripts, that is, launch everything listed in the `config.vm.provision` setting. In our case, provisioning is split into three separate shell scripts, which must be run in exactly that order.
5. Perform the port forwarding as we specified.

It can do some other things as well, but we are interested only in these steps.

At the second and later invocations of `vagrant up`, Vagrant will just launch the virtual machine and forward the ports. As was already said, the code base will be constantly shared between virtual and host machines.

It is obvious that Vagrant provides an enormously useful local deployment environment for developers. The most complex part is to correctly craft the provisioning scripts.

Fine-tuning the virtual machine

As specified in the `Vagrantfile`, we are going to have three provisioning scripts to be run in that order. Let's look at them in the same order.

In the code bundle provided with this book, these scripts are in the `bootstrap` directory exactly as shown in the preceding code example. In this folder, a bunch of other files also exists; these are various configuration files for different programs that Vagrant will place in appropriate places according to our bootstrap scripts.

Preparing the guest OS

We are using Ubuntu 12.04, which does not have PHP 5.4 in its installation repositories. On the other hand, we are in an environment where `apt-get` is available, so we can install absolutely anything we need relatively easily.

We will not provide the complete content of the `01-prepare-precise64.sh` file as it's quite long and boring. You can look at it in the code provided with this book in the `bootstrap` folder. Here is what this script does in the order in which it appears here:

1. Adds the special **Personal Package Archives** (**PPA**) with the latest PHP versions.
2. Updates the package database of `apt-get`.
3. Installs the latest Apache 2, PHP 5.4, `vim`, latest MySQL (both server and client), `git` (it's required to launch the composer inside the virtual machine), and the CURL package for PHP5 (it's required for almost anything related to PHP nowadays).
4. Installs the X virtual framebuffer, Java runtime, and the Firefox browser so you will be able to run acceptance tests directly inside the Vagrant box.
5. Removes the default virtual host defined for Apache.
6. Enables the `mod_rewrite` option for Apache as it's required for Yii 2 to deal with some URLs.

7. Makes Apache run under the Vagrant user account, so we'll get the application code base writable by the web server (no need for strict security in this setup).
8. Does a small additional tweak to suppress the default warning about the server name not being defined.

Essentially, this script is for setting up the platform, not an application. However, as we will, for simplicity, use the root password to access the database, it will leak from the level next to this one.

> You should know about the VirtualBox Guest Additions plugin to comfortably work with Vagrant over VirtualBox. Vagrant relies on this plugin to perform the magic code base synchronization, and the catch is that it must be installed both on the host and the guest machines, *and their versions must exactly match*. Most probably, the VirtualBox Guest Additions plugin on your host machine will be newer than the one installed inside the guest box. As a result, Vagrant will complain and nothing good will happen. To resolve this problem, the Vagrant team has provided us with their own plugin called vagrant-vbguest. You need to call `vagrant plugin install vagrant-vbguest`, and after that at every call, `vagrant up`, so that this plugin checks the versions of Guest Additions and installs the correct version into the guest machine if necessary (it can take significant time, though).

Preparing the database and web server

The second level of provisioning is a bridge between a platform and an application. It is quite short:

```
# Separately specified settings for database
# NOTE that the password was already specified before in previous
bootstrap script!
DB_USER=root
DB_PASS=mysqlroot
DB_NAME=crmapp

# Creating database
# NOTE the absence of the space between `-p` flag and the password!
mysql -u ${DB_USER} -p${DB_PASS} -e "create database if not exists
`${DB_NAME}` default character set utf8";
mysql -u ${DB_USER} -p${DB_PASS} -e "create database if not exists
${DB_NAME}_test default character set utf8 default collate utf8_
unicode_ci";
```

```
# Copy the prepared Apache config from codebase to the Apache config
folder.
cp -f /vagrant/bootstrap/frontend.apache2.conf /etc/apache2/sites-
enabled/
# Restart Apache so new virtual host will be published.
/etc/init.d/apache2 restart
```

All we do is create the databases for the application and the functional tests and copy the already-prepared Apache configuration from the `bootstrap` folder inside the code base to the place expected by Apache. This configuration holds the definition of a port-based virtual host, which is the reason why we removed the default configuration at the previous level.

Preparing the application

The last provisioning script is the most semantically complex. Here are its contents, at the final stage of the example application development, after *Chapter 13*, *Collaborative Work*:

```
# Go to the root project folder
cd /vagrant

# Install all prerequisites, including Yii
php composer.phar install --prefer-dist

# Copy the prepared config snippets to the configuration tree
cp bootstrap/local.php config/overrides/

# Copy the prepared config snippet for test database connection to the
configuration tree
cp bootstrap/test.php config/

# Initialize the RBAC tables
./yii migrate --migrationPath='@yii/rbac/migrations' --interactive=0

# Initialize the database overall
./yii migrate --interactive=0
```

Here, we proceed in the following order:

1. Install all dependencies managed by the composer, including Yii 2 itself.
2. Copy the configuration for this local deploy target to the location expected by our application (see *Chapter 13*, *Collaborative Work*, for details).

3. Run the built-in migration in Yii to set up tables for the database-backed RBAC manager (see *Chapter 6*, *User Authorization and Access Control*, for details).
4. Run all of our migrations collected over the course of the book.

All three of these scripts are relatively harmless, and as a result, you can launch provisioning manually and safely without breaking anything. This is done by calling `vagrant provision`. It's useful to quickly reload the Apache configuration, for example.

Using the virtual machine as a local deploy target

You stop the virtual machine managed by Vagrant by calling `vagrant halt`. If you want to completely get rid of the machine, you call `vagrant destroy`, which can be really useful when preparing the local setup.

Automatic sharing of the code base between the guest and host machines relieves you from manually deploying the code base. Basically, your deploy action reduces to just running migrations from time to time.

You access your newborn local deploy target by issuing `vagrant ssh` at the root of the code base, after which you are free to do anything. With the MySQL configuration shown previously in this appendix, you can access the database from within the virtual machine by issuing the following command:

```
$ mysql -u root -pmysqlroot crmapp
```

Do remember that your code base is not in the home directory of the Vagrant user, at which you end right after the `vagrant ssh` call, but in the `/vagrant` one, so doing `cd /vagrant` can easily become your habit.

The most important setup is the test suite carefully constructed in *Chapter 2*, *Making a Custom Application with Yii 2*, and *Chapter 3*, *Automatically Generating the CRUD Code*.

As was stated back in *Chapter 3*, *Automatically Generating the CRUD Code*, running acceptance tests too many times (more than 10, actually) will overfill the grid views in the CRUD interfaces, which are configured to show only 20 items per page. After that, tests start to fail.

So, it's up to you to manually clean the database after acceptance tests as they have no means to clean up themselves. Inside the code bundle provided for this book there is a script called `reset_database.sh`, which helps you in this task. It contains just four lines of shell code as follows:

```
# Drop the production database
mysql -u root -pmysqlroot -e "drop database if exists crmapp; create
database crmapp default character set utf8 default collate utf8_
unicode_ci";

# Initialize the RBAC setup in empty database
./yii migrate --interactive=0 --migrationPath='@yii/rbac/migrations'

# Run all our own migrations
./yii migrate --interactive=0

# Make the SQL data dump for Codeception
mysqldump -u root -pmysqlroot crmapp > tests/_data/dump.sql
```

In effect, running this script will completely rebuild your database to clean its state. Do not use something like that on the application deployed in production! The second helper script that is provided with this book is `selenium.sh`. This script will launch the local instance of the Selenium server configured in such a way that you will be able to run acceptance tests right there in the Vagrant box. It's not as reliable as using an actual remote connection to separate a deploy target (because a connection is being made to the localhost, bypassing DNS and other things like proxies), but it's convenient, as you are able to use it immediately. Launch this helper script in a separate command line, and then run the acceptance tests in another one.

The third helper script is called `minify_assets.sh`, and it's just running the correct invocation of the `./yii asset` command described in *Chapter 8*, *Overall Behavior*, in the section about minifying assets.

Overall, the code bundle provided with this book is configured in such a way that you will be able to launch Vagrant box and then immediately be able to launch the full test suite using the single command:

```
$ ./cept run
```

This `cept` script is a shorthand to call the Codeception executable inside the `vendor` directory.

B
The Active Form Primer

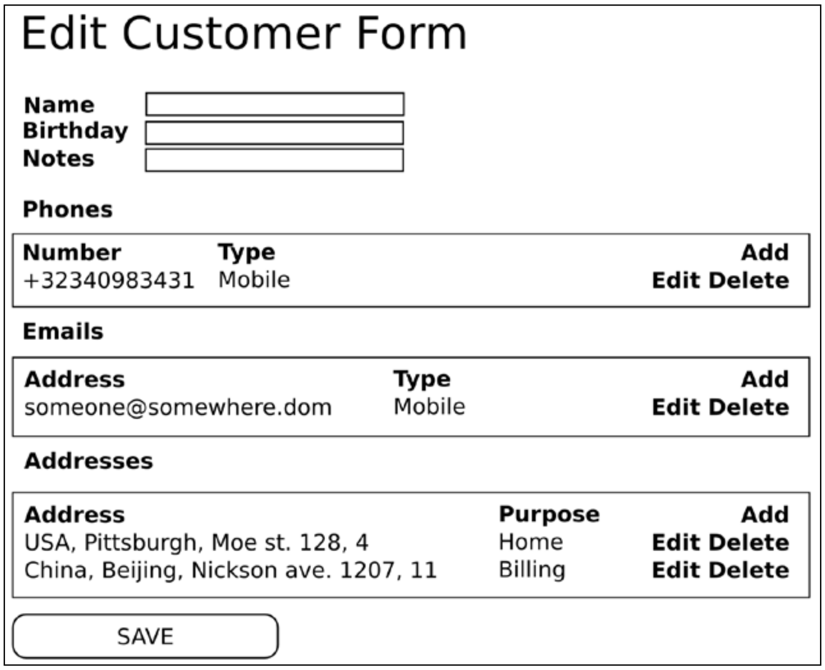
When making the Yii-flavored user interface in *Chapter 11*, *The Grid*, we centered only around the GridView widget, as it was the topic of the chapter. But there is another important part of any web application, and it's an HTML form.

The Yii 2 framework provides a very robust and convenient-to-use widget called ActiveForm, which can semi-automatically construct an HTML form for us given the `ActiveRecord` instance describing the model being manipulated. We'll just continue from where we stopped, at the very end of *Chapter 11*, *The Grid*.

Making the Edit form for customer

Our Customer models are already being stored in two separate tables in the database, and two different active records are used to create a single customer. With the addition of the Address and Email models, we are meeting the problem of the need for a user interface to create and update Customer models.

We don't have either the book volume or the intention necessary to implement some JavaScript-heavy **rich UI**, even if it would undoubtedly be more responsive and visually pleasing. So, let's go old school and make a web interface traditional to the world of static HTML pages. Here is the sketch:



So, phones, e-mails, and addresses will be presented as tables, with the buttons **Add**, **Edit**, and **Delete**. These tables will behave in exactly the same manner as the user interface used at `/user/index` and `/services/index`, which we have made using the Gii automatic CRUD generator. The **Add** and **Edit** buttons will transfer us to the add/edit pages corresponding to the submodel we want to add/edit, and when we click on **Save** on those pages, we'll be transferred back to this **Edit Customer Form** page.

Active query

What is the active query concept? It is a **domain-specific language** (**DSL**) for various tasks used to query the database for active records, which are possibly related to one other. On the one hand, it hides the intricacies of constructing the appropriate SQL and on the other, constructs the `ActiveRecord` instances from raw datasets. This concept is implemented as a `\yii\db\ActiveQuery` class, which is an extension of `\yii\db\Query`.

While a simple query returns data as associative arrays from the database, an active query is tailored to return iterable collections of active records, which is useful when you are on a level higher than raw data from the database. Of course, constructing the full `ActiveRecord` instance for each record returned by a query is a costly operation, so with convenience comes a performance penalty. However, generally it's wiser to think about your architecture first and tweak performance later when needed, because if you don't use ActiveRecords or Repository pattern over ActiveRecords, you will use the API for direct database access, and it's a lot harder to maintain it in the long term.

> The `ActiveQuery` is a *very* large class ultimately. It inherits from the `Query` class, which uses the trait `\yii\db\QueryTrait`, which itself uses two traits, `\yii\db\ActiveQueryTrait` and `\yii\db\ActiveRelationTrait`. There can be a whole book written only for the sake of explaining the intricacies of using the `ActiveQuery` class of Yii 2. You are encouraged to look at the documentation and source code for this class and explore it yourself.

We don't need all the functionality of `ActiveQuery`. To show a simple, expressive example, here is how you can get the customers who should be congratulated with their birthday in this week, but only those who were registered by the manager currently being logged in:

```
        $week_ago = (new DateTime)->sub(new DateInterval('P1W'))-
>format( 'Y-m-d' );
        $current_user = Yii::$app->user->id;
        $customers = CustomerRecord::find()
            ->where(
                ['and', 'created_by=:current_user', 'birth_
date>=:week_ago'],
                compact('current_user', 'week_ago')
            )->all();
```

What does this code do? Here's what it does:

1. The call to `ActiveRecord.find()` returns us an `ActiveQuery` instance configured for the `CustomerRecord` active record.
2. Its `where()` method configures this `ActiveQuery` instance to filter the records according to two of our conditions.
3. Finally, the `all()` method returns the array of `ActiveRecord` instances to us.

We already mentioned long ago the built-in PHP function named `compact()`.

This is usually the way to use `ActiveQuery`: we enter the querying mode by calling `find()` on the `ActiveRecord` class, and when we finish chaining all the methods required, we pull out the `ActiveRecord` instances back, returning from the `ActiveQuery` DSL.

Another way to use `ActiveQuery` is to manually construct and pass them to the methods that expect it. One such place is the `DataProvider` instance, which was described back in *Chapter 2*, *Making a Custom Application with Yii 2*.

Customizing the autogenerated form

Let's look at what the **Create Customer Record** form looks like by opening the `/customer-records` page and clicking on the big green **Create Customer Record** button above the table:

How is it implemented? Tracing the `/customer-records/create` route, we end in the `views/customer-records/create.php` view file, which calls `_form.php` in the same folder. This view file is what governs the rendering of this form. You can note that `update.php` and `create.php` are almost the same and use the same form inside the `_form.php` script. This is intentional, as in Yii 2 ideology, the creating and updating of an `ActiveRecord` are very similar actions.

If you open the view file, you will see quite a simple structure, ignoring the raw HTML code:

1. We initialize the `ActiveForm` by calling `$form = ActiveForm::begin()`.
2. Then we output input fields by calling `$form->field($model, $attr)->textInput($setup)`.
3. Then, we do some magic to render either the **Create** or **Update** submit button with proper CSS classes.
4. Then, we end the `ActiveForm` by calling `ActiveForm::end()` (note that we call the static method and not the method of `$form` instance).

First, let's change this form layout to horizontal according to our sketch. As we already have the Yii 2-bootstrap extension attached to our application, this is extremely simple. We just have to declare our form to be not of the class `\yii\widgets\ActiveForm`, but of the class `\yii\bootstrap\ActiveForm`. This can be done in the block of `use` clauses at the top of the view file. Find the following declaration:

```
use yii\widgets\ActiveForm;
```

Replace it with the following line of code:

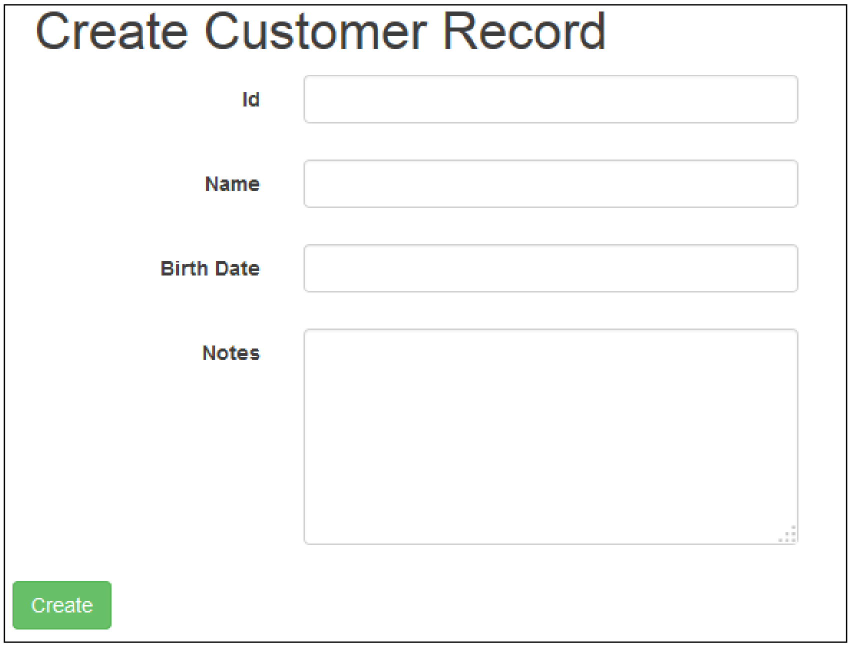
```
use yii\bootstrap\ActiveForm;
```

With this declaration substitution, we don't even need to change the code in the view file itself.

After that, we need to modify the `ActiveForm::begin()` call by adding a `layout` setting to the widget config:

```
<?php $form = ActiveForm::begin(['layout' => 'horizontal']);?>
```

That's all, our form now looks like the one shown in the following screenshot:



We certainly don't need an editable ID of the customer, so we need to remove the following line:

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

We have to introduce an important guard case here. To create the new customer record, we need to remove the subtables for phones, addresses, and e-mails, because we need an ID that will be assigned to the customer record only after it is saved to the database.

Put the following `if-endif` brackets into the `_form.php` file:

```
<?= $form->field($model, 'notes')->textarea(['rows' => 6]) ?>

<?php if (!$model->isNewRecord):?>
<!-- subtables will be here… -->
<?php endif?>

<div class="form-group">
```

This will prevent Yii from rendering the tables we're going to describe next just in case the customer record is a new one. All the following code examples are assumed inside the `if-endif` brackets!

Next, let's turn to the topic of adding a grid view for phone records associated with the customer being updated. We will start simple:

```
<h2>Phones</h2>
<?= \yii\grid\GridView::widget([
    'dataProvider' => new \yii\data\ActiveDataProvider([
        'query' => $model->getPhones(),
        'pagination' => false
    ]),
    'columns' => ['number']
]);?>
```

This will give us the following output when phones are to be attached to this new customer record:

We will use the relation method created in *Chapter 11*, *The Grid*, on the `CustomerRecord` class to return an `ActiveQuery` instance to find the `PhoneRecord` instances.

We don't need pagination in this grid view, as the number of records most probably will be small anyway. While `GridView` has a special `pager` setting, it is to customize the whole `\yii\widgets\LinkPager` widget, which is responsible for rendering all of those numbers and arrows inside squares. We, on the other hand, want to disable the whole *notion* of pagination for the Phones list. Thus, we need to use the `pagination` setting on `ActiveDataProvider` one abstraction deeper.

We don't need anything apart from the phone number, so only one column is explicitly declared. Without the declaration of columns, this grid view will break inside the **Create Customer Record** form, because there will no `PhoneRecord` instances to extract the column schema from.

Now, we'll do the real magic and wire this table to the CRUD functionality we have made for `PhoneRecord` in the previous section. For this, we need to add a special column that defines actions available to be performed on the corresponding active phone records as follows:

```
<?= \yii\grid\GridView::widget([
    // ...
    'columns' => [
        'number',
        [
            'class' => \yii\grid\ActionColumn::className(),
        ]
    ]
]);?>
```

You can guess now that the column definition can be the configuration array to be consumed by `Yii::createObject()`. Here is how this column will look:

If you look at the URLs behind the icons, you can see that it's the view, update, and delete actions we need, but they're tied to the current controller; we want `phones` instead of `customer-records`. This is solved quite simply by telling the `ActionColumn` class which controller it should relate itself to:

```
<?= \yii\grid\GridView::widget([
    // ...
    'columns' => [
        'number',
        [
            'class' => \yii\grid\ActionColumn::className(),
            'controller' => 'phones'
        ]
    ]
]);?>
```

Last, but not least, we need the **Add Phone** button. Let's do this now and put the button link right inside the column header as we sketched it previously:

```
<?= \yii\grid\GridView::widget([
    // ...
    'columns' => [
        'number',
        [
            'class' => \yii\grid\ActionColumn::className(),
            'controller' => 'phones',
            'header' => Html::a('Add New', ['phones/create']),
        ]
    ]
]);?>
```

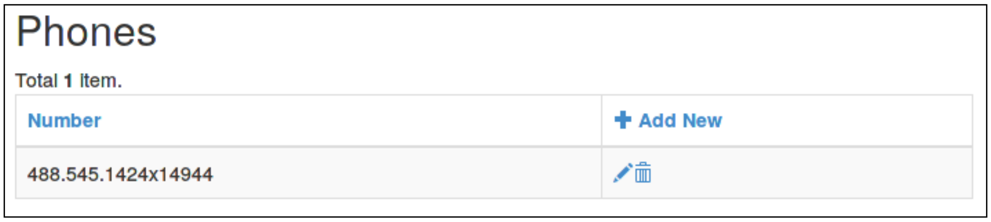
To make it better, let's stuff the icon for the plus sign beside the **Add New** label as follows:

```
'header' => Html::a(
    '<i class="glyphicon glyphicon-plus"></i> Add New',
    ['phones/create']
),
```

Also, we don't need the view icon (the one with the eye), as the number is shown to us anyway, and we can see the details in the **Update Phone** form in the same way. Have a look at the following code:

```
<?= \yii\grid\GridView::widget([
    // ...
    'columns' => [
        'number',
        [
            'class' => \yii\grid\ActionColumn::className(),
            // ...
            'template' => '{update}{delete}',
        ]
    ]
]);?>
```

Now it's perfect. The preceding code will give us the following output:

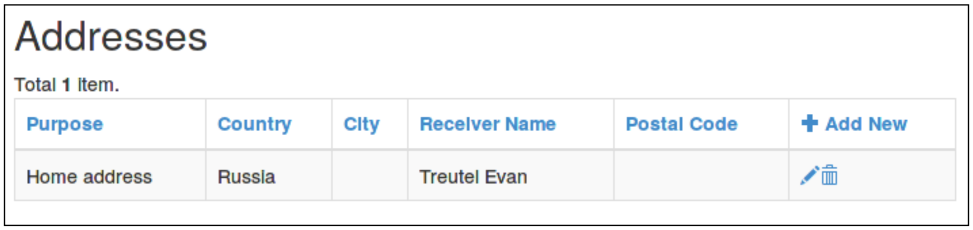


The column width can be corrected by applying CSS, which we don't care about now.

Then we add the `Addresses` grid view as follows:

```
<h2>Addresses</h2>
<?= \yii\grid\GridView::widget([
    'dataProvider' => new \yii\data\ActiveDataProvider(
        ['query' => $model->getAddresses(), 'pagination' => false]
    ),
    'columns' => [
        'purpose',
        'country',
        'city',
        'receiver_name',
        'postal_code',
        [
            'class' => \yii\grid\ActionColumn::className(),
            'controller' => 'addresses',
            'template' => '{update}{delete}',
            'header' => Html::a(
                '<i class="glyphicon glyphicon-plus"></i> Add 
New',
                ['addresses/create']
            ),
        ],
    ],
]);?>
```

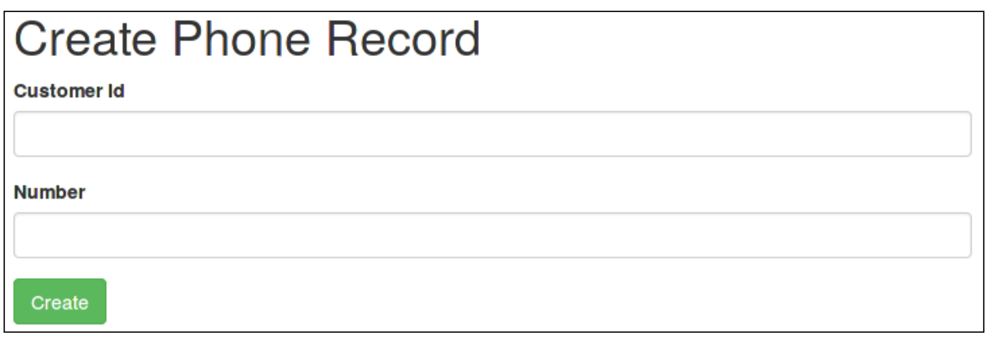
The pieces different from the `Phones` grid view are highlighted. This should result in the following table:



Similarly, you can craft the code yourself for the **Emails** grid view given this empty reference table:

Passing the customer ID to submodels

We have a small problem in our UI. Let's click on this new **Add New** button that we have created just now for the **Phones** grid view:



We don't pass the customer ID to the phone record being created! And we don't need this field at all, actually, especially if we are passing the ID automatically. We need a way to pass the customer ID to the Create Phone Record submodel (and other submodels too).

The simplest way is to modify `SubmodelController.actionCreate()` in such a way that it accepts an additional parameter. Let's name it in a generic way as follows:

```
public function actionCreate($relation_id)
```

Then, in the configuration for the header of the last action column, we can add the `relation_id` parameter to the URL being created:

```
'header' => Html::a(
    '<i class="glyphicon glyphicon-plus"></i> Add New',
    ['phones/create', 'relation_id' => $model->id]
),
```

It should be clear now after explanations in *Chapter 12*, *Route Management*, that arguments to the `action*` methods on the controllers become mandatory `GET` or `POST` parameters for the associated route with the same name.

Then, we can correctly place the given `relation_id` in the record being created:

```
public function actionCreate($relation_id)
{
    /** @var ActiveRecord $model */
    $model = new $this->recordClass;
    $model->customer_id = $relation_id;
// … other code ...
```

However, as we already named our input argument in a generic way, let's abstract from the customer concept here and make the target field generic too as follows:

```
$model->{$this->relationAttribute} = $relation_id;
```

This is another example of the capabilities of PHP in metaprogramming, as we have just used the string stored inside a variable as an attribute name of an object (whose class itself was deduced from a string stored inside a variable). Of course, we have to declare this property now. Have a look at the following code:

```
    /** @var string Name of the attribute which will store the given
relation ID */
    public $relationAttribute;
```

Also, correct the definition of `AddressesController`, `EmailsController`, and `PhonesController` by adding the following declaration inside them:

```
public $relationAttribute = 'customer_id';
```

> This is an example of the harm caused by over-generalizing in your code. What began with seemingly small and nice generalization now cost us triple code duplication. And we cannot default the value to `customer_id`, because we have `SubmodelController` completely decoupled from our domain now, and by doing so, we will break the abstraction, which looks even worse than blunt code duplication.

We can now remove the fields for the `customer_id` property from `views/addresses/_form.php`, `views/emails/_form.php`, and `views/phones/_form.php` in the same way we removed the `id` field from the **Update Customer** form.

Returning to the Update Customer form after updating the submodel

That's not all, though. When we create or update the phone, address, or e-mail, after clicking on the submit button, we are redirected to the `/phone/view`, `/address/view`, or `/email/view`, respectively, which is not what we want. It's better if we would be redirected to the page we were on before, that is, the **Update Customer Record** form page. This is quite easy to achieve using the tools Yii 2 provides us with.

First, we need to record the **Update Record** page URL by doing this in the `\app\controllers\CustomerRecordsController::actionUpdate()` method as follows:

```
$this->storeReturnUrl();
```

The `storeReturnUrl()` is a well-named function, which we define ourselves as follows:

```
    private function storeReturnUrl()
    {
        Yii::$app->user->returnUrl = Yii::$app->request->url;
    }
```

We are storing the URL from `\yii\web\Request.getUrl()` with a `\yii\web\User.setReturnUrl()` call using a nice syntax provided to us by the `__get()` and `__set()` magic methods from Yii 2.

Then we go to the `SubmodelController` class again to the method `actionCreate()`, and find the following redirection:

```
            return $this->redirect(['view', 'id' => $model->id]);
```

We need to change it to the following code:

```
                return $this->goBack();
```

This is simpler and does what we need. The `\yii\web\Controller.goBack()` method does redirect to the URL stored in `\yii\web\User.getReturnUrl()`, and it's exactly what we set before. Documentation for this property of the User component states that it's the URL we should redirect the user to after a successful login, but in reality, you can call `goBack()` in any place, thus this `returnUrl` is general-purpose.

You should perform the same replacement in the `actionUpdate()` method. Additionally, inside the `actionDelete()` method, you should replace the following redirect:

```
        return $this->redirect(['index']);
```

The preceding code should be replaced with the following code:

```
        return $this->goBack();
```

Or else, after you click on the **Delete** button beside any phone, address, or e-mail and confirm deletion, the system will try to redirect you to the nonexistent `actionIndex()` for the corresponding model.

Custom column value for the addresses table

We have only one discrepancy left between our sketch and the current state of the Update Customer form: the look of the addresses subtable. We need only a single column with the whole address written in one line.

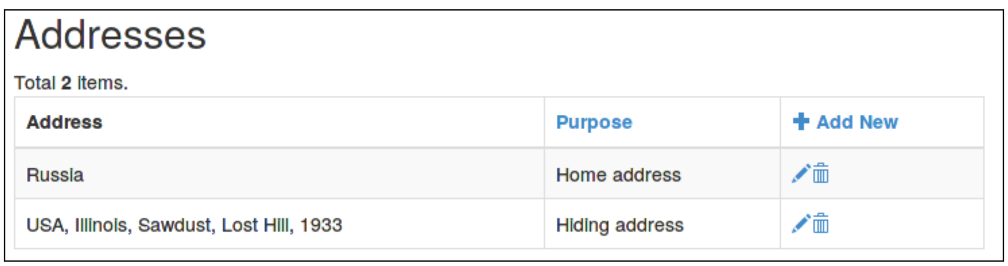
Yii 2 provides two options for us here, which are as follows:

1. The first option is to subclass `\yii\grid\DataColumn`, which is the default class for the `GridView` column, and write our own code to make the cell contents.
2. The second option is to use just the `\yii\grid\DataColumn.value` property to display the value we need. This is perfectly suitable for our problem as we don't need to do much apart from gluing together pieces of information from `AddressRecord`.

So, let's make the following straightforward implementation of the value for the **Address** column in the **Addresses** subtable:

```
            'columns' => [
                [
                    'label' => 'Address',
                    'value' => function ($model) {
                        return implode(', ',
                            array_filter(
                                $model->getAttributes(
                                    ['country', 'state', 'city', 'street',
    'building', 'apartment'])));
                    }
                ],
                'purpose',
                [
                    // … ActionColumn here which we don't care about...
                ],
            ],
```

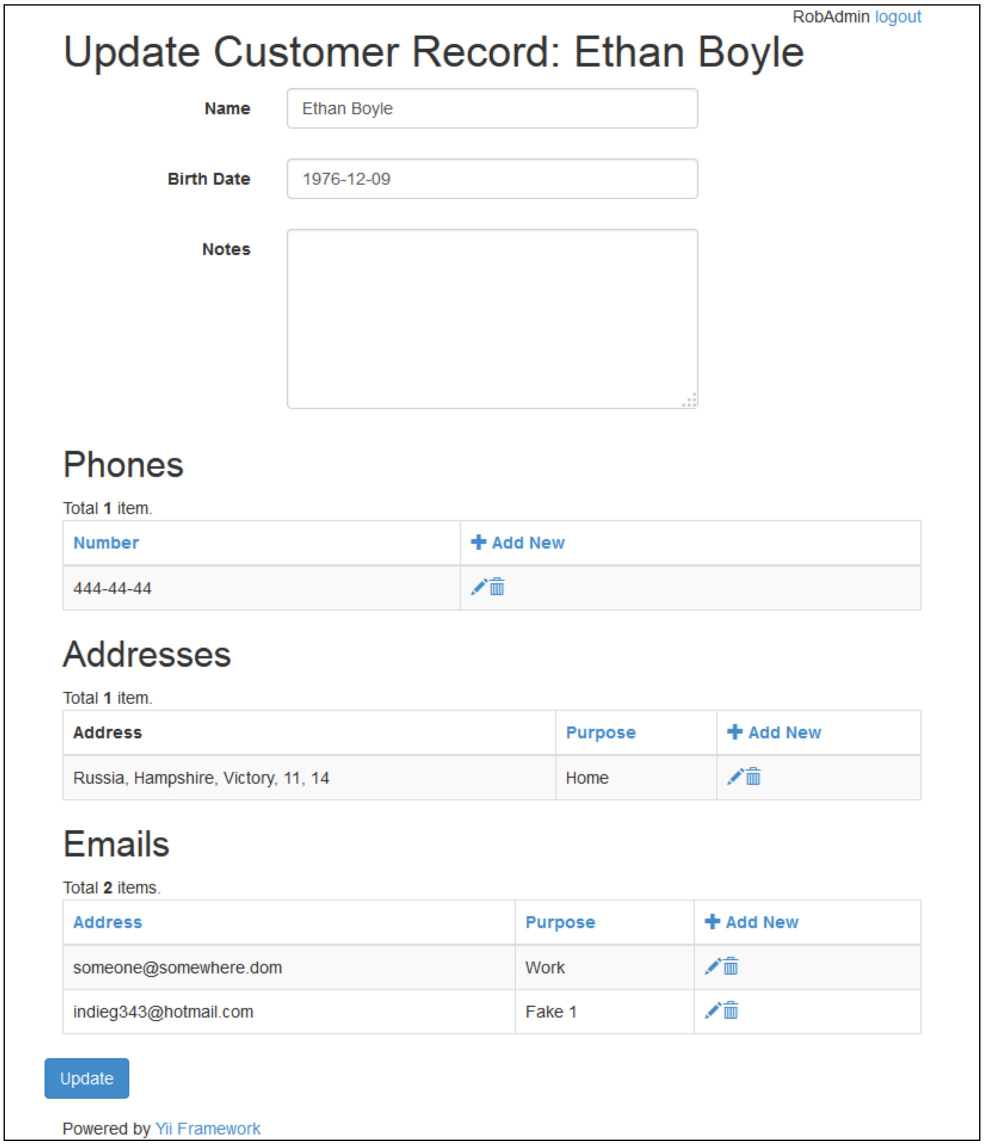
You can see that we can pass arbitrary callables to the `value` property. Also, the `label` property controls the text in the header of the column. In the end, here is what we get with this column definition:


Addresses

Total **2** items.

Address	Purpose	+ Add New
Russia	Home address	
USA, Illinois, Sawdust, Lost Hill, 1933	Hiding address	

In the end, we get the result pretty close to what we sketched at the beginning of this section:



We have not covered in detail the important topic of the active form handling: the validators. Do not miss the opportunity to learn about form validation in official documentation, as it's a really useful feature of the framework.

Index

A

F

S

T

U

Z

Thank you for buying
Web Application Development with Yii 2 and PHP

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

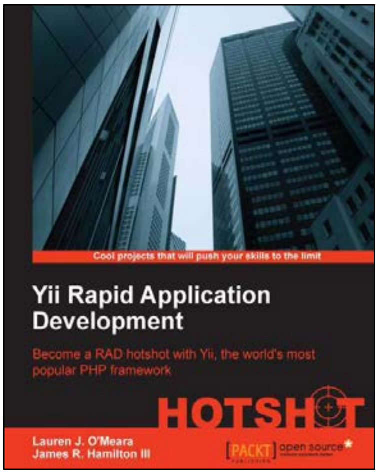
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Yii Rapid Application Development HOTSHOT

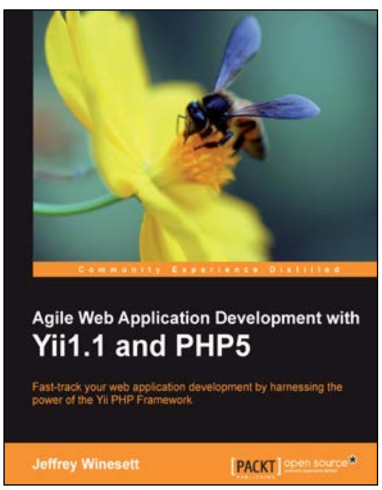


ISBN: 978-1-84951-750-8 Paperback: 340 pages

Become a RAD hotshot with Yii, the world's most popular PHP framework

1. A series of projects to help you learn Yii and Rapid Application Development.
2. Learn how to build and incorporate key web technologies.
3. Use as a cookbook to look up key concepts, or work on the projects from start to finish for a complete web application.

Agile Web Application Development with Yii1.1 and PHP5



ISBN: 978-1-84719-958-4 Paperback: 368 pages

Fast-track your web application development by harnessing the power of the Yii PHP Framework

1. A step-by-step guide to creating a modern, sophisticated web application using an incremental and iterative approach to software development.
2. Build a real-world, user-based, database-driven project task management application using the Yii development framework.
3. Take a test-driven design (TDD) approach to software development utilizing the Yii testing framework.

Please check **www.PacktPub.com** for information on our titles