Quick answers to common problems

Play Framework Cookbook

Over 60 incredibly effective recipes to take you under the hood and leverage advanced concepts of the Play framework

Alexander Reelsen

[PACKT] PUBLISHING open source* community experience distilled

Play Framework Cookbook

Over 60 incredibly effective recipes to take you under the hood and leverage advanced concepts of the Play framework

Alexander Reelsen

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Play Framework Cookbook

First published: July 2011

Production Reference: 2290711

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

ISBN 978-1-849515-52-8

www.packtpub.com

Cover Image by Fillippo (filosarti@tiscali.it)

Credits

Author

Alexander Reelsen

Reviewers

Erik Bakker

Guillaume Bort

Steve Chaloner

Pascal Voitot

Acquisition Editor

Eleanor Duffy

Development Editor

Roger D'souza

Technical Editor

Kavita Iyer

Copy Editor

Neha Shetty

Project Coordinator

Joel Goveya

Proofreader

Aaron Nash

Indexer

Hemangini Bari

Tejal Daruwale

Graphics

Nilesh Mohite

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

Foreword

Why Play is a small revolution in the Java world

Looking at the past years of application development, as a developer you might have noticed a significant shift from desktop applications to web applications. The Web has evolved as the major platform for applications and is going to take over many facets—not only in development but also in everyday life, resulting in this shift accelerating. Who would have thought 10 years ago that current mobile phones are indeed only very strong ironed notebooks with a permanent Internet connection?

The Internet provides a very direct connection between consumer and producer. For application developers this implies a very easy -to- use- and- handle platform. Looking around, many application frameworks have evolved in recent years in order to be very Internet-centric. These frameworks interpret the Web as an ubiquitous platform for providing not only ordinary web pages, as it was done 10 years ago. The web has become a data provider on top of one of the most proven protocols in industry, the HyperText Transfer Protocol (HTTP). The core concepts of the Internet being a decentralized highly available network with HTTP as a protocol on top of it are the inner core of a big part of today's applications. Furthermore, another development took place in the last years. The browser became more and more a replacement of the operating system. Fully fledged web applications like Google Docs, which act and look like desktop applications, are becoming more popular. JavaScript engines like Google V8 or SpiderMonkey are getting insanely fast to deliver web browser performance not thought of several years ago. This means current web applications are now capable of delivering a real user experience similar to applications locally installed on your system.

So many software engineers today are also web engineers as well. This poses a very big problem. As most of software engineering is based on abstraction, many tools, frameworks, and languages try to hide complexity from the engineer, which sounds good at first. No web engineer cares about the fragmentation of the IP packets which are sent throughout the whole world in milliseconds. By abstracting and layering your software to reduce the complexity per layer, you inadvertently might hide features of the underlying protocol your software is built upon. Many frameworks try to resemble the style of programming desktop applications, with the exception of application being in the Web. However, in order to make use of HTTP and its features you cannot easily hide them away in your application framework.

This is especially a problem in the Java world. The defined standard is the servlet spec, which defines how web applications have to be accessible in a standard way. This implies the use of classes like `HttpServletRequest`, `HttpServletResponse`, `HttpServlet`, or `HttpSession` on which most of the available web frameworks are built upon. The servlet specification defines the abstraction of the HTTP protocol into Java applications. Though it is quite a good spec as HTTP carries quite some complexity around, it forces frameworks to obey certain conventions which never got challenged in the past.

While many web frameworks like Django, Rails, or Symfony do not carry the burden of having to implement a big specification and do not need to fit into a big standardized ecosystem, most Java web frameworks have never questioned this. There are countless excellent web frameworks out there which implement the servlet specification, Grails, Tapestry, Google Web Toolkit, Spring Web MVC, and Wicket to name a few. However, there always was one gap: having a framework which allows quick deployment like Django or rails while still being completely Java based. This is what the Play framework finally delivers.

This feature set does not sound too impressive, but it is. Being Java based implies two things:

- Using the JVM and its ecosystem: This implies access to countless libraries, proven threading, and high performance.
- Developer reusability: There are many Java developers who actually like this language. You can count me in as well. Have you ever tried to convince Java developers to use JavaScript as a backend language? Or PHP? Though Groovy and Scala are very nice languages, you do not want your developers to learn a new framework and a new language for your next project. And I do not talk about the hassle of IDE support for dynamic languages.

Shortening development cycles is also an economic issue. As software engineers are quite expensive you do not want to pay them to wait for another “compile-deploy-restart” cycle. The Play framework solves this problem.

All of the new generation web frameworks (Django in Python, Rails on Ruby, expressjs on top of nodejs in JavaScript) impose their own style of architecture, where HTTP is a first class citizen. In Java, HTTP is only another protocol that a Java application has to run on.

So the Play framework is a pure Java-based solution, which brings a real HTTP focused framework with lots of helpers to speed up development resulting in shorter iterations and faster deployments.

This book should help you to get the most out of the Play framework and to be as fast as any other developer on any platform when creating web applications.

Prerequisites

I made several assumptions about the persons reading this book. One of the first assumptions is that you already have used Play a little bit. This does not mean that you have deployed a 20 node cluster and are running a shop on top of it. It means that you downloaded the framework, took a brief look at the documentation, and ran through a few of the examples. While reading the documentation you will also take a first look at the source, which is surprisingly short. I will try to repeat introductory stuff only when it is necessary and I will try to keep new things as short as possible, as this is a cookbook and should come with handy solutions in more complex situations.

What is missing: A Scala chapter

No book is perfect. Neither is this. Many people would be eager to read a chapter about integration of Play and Scala. When I started writing this book, my Scala knowledge was far from competitive (and still is in many areas). Furthermore I currently do not think about using Scala in a production web application together with Play. This will change with growing maturity of the integration of these two technologies.

Alexander Reelsen

About the Author

Alexander Reelsen is a software engineer living in Munich, Germany, where he has been working on different software systems, for example, a touristic booking engine, a campaign management and messaging platform, and a b2b ecommerce portal. He has been using the Play framework since 2009 and was immediately astonished by the sheer simplicity of this framework, while still being pure Java. His other interests includes scaling shared-nothing web architectures and NoSQL databases.

Being a system engineer most of the time, when he started playing around with Linux at the age of 14, Alexander got to know software engineering during studies and decided that web applications are more interesting than system administration.

If not hacking in front of his notebook, he enjoys playing a good game of basketball or streetball.

Sometimes he even tweets at `http://twitter.com/spinscale` and can be reached anytime at `alexander@reelsen.net`.

If I do not thank my girlfriend for letting me spend more time with the laptop than with her while writing this book, I fear unknown consequences. So, thanks Christine!

Uncountable appreciation goes out to my parents for letting me spent days and (possibly not knowing) nights in front of the PC, and to my brother Stefan, who introduced me into the world of IT - which worked pretty well until now.

Thanks for the inspiration, fun, and fellowship to all my current and former colleagues, mainly of course to the developers. They always open up views and opinions to make developing enjoyable.

Many thanks go out to the Play framework developers and especially Guillaume, but also to the other core developers. Additionally, thanks to all of the people on the mailing list providing good answers to many questions and all the people working on tickets and helping to debug issues I had while writing this book.

My last appreciation goes to everyone at Packt Publishing, who was involved in this book.

About the Reviewers

Erik Bakker works as a software engineer for Lunatech Research in Rotterdam, The Netherlands. Lunatech provides consulting and application development services for Play and a range of other Java technologies, with small teams of highly skilled people, and employs two Play committers. Erik has a long-time interest in web application development and has been building web application for years, both small and large, using various languages and frameworks. In 2010, he got hooked to the Play framework, which he has been using for various projects, and has since written several editorials on the subject. Erik excels at translating business needs into technical requirements and solutions. He holds a master's degree in physics from Utrecht University and you can write to him at `erik@lunatech.com`.

Guillaume Bort is co-founder and CTO of Zenexity, a French "web-oriented architecture" company. He is the creator and lead developer of the Play framework, which makes it easier to build Web applications in Java. Made by web developers, Play focuses on developer productivity and targets RESTful architectures.

Steve Chaloner, a Brit living in Belgium, has been developing in Java since 1996, and has been an avid user of the Play framework since 2010. Steve has introduced Play into several companies for projects ranging from the fairly small to the extremely large. He is the author of several Play modules, including the Deadbolt authorization system.

His company, Objectify (`http://www.objectify.be`), specializes in rapid development using JVM languages and runs training courses on Play.

Greet and Lotte – thanks for putting up with the late nights when I was working on the review. Love you both.

Pascal Voitot lives in Paris, France and is 34 years old. He is a senior developer and technical expert mainly focused on open source server solutions. He has been working in the mobile telecommunication and banking domains developing industrial solutions and then in R&D on secured mobile services. In parallel, he has become an enthusiast open source supporter and contributor. He currently works as technical expert in Java open source business services, mainly focused on web content management solutions.

His favorite subjects nowadays are nowadays lightweight and distributed servers, rich client and fast development web/mobile interfaces, and simple and scalable content management systems.

He's currently lead developer of Siena project (`http://www.sienaproject.com`), a Java lightweight object mapping API bridging SQL and NoSQL databases based on active record design. He is also an active supporter and contributor of Play, a "Rails inspired" Java/Scala framework.

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.

`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

The Play framework is the new kid on the block of Java frameworks. By breaking the existing standards it tries not to abstract away from HTTP as with most web frameworks, but tightly integrates with it. This means quite a shift for Java programmers. Understanding the concepts behind this shift and its impact on web development with Java are crucial for fast development of Java web applications.

The Play Framework Cookbook starts where the beginner's documentation ends. It shows you how to utilize advanced features of the Play framework—piece by piece and completely outlined with working applications!

The reader will be taken through all layers of the Play framework and provided with in-depth knowledge with as many examples and applications as possible. Leveraging the most from the Play framework means, learning to think simple again in a Java environment. Think simple and implement your own renderers, integrate tightly with HTTP, use existing code, and improve sites' performance with caching and integrating with other web 2.0 services. Also get to know about non-functional issues like modularity, integration into production, and testing environments. In order to provide the best learning experience during reading of Play Framework Cookbook, almost every example is provided with source code. Start immediately integrating recipes into your Play application.

What this book covers

Chapter 1, *Basics of the Play Framework*, explains the basics of the Play framework. This chapter will give you a head start about the first steps to carry out after you create your first application. It will provide you with the basic knowledge needed for any advanced topic.

Chapter 2, *Using Controllers*, will help you to keep your controllers as clean as possible, with a well defined boundary to your model classes.

Chapter 3, *Leveraging Modules*, gives a brief overview of some modules and how to make use of them. It should help you to speed up your development when you need to integrate existing tools and libraries.

Chapter 4, *Creating and Using APIs*, shows a practical example of integrating an API into your application, and provides some tips on what to do when you are a data provider yourself, and how to expose an API to the outside world.

Chapter 5, *Introduction to Writing Modules*, explains everything related to writing modules.

Chapter 6, *Practical Module Examples*, shows some examples used in productive applications. It also shows an integration of an alternative persistence layer, how to create a Solr module for better search, and how to write an alternative distributed cache implementation among others.

Chapter 7, *Running in Production*, explains the complexity that begins once the site goes live. This chapter is targeted towards both groups, developers, as well as system administrators.

Appendix, *Further Information About the Play Framework*, gives you more information about where you can find help with Play.

What you need for this book

Everything you need is listed in each recipe.

Who this book is for

This is the ideal book for people who have already written a first application with the Play Framework or have just finished reading through the documentation. In other words - anyone who is ready to get to grips with Play. Having a basic knowledge of Java is good, as well some web developer skills—HTML and JavaScript.

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 are shown as follows: "Create an `conf/application-context.xml` file, where you define your beans."

A block of code is set as follows:

```
require:
    - play
    - play -> router head
```

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: "Click on the **Downloads** menu and get the latest version.".

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 book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

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/support`, 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
Basics of the Play Framework

In this chapter, we will cover:

- Downloading and installing the Play framework
- Creating a new application
- Defining routes as the entry point to your application
- Configuring your application via `application.conf`
- Defining your own controllers
- Defining your own models
- Using Fixtures to provide initial data
- Defining your own views
- Writing your own tags
- Using Java Extensions to format data in your views
- Adding modules to extend your application
- Using Oracle or other databases with Play
- Understanding suspendable requests
- Understanding session management

Introduction

Let's begin with Play framework. There is no need for XML configuration, no need to create a war file, just start off with a commandline tool. Play is a full stack. Everything is bundled. All you need to do is download the Play framework, unzip it, and use it.

Once you install it, this chapter will give you a head start about the first steps to carry out after you create you first application. It will provide you the basic knowledge needed for any advanced topic, which is described in the later chapters. After this chapter you know where to look for certain files and how to change them.

Some features presented here are also shown in the only example application for the first chapter, which you can find at `examples/chapter1/basic-example`.

Downloading and installing the Play framework

This recipe will help you to install the Play framework as quickly and unobtrusively as possible in your current system.

Getting ready

All you need is a browser and some basic knowledge about unzipping and copying files in your operating system. Also be aware that you can install Play on Microsoft Windows, Linux as well as Mac OS X or even BSD.

How to do it...

Open up a browser and go to `http://www.playframework.org/download` and download the most up-to-date stable version `http://download.playframework.org/releases/play-1.2.zip` (at the time of writing this recipe `play 1.2` was the latest stable version).

After downloading it, unzip it, either with a GUI tool or via command line zip:

```
unzip play-1.1.zip
```

If you are using Linux or MacOS you might want to put the unzipped directory in `/usr/local/` in order to make Play available to all the users on your system; however, this is optional and requires the root access on the particular system:

```
mv play-1.1 /usr/local/
```

As a last step adding the Play binary inside the `play-1.1` directory to the `PATH` environment variable is encouraged. This is easily possible with a symlink:

```
ln -s /usr/local/play-1.1/play /usr/local/bin/play
```

If you enter `play` on your commandline, you should get an ASCII art output along with some help and, most importantly, the version of Play you just called. If you do not want to create a symlink, you can also copy the Play binary to a path, which is already included in the PATH variable. In the preceding example you could have copied it to `/usr/local/bin`.

How it works...

As just mentioned, Play would also work by just unzipping the Play framework archive and always using the absolute path of your installation. However, as this is not very convenient, you should put your installation at the defined location. This also makes it quite easy for you to replace old Play framework versions against newer ones, without having to change anything else than the created symlink.

If you are on a Linux system and you do not see the ASCII art output as mentioned some time back, it might very well be possible that you already have a Play binary on your system, installed. For example, the sox package, which includes several tools for audio processing, also includes a Play binary, which surprisingly plays an audio file. If you do not want to have this hassle, the simplest way is just to create the symlink with another name such as:

```
ln -s /usr/local/play-1.1/play /usr/local/bin/play-web
```

Now calling `play-web` instead of `play` will for sure always call the Play framework specific script.

Creating a new application

After installing the necessary parts to start with Play, the next step is to create a new application. If you are a Java developer you would most likely start with creating a Maven project, or alternatively create some custom directory structure and use Ant or scripts to compile your sources. Furthermore, you would likely create a WAR file which you could test in your web application server. All this is not the case with the Play framework, because you use a command line utility for many tasks dealing with your web application.

How to do it...

Change into a directory where you want to create a new application and execute the following command:

```
play new myApp
```

How it works...

This command will create a new directory named `myApp` and copy all needed resources for any project into it. After this is done, it should be finished in almost no time. The following file system layout exists inside the `myApp` directory:

```
./conf
./conf/dependencies.yml
./conf/routes
./conf/application.conf
```

```
./conf/messages
./test
./lib
./public
./app
./app/models
./app/controllers
./app/views
```

If you are familiar with a rails application, you might be able to orientate very quickly. Basically, the `conf` directory contains configuration and internationalization files, where as the `app` folder has a subdirectory for its model definitions. Its controllers contain the business logic and its views, being a mix of HTML and the Play template language. The `lib` directory contains jar libraries needed to run your application. The `public` folder contains static content like JavaScript, CSS, and images; and finally the `test` folder contains all types of tests.

There's more...

Generally speaking, you can add arbitrary content in the form of directories and files in the application directory; for example, the files needed to support Eclipse, or NetBeans will be put here as well. However, you should never remove data which has been copied during the creation of the application unless you really know what you are doing.

Support for various IDEs

You can add support for your IDE by entering: `playeclipsify`, `playidealize`, or `playnetbeansify`. Every command generates the files needed to import a Play application into your favorite IDE.

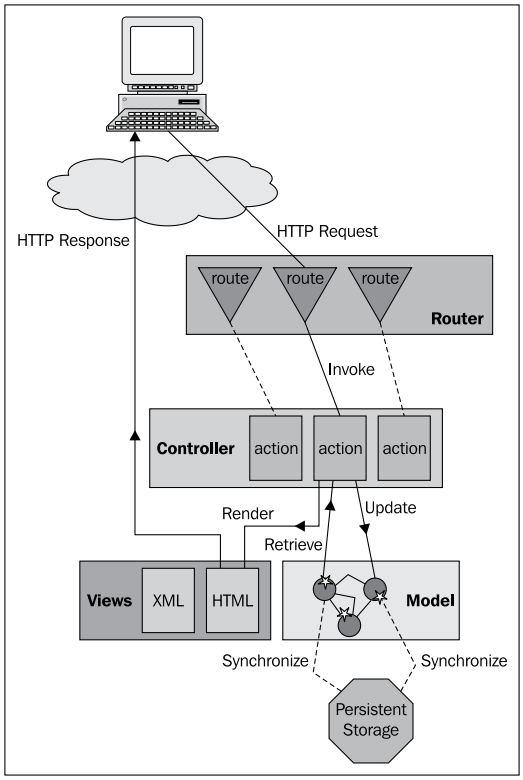
Defining routes as the entry point to your application

If you analyze a HTTP based application, every execution of logic consists of three things. First there is a HTTP method called as well as a specific URL. Optionally, the request may include a payload like request parameters or a request body. No matter what you look at: a news or a blog article, your favorite social network, you are bidding for some item during an online auction, or delete e-mails in your web mail account, you always use an URL an a HTTP method. Any unbelievably complex application such as Google Mail basically boils down to this contract. So what would be easier for a web application than to resemble this contract by mapping the tuple of HTTP method and HTTP URL to certain actions. This is what the routes file is for in Play.

Getting ready

As seen some time back in the filesystem layout, after creating a new application, there is a `conf/routes` file. This file can be seen as the central point of your application. In order to have a truly REST based architecture, the combination of the HTTP method and URL define an implicit action. Using HTTP GET on any URL should never ever change any resource because such calls are seen as idempotent calls and should always return the same result.

In order to fully understand the importance of the routes file, this graphic illustrates that it is the starting point for every incoming HTTP request:



The image is also available at `http://www.playframework.org/documentation/1.2/images/diagrams_path`.

Basically the router component parses the routes file on startup and does the mapping to the **Controller**.

Always be aware that no logic can be executed from an external source if not triggered by the router component, which in turn is configured by default in the `conf/routes` file.

How to do it...

Edit your routes file as shown in the following code snippet:

```
GET      /              Application.index
POST     /users         Application.createUser
GET      /user/{id}     Application.showUser
DELETE   /user/{id}     Application.deleteUser

# Map static resources from the /app/public folder to the /public path

GET     /public         staticDir:public
```

How it works...

The preceding example features a basic application for user management. It utilizes HTTP methods and URIs appropriately. For the sake of simplicity, updating a user is not intended in this example. Every URI (alternatively called a resource) maps to a Java method in a controller, which is also called an action. This method is the last part of the line, with the exception to the HTTP resource `/public`, where the public directory is mapped to the public URI. You might have already noticed the usage of some sort of expression language in the URI. The ID variable can be used in the controller and will contain that part of the URI. So `/user/alex` will map alex to the ID variable in the `showUser` and `deleteUser` methods of the controller.

Please be aware that some browsers currently only support GET and POST methods. However, you can freely use PUT and DELETE as well, because Play has a built-in workaround for this which uses POST and setting the X-HTTP-Method-Override header telling the framework to execute the code as needed. Be aware to set this request header when writing applications yourself, that connect to a play-based application.

There's more...

As seen in the preceding screenshot, the router component can do more than parsing the routes file. It is possible to have more complex rules such as using regular expressions.

Using regular expressions in the URL is actually pretty simple, as you can just include them:

```
GET     /user/{<[0-9]+>id}          Application.showUser
```

This ensures that only numbers are a valid user ID. Requesting a resource like `/user/alex` now would not work anymore, but `/user/1234` instead would work. You can even create a List from the arguments in the URL with the following line of code:

```
GET     /showUsers/{<[\0-9]+>ids}/?  Application.showUsers
```

In your application code you know you could use a `List<Integer>` IDs and show several users at once, when the URL `/showUsers/1234/1/2` is called. Your controller code would start like this:

```
public static void showUsers(@As("/") List<Integer> ids) {
```

This introduces some new complexity in your application logic, so always be aware if you really want to do this. One of the usecases where this is useful is when you want to use some sort of hierarchical trees in your URLs, like when displaying a mailbox with folders and arbitrary subfolders.

See also

You can also use annotations to create routing, which offers you some more flexibility. See the first recipe in *Chapter 2*. Furthermore, routing can also be done for virtual host, and this will also be presented later on.

Configuring your application via application.conf

Though Play does not require a lot of configuration to run, there has to be one file where basic information such as database connection strings, log levels, modules to enable additional functionality, supported application languages, or the setting of the application mode is configured. This file is `conf/application.conf`, though it looks like a properties file, it really is not because of its UTF-8 encoding.

How to do it...

Just open `conf/application.conf` with your any editor supporting UTF-8, be it Eclipse, Vim, textmate, or even notepad.

Now every configuration option follows the scheme:

```
# Some comment
key = value
```

Notable for application start is the `http.port` option, which runs your Play application per default on port 9000; this can be changed to any port above 1024 if you are a non-root user, which is highly recommended anyway.

How it works...

By definition Java property files are ISO-8859-1 and nothing else. Play, however, is thought of as an everything-UTF-8 framework; hence, the application configuration filename does not have a `.properties` suffix. For more info about standard Java properties, please refer to:

`http://download.oracle.com/javase/6/docs/api/java/util/Properties.html`

As the documentation covers most of the possible parameters in the configuration file pretty well, this file will only be mentioned if the default configuration has to be changed.

Most importantly, adding and configuring modules in order to enhance the basic functionality of Play is part of the `application.conf`, and each module requires enabling it via defining its path:

```
module.foo=${play.path}/modules/foo
```

After starting your Play application, the console output should include information about which of your configured modules have been loaded successfully.

Please be aware that from play 1.2 modules are not configured via this mechanism, but via the new `dependencies.yml` file. You can still configure modules this way, but this is deprecated from then on.

Another important setup is the log level of your application when using `log4j`, which is used by Play framework all over the place. When in production mode, it should be set to INFO or ERROR; however, in testing mode the following line might help you to discover problems:

```
application.log=DEBUG
```

See also

We will refer to the `application.conf` file when setting up special databases later on in this chapter. Also there is an own *Configuring log4j for log rotation* recipe in the *Chapter 7*, *Running in Production*.

Defining your own controllers

Controllers represent the thin layer between HTTP and business logic. They are called by the router, once the requested resource is mapped successfully to a controller method specified in the `conf/routes` file.

Getting ready

In order to follow this recipe, you should use the `conf/routes` file defined in the recipe *Defining routes as the entry point to your application* in this chapter.

How to do it...

Fire up your favorite editor, open `app/controllers/Application.java`, and put the following into the file:

```
package controllers;

import play.*;
import play.mvc.*;
public class Application extends Controller {

    public static void index() {
        render();
    }

    public static void showUser(String id) {
         render();
    }

    public static void deleteUser(String id) {
        render();
    }

    public static void createUser(User user) {
        render();
    }
}
```

How it works...

Absolutely no business logic happens here. All that is done here is to create a possibility to execute business logic. When looking back at the `conf/routes` file you see the use of the `id` parameter, which is again used here as a parameter for the static method inside the `Application` class. Due to the name of the parameter it is automatically filled with the corresponding part of the URL in the request; for example, calling `GET /user/1234` will result in putting "1234" in the ID parameter of the `showUser` method. The most important fact to make a Java class work as controller is to have it extend from `play.mvc.Controller`. Furthermore, all your actions have to be static in order to be executed as controller methods.

As no business logic is executed here (such as creating or deleting a user from some database) the `render()` method is called. This method is again defined in the controller class and tells the controller to start the rendering phase. A template is looked up and rendered. As Play also follows the convention over configuration pattern, a default template location is assumed, which follows an easy naming scheme:

```
./app/views/${controller}/{method}.html
```

In the case of showing a user it would be:

```
./app/views/Application/showUser.html
```

There's more...

This not only looks pretty simple, it actually is. As Play framework follows the MVC principle, you should be aware that the controller layer should be as thin as possible. This means that this layer is not for business logic but merely for validation in order to ensure the model layer will only get valid data.

Using POJOs for HTTP mapping

As it is not convenient for any web developer to construct the objects by hand from the HTTP parameters, Play can easily do this task for you like this:

```
public static void createUser(User user) {
    // Do something with the user object
    // ...
    render();
}
```

This requires a certain naming convention of your form elements in the HTML source, which will be shown later.

Using HTTP redirects

Instead of just rendering HTML pages there is another great feature. You can trigger a HTTP redirect by just calling the Java method. Imagine the following code for creating a new user:

```
public static void createUser(User user) {
    // store user here..., then call showUser()
    showUser(user.id);
}
```

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.

Now the last line of code will not call the static `showUser` method directly, but instead issue a HTTP 304 redirect response to the client, which includes a `Location: /show/1234` response header. This allows easy implementation of the common redirect-after-post pattern, without cluttering your application logic. You only need to be aware that it is not possible to directly call methods marked as public in your controller classes, as the framework intercepts them.

Thread safety

Some Java developers might want to scream in pain and agony now that "Static methods in a controller are not threadsafe!". However, the Controller is bytecode enhanced in order to make certain calls threadsafe, so the developer has not to worry about such issues. If you are interested in knowing more, you might want to check the class `play.classloading.enhancers.ControllerEnhancer`.

See also

Many recipes will change controller logic. Consider dealing with controllers which is absolute and essential core knowledge.

Defining your own models

As soon as you have to implement business logic or objects which should be persisted, the implementation should be done in the model. Note that the default implementation of this layer is implemented in Play with the use of JPA, Hibernate, and an SQL database in the background. However, you can of course implement an arbitrary persistence layer if you want.

Getting ready

Any model you define should go into the models package, which resides in the `app/models` directory.

How to do it...

As in the recipes before, this was already a reference to a user entity. It is the right time to create one now. Store this in the file `app/models/User.java`:

```
package models;

import javax.persistence.Entity;
import play.data.validation.Email;
import play.data.validation.Required;
import play.db.jpa.Model;
```

```
@Entity
public class User extends Model {

        public String login;

        @Required @Email
        public String email;
}
```

How it works...

Although there are not many lines of code, lots of things are included here. First, there are JPA annotations marking this class to be stored in a database. Second, there are validation annotations, which can be used to ensure which data should be in the object from an application point of view and not dependent on any database.

Remember: If you do as many tasks as possible such as validation in the application instead of the database it is always easier to scale. Annotations can be mixed up without problems.

The next crucially important point is the fact that the `User` class inherits from `Model`. This is absolutely essential, because it allows you to use the so-called `ActiveRecord` pattern for querying of data.

Also, by inheriting from the `Model` class you can use the `save()` method to persist the object to the database. However, you should always make sure you are importing the correct `Model` class, as there exists another `Model` class in the Play framework, which is an interface.

The last important thing which again will be mainly noticed by the Java developers is the fact, that all fields in the example code are public. Though the preceding code does not explicitly define getters and setters, they are injected at runtime into the class. This has two advantages. First, you as a developer do not have to write them, which means that your entity classes are very short and concise and not filled with setters and getters irrelevant to your logic. Second, if you really want to put logic into setters such as adding some complex check or changing the result before really storing it to the database, then it is possible without any problem. If you want you can also reuse your existing JPA entities, which are likely to have getters and setters. It is all a matter of choice. But the shorter your models are the more concise and easy to understand they will be.

There's more...

Now let's talk about some other options, or possibly some pieces of general information that are relevant to this task.

Using finders

Finders are used to query for existing data. They are a wonderful syntactic sugar on top of the Model entity. You can easily query for an attribute and get back a single object or a list of objects. For example:

```
User user = User.find("byName", name).).).first();
```

Or you can get a list of users with an e-mail beginning with a certain string:

```
List<User> users = User.find("byEmailLike", "alexander@%").fetch();
```

You can easily add pagination:

```
List<User> users = User.find("byEmailLike", "alexander@%")
                                          .from(20).fetch(10);
```

Or just add counting:

```
long results = User.count("byEmailLike", "alexander@%");
```

Never be anemic

Play has a generic infrastructure to support as many databases as possible. If you implement other persistence solutions, for example, JPA, GAE, or MongoDB, then always try to use the ActiveRecord pattern, because most of the Play developers will be used to it and it is very easy to grasp and understand. If you cannot do this for whatever reasons, like some completely different query language, then still do not use something like the DAO pattern in Play, as this is not natural for the framework and would pretty much break its flow. The anemic domain model—pulling logic from the object into data access objects—should be an absolute no-go when developing with Play.

Learning from the existing examples

Please check the Play examples and the Play documentation at `http://www.playframework.org/documentation/1.2/jpa` for an extensive introduction about models before reading further as this will be essential as well before going on with more complex topics. You will also find much more info about finders.

Regarding JPA and transactions

This is a short excursion into the JPA world but well worth it. Whenever you deal with databases you will be forced to implement some sort of commit/rollback transaction mechanism. As the standard Play persistence is based on JPA and Hibernate, the problem of course exists as well.

However, in order to simplify things, the HTTP request has been chosen as the transaction boundary. You should keep that in mind when having problems with data you thought should have been committed but is not persisted, because the request is not yet finished. A minor solution to this problem is to call `JPA.em().flush()`, which synchronizes changes to the database. If you want to make sure that you do not change data which has just been created in another request, you should read a Hibernate documentation about optimistic and pessimistic locking found at `http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/transactions.html`.

See also

For more information on the active record pattern you might want to check the Wikipedia entry `http://en.wikipedia.org/wiki/Active_record` or the more Ruby on Rails specific active record API at `http://ar.rubyonrails.org/`. There is also an active record implementation in pure Java at `http://code.google.com/p/activejdbc`.

There is a recipe for encrypting passwords before storing them on the database which makes use of creating an own setter.

Using fixtures to provide initial data

Fixtures are the Swiss Army knife of database independent seed data. By defining and describing your data entities in a text file it is pretty simple to load it into an arbitrary database.

This serves two purposes. First, you can make sure in your tests, that certain data exists when running the tests. Second, you can ensure that the must-have data like a first administrative account in your application exists, when deploying and starting your application in production.

How to do it...

Define a fixtures file and store it under `conf/initial-data.yml`:

```
User(alr):
        login: alr
        password: test
        email: alr@spinscale.de

Tweet(t1):
        content: Lets get ready to tweet
        postedAt: 2010-11-22T08:23:15
        login: alr
```

How it works...

As you can see in the preceding snippet, there are two entities defined. The first one only consists of strings, whereas the second one consists of a date and a reference to the first one, which uses the name in parentheses after the type as a reference.

There's more...

Fixtures are helpful in two cases. For one you can ensure the same test data in your unit, functional, and selenium tests. Also you can make sure, that your application is initialized with a certain set of data, when the application is loaded for the first time.

Using a bootstrap job to load seed data

If you need to initialize your application with some data, you can execute a job loading this data at application startup with the following code snippet:

```
@OnApplicationStart
public class Bootstrap extends Job {

    public void doJob() {
        // Check if the database is empty
        if(User.count() == 0) {
          Fixtures.load("initial-data.yml");
        }
    }
```

You should put the referenced `initial-data.yml` file into the `./conf` directory of your application. If you reference it with its filename only in any class like in the `doJob()` method that we saw some time back, it will be found and loaded in your current database by using the `count()` method of the User entity. Also by extending this class from `Job` and putting the `@OnApplicationStart` annotation at the top, the `doJob()` method is executed right at the start of the application.

More information about YAML

Play uses SnakeYAML as an internal YAML parser. You can find out more about the integration at either `http://www.playframework.org/documentation/1.2/yaml` or `http://code.google.com/p/snakeyaml/`.

Using lists in YAML

Fixtures are quite flexible, they also allow lists; for example, if the tags field is from type `List<Tag>`, this works:

```
tags:             [tag1, tag2, tag3, tag4]
```

Defining your own views

After getting a closer look at controllers and models, the missing piece is views. Views can essentially be anything: plain text, HTML, XML, JSON, vCard, binary data such as images, whatever you can imagine. Generally speaking, the templating component in Play is kept very simple. This has several advantages. First, you are not confronted with a new tag library, like you are in JSF with every new component. Second, every web developer will dig his way through this templating language quite fast. Third, the templating language is very nice and also very easy to extend. Even though the template language is based on Groovy and uses some Groovy expressions, there is absolutely no need to have any deep knowledge in Groovy. Even though you can use Groovy expressions, there is absolutely no need for it.

In this example, we will put together a small view showing our user entity.

How to do it...

The first step is to get the user inside the controller and allow it in the view to be used. Edit `app/controllers/Application.java` and change the `showUser()` method to this:

```
public static void showUser(Long id) {
        User user = User.findById(id);
        notFoundIfNull(user);
        render(user);
}
```

After that create an HTML template file in `./app/views/Application/showUser.html`:

```
#{extends 'main.html' /}
#{set title:'User info' /}

<h1>${user.login}</}</}</h1>

Send <a href="mailto:${user.email}">}">}">mail</a>
```

How it works...

Though again only a couple of lines have been written, a lot of things have happened. Beginning with the name and path of the template location, it is just the name of the class (including the package, which the controller does not have in this case) and its method name appended with a file format. Of course, you can change it, but this way it is pretty easy to understand which view code belongs to which controller action.

Regarding the controller logic all that has been done is to query the database for the user with a specific ID (the one specified in the URL) and to return a HTTP 404 error, if the returned object is null. This eliminates the nasty null checks from your code to keep it as clean as possible. The last part triggers the rendering. The argument handed over (you can choose an arbitrary amount of arguments) can be referenced in the HTML template under the name you put in the `render()` method. If you used `render(userObj)` you could reference it as `userObj` in the template.

The template contains lots of information in the four lines of code. First, Play template specific tags always use a `#{}` notation. Second, Play templates support some sort of inheritance with the `#{extends}` tag, as the `main.html` has been chosen here as a template into which the rest of the code is embedded. Third, you can set variables in this template, which are parsed in the `main.html` template, like the variable title, which is set in line two. Lastly you can easily output fields from the user object by writing the name of the object inside the template and its field name. As already done before, the field is not accessed directly, but the getter is called.

There's more...

Templating is covered fairly well in the documentation and in the example, so be sure to check it out.

Check out more possible template tags

There are more than two dozen predefined tags which can be used. Most of them are pretty simple, but still powerful. There is a special `#{a}` tag for creating links, which inserts real URLs from a controller action. There are of course `#{if}` structures and `#{list}` tags, form helper tags, i18n and JavaScripts helpers, as well as template inheriting tags and some more:

`http://www.playframework.org/documentation/1.2/tags`

Check out more predefined variables

There are some variables which are always defined inside a template, which help you to access data that are always needed without putting it explicitly into the render call. For example, `request`, `session`, `params`, `errors`, `out`, `messages`, `flash`, and `lang`. You can have a look at the documentation for more details:

`http://www.playframework.org/documentation/1.2/templates#implicits`

Supporting multiple formats

There are also more predefined `render()` methods with different output formats than HTML defined. Most known are `renderText()`, `renderXML()`, `renderJSON()`, and `renderBinary()` for images. Be aware that all of these methods do not use templates, but write out the result directly to the client without invoking any template specific code.

See also

It is very easy to write your own tags, so be sure to follow the next recipe as well as get some information about mixins, which allows you to define some more logic for displaying data without changing it in the model; for example, replacing the last digits with XXX for privacy issues.

Furthermore, a recipe with an own `renderRSS()` is shown as last recipe in *Chapter 2*, which is about controllers.

Writing your own tags

In order to keep repetitive tasks in your template short, you can easily define your own tags. As all you need to know is HTML and the built-in templating language, even pure web developers without backend knowledge can do this.

Getting ready

In this example, we will write a small tag called `#{loginStatus /}`, which will print the username or write a small note, that the user is not logged in. This is a standard snippet, which you might include in all of your pages, but do not want to write over again.

How to do it...

The following logic is assumed in the controller, here in `Application.java`:

```
public static void login(String login, String password) {
   User user = User.find("byLoginAndPassword", login, password).
     first();
   notFoundIfNull(user);
   session.put("login", user.login);
}
```

A new tag needs to be created in `app/views/tags/loginStatus.html`:

```
<div class="loginStatus">
#{if session.login}
Logged in as ${session.login}
#{/if}
#{else}
Not logged in
#{/else}
</div>
```

Using it in your own templates is now easy, just put the following in your templates:

```
#{loginStatus /}
```

How it works...

The controller introduces the concept of state in the web application by putting something in the session. The parameters of the login method have been (if not specified in routes file) constructed from the request parameters. In this case, from a request, which has most likely been a form submit. Upon calling the controller, the user is looked up in the database and the user's login name is stored in the session, which in turn is stored on the client side in an encrypted cookie.

Every HTML file in the `app/views/tags` directory is automatically used as a tag, which makes creating tags pretty simple. The tag itself is quite self explanatory, as it just checks whether the login property is set inside the session.

As a last word about sessions, please be aware that the session referenced in the code is actually not a HttpSession as in almost all other Java based frameworks. It is not an object stored on the server side, but rather its contents are stored as an encrypted cookie on the client side. This means you cannot store an arbitrary amount of data in it.

There's more...

You should use tags whenever possible instead of repeating template code. If you need more performance you can even write them in Java instead of using the templating language.

Using parameters and more inside tags

The preceding discussion was the absolute basic usage of tag. It can get somewhat more complex by using parameters or the same sort of inheritance, which is also possible with templates.

Check out `http://www.playframework.org/documentation/1.2/templates#tags` for more about this topic.

Higher rendering performance by using FastTags

If you need more performance, there is a mechanism called FastTags inside of Play. These tags are actually compiled, as they are written in pure Java. This speeds up execution time. Most of the native supported tags are FastTags in order to keep performance high. For example, take a look at your Play installation inside `samples-and-tests/just-test-cases/app/utils/MyTags.java` and you will see that these tags are also very simple to implement.

See also

Keep on reading the next recipe, where we will reformat a Date type from boring numbers to a string without using a tag, but a so-called extension.

Using Java Extensions to format data in your views

Java Extensions are a very nice helper inside your templates, which will help you to keep your template code as well as your model code clean from issues such as data formatting. Reformatting values such as dates is a standard problem at the view layer for most web developers. For example, the problem of having a date with millisecond exactness, though only the year should be printed. This is where these extensions start. Many web developers also do this by using JavaScript, but this often results in code duplication on frontend and backend.

This recipe shows a pretty common example, where a date needs to be formatted to show some relative date measured from the current time. This is very common in the Twitter timeline, where every Tweet in the web interface has no correct date, but merely a "n hours ago" or "n days ago" flag.

Getting ready

Just create a tiny application. You will need to create a new application and add a database to the application configuration, so entities can be specified.

How to do it...

You need a route to show your tweets in `conf/routes`:

```
GET      /{username}/timeline           Application.showTweet
```

After that we can model a tweet model class:

```
package models;


import java.util.Date;
import javax.persistence.Entity;
import play.data.validation.Max;
import play.db.jpa.Model;

@Entity
```

```
public class Tweet extends Model {

        @Max(140) public String content;
        public Date postedAt;
        public User user;
}
```

As well as a simple user entity:

```
@Entity
public class User extends Model {

        @Column(unique=true)
        public String login;
}
```

The controller is quite short. It uses an alternative query for the 20 newest tweets, which is more JPA like:

```
public static void showTweets(String username) {
        User user = User.find("byLogin", username).first();
        notFoundIfNull(user);
        List<Tweet> tweets = Tweet.find("user = ? order by postedAt
DESC", user).fetch(20);
        render(tweets, user);
}
```

The rendering code will look like this:

```
#{extends 'main.html' /}
#{set 'title'}${user.login} tweets#{/set}

#{list tweets, as:'tweet'}
<div><h3>${tweet.content}</h3> by ${tweet.user.login} at <i>${tweet.
postedAt.since()}</i></h3></div>
#{/list}
```

Now this code works. However, the `since()` Java Extension, which is built in with Play only works when you hand over a date in the past as it calculates the difference from now. What if you want to add a feature of a future tweet which is blurred, but will show a time when it is shown? You need to hack up your own extensions to do this. Create a new class called `CustomExtensions` in the extensions package inside your application directory (so the file is `./app/extensions/CustomExtension.java`)

```
public class CustomExtensions extends JavaExtensions {

        private static final long MIN   = 60;
```

```
        private static final long HOUR  = MIN * 60;
        private static final long DAY   = HOUR * 24;
        private static final long MONTH = DAY * 30;
        private static final long YEAR  = DAY * 365;

        public static String pretty(Date date) {
                Date now = new Date();
                if (date.after(now)) {
                   long delta = (date.getTime() - now.getTime()) /
1000;

                if (delta < 60) {
                    return Messages.get("in.seconds", delta,
 pluralize(delta));
                }

                if (delta < HOUR) {
                    long minutes = delta / MIN;
                    return Messages.get("in.minutes", minutes,
pluralize(minutes));
                }

                if (delta < DAY) {
                    long hours = delta / HOUR;
                    return Messages.get("in.hours", hours,
 pluralize(hours));
                }

                if (delta < MONTH) {
                    long days = delta / DAY;
                    return Messages.get("in.days", days,
pluralize(days));
                }

                if (delta < YEAR) {
                    long months = delta / MONTH;
                    return Messages.get("in.months", months,
 pluralize(months));
                }

                long years = delta / YEAR;
                return Messages.get("in.years", years,
pluralize(years));
```

```
                } else {
                        return JavaExtensions.since(date);
                }

        }
}
```

Update your `./app/conf/messages` file for successful internationalization by appending to it:

```
in.seconds = in %s second%s
in.minutes = in %s minute%s
in.hours   = in %s hour%s
in.days    = in %s day%s
in.months  = in %s month%s
in.years   = in %s year%s
```

The last change is to replace the template code to:

```
#{list tweets, as:'tweet'}
<div><h3>${tweet.content}</h3> by ${tweet.user.login} at <i>${tweet.
postedAt.pretty()}</i></h3></div>
#{/list}
```

How it works...

A lot of code has been written for an allegedly short example. The entity definitions, routes configuration, and controller code should by now be familiar to you. The only new thing is the call of `${tweet.postedAt.since()}` in the template, which does call a standard Java Extension already shipped with Play. When calling the `since()` method, you must make sure that you called it on an object from the `java.util.Date` class. Otherwise, this extension will not be found, as they are dependent on the type called on. What the `since()` method does, is to reformat the boring date to a pretty printed and internationalized string, how long ago this date is from the current time. However this functionality only works for dates in the past and not for future dates.

Therefore the `CustomExtensions` class has been created with the `pretty()` method in it. Every class which inherits from JavaExtensions automatically exposes its methods as extension in your templates. The most important part of the `pretty()` method is actually its signature. By marking the first parameter as type `java.util.Date` you define for which data type this method applies. The logic inside the method is pretty straightforward as it also reuses the code from the `since()` extension. The only unknown thing is the call to `Messages.get()`, which just returns the string in the correct language, such as"3 days ago" in English and "vor 3 Tagen" in German.

Last but not least, the template code is changed to use the new extension instead of `since()`.

There's more...

Java Extensions can be incredibly handy if used right. You should also make sure that this area of your application is properly documented, so frontend developers know what to search for, before trying to implement it somehow in the view layer.

Using parameters in extensions

It is pretty simple to use parameters as well, by extending the method with an arbitrary amount of parameters like this:

```
public static void pretty(Date date, String name) {
```

Using it in the template is as simple as `${tweet.postedAt.pretty("someStr")}`

Check for more built in Java Extensions

There are tons of useful helpers already built-in. Not only for dates, but also for currency formatting, numbers, strings, or list. Check it out at `http://www.playframework.org/documentation/1.2/javaextensions`.

Check for internationalization on plurals

Play has the great feature and possibility of definin a plural of internationalized strings, which is incidentally also defined in the built-in `JavaExtensions` class.

Adding modules to extend your application

Modules are the way to implement reusability in your application. Code which does not belong to your core functionality can be combined into a single module and also reused in other applications, or maybe even made open source. Furthermore, there are already quite a lot of modules in Play, and since the release of play 1.1, there is quite a rise of new modules every week. Using other modules is actually pretty easy and requires only one command and one configuration change to get it working.

Basically modules are Play applications themselves, so you are embedding another Play application into your own.

Getting ready

In this example the "search" module is going to be installed. It is a great module which allows you to integrate with Apache Lucene by just adding two annotations to your JPA models. From then on you can do lightning fast queries instead of using your database for full text search queries. However, if you want to install any module, you need to have an Internet connection for the first installation. Furthermore, this search module is used as an example in this case; there will be no implementation hits on using it. Please refer to the documentation for this.

How to do it...

Check whether the module is already installed. This should be executed in the directory of a Play application in order to return useful data:

```
play modules
```

Check whether the module you want to install is available:

```
play list-modules
```

Put this in your `conf/dependencies.yml` file:

```
require:
    - play
    - play -> search head
```

Then run `play deps`. After you have run and downloaded the module, you will have a `./modules/search-head` directory in your application, which gets automatically loaded on application startup.

When starting your application the next time you should see the following startup message:

```
10:58:48,825 INFO  ~ Module search is available (/path/to/app/modules/
search-head)
```

The next alternative possibility of installing modules is deprecated!

In case you are using an older version of Play than version 1.2, there is another mechanism to install a module, which needs further configuration. Make sure you are inside of the Play application where you want to install the module:

```
play install search
```

You are asked whether you are sure you want to install the module, because you need to check whether this module is compatible with the version of Play you are using. The installation tries to install the latest version of the module, but you can choose the module version in case you need an older one.

Follow the hint in the last line and put it into the `conf/application.conf` file:

```
module.search=${play.path}/modules/search-head
```

When starting your application the next time you should see the following startup message:

```
10:58:48,825 INFO  ~ Module search is available (/home/alex/devel/
play/play-1.1/modules/search-head)
```

From now on you can use the module and its functionality in your code.

How it works...

The steps are pretty straightforward as it is automated as much as possible. When calling Play install, everything is downloaded as a big package from the Web, unpacked in your Play installation (not your application) and from then on, ready to run in any Play web, once enabled in the configuration. The main difference between the old and new way of adding modules is the fact that the old mechanism stored the modules not in the application but in the framework directory, where as the new mechanism only stores modules inside of the application directory.

There's more...

Many modules require additional configuration in the `conf/application.conf` file. For example, if you install a module which persists your models in a MongoDB database, you will need to configure the database connection additionally. However, such cases are always documented, so just check the module documentation in case.

Also if modules do not work, first check whether they work in your version of Play. If this is the case, you should also file a bug report or inform the module maintainer. Many modules are not maintained by the core developers of Play, but instead by users of the Play framework.

Module documentation

As soon as you have added a new module and it includes documentation (most modules do), it will always be available in development mode under `http://localhost:9000/@documentation`.

Updating modules

There is currently no functionality to update your modules automatically. This is something you have to do manually. In order to keep it up-to-date you can either read the mailing list or alternatively just check the source repository of the module. This should always be listed in the module description page.

More on the search module

Go to `http://www.playframework.org/modules/search-head/home` for more information about this module.

See also

If you are interested in modules, check out the *Chapter 5*, *Introduction to Writing Modules* in this book. It also includes a recipe *Using Solr for indexing*, which takes up the search topic again.

Using Oracle or other databases with Play

This is just a quick recipe to show that any database with a JDBC driver can be used as persistence storage in Play, though it has been mainly developed with MySQL in mind. The most simple configuration of a database is to use the in memory H2 database by specifying `db=mem` in the `application.conf` file. You can ensure persistence by specifying `db=fs`, which also uses the H2 database. Both of these options are suitable for development mode as well as automated test running. However, in other cases you might want to use a real SQL database like MySQL or PostgreSQL.

How to do it...

Just add driver-specific configuration in your configuration file. In order to support PostgreSQL, this is the way:

```
db.url=jdbc:postgresql:accounting_db
db.driver=org.postgresql.Driver

db.user=acct
db.pass=Bdgc54S
```

Oracle can also be configured without problems:

```
db.url=jdbc:oracle:thin:@db01.your.host:1521:tst-db01

db.driver=oracle.jdbc.driver.OracleDriver
```

How it works...

As the JDBC mechanism already provides a generic way to unify the access to arbitrary databases, the complexity to configure different database is generally pretty low in Java. Play supports this by only needing to configure the `db.url` and `db.driver` configuration variables to have support for most databases, which provide a JDBC driver.

There's more...

You can even go further and configure your connection pool (if the JDBC driver has support for that), reusing application server resources, or check whether changing your JPA dialect is also needed when changing your default database.

Using application server datasources

It is also possible to use datasources provided by the underlying application server, just put the following line in your `config` file:

```
db=java:/comp/env/jdbc/myDatasource
```

Using connection pools

Connection pools are a very important feature to ensure a performant and resource saving link to the database from your application. This means saving resources by not creating a new TCP connection every time you issue a query. Most JDBC drivers come with this out of the box, but you can also tweak the settings in your `config` file:

```
# db.pool.timeout=1000
# db.pool.maxSize=30
# db.pool.minSize=10
```

Configuring your JPA dialect

It might also be necessary to configure your JPA dialect for certain databases. As Play uses hibernate, you need to specify a hibernate dialect:

```
jpa.dialect=org.hibernate.dialect.Oracle10gDialect
```

For more information about dialects, check out `http://docs.jboss.org/hibernate/core/3.3/reference/en/html/session-configuration.html#configuration-optional-dialects`.

Understanding suspendable requests

Most web frameworks bind an incoming HTTP request via the servlet to a thread. However, this means that each connection requires a thread, resulting in 300 concurrent connections needing 300 threads. Often, having that many threads does not scale when switching threads. This means that the computation time to manage threads and grant resources to them in a synchronized and fair mode increases with the amount of threads. Especially, when you consider the fact that most web connections are blocking and are waiting for something to happen in the background, like persisting an entity or executing another web request. This means that you can keep a small thread pool for accepting HTTP requests and have a bigger thread pool for executing business logic. Here comes the feature of continuations and suspendable requests into Play.

Getting ready

In order to have a simple test, you could create a small application which creates a big PDF report. Then access the URL mapped to the PDF report creation more often simultaneous than you have CPU cores. So you would have to request this resource three times at once on a duo core machine. You will see that a maximum two HTTP connections are executed simultaneously; in development mode it will be only one, regardless of your CPU count.

How to do it...

Play 1.2 introduces a new feature called continuations, which allows transparent suspension of threads including recovery without writing any additional code to do this:

```
public static void generateInvoice(Long orderId) {
    Order order = Order.findById(orderId);
    InputStream is = await(new OrderAsPdfJob(order).now());
    renderBinary(is);
}
```

Of course, the `OrderAsPdfJob` needs a signature like this:

```
public void OrderAsPdfJob extends Job<InputStream> {

    public InputStreamdoJobWithResult() {
        // logic goes here
    }
}
```

There is an alternative approach in play before version 1.2, which needed a little bit more core but still allowed asynchronous and non thread bound code execution.

You can suspend your logic for a certain amount of time like this:

```
public static void stockChanges() {
   List<Stock> stocks = Stock.find("date > ?", request.date).fetch();
   if (stocks.isEmpty()) {
       suspend("1s");
   }
renderJSON(stocks);
}
```

Alternatively, you can wait until a certain job has finished its business logic:

```
public static void generateInvoice(Long orderId) {
    if(request.isNew) {
        Order order = Order.findById(orderId);
        Future<InputStream> task = new OrderAsPdfJob(order).now();
        request.args.put("task", task);
        waitFor(task);
    }
    renderBinary((Future<InputStream>)request.args.get("task").get());
}
```

How it works...

Following the three lines of code in the first example, you see that there is actually no invocation telling the framework to suspend the thread. The `await()` method takes a so-called `Promise` as argument, which is returned by the `now()` method of the job. A `Promise` is basically a standard Java Future with added functionality for invocation inside of the framework, when the task is finished.

The `stockChanges()` example is pretty self explanatory as it waits the defined amount of time before it is called again. This means that the operation is only called again if there was no updated stock available and it is very important it is called again from the beginning. Otherwise it will happily render the JSON output and has to be triggered by the client again. As you can see, this would be a pretty interesting starting point for implementing SLAs for your customers in a stock rate application, as you could allow your premium customers quicker updates.

The second example takes another approach. The controller logic is actually run twice. In the first run, the `isNew` parameter is true and starts a Play job to create the PDF of an invoice. This parameter is automatically set by the framework depending on the status of the request and gives the developer the possibility to decide what should happen next. The `waitFor()` tells the framework to suspend here. Again, after the task is finished, the whole controller method will be called again, but this time only the `renderBinary()` method is called as `isNew` is false, which returns the result by calling `get()` on the `Future` type.

There's more...

Always think whether introducing such a suspended response is really what you want or whether this just shows certain problems in your application design. Most of the time such a mechanism is not needed.

More about promises

Promises are documented in the javadoc at `http://www.playframework.org/documentation/api/1.2/index.html?play/libs/F.Promise.html` as well as in the play 1.2 release notes at `http://www.playframework.org/documentation/1.2/releasenotes-1.2#Promises`. There are even better features like waiting for the end of a list of promises or even waiting for only one result of a list of promises.

More about jobs

The job mechanism inside a Play is used to execute any business logic either on application startup or on regular intervals and has not been covered yet. It is however pretty well documented in the public documentation at `http://www.playframework.org/documentation/1.2/jobs`.

More information about execution times

In order to find out whether parts of your business logic need such a suspendable mechanism, use `playstatus` in your production application. You can check how long each controller execution took in average and examine bottlenecks.

See also

The recipe *Integration with Munin* in *Chapter 7* shows how to monitor your controller execution times in order to make sure you are suspending the right requests.

Understanding session management

Whenever you read about Play, one of the first advantages you will hear is that it is stateless. But what does this mean actually? Does it mean you do not have a session object, which can be used to store data while a visitor is on your website? No, but you have to rethink the way sessions are used.

Usually a session in a servlet-based web application is stored on a server side. This means, every new web request is either matched to a session or a new one is created. This used to happen in memory, but can also be configured to be written on disk in order to be able to restart the servlet container without losing session data. In any scenario there will be resources used on the server side to store data which belongs to a client.

Imagine a server environment with two servers receiving all HTTP connections. After the session has been created on server A, some logic has to redirect the user with this session always to server A, because otherwise a new session would have to be created on server B without taking over the session data from server A. Alternatively, the sessions have to be shared somehow. Both solutions require additional logic somewhere.

Play goes the way of sharing the session, but in a slightly different way. First, the real session used to identify the client is stored as a Cookie on the client. This cookie is encrypted and cannot be tampered with. You can store data in this cookie; however, the maximum cookie size is only 4KB. Imagine you want to store big data in this session, like a very large shopping cart or a rendered graphic. This would not work.

Play has another mechanism to store big data, basically a dumb cache. Caches are good at storing temporary data as efficient and fast accessible as possible. Furthermore, this allows you to have a scaling caching server, as your application scales. The maximum session size is 4KB. If you need to store more data, just use a cache.

How to do it...

Use the session object inside the controller to write something into it. This is a standard task during a login:

```
public static void login(String login, String password) {
    User user = User.find("byLoginAndPassword", login, password).
first();
    notFoundIfNull(user);
    session.put("login", user.login);
    index();
}
```

The session variable can now be accessed from any other controller method as long as it is not deleted. This works for small content, like a login:

```
String login = session.get("login");
```

Now, you can also use the built-in cache functionality instead of the session to store data on the server side. The cache allows you to put more data than the session maximum of 4 kilobytes into the cache (for the sake of having a lot of data assume that you are a subcontractor of Santa Claus, responsible for the EMEA region and constantly filling your shopping cart without checking out):

```
Cache.set(login, shoppingCart, "20mn");
```

Querying is as easy as calling:

```
Cache.get(login)
```

How it works...

Adding data to the session is as easy as using a regular session object. However, there is no warning if there is data put into the session, which is bigger than the maximum allowed cookie size. Unfortunately, the application will just break when getting the data out of the cookie, as it is not stored in the cookie, and the `session.get()` call will always fail.

In order to avoid this problem, just use the `Cache` class for storing such data. You can also add a date when the data should expire out of the cache.

There's more...

Caching is a very powerful weapon in the fight for performance. However, you always gain performance at the cost of reducing the actuality of your data. Always decide what is more important. If you can keep your data up-to-date by scaling out and adding more machines, this might be more useful in some cases, than caching it. As easy as caching is, it should always be the last resort.

Configuring different cache types

If you have a setup with several Play nodes, there is a problem if every instance uses its own cache, as this can lead to data inconsistency among the nodes. Therefore, Play comes with support to offload cache data to memcached instead of using the built-in Java-based EhCache. You will not have to change any of your application code to change to memcached. The only thing to change is the configuration file:

```
memcached=enabled
memcached.host=127.0.0.1:11211
```

Using the cache to offload database load

You can store arbitrary data in your cache (as long as it is serializable). This offers you the possibility to store queries to your persistence engine in the cache. If 80 percent of your website visits only hit the first page of your application, where the 10 most recent articles are listed, it makes absolute sense to cache them for a minute or 30 seconds. However, you should check whether it is really necessary as many databases are optimizing for this case already; please check your implementation for that.

See also

There is a recipe on writing your own cache implementation in *Chapter 4* called *Writing your own cache implementation*, where Hazelcast is used to store data. In *Chapter 3* there is also a general recipe *Basics of caching* about caching different aspects of your application.

2
Using Controllers

In this chapter, we will cover:

- URL routing using annotation-based configuration
- Basics of caching
- Using HTTP digest authentication
- Generating PDFs in your controllers
- Binding objects using custom binders
- Validating objects using annotations
- Adding annotation-based right checks to your controller
- Rendering JSON output
- Writing your own renderRSS method as controller output

Introduction

This chapter will help you to keep your controllers as clean as possible, with a well defined boundary to your model classes. Always remember that controllers are really only a thin layer to ensure that your data from the outside world is valid before handing it over to your models, or something needs to be specifically adapted to HTTP. The chapter will start with some basic recipes, but it will cover some more complex topics later on with quite a bit code, of course mostly explained with examples.

URL routing using annotation-based configuration

If you do not like the routes file, you can also describe your routes programmatically by adding annotations to your controllers. This has the advantage of not having any additional `config` file, but also poses the problem of your URLs being dispersed in your code.

You can find the source code of this example in the `examples/chapter2/annotation-controller` directory.

How to do it...

Go to your project and install the router module via `conf/dependencies.yml`:

```
require:
    - play
    - play -> router head
```

Then run `playdeps` and the router module should be installed in the `modules/` directory of your application. Change your controller like this:

```
@StaticRoutes({
        @ServeStatic(value="/public/", directory="public")
})
public class Application extends Controller {

    @Any(value="/", priority=100)
    public static void index() {
        forbidden("Reserved for administrator");
    }

    @Put(value="/", priority=2, accept="application/json")
    public static void hiddenIndex() {
        renderText("Secret news here");
    }

    @Post("/ticket")
    public static void getTicket(String username, String password) {
        String uuid = UUID.randomUUID().toString();
        renderJSON(uuid);
    }
}
```

How it works...

Installing and enabling the module should not leave any open questions for you at this point. As you can see in the controller, it is now filled with annotations that resemble the entries in the `routes.conf` file, which you could possibly have deleted by now for this example. However, then your application will not start, so you have to have an empty file at least.

The `@ServeStatic` annotation replaces the static command in the routes file. The `@StaticRoutes` annotation is just used for grouping several `@ServeStatic` annotations and could be left out in this example.

Each controller call now has to have an annotation in order to be reachable. The name of the annotation is the HTTP method, or `@Any`, if it should match all HTTP methods. Its only mandatory parameter is the value, which resembles the URI—the second field in the `routes.conf`. All other parameters are optional. Especially interesting is the `priority` parameter, which can be used to give certain methods precedence. This allows a lower prioritized catch-all controller like in the preceding example, but a special handling is required if the URI is called with the PUT method. You can easily check the correct behavior by using curl, a very practical command line HTTP client:

```
curl -v localhost:9000/
```

This command should give you a result similar to this:

```
> GET / HTTP/1.1
> User-Agent: curl/7.21.0 (i686-pc-linux-gnu) libcurl/7.21.0
OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
> Host: localhost:9000
> Accept: */*
>

< HTTP/1.1 403 Forbidden
< Server: Play! Framework;1.1;dev
< Content-Type: text/html; charset=utf-8
< Set-Cookie: PLAY_FLASH=;Path=/
< Set-Cookie: PLAY_ERRORS=;Path=/
< Set-Cookie: PLAY_SESSION=0c7df945a5375480993f51914804284a3bb
ca726-%00___ID%3A70963572-b0fc-4c8c-b8d5-871cb842c5a2%00;Path=/
< Cache-Control: no-cache
< Content-Length: 32
<

<h1>Reserved for administrator</h1>
```

You can see the HTTP error message and the content returned. You can trigger a PUT request in a similar fashion:

```
curl -X PUT -v localhost:9000/

> PUT / HTTP/1.1
> User-Agent: curl/7.21.0 (i686-pc-linux-gnu) libcurl/7.21.0
OpenSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
> Host: localhost:9000
> Accept: */*
>

< HTTP/1.1 200 OK
< Server: Play! Framework;1.1;dev
< Content-Type: text/plain; charset=utf-8
< Set-Cookie: PLAY_FLASH=;Path=/
< Set-Cookie: PLAY_ERRORS=;Path=/
< Set-Cookie: PLAY_SESSION=f0cb6762afa7c860dde3fe1907e8847347
6e2564-%00___ID%3A6cc88736-20bb-43c1-9d43-42af47728132%00;Path=/
< Cache-Control: no-cache
< Content-Length: 16

Secret news here
```

As you can see now, the priority has voted the controller method for the PUT method which is chosen and returned.

There's more...

The router module is a small, but handy module, which is perfectly suited to take a first look at modules and to understand how the routing mechanism of the Play framework works at its core. You should take a look at the source if you need to implement custom mechanisms of URL routing.

Mixing the configuration file and annotations is possible

You can use the router module and the routes file—this is needed when using modules as they cannot be specified in annotations. However, keep in mind that this is pretty confusing. You can check out more info about the router module at `http://www.playframework.org/modules/router`.

Basics of caching

Caching is quite a complex and multi-faceted technique, when implemented correctly. However, implementing caching in your application should not be complex, but rather the mindwork before, where you think about what and when to cache, should be. There are many different aspects, layers, and types (and their combinations) of caching in any web application. This recipe will give a short overview about the different types of caching and how to use them.

You can find the source code of this example in the `chapter2/caching-general` directory.

Getting ready

First, it is important that you understand where caching can happen—inside and outside of your Play application. So let's start by looking at the caching possibilities of the HTTP protocol. HTTP sometimes looks like a simple protocol, but is tricky in the details. However, it is one of the most proven protocols in the Internet, and thus it is always useful to rely on its functionalities.

HTTP allows the caching of contents by setting specific headers in the response. There are several headers which can be set:

- **Cache-Control**: This is a header which must be parsed and used by the client and also all the proxies in between.
- **Last-Modified**: This adds a timestamp, explaining when the requested resource had been changed the last time. On the next request the client may send an If-Modified-Since header with this date. Now the server may just return a HTTP 304 code without sending any data back.
- **ETag**: An ETag is basically the same as a Last-Modified header, except it has a semantic meaning. It is actually a calculated hash value resembling the resource behind the requested URL instead of a timestamp. This means the server can decide when a resource has changed and when it has not. This could also be used for some type of optimistic locking.

So, this is a type of caching on which the requesting client has some influence on. There are also other forms of caching which are purely on the server side. In most other Java web frameworks, the HttpSession object is a classic example, which belongs to this case.

Play has a cache mechanism on the server side. It should be used to store big session data, in this case any data exceeding the 4KB maximum cookie size. Be aware that there is a semantic difference between a cache and a session. You should not rely on the data being in the cache and thus need to handle cache misses.

You can use the Cache class in your controller and model code. The great thing about it is that it is an abstraction of a concrete cache implementation. If you only use one node for your application, you can use the built-in ehCache for caching. As soon as your application needs more than one node, you can configure a memcached in your `application.conf` and there is no need to change any of your code.

Furthermore, you can also cache snippets of your templates. For example, there is no need to reload the portal page of a user on every request when you can cache it for 10 minutes.

This also leads to a very simple truth. Caching gives you a lot of speed and might even lower your database load in some cases, but it is not free. Caching means you need RAM, lots of RAM in most cases. So make sure the system you are caching on never needs to swap, otherwise you could read the data from disk anyway. This can be a special problem in cloud deployments, as there are often limitations on available RAM.

The following examples show how to utilize the different caching techniques. We will show four different use cases of caching in the accompanying test. First test:

```
public class CachingTest extends FunctionalTest {

    @Test
    public void testThatCachingPagePartsWork() {
        Response response = GET("/");
        String cachedTime = getCachedTime(response);
        assertEquals(getUncachedTime(response), cachedTime);

        response = GET("/");
        String newCachedTime = getCachedTime(response);
        assertNotSame(getUncachedTime(response), newCachedTime);
        assertEquals(cachedTime, newCachedTime);
    }

    @Test
    public void testThatCachingWholePageWorks() throws Exception {
        Response response = GET("/cacheFor");
        String content = getContent(response);
        response = GET("/cacheFor");
        assertEquals(content, getContent(response));
        Thread.sleep(6000);
        response = GET("/cacheFor");
        assertNotSame(content, getContent(response));
    }

    @Test
    public void testThatCachingHeadersAreSet() {
```

```
        Response response = GET("/proxyCache");
        assertIsOk(response);
        assertHeaderEquals("Cache-Control", "max-age=3600", response);
    }

    @Test
    public void testThatEtagCachingWorks() {
        Response response = GET("/etagCache/123");
        assertIsOk(response);
        assertContentEquals("Learn to use etags, dumbass!", response);

        Request request = newRequest();

        String etag = String.valueOf("123".hashCode());
        Header noneMatchHeader =  new Header("if-none-match", etag);
        request.headers.put("if-none-match", noneMatchHeader);

        DateTime ago = new DateTime().minusHours(12);
        String agoStr = Utils.getHttpDateFormatter().format(ago.
          toDate());
        Header modifiedHeader = new Header("if-modified-since",
          agoStr);
        request.headers.put("if-modified-since", modifiedHeader);

        response = GET(request, "/etagCache/123");
        assertStatus(304, response);
    }


    private String getUncachedTime(Response response) {
        return getTime(response, 0);
    }

    private String getCachedTime(Response response) {
        return getTime(response, 1);
    }

    private String getTime(Response response, intpos) {
        assertIsOk(response);
        String content = getContent(response);
        return content.split("\n")[pos];
    }
}
```

The first test checks for a very nice feature. Since play 1.1, you can cache parts of a page, more exactly, parts of a template. This test opens a URL and the page returns the current date and the date of such a cached template part, which is cached for about 10 seconds. In the first request, when the cache is empty, both dates are equal. If you repeat the request, the first date is actual while the second date is the cached one.

The second test puts the whole response in the cache for 5 seconds. In order to ensure that expiration works as well, this test waits for six seconds and retries the request.

The third test ensures that the correct headers for proxy-based caching are set.

The fourth test uses an HTTP ETag for caching. If the `If-Modified-Since` and `If-None-Match` headers are not supplied, it returns a string. On adding these headers to the correct ETag (in this case the `hashCode` from the string `123`) and the date from 12 hours before, a 302 Not-Modified response should be returned.

How to do it...

Add four simple routes to the configuration as shown in the following code:

```
GET     /                   Application.index
GET     /cacheFor           Application.indexCacheFor
GET     /proxyCache         Application.proxyCache
GET     /etagCache/{name}   Application.etagCache
```

The application class features the following controllers:

```
public class Application extends Controller {

    public static void index() {
        Date date = new Date();
        render(date);
    }

    @CacheFor("5s")
    public static void indexCacheFor() {
        Date date = new Date();
        renderText("Current time is: " + date);
    }

    public static void proxyCache() {
        response.cacheFor("1h");
        renderText("Foo");
    }
```

```
    @Inject
    private static EtagCacheCalculator calculator;

    public static void etagCache(String name) {
        Date lastModified = new DateTime().minusDays(1).toDate();
        String etag = calculator.calculate(name);
        if(!request.isModified(etag, lastModified.getTime())) {
            throw new NotModified();
        }
        response.cacheFor(etag, "3h", lastModified.getTime());
        renderText("Learn to use etags, dumbass!");
    }
}
```

As you can see in the controller, the class to calculate ETags is injected into the controller. This is done on startup with a small job as shown in the following code:

```
@OnApplicationStart
public class InjectionJob extends Job implements BeanSource {

    private Map<Class, Object>clazzMap = new HashMap<Class, Object>();

    public void doJob() {
        clazzMap.put(EtagCacheCalculator.class, new
          EtagCacheCalculator());
        Injector.inject(this);
    }

    public <T> T getBeanOfType(Class<T>clazz) {
        return (T) clazzMap.get(clazz);
    }
}
```

The calculator itself is as simple as possible:

```
public class EtagCacheCalculator implements ControllerSupport {

    public String calculate(String str) {
        return String.valueOf(str.hashCode());
    }
}
```

The last piece needed is the template of the `index()` controller, which looks like this:

```
Current time is: ${date}
#{cache 'mainPage', for:'5s'}
Current time is: ${date}
#{/cache}
```

How it works...

Let's check the functionality per controller call. The `index()` controller has no special treatment inside the controller. The current date is put into the template and that's it. However, the caching logic is in the template here because not the whole, but only a part of the returned data should be cached, and for that a `#{cache}` tag used. The tag requires two arguments to be passed. The `for` parameter allows you to set the expiry out of the cache, while the first parameter defines the key used inside the cache. This allows pretty interesting things. Whenever you are in a page where something is exclusively rendered for a user (like his portal entry page), you could cache it with a key, which includes the user name or the session ID, like this:

```
#{cache 'home-' + connectedUser.email, for:'15min'}
${user.name}
#{/cache}
```

This kind of caching is completely transparent to the user, as it exclusively happens on the server side. The same applies for the `indexCacheFor()` controller. Here, the whole page gets cached instead of parts inside the template. This is a pretty good fit for non-personalized, high performance delivery of pages, which often are only a very small portion of your application. However, you already have to think about caching before. If you do a time consuming JPA calculation, and then reuse the cache result in the template, you have still wasted CPU cycles and just saved some rendering time.

The third controller call `proxyCache()` is actually the most simple of all. It just sets the proxy expire header called `Cache-Control`. It is optional to set this in your code, because your Play is configured to set it as well when the `http.cacheControl` parameter in your `application.conf` is set. Be aware that this works only in production, and not in development mode.

The most complex controller is the last one. The first action is to find out the last modified date of the data you want to return. In this case it is 24 hours ago. Then the ETag needs to be created somehow. In this case, the calculator gets a String passed. In a real-world application you would more likely pass the entity and the service would extract some properties of it, which are used to calculate the ETag by using a pretty-much collision-safe hash algorithm. After both values have been calculated, you can check in the request whether the client needs to get new data or may use the old data. This is what happens in the `request.isModified()` method.

If the client either did not send all required headers or an older timestamp was used, real data is returned; in this case, a simple string advising you to use an ETag the next time. Furthermore, the calculated ETag and a maximum expiry time are also added to the response via `response.cacheFor()`.

A last specialty in the `etagCache()` controller is the use of the `EtagCacheCalculator`. The implementation does not matter in this case, except that it must implement the `ControllerSupport` interface. However, the initialization of the injected class is still worth a mention. If you take a look at the `InjectionJob` class, you will see the creation of the class in the `doJob()` method on startup, where it is put into a local map. Also, the `Injector.inject()` call does the magic of injecting the `EtagCacheCalculator` instance into the controllers. As a result of implementing the `BeanSource` interface, the `getBeanOfType()` method tries to get the corresponding class out of the map. The map actually should ensure that only one instance of this class exists.

There's more...

Caching is deeply integrated into the Play framework as it is built with the HTTP protocol in mind. If you want to find out more about it, you will have to examine core classes of the framework.

More information in the ActionInvoker

If you want to know more details about how the `@CacheFor` annotation works in Play, you should take a look at the `ActionInvoker` class inside of it.

Be thoughtful with ETag calculation

Etag calculation is costly, especially if you are calculating more then the last-modified stamp. You should think about performance here. Perhaps it would be useful to calculate the ETag after saving the entity and storing it directly at the entity in the database. It is useful to make some tests if you are using the ETag to ensure high performance. In case you want to know more about ETag functionality, you should read RFC 2616.

You can also disable the creation of ETags totally, if you set `http.useETag=false` in your `application.conf`.

Use a plugin instead of a job

The job that implements the `BeanSource` interface is not a very clean solution to the problem of calling `Injector.inject()` on start up of an application. It would be better to use a plugin in this case.

See also

The cache in Play is quite versatile and should be used as such. We will see more about it in all the recipes in this chapter. However, none of this will be implemented as a module, as it should be. This will be shown in *Chapter 6*, *Practical Module Examples*.

Using HTTP digest authentication

As support for HTTP, basic authentication is already built-in with Play. You can easily access `request.user` and `request.password` in your controller as using digest authentication is a little bit more complex. To be fair, the whole digest authentication is way more complex.

You can find the source code of this example in the `chapter2/digest-auth` directory.

Getting ready

Understanding HTTP authentication in general is quite useful, in order to grasp what is done in this recipe. For every HTTP request the client wants to receive a resource by calling a certain URL. The server checks this request and decides whether it should return either the content or an error code and message telling the client to provide needed authentication. Now the client can re-request the URL using the correct credentials and get its content or just do nothing at all.

When using HTTP basic authentication, the client basically just sends some user/password combination with its request and hopes it is correct. The main problem of this approach is the possibility to easily strip the username and password from the request, as there are no protection measures for basic authentication. Most people switch to an SSL-encrypted connection in this case in order to mitigate this problem. While this is perfectly valid (and often needed because of transferring sensitive data), another option is to use HTTP digest authentication. Of course digest authentication does not mean that you cannot use SSL. If all you are worrying about is your password and not the data you are transmitting, digest authentication is just another option.

In basic authentication the user/password combination is sent in almost cleartext over the wire. This means the password does not need to be stored as cleartext on the server side, because it is a case of just comparing the hash value of the password by using MD5 or SHA1. When using digest authentication, only a hash value is sent from client to server. This implies that the client and the server need to store the password in cleartext in order to compute the hash on both sides.

How to do it...

Create a user entity with these fields:

```
@Entity
public class User extends Model {

        public String name;
        public String password;      // hashed password
        public String apiPassword; // cleartext password
}
```

Create a controller which has a `@Before` annotation:

```
public class Application extends Controller {

    @Before
    static void checkDigestAuth() {
        if (!DigestRequest.isAuthorized(request)) {
            throw new UnauthorizedDigest("Super Secret Stuff");
        }
    }

    public static void index() {
        renderText("The date is " + new Date());
    }
}
```

The controller throws an `UnauthorizedDigest` exception, which looks like this:

```
public class UnauthorizedDigest extends Result {

    String realm;

    public UnauthorizedDigest(String realm) {
        this.realm = realm;
    }

    @Override
    public void apply(Request request, Response response) {
        response.status = Http.StatusCode.UNAUTHORIZED;
        String auth = "Digest realm=" + realm + ", nonce=" +
           Codec.UUID();
        response.setHeader("WWW-Authenticate", auth);
    }
}
```

The digest request handles the request and checks the authentication:

```
class DigestRequest {

    private Map<String,String>params = new HashMap<String,String>();
    private Request request;

    public DigestRequest(Request request) {
        this.request = request;
    }
```

```
    public booleanisValid() {
...
    }

    public booleanisAuthorized() {
        User user = User.find("byName", params.get("username")).
                    first();
        if (user == null) {
            throw new UnauthorizedDigest(params.get("realm"));
        }

        String digest = createDigest(user.apiPassword);
        return digest.equals(params.get("response"));
    }

    private String createDigest(String pass) {
...
    }

    public static booleanisAuthorized(Http.Request request) {
        DigestRequest req = new DigestRequest(request);
        return req.isValid() && req.isAuthorized();
    }
}
```

How it works...

As you can see, all it takes is four classes. The user entity should be pretty clear, as it only exposes three fields, one being a login and two being passwords. This is just to ensure that you should never store a user's master password in cleartext, but use additional passwords if you implement some cleartext password dependant application.

The next step is a controller, which returns a HTTP 403 with the additional information requiring HTTP digest authentication. The method annotated with the `Before` annotation is always executed before any controller method as this is the perfect place to check for authentication. The code checks whether the request is a valid authenticated request. If this is not the case an exception is thrown. In Play, every `Exception` which extends from `Result` actually can return the request or the response.

Taking a look at the `UnauthorizedDigest` class you will notice that it only changes the HTTP return code and adds the appropriate WWW-Authenticate header. The WWW-Authenticate header differs from the one used with basic authentication in two ways. First it marks the authentication as "Digest", but it also adds a so-called nonce, which should be some random string. This string must be used by the client to create the hash and prevents bruteforce attacks by never sending the same `user/password/nonce` hash combination.

The heart of this recipe is the `DigestRequest` class, which actually checks the request for validity and also checks whether the user is allowed to authenticate with the credentials provided or not. Before digging deeper, it is very useful to try the application using `curl` and observing what the headers look like. Call `curl` with the following parameters:

```
curl --digest --user alex:test -v localhost:9000
```

The response looks like the following (unimportant output and headers have been stripped):

```
> GET / HTTP/1.1
> Host: localhost:9000
> Accept: */*
>

< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Digest realm=Super Secret Stuff, nonce=3ef81305-
745c-40b9-97d0-1c601fe262ab
< Content-Length: 0
<
* Connection #0 to host localhost left intact
* Issue another request to this URL: 'HTTP://localhost:9000'

> GET / HTTP/1.1
> Authorization: Digest username="alex", realm="Super Secret Stuff",
nonce="3ef81305-745c-40b9-97d0-1c601fe262ab", uri="/", response="6e97a
12828d940c7dc1ff24dad167d1f"
> Host: localhost:9000
> Accept: */*
>

< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 20
<
This is top secret!
```

Curl actually issues two requests. The first returns a 403 "not authorized" error, but also the nonce, which is used together with the username and password in the second request to create the response field inside the WWW-Authenticate header. As the client also sends the nonce and the username inside the header, the server can reconstruct the whole response on the server side. This means it is actually stateless and the server does not need to store any data between two requests.

Looking at the `DigestRequest` class, it is comprised of three core methods: `isValid()`, `isAuthorized()`, and `createDigest()`. The `isValid()` method checks whether a request contains all the needed data in order to be able to compute and compare the hash. The `isAuthorized()` method does a database lookup of the user's cleartext password and hands it over to the `createDigest` method, which computes the response hash and returns true if the computed hash with the local password is the same as the hash sent in the request. If they are not, the authentication has to fail.

The static `DigestRequest.isAuthorized()` method is a convenient method to keep the code in the controller as short as possible.

There are two fundamental disadvantages in the preceding code snippet. First, it is implementation dependent, because it directly relies on the user entity and the password field of this entity. This is not generic and has to be adapted for each implementation. Secondly, it only implements the absolute minimum subset of HTTP digest authentication. Digest authentication is quite complex if you want to support it with all its variations and options. There are many more options and authentication options, hashing algorithms, and optional fields which have to be supported in order to be RFC-compliant. You should see this only as a minimum starting point to get this going. Also this should not be thought of as secure, because without an additional header called "qop", every client will switch to a less secure mode. You can read more about that in RFC2069 and RFC2617.

There's more...

You can also verify this recipe in your browser by just pointing it to `http://localhost:9000/`. An authentication window requiring you to enter username and password will popup.

Get more info about HTTP digest authentication

As this recipe has not even covered five percent of the specification, you should definitely read the corresponding RFC at `http://tools.ietf.org/html/rfc2617` as well as RFC2069 at `http://tools.ietf.org/html/rfc2617`.

See also

In order to be application-independent you could use annotations to mark the field of the entity to be checked. The recipe *Rendering JSON output* will show you how to use an annotation to mark a field not to be exported via JSON.

Generating PDFs in your controllers

Generating binary content is a standard procedure in every web application, be it a dynamic generated image such as a CAPTCHA or user-specific document such as an invoice or an order confirmation. Play already supports the `renderBinary()` command in the controller to send binary data to the browser, however this is quite low level. This recipe shows how to combine the use of Apache FOP – which allows creation of PDF data out of XML-based templates – and the Play built-in templating mechanism to create customized PDF documents in real time.

You can find the source code of this example in the `chapter2/pdf` directory.

Getting ready

As there is already a PDF module included in Play, you should make sure you disable it in your application in order to avoid clashes. This of course only applies, if it has already been enabled before.

How to do it...

First you should download Apache FOP from `http://www.apache.org/dyn/closer.cgi/xmlgraphics/fop` and unpack it into your application. Get the ZIP file and unzip it so that there is a `fop-1.0` directory in your application depending on your downloaded version.

Now you have to copy the JAR files into the `lib/` directory, which is always included in the classpath when your application starts.

```
cp fop-1.0/build/fop.jar lib/
cp fop-1.0/lib/*.jar lib/
cp fop-1.0/examples/fo/basic/simple.fo app/views/Application/index.fo
rm lib/commons*
```

Make sure to remove the commons JAR files from the `lib` directory, as Play already provides them. In case of using Windows, you would have to use `copy` and `del` as commands instead of the Unix commands `cp` and `rm`. Instead of copying these files manually you could also add the entry to `conf/dependencies.yml`. However, you would have to exclude many dependencies manually, which can be removed as well.

Create a dummy User model, which is rendered in the PDF:

```
public class User {

        public String name = "Alexander";
        public String description = "Random: " +
RandomStringUtils.randomAlphanumeric(20);
}
```

You should now replace the content of the freshly copied `app/views/Application/index.fo` file to resemble something from the user data like you would do it in a standard HTML template file in Play:

```
<fo:block font-size="18pt"
            ...
            padding-top="3pt">
    ${user.name}
</fo:block>

<fo:block font-size="12pt"
            ...
            text-align="justify">
    ${user.description}
</fo:block>
```

Change the application controller to call `renderPDF()` instead of `render()`:

```
import static pdf.RenderPDF.renderPDF;

public class Application extends Controller {

    public static void index() {
        User user = new User();
        renderPDF(user);
    }
}
```

Now the only class that needs to be implemented is the `RenderPDF` class in the PDF package:

```
public class RenderPDF extends Result {

        private static FopFactoryfopFactory = FopFactory.
         newInstance();
        private static TransformerFactorytFactory =
         TransformerFactory.newInstance();
        private VirtualFiletemplateFile;

        public static void renderPDF(Object... args) {
            throw new RenderPDF(args);
        }

        public RenderPDF(Object ... args) {
            populateRenderArgs(args);
```

```
            templateFile = getTemplateFile(args);
        }

        @Override
        public void apply(Request request, Response response) {
            Template template = TemplateLoader.load(templateFile);

            String header = "inline; filename=\"" + request.
actionMethod + ".pdf\"";
            response.setHeader("Content-Disposition", header);

            setContentTypeIfNotSet(response, "application/pdf");

            try {
                    Fop fop = fopFactory.newFop(MimeConstants.MIME_PDF,
response.out);
                    Transformer transformer = tFactory.
newTransformer();
                    Scope.RenderArgsargs = Scope.RenderArgs.current();
                    String content = template.render(args.data);
                    InputStream is = IOUtils.toInputStream(content);
                    Source src = new StreamSource(is);
                    javax.xml.transform.Result res = new SAXResult(fop.
getDefaultHandler());
                    transformer.transform(src, res);
            } catch (FOPException e) {
                    Logger.error(e, "Error creating pdf");
            } catch (TransformerException e) {
                    Logger.error(e, "Error creating pdf");
            }
        }

        private void populateRenderArgs(Object ... args) {
            Scope.RenderArgsrenderArgs = Scope.RenderArgs.current();
            for (Object o : args) {
                    List<String> names = LocalVariablesNamesTracer.
getAllLocalVariableNames(o);
                    for (String name : names) {
                        renderArgs.put(name, o);
                    }
            }
            renderArgs.put("session", Scope.Session.current());
```

```
            renderArgs.put("request", Http.Request.current());
            renderArgs.put("flash", Scope.Flash.current());
            renderArgs.put("params", Scope.Params.current());
            renderArgs.put("errors", Validation.errors());
        }

        private VirtualFilegetTemplateFile(Object ... args) {
            final Http.Request request = Http.Request.current();
            String templateName = null;
            List<String>renderNames = LocalVariablesNamesTracer.getAll
LocalVariableNames(args[0]);
            if (args.length> 0 &&args[0] instanceof String
&&renderNames.isEmpty()) {
                templateName = args[0].toString();
            } else {
                templateName = request.action.replace(".", "/") +
".fo";
            }

            if (templateName.startsWith("@")) {
                templateName = templateName.substring(1);
                if (!templateName.contains(".")) {
                    templateName = request.controller + "." +
templateName;
                }
                templateName = templateName.replace(".", "/") + ".fo";
            }

            VirtualFile file = VirtualFile.search(Play.templatesPath,
templateName);
            return file;
        }

}
```

How it works...

Before trying to understand how this example works, you could also fire up the included example of this application under `examples/chapter2/pdf` and open `http://localhost:9000/` which will show you a PDF that includes the user data defined in the entity.

When opening the PDF, an XML template is rendered by the Play template engine and later processed by Apache FOP. Then it is streamed to the client. Basically, there is a new `renderPDF()` method created, which does all this magic. This method is defined in the `pdf.RenderPDF` class. All you need to hand over is a user to render.

The `RenderPDF` is only a rendering class, similar to the `DigestRequest` class in the preceding recipe. It consists of a static `renderPDF()` method usable in the controller and of three additional methods.

The `getTemplateFile()` method finds out which template to use. If no template was specified, a template with the name as the called method is searched for. Furthermore it is always assumed that the template file has a `.fo` suffix. The `VirtualFile` class is a Play helper class, which makes it possible to use files inside archives (like modules) as well. The `LocalVariablesNamesTracer` class allows you to get the names and the objects that should be rendered in the template.

The `populateRenderArgs()` method puts all the standard variables into the list of arguments which are used to render the template, for example, the session or the request.

The heart of this recipe is the `apply()` method, which sets the response content type to `application/pdf` and uses the Play built-in template loader to load the `.fo` template. After initializing all required variables for ApacheFOP, it renders the template and hands the rendered string over to the FOP transformer. The output of the PDF creation has been specified when calling the FopFactory. It goes directly to the output stream of the response object.

There's more...

As you can see, it is pretty simple in Play to write your own renderer. You should do this whenever possible, as it keeps your code clean and allows clean splitting of view and controller logic. You should especially do this to ensure that complex code such as Apache FOP does not sneak in to your controller code and make it less readable.

This special case poses one problem. Creating PDFs might be a long running task. However, the current implementation does not suspend the request. There is a solution to use the `await()` code from the controller in your own responses as seen in *Chapter 1*.

More about Apache FOP

Apache FOP is a pretty complex toolkit. You can create really nifty PDFs with it; however, it has quite a steep learning curve. If you intend to work with it, read the documentation under `http://xmlgraphics.apache.org/fop/quickstartguide.html` and check the examples directory (where the `index.fo` file used in this recipe has been copied from).

Using other solutions to create PDFs

There are many other solutions, which might fit your needs better than one based on `xsl-fo`. Libraries such as iText can be used to programmatically create PDFs. Perhaps even the PDF module available in the module repository will absolutely fit your needs.

See also

There is also the recipe *Writing your own renderRSS method as controller output* for writing your own RSS renderer at the end of this chapter.

Binding objects using custom binders

You might already have read the Play documentation about object binding. As validation is extremely important in any application, it basically has to fulfill several tasks.

First, it should not allow the user to enter wrong data. After a user has filled a form, he should get a positive or negative feedback, irrespective of whether the entered content was valid or not. The same goes for storing data. Before storing data you should make sure that storing it does not pose any future problems as now the model and the view layer should make sure that only valid data is stored or shown in the application. The perfect place to put such a validation is the controller.

As a HTTP request basically is composed of a list of keys and values, the web framework needs to have a certain logic to create real objects out of this argument to make sure the application developer does not have to do this tedious task.

You can find the source code of this example in the `chapter2/binder` directory.

How to do it...

Create or reuse a class you want created from an item as shown in the following code snippet:

```
public class OrderItem {

    @Required public String itemId;
    public Boolean hazardous;
    public Boolean bulk;
    public Boolean toxic;
    public Integer piecesIncluded;

    public String toString() {
       return MessageFormat.format("{0}/{1}/{2}/{3}/{4}", itemId,
piecesIncluded, bulk, toxic, hazardous);
    }
}
```

Create an appropriate form snippet for the `index.xml` template:

```
#{form @Application.createOrder()}
<input type="text" name="item" /><br />
<input type="submit" value="Create Order">
#{/form}
```

Create the controller:

```
public static void createOrder(@Valid OrderItem item) {
    if (validation.hasErrors()) {
        render("@index");
    }

        renderText(item.toString());
}
```

Create the type binder doing this magic:

```
@Global
public class OrderItemBinder implements TypeBinder<OrderItem> {

    @Override
    public Object bind(String name, Annotation[] annotations, String
value,
            Class actualClass) throws Exception {

        OrderItem item = new OrderItem();
        List<String> identifier = Arrays.asList(value.split("-", 3));

        if (identifier.size() >= 3) {
            item.piecesIncluded = Integer.parseInt(identifier.get(2));
}

        if (identifier.size() >= 2) {
            int c = Integer.parseInt(identifier.get(1));
            item.bulk = (c & 4) == 4;
            item.hazardous = (c & 2) == 2;
            item.toxic = (c & 1) == 1;
        }

        if (identifier.size() >= 1) { item.itemId = identifier.get(0);
}

        return item;
    }
}
```

How it works...

With the exception of the binder definition all of the preceding code has been seen earlier. By working with the Play samples you already got to know how to handle objects as arguments in controllers. This specific example creates a complete object out of a simple String. By naming the string in the form value (`<input ...name="item" />`) the same as the controller argument name (`createOrder(@Valid OrderItem item)`) and using the controller argument class type in the `OrderItemBinder` definition (`OrderItemBinder implements TypeBinder<OrderItem>`), the mapping is done.

The binder splits the string by a hyphen, uses the first value for item ID, the last for `piìesIncluded`, and checks certain bits in order to set some Boolean properties.

By using `curl` you can verify the behavior very easily as shown:

```
curl -v -X POST --data "item=Foo-3-5" localhost:9000/order

Foo/5/false/true/true
```

Here `Foo` resembles the item ID, `5` is the `piecesIncluded` property, and `3` is the argument means that the first two bits are set and so the hazardous and toxic properties are set, while bulk is not.

There's more...

The `TypeBinder` feature has been introduced in Play 1.1 and is documented at `http://www.playframework.org/documentation/1.2/controllers#custombinding`.

Using type binders on objects

Currently, it is only possible to create objects out of one single string with a `TypeBinder`. If you want to create one object out of several submitted form values you will have to create your own plugin for this as workaround. You can check more about this at:

`http://groups.google.com/group/play-framework/browse_thread/thread/62e7fbeac2c9e42d`

Be careful with JPA using model classes

As soon as you try to use model classes with a type binder you will stumble upon strange behavior, as your objects will always only have null or default values when freshly instanced. The JPA plugin already uses a binding and overwrites every binding you are doing.

Validating objects using annotations

Basic validation should be clear for you now. It is well-documented in the official documentation and shows you how to use the different annotations such as `@Min`, `@Max`, `@Url`, `@Email`, `@InFuture`, `@InPast`, or `@Range`. You should go a step forward and add custom validation. An often needed requirement is to create some unique string used as identifier. The standard way to go is to create a UUID and use it. However, validation of the UUID should be pretty automatic and you want to be sure to have a valid UUID in your models.

You can find the source code of this example in the `chapter2/annotation-validation` directory.

Getting ready

As common practice is to develop your application in a test driven way, we will write an appropriate test as first code in this recipe. In case you need more information about writing and using tests in Play, you should read `http://www.playframework.org/documentation/1.2/test`.

This is the test that should work:

```
public class UuidTest extends FunctionalTest {

    @Test
    public void testThatValidUuidWorks() {
        String uuid = UUID.randomUUID().toString();
        Response response = GET("/" + uuid);
        assertIsOk(response);
        assertContentEquals(uuid + " is valid", response);
    }

    @Test
    public void testThatInvalidUuidWorksNot() {
        Response response = GET("/absolutely-No-UUID");
        assertStatus(500, response);
    }
}
```

So whenever a valid UUID is used in the URL, it should be returned in the body and whenever an invalid UUID has been used, it should return HTTP error 500.

How to do it...

Add an appropriate configuration line to your `conf/routes` file:

```
GET     /{uuid}    Application.showUuid
```

Create a simple @UUID annotation, practically in its own annotations or validations package:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Constraint(checkWith = UuidCheck.class)
public @interface Uuid {
        String message() default "validation.invalid.uuid";
}
```

Create the appropriate controller, which uses the @Uuid annotation:

```
public class Application extends Controller {

    public static void showUuid(@Uuid String uuid) {
        if (validation.hasErrors()) {
            flash.error("Fishy uuid");
            error();
        }

        renderText(uuid + " is valid");
    }
}
```

Create the check, which is triggered by the validation. You might want to put it into the checks package:

```
public class UuidCheck extends AbstractAnnotationCheck<Uuid> {

    @Override
    public booleanisSatisfied(Object validatedObject, Object value,
    OValContext context, Validator validator)
    throws OValException {

        try {
            UUID.fromString(value.toString());
            return true;
        } catch (IllegalArgumentException e) {}

        return false;
    }
}
```

How it works...

When starting your application via `play test` and going to `http://localhost:9000/@tests` you should be able to run the `UuidTest` without problems.

Except the `UuidCheck` class, most of this here is old stuff. The Uuid annotation has two specialties. First it references the `UuidCheck` with a constraint annotation and second you can specify a message as argument. This message is used for internationalization.

The UuidCheck class is based on an Oval class. Oval is a Java library and used by the Play framework for most of the validation tasks and can be pretty easily extended as you can see here. All you need to implement is the `isSatisfied()` method. In this case it has tried to convert a String to a UUID. If it fails, the runtime exception thrown by the conversion is caught and false is returned, marking the check as invalid.

There's more...

The oval framework is pretty complex and the logic performed here barely scratches the surface. For more information about oval, check the main documentation at `http://oval.sourceforge.net/`.

Using the configure() method for setup

The `AbstractAnnotationCheck` class allows you to overwrite the configure(T object) method (where T is generic depending on your annotation). This allows you to set up missing annotation parameters with default data; for example, default values for translations. This is done by many of the already included Play framework checks as well.

Annotations can be used in models as well

Remember that the annotation created above may also be used in your models, so you can label any String as a UUID in order to store it in your database and to make sure it is valid when validating the whole object.

```
@Uuid public String registrationUuid;
```

Adding annotation-based right checks to your controller

Sooner or later any business application will have some login/logout mechanism and after that there are different types of users. Some users are allowed to do certain things, others are not. This can be solved via a check in every controller, but is way too much overhead. This recipe shows a clean and fast (though somewhat limited) solution to creating security checks, without touching anything of the business logic inside your controller.

You can find the source code of this example in the `chapter2/annotation-rights` directory.

Getting ready

Again we will start with a test, which performs several checks for security:

```
public class UserRightTest extends FunctionalTest {

    @Test
    public void testSecretsWork() {
        login("user", "user");
        Response response = GET("/secret");
        assertIsOk(response);
        assertContentEquals("This is secret", response);
    }

    @Test
    public void testSecretsAreNotFoundForUnknownUser() {
        Response response = GET("/secret");
        assertStatus(404, response);
    }

    @Test
    public void testSuperSecretsAreAllowedForAdmin() {
        login("admin", "admin");
        Response response = GET("/top-secret");
        assertIsOk(response);
        assertContentEquals("This is top secret", response);
    }

    @Test
    public void testSecretsAreDeniedForUser() {
        login("user", "user");
        Response response = GET("/top-secret");
        assertStatus(403, response);
    }

    private void login(String user, String pass) {
        String data = "username=" + user + "&password=" + pass;
        Response response = POST("/login",
                APPLICATION_X_WWW_FORM_URLENCODED, data);
        assertIsOk(response);
    }
}
```

As you can see here, every test logs in with a certain user first, and then tries to access a resource. Some are intended to fail, while some should return a successful access. Before every access, a login is needed using the `login()` method. In case you wonder why a simple HTTP request stores the returned login credentials for all of the next requests, this is actually done by the logic of the `FunctionalTest`, which stores all returned cookies from the login request during the rest of the test.

How to do it...

Add the needed routes:

```
POST    /login          Application.login
GET      /secret        Application.secret
GET      /top-secret    Application.topsecret
```

Create User and Right entities:

```
@Entity
public class User extends Model {

    public String username;
    public String password;
    @ManyToMany
    public Set<Right> rights;

    public booleanhasRight(String name) {
        Right r = Right.find("byName", name).first();
        return rights.contains(r);
    }
}
```

A simple entity representing a right and consisting of a name is shown in the following code:

```
@Entity
public class Right extends Model {

    @Column(unique=true)
    public String name;
}
```

Create a Right annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Right {
    String value();
}
```

Lastly, create all the controller methods:

```
public class Application extends Controller {

    public static void index() {
        render();
    }

    @Before(unless = "login")
    public static void checkForRight() {
        String sessionUser = session.get("user");
        User user = User.find("byUsername", sessionUser).first();
        notFoundIfNull(user);

        Right right = getActionAnnotation(Right.class);
        if (!user.hasRight(right.value())) {
            forbidden("User has no right to do this");
        }
    }

    public static void login(String username, String password) {
        User user = User.find("byUsernameAndPassword", username,
password).first();

        if (user == null) {
            forbidden();
        }

        session.put("user", user.username);
    }

    @Right("Secret")
    public static void secret() {
        renderText("This is secret");
    }

    @Right("TopSecret")
    public static void topsecret() {
        renderText("This is top secret");
    }
}
```

How it works...

Going through this step by step reveals surprisingly few new items, but rather a simple and concise change at the core of each controller call. Neither the routes are new, nor the entity definitions, or its possibility to create the `hasRight()` method. The only real new logic is inside the controller. The logic here is not meant as business logic of your application but rather permission checking. On the one hand every security aware controller has a `@Right` annotation at its definition, which defines the required right as a text string.

On the other hand all the logic regard permissions is executed at the `checkForRight()` method before every controller call. It inspects the annotation value and checks whether the currently logged-in user has this specific annotation value as a right set using the `hasRight()` method defined in the user entity.

There's more...

This is a pretty raw method to check for rights. It imposes several design weaknesses and severe performance issues. But it is a start to go further.

Be flexible with roles instead of rights

The security model here is pretty weak. You should think of using roles on user level instead of rights, and check these roles for the rights called. This allows you to create less fine-grained permission checks such as a "Content editor" and a "publisher" role for example.

More speed with caching

The whole code presented here can be pretty slow. First you could cache the roles or rights of a certain user. Furthermore you could cache the security right of the controller action and the login credentials, which are looked up on every request.

Increased complexity with context-sensitive rights

The security checks compared here are very simple. If you want to have a right, then only the owner of an object can change it, you are not completely off with the solution presented here. You need to define more logic inside your controller call.

Check out the deadbolt module

There is a module which already does more checks and has more options than this example. You should take a look at the deadbolt module at `http://www.playframework.org/modules/deadbolt`. This module offers restrictions not only on the controller level, but also inside Views. Deadbolt also provides arbitrary dynamic security, which allows you to provide application-specific security strategies.

Rendering JSON output

As soon as a web application consists of a very fast frontend, no or seldom complete page reloads occur. This implies a complete rendering by the browser, which is one of the most time consuming tasks when loading a webpage in the browser. This means you have to get the data as fast as possible to the client. You can either send them as pre-rendered HTML, XML, or JSON format. Sending the data in either JSON or XML means the application on the browser side can render the data itself and decide how and when it should be displayed. This means your application should be able to create JSON or XML-based responses.

As JSON is quite popular, this example will not only show you how to return the JSON representation of an entity, but also how to make sure sensitive data such as a password will not get sent to the user.

Furthermore some hypermedia content will be added to the response, like an URL where more information can be found.

You can find the source code of this example in the `chapter2/json-render-properties` directory.

Getting ready

Beginning with a test is always a good idea:

```
public class JsonRenderTest extends FunctionalTest {

    @Test
    public void testThatJsonRenderingWorks() {
        Response response = GET("/user/1");
        assertIsOk(response);

        User user = new Gson().fromJson(getContent(response), User.
class);
        assertNotNull(user);
        assertNull(user.password);
        assertNull(user.secrets);
        assertEquals(user.login, "alex");
        assertEquals(user.address.city, "Munich");
        assertContentMatch("\"uri\":\"/user/1\"", response);
    }
}
```

This expects a JSON reply from the request and parses it into a User instance with the help of gson, a JSON library from Google, which is also used by Play for serializing. As we want to make sure that no sensitive data is sent, there is a check for nullified values of password and secrets properties. The next check goes for a user property and for a nested property inside another object. The last check has to be done by just checking for an occurrence of the string, because the URL is not a property of the user entity and is dynamically added by the special JSON serializing routine used in this example.

How to do it...

Create your entities first. This example consists of a user, an address, and a `SuperSecretData` entity:

```
@Entity
public class User extends Model {

    @SerializedName("userLogin")
    public String login;
    @NoJsonExport
    public String password;
    @ManyToOne
    public Address address;
    @OneToOne
    public SuperSecretData secrets;

    public String toString() {
        return id + "/" +  login;
    }
}


@Entity
public class Address extends Model {
    public String street;
    public String city;
    public String zip;
}


@Entity
public class SuperSecretData extends Model {
    public String secret = "foo";
}
```

The controller is simple as well:

```
public static void showUser(Long id) {
    User user = User.findById(id);
    notFoundIfNull(user);
    renderJSON(user, new UserSerializer());
}
```

The last and most important part is the serializer used in the controller above:

```
public class UserSerializer implements JsonSerializer<User> {

    public JsonElement serialize(User user, Type type,
        JsonSerializationContext context) {
        Gsongson = new GsonBuilder()
            .setExclusionStrategies(new LocalExclusionStrategy())
            .create();

      JsonElementelem = gson.toJsonTree(user);
      elem.getAsJsonObject().addProperty("uri", createUri(user.id));
      return elem;
    }

    private String createUri(Long id) {
      Map<String,Object> map = new HashMap<String,Object>();
      map.put("id", id);
      return Router.reverse("Application.showUser", map).url;
    }


    public static class LocalExclusionStrategy implements
ExclusionStrategy {

      public booleanshouldSkipClass(Class<?>clazz) {
        return clazz == SuperSecretData.class;
      }

      public booleanshouldSkipField(FieldAttributes f) {
        return f.getAnnotation(NoJsonExport.class) != null;
      }
    }
}
```

How it works...

The entities used in this example are simple. The only differences are the two annotations in the User entity. First there is a `SerializedNamed` annotation, which uses the annotation argument as field name in the json output – this annotation comes from the gson library. The `@NoJsonExport` annotation has been specifically created in this example to mark fields that should not be exported like a sensitive password field in this example. The address field is only used as an example to show how many-to-many relations are serialized in the JSON output.

As you might guess, the `SuperSecretData` class should mark the data as secret, so this field should not be exported as well. However, instead of using an annotation, the functions of the Google gson library will be utilized for this.

The controller call works like usual except that the `renderJson()` method gets a specific serializer class as argument to the object it should serialize.

The last class is the `UserSerializer` class, which is packed with features, although it is quite short. As the class implements the `JsonSerializer` class, it has to implement the `serialize()` method. Inside of this method a gson builder is created, and a specific exclusion strategy is added. After that the user object is automatically serialized by the gson object. Lastly another property is added. This property is the URI of the `showUser()` controller call, in this case something like `/user/{id}` . You can utilize the Play internal router to create the correct URL.

The last part of the serializer is the `ExclusionStrategy`, which is also a part of the gsonserializer. This strategy allows exclusion of certain types of fields. In this case the method `shouldSkipClass()` excludes every occurrence of the `SuperSecretData` class, where the method `shouldSkipFields()` excludes fields marked with the `@NoJsonExport` annotation.

There's more...

If you do not want to write your own JSON serializer you could also create a template ending with `.json` and write the necessary data like in a normal HTML template. However there is no automatic escaping, so you would have to take care of that yourself.

More about Google gson

Google gson is a pretty powerful JSON library. Once you get used to it and learn to utilize its features it may help you to keep your code clean. It has very good support for specific serializers and deserializers, so make sure you check out the documentation before trying to build something for yourself. Read more about it at `http://code.google.com/p/google-gson/`.

Alternatives to Google gson

Many developers do not like the gson library at all. There are several alternatives. There is a nice example of how to integrate FlexJSON. Check it out at `http://www.lunatech-research.com/archives/2011/04/20/play-framework-better-json-serialization-flexjson`.

Writing your own renderRSS method as controller output

Nowadays, an almost standard feature of web applications is to provide RSS feeds, irrespective of whether it is for a blog or some location-based service. Most clients can handle RSS out of the box. The Play examples only carry an example with hand crafted RSS feeds around. This example shows how to use a library for automatic RSS feed generation by getting the newest 20 post entities and rendering it either as RSS, RSS 2.0 or Atom feed.

You can find the source code of this example in the `chapter2/render-rss` directory.

Getting ready

As this recipe makes use of the ROME library to generate RSS feeds, you need to download ROME and its dependency JDOM first. You can use the Play dependency management feature again. Put this in your `conf/dependencies.yml`:

```
require:
    - play
    - net.java.dev.rome -> rome 1.0.0
```

Now as usual a test comes first:

```
public classFeedTest extends FunctionalTest {

    @Test
    public void testThatRss10Works() throws Exception {
        Response response = GET("/feed/posts.rss");
        assertIsOk(response);
        assertContentType("application/rss+xml", response);
        assertCharset("utf-8", response);
        SyndFeed feed = getFeed(response);
        assertEquals("rss_1.0", feed.getFeedType());
    }

    @Test
    public void testThatRss20Works() throws Exception {
        Response response = GET("/feed/posts.rss2");
```

```
        assertIsOk(response);
        assertContentType("application/rss+xml", response);
        assertCharset("utf-8", response);
        SyndFeed feed = getFeed(response);
        assertEquals("rss_2.0", feed.getFeedType());
    }

    @Test
    public void testThatAtomWorks() throws Exception {
        Response response = GET("/feed/posts.atom");
        assertIsOk(response);
        assertContentType("application/atom+xml", response);
        assertCharset("utf-8", response);
        SyndFeed feed = getFeed(response);
        assertEquals("atom_0.3", feed.getFeedType());
    }

    private SyndFeedgetFeed(Response response) throws Exception {
        SyndFeedInput input = new SyndFeedInput();
        InputSource s = new InputSource(IOUtils.toInputStream
                          (getContent(response)));
        return input.build(s);
    }
}
```

This test downloads three different kinds of feeds, `rss1`, `rss2`, and `atom` feeds, and checks the feed type for each. Usually you should check the content as well, but as most of it is made up of random chars at startup, it is dismissed here.

How to do it...

The first definition is an entity resembling a post:

```
@Entity
public class Post extends Model {

    public String author;
    public String title;
    public Date createdAt;
    public String content;

    public static List<Post>findLatest() {
        return Post.findLatest(20);
    }
```

```
    public static List<Post>findLatest(int limit) {
        return Post.find("order by createdAt DESC").fetch(limit);
    }
}
```

A small job to create random posts on application startup, so that some RSS content can be rendered from application start:

```
@OnApplicationStart
public class LoadDataJob extends Job {

    // Create random posts
    public void doJob() {
        for (int i = 0 ; i < 100 ; i++) {
            Post post = new Post();
            post.author = "Alexander Reelsen";
            post.title = RandomStringUtils.
             randomAlphabetic(RandomUtils.nextInt(50));
            post.content = RandomStringUtils.
             randomAlphabetic(RandomUtils.nextInt(500));
            post.createdAt = new Date(new Date().getTime() +
             RandomUtils.nextInt(Integer.MAX_VALUE));
            post.save();
        }
    }
}
```

You should also add some metadata in the `conf/application.conf` file:

```
rss.author=GuybrushThreepwood
rss.title=My uber blog
rss.description=A blog about very cool descriptions
```

The `routes` file needs some controllers for rendering the feeds:

```
GET     /                 Application.index
GET     /feed/posts.rss   Application.renderRss
GET     /feed/posts.rss2  Application.renderRss2
GET     /feed/posts.atom  Application.renderAtom
GET     /post/{id}        Application.showPost
```

The Application controller source code looks like this:

```
import static render.RssResult.*;

public class Application extends Controller {

    public static void index() {
```

```
            List<Post> posts = Post.findLatest(100);
        render(posts);
    }

    public static void renderRss() {
            List<Post> posts = Post.findLatest();
        renderFeedRss(posts);
    }

    public static void renderRss2() {
            List<Post> posts = Post.findLatest();
        renderFeedRss2(posts);
    }

    public static void renderAtom() {
            List<Post> posts = Post.findLatest();
        renderFeedAtom(posts);
    }

    public static void showPost(Long id) {
            List<Post> posts = Post.find("byId", id).fetch();
            notFoundIfNull(posts);
            renderTemplate("Application/index.html", posts);
    }
}
```

You should also adapt the `app/views/Application/index.html` template to show posts and to put the feed URLs in the header to make sure a browser shows the RSS logo on page loading:

```
#{extends 'main.html' /}
#{set title:'Home' /}
#{set 'moreHeaders' }
<link rel="alternate" type="application/rss+xml" title="RSS 1.0 Feed"
href="@@{Application.renderRss2()}" />
<link rel="alternate" type="application/rss+xml" title="RSS 2.0 Feed"
href="@@{Application.renderRss()}" />
<link rel="alternate" type="application/atom+xml" title="Atom Feed"
href="@@{Application.renderAtom()}" />
#{/set}

#{list posts, as:'post'}
<div>
<h1>#{a @Application.showPost(post.id)}${post.title}#{/a}</h1><br />
by ${post.author} at ${post.createdAt.format()}
<div>${post.content.raw()}</div>
</div>
#{/list}
```

You also have to change the default `app/views/main.html` template, from which all other templates inherit to include the `moreHeaders` variable:

```
<html>
    <head>
        <title>#{get 'title' /}</title>
        <meta http-equiv="Content-Type" content="text/html;
          charset=utf-8">
        #{get 'moreHeaders' /}
        <link rel="shortcut icon" type="image/png" href="@{'/public/
images/favicon.png'}">
    </head>
    <body>
        #{doLayout /}
    </body>
</html>
```

The last part is the class implementing the different `renderFeed` methods. This is again a `Result` class:

```
public class RssResult extends Result {

    private List<Post> posts;
    private String format;

    public RssResult(String format, List<Post> posts) {
        this.posts = posts;
        this.format = format;
    }

    public static void renderFeedRss(List<Post> posts) {
        throw new RssResult("rss", posts);
    }

    public static void renderFeedRss2(List<Post> posts) {
        throw new RssResult("rss2", posts);
    }

    public static void renderFeedAtom(List<Post> posts) {
        throw new RssResult("atom", posts);
    }

    public void apply(Request request, Response response) {
        try {
            SyndFeed feed = new SyndFeedImpl();
            feed.setAuthor(Play.configuration.getProperty
               ("rss.author"));
```

```
            feed.setTitle(Play.configuration.getProperty
               ("rss.title"));
            feed.setDescription(Play.configuration.getProperty
               ("rss.description"));
            feed.setLink(getFeedLink());

            List<SyndEntry> entries = new ArrayList<SyndEntry>();
            for (Post post : posts) {
              String url = createUrl("Application.showPost", "id",
                                     post.id.toString());
              SyndEntry entry = createEntry(post.title, url,
                                     post.content, post.createdAt);
              entries.add(entry);
            }

            feed.setEntries(entries);

            feed.setFeedType(getFeedType());
            setContentType(response);

              SyndFeedOutput output = new SyndFeedOutput();
              String rss = output.outputString(feed);
              response.out.write(rss.getBytes("utf-8"));
        } catch (Exception e) {
              throw new UnexpectedException(e);
        }
    }

    private SyndEntrycreateEntry (String title, String link, String
description, Date createDate) {
        SyndEntry entry = new SyndEntryImpl();
        entry.setTitle(title);
        entry.setLink(link);
        entry.setPublishedDate(createDate);

        SyndContententryDescription = new SyndContentImpl();
        entryDescription.setType("text/html");
        entryDescription.setValue(description);

        entry.setDescription(entryDescription);

        return entry;
    }

    private void setContentType(Response response) {
        ...
    }
```

```
        private String getFeedType() {
            ...
        }

        private String getFeedLink(){
            ...
        }

        private String createUrl(String controller, String key, String
    value) {
            ...
        }
    }
```

How it works...

This example is somewhat long at the end. The post entity is a standard model entity with a helper method to find the latest posts. The `LoadDataJob` fills the in-memory database on startup with hundreds of random posts.

The `conf/routes` file features showing an index page where all posts are shown, as well as showing a specific post and of course showing all three different types of feeds.

The controller makes use of the declared `findLatest()` method in the post entity to get the most up-to-date entries. Furthermore the `showPost()` method also utilizes the `index.html` template so you do not need to create another template to view a single entry. All of the used `renderFeed` methods are defined in the `FeedResult` class.

The `index.html` template file features all three feeds in the header of the template. If you take a look at `app/views/main.html`, you might notice the inclusion of the `moreHeaders` variable in the header. Using the @@ reference to a controller in the template creates absolute URLs, which can be utilized by any browser.

The FeedResult class begins with a constructor and the three static methods used in the controller, which render RSS, RSS 2.0, or Atom feeds appropriately.

The main work is done in the `apply()` method of the class. A SyndFeed object is created and filled with meta information like blog name and author defined in the `application.conf` file.

After that a loop through all posts generates entries, each including the author, content, title, and the generated URL. After setting the content type and the feed type—RSS or atom—the data is written to the response output stream.

The helper methods have been left out to save some lines inside this example. The `setContentType()` method returns a specific content type, which is different for RSS and atom feeds. The `getFeedType()` method returns "rss_2.0", "rss_1.0", or "atom_0.3" depending on the feed to be returned. The `getFeedLink()` method returns an absolute URL for any of the three feed generating controller actions. The `createUrl()` method is a small helper to create an absolute URL with a parameter which in this case is an ID. This is needed to create absolute URLs for each post referenced in the feed.

The example also uses ROME to extract the feed data again in the test, which is not something you should do to ensure the correct creation of your feed. Either use another library, or, if you are proficient in checking corner cases by hand, do it manually.

There's more...

This is (as with most of the examples here) only the tip of the iceberg. Again, you could also create a template to achieve this, if you wanted to keep it simple. The official documentation lists some of the preparing steps to create your own templates ending with `.rss` at `http://www.playframework.org/documentation/1.2/routes#Customformats`

Using annotations to make your code more generic

This implementation is implementation specific. You could make it far more generic with the use of annotations at the Post entity:

```
@Entity
public class Post extends Model {

    @FeedAuthor
    public String author;
    @FeedTitle
    public String title;
    @FeedDate
    public Date createdAt;
    @FeedContent
    public String content;
}
```

Then you could change the render signature to the following:

```
    public RssResult(String format, List<? extends Object> data) {
        this.data = data;
        this.format = format;
    }

    public static void renderFeedRss(List<? extends Object> data) {
        throw new RssResult("rss", data);
    }
```

By using the generics API you could check for the annotations defined in the Post entity and get the content of each field.

Using ROME modules

ROME comes with a bunch of additional modules. It is pretty easy to add GeoRSS information or MediaWiki-specific RSS. This makes it pretty simple to extend features of your feeds.

Cache at the right place

Especially RSS URLs are frequently under heavy loads – one misconfigured client type can lead to some sort of denial of service attack on your system. So again a good advice is to cache here. You could cache the SyndFeed object or the rss string created out of it. However, since play 1.1 you could also cache the complete controller output. See *Basics of caching* at the beginning of *Chapter 4* for more information about that.

3
Leveraging Modules

In this chapter, we will cover:

- Dependency injection with Spring
- Dependency injection with Guice
- Using the security module
- Adding security to the CRUD module
- Using the MongoDB module
- Using MongoDB/GridFS to deliver files

Introduction

As the core of the Play framework strives to be as compact as possible, the aim is to offer arbitrary possibilities of extension. This is what modules are for. They are small applications inside your own application and allow easy and fast extension without bloating your own source code. Modules can introduce feature-specific abilities such as adding a different persistence mechanism, just helping your test infrastructure, or integrating other view techniques.

This chapter gives a brief overview of some modules and how to make use of them. It should help you to speed up your development when you need to integrate existing tools and libraries.

In case you wonder why some of the well-known modules are not included, this is merely because there is not much that could be added by giving them their own recipes. The GAE and GWT modules, for example, already have a good basic documentation included.

Dependency injection with Spring

The Spring framework has been released in 2003, and has been very successful in introducing concepts such as dependency injections and aspect-oriented programming to a wider audience. It is one of the most comprehensive and feature-complete frameworks in the Java ecosystem. It is possible that you may need to use the Spring framework in your Play application, maybe in order to reuse some components that have dependencies on the Spring API. In this case, the Spring module will help you to integrate the two frameworks together easily.

Also, you might want to use some existing code from your application and just test some features of Play. This is where the Spring module comes in very handy.

The source code of this recipe is available in the `examples/chapter3/spring` directory.

Getting ready

Create an application. Install the Spring module by adding it to the `dependencies.yml` file and rerun `playdeps`. Optionally, you may need to rerun the command to generate your IDE specific files. And, as usual, let's go test first. This example features a simple obfuscation of a string by using a service to encrypt and decrypt it:

```
public class EncryptionTest extends FunctionalTest {

    @Test
    public void testThatDecryptionWorks() {
        Response response = GET("/decrypt?text=foo");
        assertIsOk(response);
        assertContentEquals("Doof", response);
    }

    @Test
    public void testThatEncryptionWorks() {
        Response response = GET("/encrypt?text=oof");
        assertIsOk(response);
        assertContentEquals("Efoo", response);
    }
}
```

How to do it...

Now let's define some encryption service in this example.

Create a `conf/application-context.xml` file, where you define your beans:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN""http://www.
springsource.org/dtd/spring-beans-2.0.dtd">
```

```
<beans>
<bean id="encryptionService" class="spring.EncryptionServiceImpl" />
</beans>
```

Define two routes:

```
GET     /encrypt      Application.encrypt
GET     /decrypt      Application.decrypt
```

Define an `EncryptionService` interface and create a concrete implementation:

```
package spring;

public interface EncryptionService {
    public String encrypt(String clearText);
    public String decrypt(String cipherText);
}
```

It is true that this is not the strict definition of encryption, but it serves the purpose:

```
package spring;

public class EncryptionServiceImpl implements EncryptionService {

    @Override
    public String decrypt(String cipherText) {
        return "D" + StringUtils.reverse(cipherText);
    }

    @Override
    public String encrypt(String clearText) {
        return "E" + StringUtils.reverse(clearText);
    }
}
```

The last part is the controller:

```
public class Application extends Controller {

    public static void decrypt() {
        EncryptionService encService =
        Spring.getBeanOfType(EncryptionService.class);
        renderText(encService.decrypt(params.get("text")));
    }

    public static void encrypt() {
        EncryptionService encService =
```

```
            Spring.getBeanOfType(EncryptionService.class);
            renderText(encService.encrypt(params.get("text")));
        }

    }
```

How it works...

If you have worked with Spring before, most of the work is straightforward. After defining a bean in the application context and implementing the interface, your Spring application is up and running. The Play specific part is calling `Spring.getBeanOfType()` in the controller, which returns the specific spring bean.

You can call `Spring.getBeanOfType()` either with the name of the bean as argument or with a class you want to have return.

There's more...

Unfortunately, the Spring module (version 1.0 at the time of writing) does not yet support the `@Inject` annotation. Furthermore, the Spring version used is 2.5.5, so you might need to patch the module by replacing the jars in the `lib` directory of the module, before you Play around with the spring integration.

Use the component scanning feature

If you do not want to create a Spring definition file at all, you can use annotations. Comment out any bean definitions in the application-context file (but do not remove it!) and annotate the service with the `@Service` annotation on class level.

Have Spring configurations per ID

If you have set a special ID via "play ID", you can also load a special context on startup. If your ID is set to `foo`, create a `conf/foo.application-context.xml` Spring bean definition.

Direct access to the application context

You can use `SpringPlugin.applicationContext` to access the application context anywhere in your application.

See also

If you like dependency injection and inversion of control, just read on to find out that Google guice is also supported.

Dependency injection with Guice

Guice is the new kid on the block in the context dependency injection field. It tries not to be a complete stack like Spring, but merely a very useful addition, which does not carry grown structures around like Spring does. It has been developed by Google and is used in some of their applications, for example, in Google Docs, Adwords, and even YouTube.

The source code of this recipe is available in the `chapter3/guice` directory.

Getting ready

This example implements the same encryption service as the Spring example, so the only thing that changes is actually the controller implementation. You should have installed the Guice module by now, using the `dependencies.yml` file.

How to do it...

First a Guice module needs to be defined, where the interface is glued to the implementation:

```
public class GuiceModule extends AbstractModule {

    @Override
    protected void configure() {
    bind(EncryptionService.class).to(EncryptionServiceImpl.class);
    }
}
```

After that the controller can be written in the following way:

```
public class Application extends Controller {

    @Inject
    private static EncryptionService encService;

    public static void decrypt() {
        renderText(encService.decrypt(params.get("text")));
    }

    public static void encrypt() {
        renderText(encService.encrypt(params.get("text")));
    }
}
```

How it works...

As you can see, the `@Inject` annotation helps to keep code outside of the controller methods. This means any method can access the service object as it is defined as static. This also implies that you should never store state in such an object, the same as with any Spring bean. Also, be aware that you should import `javax.inject.Inject` and not the `com.google.inject.Inject` class in to inject your service correctly.

There's more...

Now let's talk about some other options, or possibly some pieces of general information that are relevant to this task.

Default @Inject support of play

The Guice module has basic support for the `@Inject` annotation, where you do not need to specify a mapping from an interface to a concrete implementation in the class which extends `AbstractModule`, like `GuiceModule` in this example. However, it works only for classes, which are either a `Job`, a `Mailer`, or implement the `ControllerSupport` interface. The following snippet would return the current date in a controller whenever it is called:

```
@Inject
private static DateInjector dater;

  public static void time() {
    renderText(dater.getDate());
}
```

The `DateInjector` would be defined as the following:

```
public class DateInjector implements ControllerSupport {

    public Date getDate() {
        return new Date();
    }
}
```

Keep in mind that the class you are injecting is always a singleton. Never store some kind of state inside its instance variables. Also this injection still needs to have the Guice module loaded.

Creating own injectors

The Guice module also supports the creation of own injectors. Please check the official documentation for more information as it is well described in there.

Using the security module

One of the basic functions of an application is the need for authentication and authorization. If you only have basic needs and checks, you can use the security module that is already bundled with Play. This recipe shows simple use cases for it.

You can find the source code of this example in the `chapter3/secure` directory.

Getting ready

Create an application and put the security module in the configuration. Though you do need to install the module, you do not need to specify a version because it is built in with the standard Play distribution. The `conf/dependencies.yml` entry looks like the following:

```
require:
    - play
    - play -> secure
```

As usual, nothing is complete without a test, here it goes:

```
public class SecurityTest extends FunctionalTest {

    @Test
    public void testThatIndexPageNeedsLogin() {
        Response response = GET("/");
        assertStatus(302, response);
        assertLocationRedirect("/login", response);
    }

    @Test
    public void testThatUserCanLogin() {
        loginAs("user");
        Response response = GET("/");
        assertContentMatch("Logged in as user", response);
    }

    @Test
    public void testThatUserCannotAccessAdminPage() {
        loginAs("user");
        Response response = GET("/admin");
        assertStatus(403, response);
    }
```

```
    @Test
    public void testThatAdminAccessAdminPage() {
        loginAs("admin");
        Response response = GET("/admin");
        assertStatus(302, response);
    }

    private void assertLocationRedirect(String location, Response
resp) {
        assertHeaderEquals("Location", "http://localhost"+location,
resp);
    }

    private void loginAs(String user) {
        Response response = POST("/login?username=" + user +
          "&password=secret");
        assertStatus(302, response);
        assertLocationRedirect("/", response);
    }
}
```

These four tests should validate the application behavior. First, you cannot access a page without logging in. Second, after logging in as `user` you should be redirected to the login page. The login page should contain the username. The third test checks to make sure that the `user`user may not get access to the admin page, while the fourth test verifies a valid access for the admin user.

This test assumes some things, which are laid down in the implementation:

- A user with name `user` and password `secret` is a valid login
- A user with name `admin` and password `secret` is a valid login and may see the admin page
- Watching the admin page results in a redirect instead of really watching a page

You might be wondering why the Play server is running in port 9000, but there is no port specified in the location redirect. The request object is created by the tests with port 80, as default. The port number does not affect testing because any functional test calls the Java methods inside of the Play framework directly instead of connecting via HTTP to it.

How to do it...

Let's list the steps required to complete the task. The routes file needs to include a reference to the secure module:

```
*       /               module:secure
GET     /               Application.index
GET     /admin          Application.admin
```

Only one single template is used in this example. The template looks like this:

```
#{extends 'main.html' /}
#{set title:'Home' /}

<h1>Logged in as ${session.username}</h1>

<div>
Go to #{a @Application.index()}index#{/a}
<br>
Go to #{a @Application.admin()}admin#{/a}
</div>

#{a @Secure.logout()}Logout#{/a}
```

The Controller looks like the following:

```
@With(Secure.class)
public class Application extends Controller {

    public static void index() {
        render();
    }

    @Check("admin")
    public static void admin() {
        index();
    }
}
```

The last part is to create a class extending the `Secure` class. This class is used as class annotation in the controller. The task of this class is to implement a basic security check, which does not allow login with any user/pass combination like the standard implementation does:

```
public class SimpleSecurity extends Secure.Security {

    static boolean authenticate(String username, String password) {
        return "admin".equals(username) &&"secret".equals(password) ||
            "user".equals(username) &&"secret".equals(password);
    }

    static boolean check(String profile) {
        if ("admin".equals(profile)) {
            return connected().equals("admin");
        }
        return false;
```

```
        }

        static void onAuthenticated() {
            Logger.info("Login by user %s", connected());
        }
        static void onDisconnect() {
            Logger.info("Logout by user %s", connected());
        }
        static void onCheckFailed(String profile) {
            Logger.warn("Failed auth for profile %s", profile);
            forbidden();
        }
    }
```

How it works...

A lot of explanation is needed for the `SimpleSecurity` class. It is absolutely necessary to put this class into the controller package, otherwise no security checks will happen. The routes configuration puts the secure module in front of all other URLs. This means that every access will be checked, if the user is authenticated, with the exception of login and logout of course.

The template shows the logged-in user, and offers a link to the index and to the administration site as well as the possibility to log out.

The controller needs to have a `@With` annotation at class level. It is important here to refer to the `Secure` class and not to your own written `SimpleSecure` class, as this will not work at all.

Furthermore, the admin controller is equipped with a `@Check` annotation. This will make the secure module perform an extra check to decide whether the logged-in user has the needed credentials.

The most important part though is the `SimpleSecure` class, which inherits form `Secure.Security`. The `authenticate()` method executes the check where the user is allowed to log in. In the preceding example it only returns success (as in Boolean true) if the user logs in with username admin or user and password secret in both cases.

Furthermore, there are three methods which are executed only when certain events happen, in this case a successful login, a successful logout, and missing permissions even though the user is logged in. This last case can happen only when the `Check` annotation is used on a controller, like done in the `admin()` controller. Furthermore, the `check()` method in the `SimpleSecure` class needs to be defined. In this case the `check` method performs two checks. First, if the value inside of the check annotation was `admin` and the second if the currently logged-in user is `admin` as well. So this check resembles the most simple admin check that can be implemented.

There's more...

This module has been kept as simple as possible intentionally. Whenever you need more complex checks, this module might not be what you search for, and you should write something similar yourself or extend the module to fit your needs. For more complex needs, you should take a look at the deadbolt and secure permissions modules.

Declare only one security class

You should have only one class in your project which inherits from security. Due to the problem that the classloader possibly loads classes randomly, Play always picks the first it finds. There is no possibility to enforce which security class is used.

Implementing rights per controller with the secure module

In *Chapter 2* there was an example where you could put certain rights via an annotation at a controller. Actually, it is not too hard to implement the same using the secure module. Take a few minutes and try to change the example in that way.

Adding security to the CRUD module

The CRUD module is the base of any rapid prototyping module. It helps you to administer data in the backend, while still being quick to create a frontend that closely resembles your prototype. For example, when creating an online shop, your first task is to create a nice looking frontend. Still it would be useful if you could change some things such as product name, description, or ID in the backend. This is where CRUD helps. However, there is no security inside the CRUD module, so anyone can add or delete data. This is where the secure module can help.

You can find the source code of this example in the `chapter3/crud-secure` directory.

Getting ready

You should already have added controllers for CRUD editing, as well as an infrastructure for authentication or at least something similar to a user entity. If you do not know about this part, you can read about it at `http://www.playframework.org/documentation/1.2/crud`.

How to do it...

You should have your own security implementation, something like this:

```
public class Security extends Secure.Security {

    static boolean authenticate(String username, String password) {
```

```
            User user = User.find("byUserAndPassword", username, Crypto.
    passwordHash(password)).first();
            return user != null;
        }

        static boolean check(String profile) {
            if ("admin".equals(profile)) {
              User user = User.find("byUser", connected()).first();
              if (user != null) {
                return user.isAdmin;
              }
            } else if ("user".equals(profile)) {
              return connected().equals("user");
            }
            return false;
        }
    }
```

Adding users via CRUD should only be done by the admin:

```
@Check("admin")
@With(Secure.class)
public class Users extends CRUD {
}
```

However, creating `Merchants` should never be allowed for the admin, but only by an authorized user. Deleting (most data on live systems will not be deleted anyway) `Merchants`, however, should be an admin only task again:

```
@With(Secure.class)
public class Merchants extends CRUD {

    @Check("admin")
    public static void delete(String id) {
        CRUD.delete(id);
    }

    @Check("user")
    public static void create() throws Exception {
        CRUD.create();
    }
}
```

How it works...

As you can see, you can easily secure complete controllers to be only accessible for logged-in users. Furthermore you can also make only special controllers available for certain users. As these methods are static, you are not able to call `super()` in them, but need to define the static methods of the parent controller again and then manually call the methods of the CRUD controller.

There's more...

CRUD should never be a big topic in your finished business application because your business logic will be far more complex than adding or removing entities. However, it can be a base for certain tasks. This is where more advanced aspects come in handy.

Changing the design of the CRUD user interface

You can use the `play crud:ov --template Foo/bar` command line call to copy the template HTML code to `Foo/bar.html`, so you can edit it and adapt it to your corporate design.

Checking out the scaffold module

There is also the scaffold module you can take a look at. It generates controllers and templates by inferring the information of your model classes when you run `play scaffold:gen` on the command line. It currently works for JPA and Siena.

Using the MongoDB module

MongoDB is one of the many rising stars on the NoSQL horizon. MongoDB outperforms other databases in development speed once you get used to thinking in data structures again instead of somewhat more or less arbitrary split rows. If you do not use MongoDB, this recipe will not help you at all.

You can find the source code of this example in the `chapter3/booking-mongodb` directory.

Getting ready

As there is already a wonderful and well known booking application, which is also used in many other frameworks (Seam, Spring, ZK) as an example, we will take it in this example and convert it to use MongoDB as its persistence backend. So, in order to get ready, copy it from the `samples-and-tests/booking` directory of your Play installation directory to somewhere else, and follow the steps outlined here.

After copying the application, you should install the Morphia module by adding it to the `dependencies.yml` file and rerun `play deps`. Then you are ready to convert the application to store data into MongoDB using Morphia instead of using the native SQL storage of Play.

Of course, you should have an up and running MongoDB instance. You can find some help installing it at `http://www.mongodb.org/display/DOCS/Quickstart`.

How to do it...

The first part is to convert the models of the application to use Morphia instead of JPA annotations. The simplest model is the user entity, which should look like this:

```
import play.modules.morphia.Model;
import com.google.code.morphia.annotations.Entity;

@Entity
public class User extends Model {

    @Required
    @MaxSize(15)
    @MinSize(4)
    @Match(value="^\\w*$", message="Not a valid username")
    public String username;

    @Required
    @MaxSize(15)
    @MinSize(5)
    public String password;

    @Required
    @MaxSize(100)
    public String name;

    public User(String name, String password, String username) {
        this.name = name;
        this.password = password;
        this.username = username;
    }

    public String toString()  {
        return "User(" + username + ")";
    }

}
```

In order to keep the recipe short, only the required changes will be outlined in the other entities instead of posting them completely. No JPA annotations should be present in your models following these changes. Always make sure you are correctly checking your imports as the annotations' names are often the same.

Remove the `@Temporal`, `@Table`, `@Column`, `@ManyToOne`, `@Entity` JPA annotations from the entities. You can replace `@ManyToOne` with `@Reference` in the Booking entity.

One last important point is to set the `BigDecimal` typed price to a Float type. This means losing precision. You should not do this in live applications if you need exact precision scale numbers. Currently Morphia does not support `BigDecimal` types. If you need precision arithmetic, you could create an own data type for such a task. Then replace this code from the original Hotel entity:

```
@Column(precision=6, scale=2)
public BigDecimal price;
```

By removing the annotation and setting the price as a float as in:

```
public Float price;
```

The next step is to replace some code in the Hotel controller. Whenever an ID is referenced in the routes file, it is not a Long but an `ObjectId` from MongoDB represented as a String, which consists of alphanumeric characters. This needs to be done in the signature of the `show()`, `book()`, `confirmBooking()`, and `cancelBooking()` methods. You can also set the ID field to be a Long type instead of an `ObjectID` via the `morphia.id.type=Long` parameter in your application configuration, if you want.

Whenever `find()` is called on a mongo entity, the `fetch()` method does not return a list, but an iterable. This iterable does not get the full data from the database at once. In order to keep this example simple, we will return all bookings by a user at once. So the `index()` methods need to replace the following:

```
List<Booking> bookings = Booking.find("byUser", connected()).fetch();
```

With the following:

```
List<Booking> bookings = Booking.find("byUser", connected()).asList();
```

The last change is the call of `booking.id`, which has to be changed to `booking.getId()` because there is no direct ID property in the `Model` class based on Morphia. This needs to be changed in the `confirmBooking()` and `cancelBooking()` methods.

How it works...

The functionality stays the same, as only the persistence layer and some controller code is changed. If you wanted to separate controller code even further, you could make sure that finders are always only constructed in the model class.

If you click through the example now and compare it to the database version, you will not see any difference. You can also use the mongo command line client to check whether everything you did was actually persisted. When checking a booking, you will see it looks like this:

```
>db.Booking.findOne()
{
    "_id" : ObjectId("4d1dceb3b301127c3fc745c6"),
    "className" : "models.Booking",
    "user" : {
      "$ref" : "User",
      "$id" : ObjectId("4d1dcd6eb301127c2ac745c6")
    },
    "hotel" : {
      "$ref" : "Hotel",
      "$id" : ObjectId("4d1dcd6eb301127c2dc745c6")
    },
    "checkinDate" : ISODate("2010-12-06T23:00:00Z"),
    "checkoutDate" : ISODate("2010-12-29T23:00:00Z"),
    "creditCard" : "1234567890123456",
    "creditCardName" : "VISA",
    "creditCardExpiryMonth" : 1,
    "creditCardExpiryYear" : 2011,
    "smoking" : false,
    "beds" : 1
}
```

As you can see, there is one booking. A specialty of Morphia is to store the class, where it was mapped from into the data as well, with the property `className`. If needed, this behavior can be disabled. The user and hotel properties are references to the specific collections and reference a certain object ID there. Think of this as a foreign key, when coming from the SQL world.

There's more...

This has only scratched the surface of what is possible. The Morphia module is especially interesting, because it also supports embedded data, even collections. You can, for example, map comments to a blog post inside of this post instead of putting it into your own collection. You should read the documentation of Morphia and the play-specific Morphia module very carefully though, if you want to be sure that you can easily convert an already started project to persist into MongoDB.

Check out the Yabe example in the Morphia directory

Morphia already includes a port of the Yabe example, which uses some of the more advanced features like map reduce.

Use long based data types as unique IDs

The Morphia module also offers to use a long value as ID instead of an object ID. This would have saved changing the controller code.

Aggregation and grouping via map reduce

As there is no join support in MongoDB, you will need to use map-reduce algorithms. There is no really good support of map-reduce in the java drivers, as you have to write JavaScript code as your map-reduce algorithms. For more information about that you might want to check the MongoDB documentation at `http://www.mongodb.org/display/DOCS/MapReduce`.

Using MongoDB/GridFS to deliver files

MongoDB has a very nice feature called GridFS, which removes the need to store binary data in the filesystem. This example will feature a small (and completely unstyled) image gallery. The gallery allows you to upload a file and store it into MongoDB.

You can find the source code of this example in the `chapter3/mongodb-image` directory.

Getting ready

You should have installed the Morphia module in your application and should have a configured up-and-running MongoDB instance.

How to do it...

The `application.conf` file should feature a complete MongoDB configuration as for any Morphia-enabled application. Furthermore, a special parameter has been introduced, which represents the collection to store the binary data. The parameter is optional anyway. The `uploads` collection resembles the default.

```
morphia.db.host=localhost
morphia.db.port=27017
morphia.db.name=images
morphia.db.collection.upload=uploads
```

The routes file features four routes. One shows the index page, one returns a JSON representation of all images to the client, one gets the image from the database and renders it, and one allows the user to upload the image and store it into the database.

```
GET     /                 Application.index
GET/images.json           Application.getImages
GET/image/{id}            Application.showImage
POST   /image/            Application.storeImage
```

The controller implements the routes:

```
public class Application extends Controller {

    public static void index() {
        render();
    }

    public static void getImages() {
        List<GridFSDBFile> files = GridFsHelper.getFiles();
        Map map = new HashMap();
        map.put("items", files);
        renderJSON(map, new GridFSSerializer());
    }

    public static void storeImage(File image, String desc) {
        notFoundIfNull(image);
        try {
          GridFsHelper.storeFile(desc, image);
        } catch (IOException e) {
          flash("uploadError", e.getMessage());
        }
        index();
    }

    public static void showImage(String id) {
        GridFSDBFile file = GridFsHelper.getFile(id);
        notFoundIfNull(file);
        renderBinary(file.getInputStream(), file.getFilename(),
            file.getLength(), file.getContentType(), true);
    }
}
```

As written in the preceding code snippet, an own `Serializer` for the `GridFSDBFile` class uses its own Serializer when rendering the JSON reply:

```
public class GridFSSerializer implements JsonSerializer<GridFSDBFile>
{

    @Override
    public JsonElement serialize(GridFSDBFile file, Type type,
          JsonSerializationContextctx) {
        String url = createUrlForFile(file);
        JsonObjectobj = new JsonObject();
        obj.addProperty("thumb", url);
```

```
            obj.addProperty("large", url);
            obj.addProperty("title", (String)file.get("title"));
            obj.addProperty("link", url);

            return obj;
        }

        private String createUrlForFile(GridFSDBFile file) {
            Map<String,Object> map = new HashMap<String,Object>();
            map.put("id", file.getId().toString());
            return Router.getFullUrl("Application.showImage", map);
        }
    }
```

The `GridFSHelper` is used to store and read images as binary data from MongoDB:

```
    public class GridFsHelper {

        public static GridFSDBFilegetFile(String id) {
            GridFSDBFile file = getGridFS().findOne(new ObjectId(id));
            return file;
        }

        public static List<GridFSDBFile>getFiles() {
            return getGridFS().find(new BasicDBObject());
        }

        public static void storeFile(String title, File image) throws
    IOException {
            GridFSfs = getGridFS();
            fs.remove(image.getName()); // delete the old file
            GridFSInputFile gridFile = fs.createFile(image);
            gridFile.save();
            gridFile.setContentType("image/" + FilenameUtils.
    getExtension(image.getName()));
            gridFile.setFilename(image.getName());
            gridFile.put("title", title);
            gridFile.save();
        }

        private static GridFS getGridFS() {
            String collection = Play.configuration.getProperty("morphia.
    db.collection.upload", "uploads");
```

```
            GridFSfs = new GridFS(MorphiaPlugin.ds().getDB(), collection);
            return fs;
        }
    }
```

As the Dojo Toolkit is used in this example, the main template file needs to be changed to include a class attribute in the body tag. The Dojo Toolkit is a versatile JavaScript library, which features a nice gallery widget used in this example. So the file `app/views/main.html` needs to be changed:

```
<!DOCTYPE html>
<html>
    <head>
        <title>#{get 'title' /}</title>
        <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
        #{get 'moreStyles' /}
        <link rel="shortcut icon" type="image/png"href="@{'/public/
images/favicon.png'}">
        #{get 'moreScripts' /}
    </head>
    <body class="tundra">
        #{doLayout /}
    </body>
</html>
```

Furthermore, the index templates file itself needs to be created at `app/views/Application/index.html`:

```
#{extends 'main.html' /}
#{set title:'Gallery' /}
#{set 'moreStyles'}
<style type="text/css">
  @import "http://ajax.googleapis.com/ajax/libs/dojo/1.5/dojox/image/
resources/image.css";
  @import "http://ajax.googleapis.com/ajax/libs/dojo/1.5/dijit/themes/
tundra/tundra.css";
</style>
#{/set}
#{set 'moreScripts'}
<script src="http://ajax.googleapis.com/ajax/libs/dojo/1.5/dojo/dojo.
xd.js"djConfig="parseOnLoad:true"></script>
<script type="text/javascript">
  dojo.require("dojox.image.Gallery");
  dojo.require("dojo.data.ItemFileReadStore");
</script>
#{/set}
```

```
#{form @Application.storeImage(), enctype:'multipart/form-data'}
  <div>Title: <input type="text" name="description"></div>
  <div>File: <input type="file" name="image"></div>
  <div><input type="submit" value="Send"></div>
#{/form}

<h1>The gallery</h1>

<div jsId="imageItemStore" dojoType="dojo.data.
ItemFileReadStore"url="@{Application.getImages()}"></div>

<div id="gallery1 dojoType="dojox.image.Gallery">
  <script type="dojo/connect">
    varitemNameMap = {
      imageThumbAttr: "thumb",
      imageLargeAttr: "large"
    };
    this.setDataStore(imageItemStore, {}, itemNameMap);
  </script>
</div>
```

How it works...

The configuration and routes files are already explained above. The controller mainly uses the `GridFSHelper` and the `GridFSSerializer`.

The `GridfSHelper` calls the Morphia plugin to get the database connection. You could also do this by just using the MongoDB driver; however, it is likely that you will use the rest of the Morphia module as well. The `getGridFS()` method returns the object needed to extract GridFS files from MongoDB. The `getFile()` method queries for a certain object ID, while the `getFiles()` method returns all objects because a query by example is done with an empty object. This is the way the standard MongoDB API works as well. The `storeFile()` method deletes an already existing file (the image file name used when uploading is used here). After deletion it is stored, and its content type is set along with a metadata tag called `title`. As storing might pose problems (connection might be down, user may not have rights to store, filesystem could be full), an exception can possibly be thrown, which must be caught in the controller.

The `Serializer` for a `GridFSDBFile` is pretty dumb in this case. The format of the JSON file is predefined. Due to the use of the Dojo Toolkit, data has to be provided in a special format. This format requires having four properties set for each image:

- `thumb`: Represents a URL of a small thumbnail of the image.
- `large`: Represents a URL of the normal sized image.

- `link`: Represents a URL which is rendered as a link on the normal sized image. Could possibly be a Flickr link for example.
- `title`: Represents a comment for this particular image.

For the sake of simplicity, the thumb, large, and link URLs are similar in this special case – in a real world application this would not be an option.

The controller has no special features. After a file upload the user is redirected to the index page. This means only one template is needed. The `getImages()` method uses the special serializer, but also needs to have an `items` defined to ensure the correct JSON format is returned to the client. The `showImage()` method gets the file from the `GridFSHelper` class and then sets header data as needed in the `renderBinary()` by calling it with the right arguments. There is no need to set headers manually.

After setting the `class` attribute to the body tag in the main template, the last thing is to write the `index.html` template. Here all the needed Dojo JavaScript and CSS files are loaded from the Google CDN. This means that you do not need to download Dojo to your local system. The `dojo.require()` statements are similar to Java class imports in order to provide certain functionality. In this case the Gallery functionality itself uses a so-called `FileItemReadStore` in order to store the data of the JSON reply in a generalized format which can be used by Dojo. Whenever you want to support HTML form-based file uploads, you have to change the enctype parameter of the form tag. The rest of the HTML is Dojo specific. The first div tag maps the `FileItemReadStore` to the JSON controller. The second div tag defines the gallery itself and maps the `FileItemReadStore` to the gallery, so it uses the JSON data for display.

After you finished editing the templates, you can easily go to the main page, upload an arbitrary amount of pictures, and see it including a thumbnail and a main image on the same page.

There's more...

Of course this is the simplest example possible. There are many possibilities for improvement.

Using MongoDB's REST API

Instead of using the Morphia module, you could also use the MongoDB built-in REST API. However, as it is quite a breeze to work with the Morphia API, there is no real advantage except for the more independent layer.

Resizing images on the fly

You could possibly create thumbnails on the fly when uploading the file. There is a `play.libs.Images` class which already features a `resize()` method.

4

Creating and Using APIs

In this chapter, we will cover:

- Using Google Chart API as a tag
- Including a Twitter search in your application
- Managing different output formats
- Binding JSON and XML to objects

Introduction

Although possible, it is unlikely in today's web application environment that you will only provide data inside your own application. Chances are high that you will include data from other sources as well. This means you have to implement strategies so that your application will not suffer from downtime of other applications. Though you are dependent on other data, you should make sure your actual live system is not, or at least it has the capability to function without the data provided.

The first recipe will show a practical example of integrating an API into your application. It will use the nice Google Chart API. In order to draw such graphs, the templating system will be extended with several new tags.

Another quick example will show you how to include a Twitter search in your own page, where you have to deal with problems other than the one with the chart API example.

The last two examples provide some tips on what to do when you are being a data provider yourself, and how to expose an API to the outside world.

However, first, we should dig a little deeper into the basics of mashups.

If you are asking yourself what mashups are, but you have already built several web applications, then chances are high that you have already created mashups without knowing it. Wikipedia has a very nice and short definition about it:

> *"In web development, a mashup is a web page or application that uses and combines data, presentation or functionality from two or more sources to create new services."*

See `http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)`.

So, as seen earlier, just by putting Google maps on your page to show the exact address of some data you stored, you basically created a mashup, or maybe you have already included a Flickr search on your page?

What you create out of mashups is basically left to your power of imagination. If you need some hints on what kind of cool APIs exist, then you may go to `http://www.programmableweb.com/` and check their API listings.

You can distinguish between the two types of mashups, namely, those happening on the server side, and those happening on the client side.

You can render a special link or HTML snippet, which resembles a Google map view of your house on your home page. This is client side because the data fetching is done by the client. The only thing you are providing is the link that has to be fetched.

In contrast to this, there might be services where you have to query the service first and then present the data to the client. However, you are getting the data from the API to your application, and then using it to render a view to the client. A classic example of this might be the access to your CRM system. For example you might need to get leads or finished deals of the last 24 hours from your CRM system. However, you do not want to expose this data directly to the client, as it needs to be anonymized first, before a graphical representation is shown to the client.

The most important thing is to ensure that your application still works when the data provider is down. It should not run into timeouts, blocking your application on the client side as well as on the server side. If you keep fetching the same data, then you might want to store it locally in order to be independent of outages. If you really rely on this data, then make sure you have SLAs on your business, and a failure-proof implementation that monitors outages and keeps on trying to access the data end point.

Using Google Chart API as a tag

Sooner or later, one of your clients will ask for graphical representation of something in your application. It may be time-based (revenue per day/month/year), or more arbitrary. Instead of checking available imaging libraries like JFreeChart, and wasting your own CPU cycles when creating images, you can rely on the Google Chart API that is available at `http://code.google.com/apis/chart/`.

This API supports many charts, some of which do not even resemble traditional graphs. We will come to this later in the recipe.

The source code of the example is available at `examples/chapter4/mashup-chart-api`.

How to do it...

Some random data to draw from might be useful. A customer entity and an order entity are created in the following code snippets:

```
public class Customer {

  public String name;
  public List<Order> orders = new ArrayList<Order>();

  public Customer() {
    name = RandomStringUtils.randomAlphabetic(10);
    for (int i = 0 ; i< 6 ; i++) {
      orders.add(new Order());
    }
  }
}
```

Creating orders is even simpler, as shown in the following code snippet:

```
public class Order {
  publicBigDecimal cost = new BigDecimal(RandomUtils.nextInt(50));
}
```

The index controller in the Application needs to expose a customer on every call, as shown in the following code snippet. This saves us from changing anything in the routes file:

```
public static void index() {
  Customer customer = new Customer();
  render(customer);
}
```

Now the `index.html` template must be changed, as shown in the following code snippet:

```
#{extends 'main.html' /}
#{settitle:'Home' /}

<h2>QrCode for customer ${customer.name}</h2>

#{qrcode customer.name, size:150 /}

<h2>Current lead check</h2>

#{metertitle:'Conversion sales rate', value:70 /}

<h2>Some random graphic</h2>

#{linecharttitle:'Some data',
  labelX: 1..10, labelY:1..4,
  data:[2, 4, 6, 6, 8, 2, 2.5, 5.55, 10, 1] /}

<h2>Some sales graphic</h2>

#{chart.lctitle:'Sales of customer ' + customer.name,
  labelY:[0,10,20,30,40,50],
  labelX:['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
  data:customer.orders.subList(0, 6),
  field:'cost' /}
```

As you can see here, four new tags are used. The first two are pretty simple and represent usual tags. The `views/tags/qrcode.html` file looks similar to the following code snippet:

```
%{
  size = _size?:200
}%
<img src="http://chart.apis.google.com/chart?cht=qr&chl=${_
arg}&chs=${size

  }x${size}">
```

The `views/tags/meter.html` file writes the image tag out a little different:

```
%{
  width = _width?:300
  height = _height?:150
  title = _title?:"No title"

  encodedData = googlechart.DataEncoder.encode([_value], 100)
```

```
    out.print('<imgsrc="http://chart.apis.google.com/chart?cht=gm')
    out.print('&chs=' + width + 'x' + height)
    out.print('&chd=e:' + encodedData)
    out.println('&chtt='+title+'">')
%}
```

The `linechart` tag allows us to draw arbitrary data handed over into arrays. The file must be placed at `views/tags/linechart.html` and needs to look like the following code snippet:

```
%{
    width = _width?:300
    height = _height?:225
    title = _title?:"No title"
    colors = _colors?:"3D7930"

    out.print('<imgsrc="http://chart.apis.google.com/chart?')
    out.print('cht=lc')

    String labelX = _labelX.join("|");
    String labelY = _labelY.join("|");
    out.println("&chxl=0:|"+ labelX + "|1:|" + labelY);

    out.print('&chs=' + width + 'x' + height)
    out.print('&chtt=' + title)
    out.print('&chco=' + colors)
    dataEncoded = googlechart.DataEncoder.encode(_data)
    out.print('&chd=e:' + dataEncoded)

    maxValue = googlechart.DataEncoder.getMax(_data)
    out.print('&chxr=0,0,' + maxValue)
    out.print('&chxt=x,y')
    out.print("&chls=1,6,3");

    out.print('">')

}%
```

The remaining tag, used as `#{chart.lc}` in the index template, is a so-called fast tag and uses Java instead of the template language; therefore, it is a simple class that extends from the standard `FastTags` class, as shown in the following code snippet:

```
@FastTags.Namespace("chart")
public class ChartTags extends FastTags {

    public static void _lc(Map<?, ?>args, Closure body, PrintWriter out,
      ExecutableTemplate template, intfromLine) throws Exception {
```

```
        out.print("<imgsrc="http://chart.apis.google.com/chart?");
        out.print("cht=lc");

        out.print("&chs=" + get("width", "400", args) + "x" +
    get("height",
          "200", args));
        out.print("&chtt=" + get("title", "Standard title", args));
        out.print("&chco=" + get("colors", "3D7930", args));

        String labelX = StringUtils.join((List<String>)args.
    get("labelX"),
          "|");
        String labelY = StringUtils.join((List<String>)args.
    get("labelY"),
          "|");
        out.println("&chxl=0:|"+ labelX + "|1:|" + labelY);

        List<Object> data = (List<Object>) args.get("data");

        String fieldName = args.get("field").toString();

        List<Number>xValues = new ArrayList<Number>();
        for (Object obj : data) {
          Class clazz = obj.getClass();
          Field field = clazz.getField(fieldName);
          Number currentX = (Number) field.get(obj);
          xValues.add(currentX);
        }

        String dataString = DataEncoder.encode(xValues);

        out.print("&chd=e:" + dataString);
        out.print("&chxs=0,00AA00,14,0.5,l,676767");
        out.print("&chxt=x,y");
        out.print("&chxr=0,0," + DataEncoder.getMax(xValues));
        out.print("&chg=20,25");
        out.print("&chls=1,6,3");
        out.print("\">");
      }

      private static String get(String key, String defaultValue,
        Map<?,?>args) {
        if (args.containsKey(key)) {
          return args.get(key).toString();
      }
      return defaultValue;
    }
  }
```

If you have read the above tags and the last Java class, then you might have seen the usage of the `DataEncoder` class. The Google chart API needs the supplied data in a special format. The `DataEncoder` code snippet is as follows:

```
public class DataEncoder {

  public static String chars =

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-.";
  public static int length = chars.length();

  public static String encode(List<Number> numbers, intmaxValue) {
    String data = "";
    for (Number number : numbers) {
     double scaledVal = Math.floor(length * length * number.intValue()
        / maxValue);

      if (scaledVal> (length * length ) -1) {
        data += "..";
      }
      else if (scaledVal< 0) {
      data += "__";
      }
      else {
        int quotient = (int) Math.floor(scaledVal / length);
        int remainder = (int) scaledVal - (length * quotient);
        data += chars.charAt(quotient) + "" + chars.charAt(remainder);
      }
    }
    Logger.debug("Called with %s and %s => %s", numbers, maxValue,
      data);
    return data;
  }

  public static String encode(List<Number> numbers) {
    return encode(numbers, getMax(numbers));
  }

  public static intgetMax(List<Number> numbers) {
    Number max = numbers.get(0);
    for (Number number : numbers.subList(1, numbers.size())) {
      if (number.doubleValue() >max.doubleValue()) {
        max = number;
```

```
            }
        }

        return (int) Math.ceil(max.doubleValue());
    }
}
```

This formatter always produces the extended format, which is a little bit longer, but can represent all the data handed to it.

How it works...

A lot of code has been produced, so what has been done? Instead of writing the Google image code all over again, tags were created to ease usage. No one can remember all those parameters which are needed for this or that type of graph. Also, the template code looks much cleaner, because you actually get to know by reading what kind of graphic is supposed to be seen.

I will not explain here the myths of the Google APIs or how the data encoder is working. It is created from the example JavaScript code on the chart API web pages. The request parameters are extensively explained in the documentation. I will only show the specialties of the tags.

Taking a closer look at the `#{qrcode}` tag reveals the usage of a default parameter for the size. As long as it is not taken over as argument, as well as setting the title parameter of the graphic.

The `#{meter}` tag uses a big Groovy scriptlet for executing its logic. Inside the script you can access the request output stream via the `out` variable. Furthermore, the data encoder is called with its full class path and name as you cannot import classes inside a template.

The `#{linechart}` tag is pretty long for a single tag. You should think whether it makes more sense to write such a big logic inside the template with Groovy, or to use a fast tag direct instead. Fast tags have the possibility of being unit tested for example. As you can see by the use of the `join()` method on the `labelX` and `labelY` arrays, writing pure Groovy code is not a problem here. When used correctly, this tag allows the developer to input arbitrary data into the variable, as long as it is an array consisting of numbers. So, this is the generic version of a drawing tag.

The`#{chart.lc}` tag is a more concrete version of the `#{linechart}` tag, because you use the data of a certain field as the input. The data is a list of arbitrary objects, but additionally you specify the field where data of each object should be extracted from. The first six orders of a customer are used as data input as you can see in the index template. Inside every order, the content of the cost property is used as data. This saves the developer the hassle of always having to provide an array of data to the `render()` method. This is the big difference to the `#{linechart}` tag.

As `#{chart.lc}` is a fast tag, its implementation is in Java. Looking at the class, the `@` Namespace annotation before the class definition shows where the chart prefix is coming from. This helps you to have same named tags in your application. Every tag you want to implement in Java has to be a method, which must be public, static, return void, and must begin with an underscore. Also, the arguments must match. However, it may throw any exception or none at all. This helps to keep the code free from exceptions in this case, as no error handling is done. If the property you defined to check for does not exist, the whole tag crashes. One should, of course, never do this in a production environment. You should output nothing, but create an error log message. Basically, the tag does the same as the `#{linechart}` does. It joins the labels for x and y axis as needed, then iterates through the array of objects. For each object, the field is read from the object with the help of the reflection API.

In case you are wondering why the `DataEncoder` class has the `getMax()` method exposed, it is needed to keep the graph scaled.

There's more...

Before going on, you should delve deeper into tags by taking a look at the Play framework source code, which shows off some nice examples and tricks to keep in mind.

Getting request data inside a fast tag

It is no problem to get the request or the parameters inside a fast tag. Access the request and all its subsequent data structures via the following code:

```
Request req = Http.Request().current();
```

This ensures thread safety by always returning the request of the current thread.

The Google Chart API

The Google Chart API is really powerful and complex. I have barely scratched the surface here. You will see that when you check the documentation at `http://code.google.com/apis/chart/docs/chart_params.html`. An even better place to look at the Google Chart API is the gallery at `http://imagecharteditor.appspot.com`, where you can try out different image types in the browser. The API features several charts with magnitudes of options.

Make a graceful and more performant implementation

There is really a lot of room for improvement in this example. First, the tags are nowhere near graceful. Many different errors can occur, like not handing over mandatory parameters, class cast exceptions when converting the data to numbers, or defining a non existing property and then calling the reflection API. As a little practice, you could try to improve this before adding new graph options. If you want to create new charts, then the simplest way is to use the chart wizard provided by Google to find out which parameter changes the behavior.

Considering privacy when transmitting data

By using the Google Chart API you are actually transmitting quite a lot of data out of your system, in clear text. You should be aware that this might pose a privacy problem. Personally, I would not submit sensitive data like my daily revenue through the Internet just to have it graphed. On the other hand, I would not have a problem with the average response times of my server from yesterday. Always think about such facts before creating mashups.

Including a Twitter search in your application

This example shows you how to include the result of a Twitter search in your application. This time it is not client based as the first recipe of this chapter. The result will be downloaded to the server, and then displayed to the client. This poses a possible problem.

What happens if your server cannot reach Twitter? There might be a number of different reasons. For example, your DNS is flaky, Twitter is down, the routing to Twitter is broken, or you are pulling off too many requests resulting in a ban, and many, many more. However, this should not affect your application. It might, of course, affect what data is displayed on the page – it may however never stop, or block any of your services. Your complete system has to be unaffected by failure of any external system. This recipe shows a small example and incidentally uses the Twitter API for this. You can, however, copy the principle behind this to any arbitrary API you are connecting to. You can get more information about the Twitter API we are about to use at `http://dev.twitter.com/doc/get/search`.

The source code of the example is available at `examples/chapter4/mashup-twitter-search`.

Getting ready

All you need is a small application which gets some data from an external source.

How to do it...

In order to be a little bit dynamic add the following query to the `application.conf` file:

```
twitter.query=http://search.twitter.com/search.
json?q=playframework%20OR%20from.playframework&lang=en
```

Create a POJO (Plain Old Java Object) which models the mandatory fields of a Twitter search query response, as shown in the following code snippet:

```
public class SearchResult implements Serializable {

  @SerializedName("from_user") public String from;
```

```
  @SerializedName("created_at") public Date date;
  @SerializedName("text") public String text;
}
```

Write a job which queries the Twitter service every 10 minutes and stores the results, as shown in the following code snippet:

```
@OnApplicationStart
@Every("10min")
public class TwitterSearch extends Job {

  public void doJob() {
    String url = Play.configuration.getProperty("twitter.query");

    if (url == null) {
      return;
    }

    JsonElement element = WS.url(url).get().getJson();

    if (!element.isJsonObject()) {
      return;
    }

    JsonObject jsonObj = (JsonObject) element;

    Gson gson = new GsonBuilder().setDateFormat("EEE, dd MMM yyyy
HH:mm:ss Z").create();

    Type collectionType = new
      TypeToken<Collection<SearchResult>>(){}.getType();
    Collection<SearchResult> search =
      gson.fromJson(jsonObj.get("results"), collectionType);
    search = removeDuplicates(search);

    Cache.set("twitterSearch", search);
  }

    private Collection<SearchResult> removeDuplicates(Collection
<SearchResult> search) {
    Collection<SearchResult> nonduplicateSearches = new
LinkedHashSet();
    Set<String> contents = new HashSet();
```

```
        for (SearchResultsearchResult : search) {
          if (!contents.contains(searchResult.text)) {
            nonduplicateSearches.add(searchResult);
            contents.add(searchResult.text);
          }
        }

        return nonduplicateSearches;
      }
    }
```

Put the Twitter results in your page rendering code, as shown in the following code snippet:

```
public class Application extends Controller {

  @Before
  public static void twitterSearch() {
    Collection<SearchResult> results = Cache.get("twitterSearch",
      Collection.class);
    if (results == null) {
      results = Collections.emptyList();
    }
    renderArgs.put("twitterSearch", results);
  }

  public static void index() {
    render();
  }
}
```

The final step is to create the template code, as shown in the following code snippet:

```
#{extends 'main.html' /}
#{settitle:'Home' /}

#{cache 'twitterSearchesRendered', for:'10min'}
  <ul>
    #{list twitterSearch, as:'search'}
    <li><i>${search.text}</i>by ${search.from},
      ${search.date.since()}</li>
    #{/list}
  </ul>
#{/cache}
```

How it works...

As you can see, it is not overly complex to achieve independence from your API providers with a few tricks.

Configuration is pretty simple. The URL resembles a simple query searching for everything which contains Play framework, or is from the @playframework Twitter account. The created page should stay up-to-date with the Play framework.

The `SearchResult` class represents an entity with the JSON representation of the search reply defined in the configuration file. If you put this URL into a browser, you will see a JSON reply, which has `from_user`, `createdAt`, and text fields. As the naming scheme is not too good, the class uses the `@SerializedName` annotation for a mapping. You could possibly map more fields, if you wanted. Note that the `@SerializedName` annotation is a helper annotation from the Gson library.

The logic is placed in the `TwitterSearch` class. It is a job. This helps to decouple the query to Twitter from any incoming HTTP request. You do not want to query any API at the time new requests come in. Of course, there are special cases such as market rates that have to be live data. However, in most of the cases there is no problem, when the data provided is not real-time. Decoupling this solves several problems. It reduces wait times until the request is loaded. It reduces wait times, while the response is parsed. All this has to be done, while the client is waiting for a response. This is especially important for Play in order to not block other clients that are also requesting resources.

The `TwitterSearch doJob()` method checks whether a configuration URL has been provided. If this is the case, then it is fetched via the `WS` class, which is a very useful helper class – and directly stored as a JSON element. If the returned JSON element is a complex JSON object, a Google gson parser is created. It is created with a special date format, which automatically parses the date in the `createdAt` field, and saves the need to create a custom date serializer, as the results field inside the JSON reply contains all Twitter messages. This field should be deserialized into a collection of `SearchResult` instances. Because this is going to be a collection, a special collection type has to be created and mapped. This is done with the `TypeToken` class, which gets handed over to the `gson.fromJson()` method. Finally, the `removeDuplicates()` methods filters out all retweets by not allowing duplicate text content in the collection of `SearchResult` instances. This makes sure that only boring retweets are not displayed in your list of tweets. After the collection is cleared of doubled tweets, it is put in the cache.

The controller has a method with a `@Before` annotation, which checks the cache for a list of `SearchResult`s. If there is no cache hit, it creates an empty collection. In any case, the collection is put into the `renderArgs` variable and can be used in any template of this controller.

The last step is to display the content. If you take a look at the template, here again the caching feature is used. It is absolutely optional to use the cache here. At worst, you get a delay of 20 minutes until your data is updated, because the job only runs every ten minutes in addition to caching for ten minutes inside of the template. Think about, whether such a caching makes sense in your application before implementing it.

There's more...

Even though caching is easy and fast to implement, sometimes there are scenarios where it might not be needed, like the possibility of telling the client to get the data. As usual, try to create applications without caching and only add it if needed.

Make it a client side API

Check out the search documentation on the Twitter site at `http://dev.twitter.com/doc/get/search`—when you check the possibility of the JSON API to use a callback parameter. It actually gets pretty easy to build this as a client side search, so your servers do not have to issue the request to Twitter. You should check any API, to see whether it is actually possible to offload stuff to the client. This keeps your application even more scalable and independent – from a server point of view, not the functionality point of view.

Add caching to your code late

Whenever you are developing new features and start integrating APIs, you should actually add the caching feature as late as possible. You might stumble over strange exceptions, when putting invalid or incomplete data into the cache because of incorrect error handling, or not putting serialized objects into the cache. Keep this in mind, as soon as you stumble across error messages or exceptions when trying to read data from a cache. Again, cover everything with tests as much as possible.

Be fast with asynchronous queries

If you have a use-case where you absolutely must get live data from another system, you still have the possibility to speed things up a little bit. Imagine the following controller call, which returns two daily quotes, and queries remote hosts in order to make sure it always gets the latest quote instead of a boring cached one, as shown in the following code snippet:

```
public static void quotes() throws Exception {

  Promise<HttpResponse> promise2 =
    WS.url("http://www.iheartquotes.com/api/v1/random?format=json").
    getAsync();
  Promise<HttpResponse> promise1 =
    WS.url("http://jamocreations.com/widgets/travel-
    quotes/get.php").getAsync();
```

```
    // code here, preferably long running like db queries...
    // ...
    List<HttpResponse>resps = Promise.waitAll(promise1, promise2).get();

    if(resps.get(0) != null) {
      renderArgs.put("cite1",
      resps.get(0).getXml().getElementsByTagName("quote").
      item(0).getChildNodes().item(1).getTextContent());
    }

    if(resps.get(1) != null) {
      renderArgs.put("cite2", ((JsonObject)
      resps.get(1).getJson()).get("quote").getAsString());
    }

    render();
  }
```

This allows you to trigger both external HTTP requests in a non-blocking fashion, instead of calling them sequentially, and then perform some database queries. After that you can access the promise, where the probability of them already having ended is much higher. The preceding snippet is included in `examples/chapter5/mashup-api`.

Managing different output formats

In the preceding examples another API was consumed. When your application gets more users, the demand to get data out of your application will not only rise in web pages. As soon as machine-to-machine communication is needed, you will need to provide an API yourself.

This recipe will show you how to implement your own APIs using the Play framework. If you need an API that exposes your data in as many data formats as possible, you might not be right with selecting the Play framework. Currently, it is not that generic. You might want to go with enunciate for example, which is reachable at `http://enunciate.codehaus.org/`

Find the accompanying source example at `examples/chapter4/mashup-api`.

Getting ready

There is some preliminary information you should know before implementing anything. Play already has a built-in mechanism for finding out what type of data to return. It checks the `Accept` header of the request and puts the result in the `request.format` variable inside your controllers. Play also decides what template to render based on this information.

How to do it...

Let's start with a service to create tickets. There are several ways to provide authentication to your API. One way is to always use HTTP Basic Authentication. This means a password is supplied on every request. This implies that all your communication must be secure in order to not to leak passwords. Encryption is expensive, if you do not have special crypto hardware in your servers. If your data is not that confidential, it should be sufficient to only do the authentication via SSL and use the created ticket for the rest. This basically resembles the browser behaviour, where often login to a site is encrypted; the rest of the communication is clear text. Be aware that session hijacking is possible with this scenario. If you need security, go via HTTPS completely. The following code snippet is an example controller to generate a unique ticket:

```
public static void createTicket(String user, String pass) {
  User u = User.find("byNameAndPassword", user, pass).first();
  if (u == null) {
    error("No authorization granted");
  }

  String uuid = UUID.randomUUID().toString().replaceAll("-", "");
  Cache.set("ticket:" + uuid, u.name, "5min");
  renderText(uuid);
}
```

Now the ticket needs to be checked on every incoming request, except the ticket creation itself. This is shown in the following code snippet:

```
@Before(priority=1, unless="createTicket")
public static void checkAuth() {
  Header ticket = request.headers.get("x-authorization");
  if (ticket == null) {
    error("Please provide a ticket");
  }

  String cacheTicket = Cache.get("ticket:" + ticket.value(),
    String.class);
  if (cacheTicket == null) {
    error("Please renew your ticket");
  }

  Cache.set("ticket:" + ticket.value(), cacheTicket, "5min");
}
```

From now on, every request should have an `X-Authorization` header, where the created UUID is sent to the server. These tickets are valid for five minutes, and then they expire from the cache. On every request the expired time is reset to five minutes. You could possibly put this into the database as well, if you wanted, but the cache is a better place for such data.

As the ticket generator returns a text string by using `renderText()`, it is pretty easy to use. However, you may want to return different output formats based on the client's request. The following code snippet is an example controller that returns the user's favorite quote:

```
public static void quote() {
  String ticket = request.params.get("ticket");
  String username = Cache.get("ticket:" + ticket, String.class);
  User user = User.find("byName", username).first();
  String quote = user.quote;
  render(quote);
}
```

Now, add three templates for the controller method. The first is the HTML template which needs to be put at `app/views/Application/quote.html`:

```
<html><body>The quote is ${quote}</body></html>
```

Then comes `app/views/Application/quote.xml`:

```
<quote>${quote}</quote>
```

And finally `app/views/Application/quote.json`:

```
{ "quote": "${quote}" }
```

How it works...

It is pretty simple to test the above implemented functionality by running curl against the implementation – as an alternative to tests, which should always be the first choice. The first thing in this example is to get a valid ticket:

```
curl -X POST --data "user=test&pass=test" localhost:9000/ticket

096dc3153f774f898f122d9af3e5cfcb
```

After that you can call the quote service with different headers:

```
curl --header "X-Authorization: 096dc3153f774f898f122d9af3e5cfcb"--
header
  "Accept: application/json" localhost:9000/quote

{ "quote": "Aleaiactaest!" }
```

XML is also possible:

```
curl--header "X-Authorization: 096dc3153f774f898f122d9af3e5cfcb"
--header
  "Accept: application/xml" localhost:9000/quote

<quote>Aleaiactaest!</quote>
```

Adding no header – or an invalid one –returns the standard HTML response:

```
curl--header "X-Authorization: 096dc3153f774f898f122d9af3e5cfcb"
  localhost:9000/quote

<html><body>The quote is Aleaiactaest!</body></html>
```

The functionality of being able to return different templates based on the client `Accept` header looks pretty useful at first sight. However, it carries the burden of forcing the developer to ensure that valid XML or JSON is generated. This is usually not what you want. Both formats are a little bit picky about validation. The developer should create neither of those by hand. This is what the `renderJSON()` and `renderXml()` methods are for. So always use the alternative methods presented in this recipe with care, as they are somewhat error prone, even though they save some lines of controller code.

There's more...

It is very simple to add more text based output formats such as CSV, and combine them with the default templating engine. However, it is also possible to support binary protocols such as the AMF protocol if you need to.

Integrating arbitrary formats

It is easily possible to integrate arbitrary formats in the rendering mechanism of Play. You can add support for templates with a `.vcf` file suffix with one line of code. More information about this is provided in the official documentation at the following link: `http://www.playframework.org/documentation/1.2.1/routes#Customformats`

Getting out AMF formats

You should check the modules repository for more output formats like just JSON or XML. If you are developing a Flex application, then you might need to create some AMF renderer. In this case, you should check out `https://bitbucket.org/maxmc/cinnamon-play/overview`.

See also

Another minor problem that needs to be solved is the ability to also accept incoming XML and JSON data, and to convert it into objects. This is done in the next recipe.

Binding JSON and XML to objects

After you have explored Play a little bit and written your first apps, you might have noticed that it works excellently when binding complex Java objects out of request parameters, as you can put complex objects as controller method parameters. This type of post request deserialization is the default in Play. This recipe shows how to convert JSON and XML data into objects without changing any of your controller code.

The source code of the example is available at `examples/chapter4/mashup-json-xml`.

Getting ready

Let's start with a controller, which will not change and does not yield any surprises, as shown in the following code snippet:

```
public class Application extends Controller {

  public static void thing(Thing thing) {
    renderText("foo:"+thing.foo+"|bar:"+thing.bar+"\n");
  }
}
```

You should add a correct route for the controller as well in `conf/routes`

```
POST     /thing     Application.thing
```

Start with a test as usual:

```
public class ApplicationTest extends FunctionalTest {

  private String expectedResult = "foo:first|bar:second\n";

  @Test
  public void testThatParametersWork() {
    String html = "/thing?thing.foo=first&thing.bar=second";
    Response response = POST(html);
    assertIsOk(response);
    assertContentType("text/plain", response);
    assertContentMatch(expectedResult, response);
  }

  @Test
  public void testThatXmlWorks() {
    String xml = "<thing><foo>first</foo><bar>second</bar></thing>";
    Response response = POST("/thing", "application/xml", xml);
```

```
        assertIsOk(response);
        assertContentType("text/plain", response);
        assertContentMatch(expectedResult, response);
      }

      @Test
      public void testThatJsonWorks() {
        String json = "{ thing : { \"foo\" : \"first\", \"bar\" :
    \"second\"
          } }";
        Response response = POST("/thing", "application/json", json);
        assertIsOk(response);
        assertContentType("text/plain", response);
        assertContentMatch(expectedResult, response);
      }

    }
```

Three tests come with three different representations of the same data. The first one is the standard representation of an object and its properties come via HTTP parameters. Every parameter starting with "thing." is mapped as property to the thing object of the controller. The second example represents the thing object as XML entity, whereas the third does the same as JSON. In both cases, there is a thing root element. Inside of this element every property is mapped.

How to do it...

In order to get this to work, a small but very effective plugin is needed. In this case, the plugin will be put directly into the application. This should only be done for rapid prototyping but not in big production applications. The first step is to create an `app/play.plugins` file with the following content:

```
201:plugin.ApiPlugin
```

This ensures that the `ApiPlugin` class in the plugin package is loaded on application startup.

The next step is to modify the entity to support `JAXB` annotations:

```
@Entity
@XmlRootElement(name="thing")
@XmlAccessorType(XmlAccessType.FIELD)
public class Thing extends Model {
  @XmlElement public String foo;
  @XmlElement public String bar;
}
```

The last step is to write the plugin itself, as shown in the following code snippet:

```
public class ApiPlugin extends PlayPlugin {

  private JAXBContext jc;
  private Gson gson;

  public void onLoad() {
    Logger.info("ApiPlugin loaded");
    try {
      List<ApplicationClass>applicationClasses =
        Play.classes.getAnnotatedClasses(XmlRootElement.class);
      List<Class> classes = new ArrayList<Class>();
      for (ApplicationClassapplicationClass : applicationClasses) {
        classes.add(applicationClass.javaClass);
      }
      jc = JAXBContext.newInstance(classes.toArray(new Class[]{}));
    }
    catch (JAXBException e) {
      Logger.error(e, "Problem initializing jaxb context: %s",
        e.getMessage());
    }
    gson = new GsonBuilder().create();

  }

  public Object bind(String name, Class clazz, Type type, Annotation[]
    annotations, Map<String, String[]>params) {
    String contentType = Request.current().contentType;

    if ("application/json".equals(contentType)) {
      return getJson(clazz, name);
    }
    else if ("application/xml".equals(contentType)) {
      return getXml(clazz);
    }

    return null;
  }

  private Object getXml(Class clazz) {
    try {
      if (clazz.getAnnotation(XmlRootElement.class) != null) {
        Unmarshaller um = jc.createUnmarshaller();
```

```
            return um.unmarshal(Request.current().params.get("body"));
          }
        }
        catch (JAXBException e) {
          Logger.error("Problem rendering XML: %s", e.getMessage());
        }
        return null;
      }

      private Object getJson(Class clazz, String name) {
        try {
          String body = Request.current().params.get("body");
          JsonElement jsonElem = new JsonParser().parse(body);
          if (jsonElem.isJsonObject()) {
            JsonObject json = (JsonObject) jsonElem;

            if (json.has(name)) {
              JsonObject from = json.getAsJsonObject(name);
              Object result = gson.fromJson(from, clazz);
              return result;
            }
          }
        }
        catch (Exception e) {
          Logger.error("Problem rendering JSON: %s", e.getMessage());
        }
        return null;
      }

    }
```

How it works...

Though the presented solution is pretty compact in lines of code and amount of files touched, lots of things happen here.

First, the created `play.plugins` file consists of two fields. The first field represents a priority. This priority defines the order of modules loaded. Take a look at `framework/src/play.plugins` of your Play installation. Check the number written before `DBPlugin` is mentioned. This number represents a loading priority. In any case, the `ApiPlugin` should have a lower priority than the `DBPlugin`, resulting in being loaded first. Otherwise, this recipe will not work because the `bind()` method of the `DBPlugin` gets executed in favor of our own written one.

The next step is to extend the entity. You need to add `JAXB` annotations. The `@XmlRootElement` annotation marks the name of the root element. You have to set this annotation, because it is also used by the plugin later on. The second annotation, `@XmlAccessorType`, is also mandatory with its value `XmlAccessType.FIELD`. This ensures field access instead of methods – this does not affect your model getters. If you use getters, they are called as in any other Play application. It is merely a JAXB issue. The `@XmlElement` annotations are optional and can be left out.

The core of this recipe is the `ApiPlugin` class. It consists of four methods, and more importantly, in order to work as a plugin it must extend `PlayPlugin`. The `onLoad()` method is called when the plugin is loaded, on the start of the application. It logs a small message, and creates a Google gson object for JSON serialization as well as a JAXB context. It searches for every class annotated with `@XmlRootElement`, and adds it to the list of classes to be parsed for this JAXB context. The `bind()` method is the one called on incoming requests. The method checks the Content-Type header. If it is either `application/json` or `application/xml`, it will call the appropriate method for the content type. If none is matched, null is returned, which means that this plugin could not create an object out of the request. The `getXml()` method tries to unmarshall the data in the request body to an object, in case the handed over class has an `@XmlRootElement` annotation. The `getJson()` method acts pretty similar. It converts the response string to a JSON object and then tries to find the appropriate property name. This property is then converted to a real object and returned if successful.

As you can see in the source, there is not much done about error handling. This is the main reason why the implementation is that short. You should return useful error messages to the user, instead of just catching and logging exceptions.

There's more...

This implementation is quite rudimentary and could be made more error proof. However, another major point is that it could be made even simpler for the developer using it.

Add the XML annotations via byte code enhancement

Adding the same annotations in every model class over and over again does not make much sense. This looks like an easy to automate job. However, adding annotations is not as smooth as expected with Java. You need to perform byte code enhancement. Read more about enhancing class annotations via bytecode enhancement at the following link:

`http://www.csg.is.titech.ac.jp/~chiba/javassist/tutorial/tutorial3.html`

Put plugins where they belong

Plugins never ever belong to your main application. They will clutter your code and make your applications less readable. Put them into their own module, always.

Change your render methods to use JAXB for rendering

You could add a `renderXML(Object o)` method, which is similar to its `renderJSON(Object o)` companion. By using JAXB, you get almost everything for free:

```
public class RenderXml extends Result {

  private Marshaller m;
  private Object o;

  public static void renderXML(Object o) {
    throw new RenderXml(o);
  }

  public RenderXml(Object o) {
    this.o = o;
  }

  @Override
  public void apply(Request request, Response response) {
    try {
      setContentTypeIfNotSet(response, "text/xml");
      m = ApiPlugin.jc.createMarshaller();
      m.marshal(o, response.out);
    }
    catch (JAXBException e) {
      Logger.error(e, "Error renderXml");
    }
  }
}
```

Your controller call may now look similar to the following code snippet:

```
public static void thingXml(Thing thing) {
  renderXML(thing);
}
```

Of course, you should not forget the static import in your controller, as shown in the following line of code:

```
import static render.RenderXml.*;
```

All of this is also included in the example source code.

See also

Learn how to create modules instead of putting code as shown in the preceding examples directly into your application in the next chapter.

5
Introduction to Writing Modules

In this chapter, we will cover:

- Creating and using your own module
- Building a flexible registration module
- Understanding events
- Managing module dependencies
- Using the same model for different applications
- Understanding bytecode enhancement
- Adding private module repositories
- Preprocessing content by integrating stylus
- Integrating Dojo by adding command line options

Introduction

Modularity should be one of the main goals, when designing your application. This has several advantages from a developer's point of view: reusability and structured components are among them.

A module in Play is basically just an application which can be included in your current application. This application-in-application architecture means that there are only very few differences between your application and the included module. You implicitly know the basics of modules as an average Play developer.

In order to get to know more modules, you should not hesitate to take a closer look at the steadily increasing amount of modules available at the Play framework modules page at `http://www.playframework.org/modules`.

When beginning to understand modules, you should not start with modules implementing its persistence layer, as they are often the more complex ones.

In order to clear up some confusion, you should be aware of the definition of two terms throughout the whole chapter, as these two words with an almost identical meaning are used most of the time. The first is word is *module* and the second is *plugin*. Module means the little application which serves your main application, where as plugin represents a piece of Java code, which connects to the mechanism of plugins inside Play.

Creating and using your own module

Before you can implement your own functionality in a module, you should know how to create and build a module. This recipe takes a look at the module's structure and should give you a good start.

The source code of the example is available at `examples/chapter5/module-intro`.

How to do it...

It is pretty easy to create a new module. Go into any directory and enter the following:

```
play new-module firstmodule
```

This creates a directory called `firstmodule` and copies a set of predefined files into it. By copying these files, you can create a package and create this module ready to use for other Play applications. Now, you can run `play build-module` and your module is built. The build step implies compiling your Java code, creating a JAR file from it, and packing a complete ZIP archive of all data in the module, which includes Java libraries, documentation, and all configuration files. This archive can be found in the `dist/` directory of the module after building it. You can just press *Return* on the command line when you are asked for the required Play framework version for this module. Now it is simple to include the created module in any Play framework application. Just put this in the in the `conf/dependencies.yml` file of your application. Do not put this in your module!

```
require:
    - play
    - customModules -> firstmodule

repositories:
```

```
- playCustomModules:
    type:       local
    artifact:   "/absolute/path/to/firstmodule/"
    contains:
        - customModules -> *
```

The next step is to run `play deps`. This should show you the inclusion of your module. You can check whether the `modules/` directory of your application now includes a file `modules/firstmodule`, whose content is the absolute path of your module directory. In this example it would be `/path/to/firstmodule`. To check whether you are able to use your module now, you can enter the following:

```
play firstmodule:hello
```

This should return `Hello` in the last line. In case you are wondering where this is coming from, it is part of the `commands.py` file in your module, which was automatically created when you created the module via `play new-module`. Alternatively, you just start your Play application and check for an output such as the following during application startup:

```
INFO  ~ Module firstmodule is available (/path/to/firstmodule)
```

The next step is to fill the currently non-functional module with a real Java plugin, so create `src/play/modules/firstmodule/MyPlugin.java`:

```
public class MyPlugin extends PlayPlugin {

   public void onApplicationStart() {
      Logger.info("Yeeha, firstmodule started");
   }

}
```

You also need to create the file `src/play.plugins`:

```
1000:play.modules.firstmodule.MyPlugin
```

Now you need to compile the module and create a JAR from it. Build the module as shown in the preceding code by entering `play build-module`. After this step, there will be a `lib/play- firstmodule.jar` file available, which will be loaded automatically when you include the module in your real application configuration file. Furthermore, when starting your application now, you will see the following entry in the application log file. If you are running in development mode, do not forget to issue a first request to make sure all parts of the application are loaded:

```
INFO ~ Yeeha, firstmodule started
```

How it works...

After getting the most basic module to work, it is time go get to know the structure of a module. The filesystem layout looks like this, after the module has been created:

```
app/controllers/firstmodule
app/models/firstmodule
app/views/firstmodule
app/views/tags/firstmodule
build.xml
commands.py
conf/messages
conf/routes
lib
src/play/modules/firstmodule/MyPlugin.java
src/play.plugins
```

As you can see a module basically resembles a normal Play application. There are directories for models, views, tags, and controllers, as well as a configuration directory, which can include translations or routes. Note that there should never be an `application.conf` file in a module.

There are two more files in the root directory of the module. The `build.xml` file is an ant file. This helps to compile the module source and creates a JAR file out of the compiled classes, which is put into the `lib/` directory and named after the module. The `commands.py` file is a Python file, which allows you to add special command line directives, such as the `play firstmodule:hello` command that we just saw when executing the Play command line tool.

The `lib/` directory should also be used for additional JARs, as all JAR files in this directory are automatically added to classpath when the module is loaded.

Now the only missing piece is the `src/` directory. It includes the source of your module, most likely the logic and the plugin source. Furthermore, it features a very important file called `play.plugins`. After creating the module, the file is empty. When writing Java code in the `src/` directory, it should have one line consisting of two entries. One entry features the class to load as a plugin; where as the other entry resembles a priority. This priority defines the order in which to load all modules of an application. The lower the priority, the earlier the module gets loaded.

If you take a closer look at the `PlayPlugin` class, which `MyPlugin` inherits from, you will see a lot of methods that you can override. Here is a list of some of them accompanying a short description:

- `onLoad()`: This gets executed directly after the plugin has been loaded. However, this does not mean that the whole application is ready!

- `bind()`: There are two `bind()` methods with different parameters. These methods allow a plugin to create a real object out of arbitrary HTTP request parameters or even the body of a request. If you return anything different other than null in this method, the returned value is used as a parameter for controller whenever any controller is executed. Please check the recipe *Binding JSON and XML to objects,* for more details about usage of this, as it includes a recipe on how to create Java objects from JSON or XML request bodies.
- `getStatus()`, `getJsonStatus()`: Allows you to return an arbitrary string representing a status of the plugin or statistics about its usage. You should always implement this for production ready plugins in order to simplify monitoring.
- `enhance()`: Performs bytecode enhancement. Keep on reading the chapter to learn more about this complex but powerful feature.
- `rawInvocation()`: This can be used to intercept any incoming request and change the logic of it. This is already used in the `CorePlugin` to intercept the `@kill` and `@status` URLs. This is also used in the `DocViewerPlugin` to provide all the existing documentation, when being in test mode.
- `serveStatic()`: Allows for programmatically intercepting the serving of static resources. A common example can be found in the SASS module, where the access to the `.sass` file is intercepted and it is precomplied. This will also be used in a later recipe, when integrating stylus.
- `loadTemplate()`: This method can be used to inject arbitrary templates into the template loader. For example, it could be used to load templates from a database instead of the filesystem.
- `detectChange()`: This is only active in development mode. If you throw an exception in this method, the application will be reloaded.
- `onApplicationStart()`: This is executed on application start and if in development mode, on every reload of your application. You should initiate stateful things here, such as connections to databases or expensive object creations. Be aware, that you have to care of thread safe objects and method invocations for yourself. For an example you could check the `DBPlugin`, which initializes the database connection and its connection pool. Another example is the `JPAPlugin`, which initializes the persistence manager or the `JobPlugin`, which uses this to start jobs on application start.
- `onApplicationReady()`: This method is executed after all plugins are loaded, all classes are precompiled, and every initialization is finished. The application is now ready to serve requests.
- `afterApplicationStart()`: This is currently almost similar to `onApplicationReady()`.
- `onApplicationStop()`: This method is executed during a graceful shutdown. This should be used to free resources, which were opened during the starting of the plugin. A standard example is to close network connections to database, remove stale file system entries, or clear up caches.

- `onInvocationException()`: This method is executed when an exception, which is not caught is thrown during controller invocation. The `ValidationPlugin` uses this method to inject an error cookie into the current request.
- `invocationFinally()`: This method is executed after a controller invocation, regardless of whether an exception was thrown or not. This should be used to close request specific data, such as a connection, which is only active during request processing.
- `beforeActionInvocation()`: This code is executed before controller invocation. Useful for validation, where it is used by Play as well. You could also possibly put additional objects into the render arguments here. Several plugins also set up some variables inside thread locals to make sure they are thread safe.
- `onActionInvocationResult()`: This method is executed when the controller action throws a result. It allows inspecting or changing the result afterwards. You can also change headers of a response at this point, as no data has been sent to the client yet.
- `onInvocationSuccess()`: This method is executed upon successful execution of a complete controller method.
- `onRoutesLoaded()`: This is executed when routes are loaded from the routes files. If you want to add some routes programmatically, do it in this method.
- `onEvent()`: This is a poor man's listener for events, which can be sent using the `postEvent()` method. Another recipe in this chapter will show how to use this feature.
- `onClassesChange()`: This is only relevant in testing or development mode. The argument of this method is a list of freshly changed classes, after a recompilation. This allows the plugin to detect whether certain resources need to be refreshed or restarted. If your application is a complete shared-nothing architecture, you should not have any problems. Test first, before implementing this method.
- `addTemplateExtensions()`: This method allows you to add further `TemplateExtension` classes, which do not inherit from `JavaExtensions`, as these are added automatically. At the time of this writing, neither a plugin nor anything in the core Play framework made use of this, with the exception of the Scala module.
- `compileAll()`: If the standard compiler inside Play is not sufficient to compile application classes, you can override this method. This is currently only done inside the Scala plugin and should not be necessary in regular applications.
- `routeRequest()`: This method can be used to redirect requests programmatically. You could possibly redirect any URL which has a certain prefix or treat POST requests differently. You have to render some result if you decide to override this method.

- `modelFactory()`: This method allows for returning a factory object to create different model classes. This is needed primarily inside of the different persistence layers. It was introduced in play 1.1 and is currently only used by the JPA plugin and by the Morphia plugin. The model factory returned here implements a basic and generic interface for getting data, which is meant to be independent from the persistence layer. It is also used to provide a more generic fixtures support.
- `afterFixtureLoad()`: This method is executed after a `Fixtures.load()` method has been executed. It could possibly be used to free or check some resources after adding batch data via fixtures.

There's more...

These are the mere basics of any module. You should be aware of this, when reading any other recipe in this chapter.

Cleaning up after creating your module

When creating a module via Play new-module, you should remove any unnecessary cruft from your new module, as most often, not all of this is needed. Remove all unneeded directories or files, to make understanding the module as easy as possible.

Supporting Eclipse IDE

As `play eclipsify` does not work currently for modules, you need to set it up manually. A trick to get around this is to create and eclipsify a normal Play application, and then configure the build path and use "Link source" to add the `src/` directory of the plugin.

Building a flexible registration module

This is the first hands-on module. We will write one of the most needed functionalities of modern web applications in a module, a registration module featuring a double opt-in with a confirmation e-mail. The following tasks have to be covered by this module:

- A registration has to be triggered by a user
- An e-mail is send to an e-mail address including a URL to confirm registration
- A URL can be opened to confirm registration
- On confirmation of the registration by the user, the account should be enabled
- A Job to remove registrations, which are older than a specified time interval

You will also see that there are some problems in keeping your application modularized, which will have to be worked around and require understanding, as to when to put code into a module and when to put it into the application.

The source code of the example is available at `examples/chapter5/registration`.

Getting ready

Create an application, or you possibly already have an application, where you need the registration functionality. Inside this application should be a class resembling a user, which has an e-mail address property and a property which defines that the user is active. Create a new module via `play new-module registration` named registration.

As there will be two applications written in this example, the module as well as the application, an additional directory name will be prepended to any file path. In case of the module this will be "registration", where in the case of the real application this will be "register-app". This should sort out any possible confusion.

How to do it...

Starting with the plugin, it will feature a simple controller, which allows confirmation of the registration. This should be put into `registration/app/controllers/Registration.java`:

```
public class Registration extends Controller {

   public static void confirm(String uuid) {
      RegistrationPlugin.confirm(uuid);
      Application.index();
   }
}
```

Furthermore, this module has its own routes definitions, right in `registration/conf/routes`:

```
GET   /{uuid}/confirm   registration.Registration.confirm
```

The next step is to define an interface for the process of registration, which we will implement in the application itself. This file needs to be put in `registration/src/play/modules/registration/RegistrationService.java`:

```
public interface RegistrationService {
   public void createRegistration(Object context);
   public void triggerEmail(Object context);
   public boolean isAllowedToExecute(Object context);
   public void confirm(Object context);
}
```

Now the plugin itself can be implemented. Put it into `registration/src/play/modules/registration/RegistrationPlugin.java`:

```
public class RegistrationPlugin extends PlayPlugin {

   private static boolean pluginActive = false;
   private static RegistrationService service;

   public void onApplicationStart() {
      ApplicationClass registrationService = Play.classes.getAssignabl
eClasses(RegistrationService.class).get(0);

      if (registrationService == null) {
         Logger.error("Registration plugin disabled. No class
implements RegistrationService interface");
      } else {
         try {
            service = (RegistrationService) registrationService.
javaClass.newInstance();
            pluginActive = true;
         } catch (Exception e) {
            Logger.error(e, "Registration plugin disabled. Error when
creating new instance");
         }
      }
   }

   public void onEvent(String message, Object context) {
      boolean eventMatched = "JPASupport.objectPersisted".
equals(message);
      if (pluginActive && eventMatched && service.
isAllowedToExecute(context)) {
         service.createRegistration(context);
         service.triggerEmail(context);
      }
   }

   public static void confirm(Object uuid) {
      if (pluginActive) {
         service.confirm(uuid);
      }
   }
}
```

After creating the plugin the obligatory `play.plugins` file should not be forgotten, which must be put into `registration/src/play.plugins`:

```
900:play.modules.registration.RegistrationPlugin
```

Now the module is finished and you can create it via `play build-module` in the module directory.

In order to keep the whole application simple and the way it works together with the module, the whole application will be explained in this example.

So including the module in the `register-app/conf/dependencies.yml` is the first step. Running `play deps` after that is required:

```
require:
    - play
    - registration -> registration

repositories:
    - registrationModules:
        type:       local
        artifact:   "/absolute/path/to/registration/module"
        contains:
            - registration -> *
```

Then it needs to be enabled in the `register-app/conf/routes` file:

```
*   /registration      module:registration
```

The application itself consists of two entities, a user, and the registration entity itself:

```
@Entity
public class User extends Model {

   public String name;
   @Email
   public String email;
   public Boolean active;
}
```

The registration entity is also pretty short:

```
@Entity
public class Registration extends Model {

   public String uuid;
   @OneToOne
   public User user;
   public Date createdAt = new Date();
}
```

The controllers for the main application consist of one index controller and one controller for creating a new user. After the last one is executed, the logic of the registration plugin should be triggered:

```
public class Application extends Controller {

    public static void index() {
        render();
    }

    public static void addUser(User user) {
       user.active = false;
       if (validation.hasErrors()) {
          error("Validation errors");
       }

       user.create();
       index();
    }
}
```

When a user registers, a mail should be sent. So a mailer needs to be created, in this case at `register-app/app/notifier/Mails.java`:

```
public class Mails extends Mailer {

   public static void sendConfirmation(Registration registration) {
      setSubject("Confirm your registration");
      addRecipient(registration.user.email);
      String from = Play.configuration.getProperty("registration.mail.
from");
      setFrom(from);
      send(registration);
   }
}
```

A registration clean up job is also needed, which removes stale registrations once per week. Put it at `register-app/app/jobs/RegistrationCleanupJob.java`:

```
@Every("7d")
public class RegistrationCleanupJob extends Job {

   public void doJob() {
      Calendar cal = Calendar.getInstance();
      cal.add(Calendar.MONTH, -1);
```

```
        List<Registration> registrations = Registration.find("createdAt
< ?", cal.getTime()).fetch();
        for (Registration registration : registrations) {
            registration.delete();
        }
        Logger.info("Deleted %s stale registrations", registrations.
size());
    }
}
```

The last part is the actual implementation of the `RegistrationService` interface from the plugin. This can be put into `register-app/app/service/RegistrationServiceImpl.java`:

```
public class RegistrationServiceImpl implements RegistrationService {

    @Override
    public void createRegistration(Object context) {
        if (context instanceof User) {
            User user = (User) context;
            Registration r = new Registration();
            r.uuid = UUID.randomUUID().toString().replaceAll("-", "");
            r.user = user;
            r.create();
        }
    }

    @Override
    public void triggerEmail(Object context) {
        if (context instanceof User) {
            User user = (User) context;
            Registration registration = Registration.find("byUser",
user).first();
            Mails.sendConfirmation(registration);
        }
    }

    @Override
    public boolean isAllowedToExecute(Object context) {
        if (context instanceof User) {
            User user = (User) context;
            return !user.active;
        }
        return false;
    }
```

```
    @Override
    public void confirm(Object context) {
        if (context != null) {
            Registration r = Registration.find("byUuid", context.
toString()).first();
            if (r == null) {
                return;
            }
            User user = r.user;
            user.active = true;
            user.create();
            r.delete();
            Flash.current().put("registration", "Thanks for
registering");
        }
    }
}
```

There are only two remaining steps, the creation of two templates. The first one is the `register-app/app/views/Application/index.html` template, which features a registration mechanism and additional messages from the flash scope:

```
#{extends 'main.html' /}
#{set title:'Home' /}

${flash.registration}

#{form @Application.addUser()}
Name: <input type="text" name="user.name" /><br />
Email: <input type="text" name="user.email" /><br />
<input type="submit" value="Add" />
#{/form}
```

The last template is the one for the registration e-mail, which is very simple. And put the following under `register-app/app/views/notifier/Mails/sendConfirmation.txt`:

```
Hey there...

a very warm welcome.
We need you to complete your registration at

@@{registration.Registration.confirm(registration.uuid)}
```

Also you should configure your `register-app/conf/application.conf` file with a valid from e-mail address, which should be set in the sent out e-mails. Therefore, put the parameter `registration.mail.from=email@example.com` in your configuration.

Many things happened here in two applications. In order to make sure, you got it right and where to put what file, here is a list of each. This might help you not to be confused. First goes the module, which is in the registration directory:

```
registration/app/controllers/registration/Registration.java
registration/build.xml
registration/conf/routes
registration/lib/play-registration.jar
registration/src/play/modules/registration/RegistrationPlugin.java
registration/src/play/modules/registration/RegistrationService.java
registration/src/play.plugins
```

Note that the `play-registration.jar` will only be there, after you built the module. Your register application should consist of the following files:

```
register-app/app/controllers/Application.java
register-app/app/jobs/RegistrationCleanupJob.java
register-app/app/models/Registration.java
register-app/app/models/User.java
register-app/app/notifier/Mails.java
register-app/app/service/RegistrationServiceImpl.java
register-app/app/views/Application/index.html
register-app/app/views/main.html
register-app/app/views/notifier/Mails/sendConfirmation.txt
register-app/conf/application.conf
register-app/conf/routes
```

After checking you can start your application, go to the index page and enter a username and an e-mail address. Then the application will log the sent mail, as long as the mock mailer is configured. You can check the template and that the sent e-mail has an absolute URL to the configuration including an URL with `/registration/` in it, where the registration module is mounted. When visiting the link of the e-mail, you will be redirected to the start page, but there is a message at the top of the page. When reloading, this message will vanish, as it is only put in the flash scope.

How it works...

The preceding list, where to find which resources, should primarily show one thing. This example plugin is far from ideal. Indeed, it is very far from that. It should make you think about when you really need a plugin, and when you should include most of the stuff inside your actual project. It's time to discuss flaws and advantages of this example.

First, when you take a look at the `RegistrationPlugin` in the module, you will see a loosely coupled integration. The plugin searches for an implementation of the `RegistrationService` interface on startup, and will be marked active, if it finds such an implementation. The implementation in turn is completely independent of the module and therefore done in the application. When you take another look at the plugin, there is an invocation of the service, if a certain JPA event occurs, like the creation of a new object in this case. Again the actual implementation of the service should decide, whether it should be invoked or not. This happens via the `isAllowedToExecute()` method.

In case you are wondering, why there is no concrete implementation and especially no use of the registration entity: this is a limitation of the Play framework. You cannot use models in modules currently. You cannot import them in your plugin, as they are not visible at compile time of your module. Instead you would have to get them via the Play classloader. Using an interface, which is implemented in the application itself, was a design decision to circumvent exactly this problem with models. Still this means a certain decoupling from your concrete application and is not too good either. This is also the reason, why the `RegistrationCleanupJob` is also in the application instead of being at the module. Otherwise, it would also have to be configurable, like adding the time, how often it should run, and what entities should be cleaned. As opposed in this example, any user who has not registered might have to be cleaned as well. As all mailer classes are enhanced, they also do not fit into the module. The same applies for the e-mail template due to its flexibility, which implies it should not be packaged in any JAR files or external module, because it cannot be changed in the application then.

So, as you can see in this example, there is no clear and implicit structure. Even though the integration and writing of the plugin is nice, as it is completely decoupled from the storing of a user entity, it would make more sense in this example, to write the logic of this plugin directly in your application instead of building an own module with own packaging, as you could use all your models easily.

So when does writing a module make more sense? It makes more sense whenever you provide infrastructure code. This code is like your own persistence layer, specific templating, specific parsing of incoming data, external authentication or general integration of external services like a search engine. There are going to be enough examples inside this chapter to give a grip what belongs in a module and what belongs in your application.

A last important note and weakness of modules are currently the test capabilities. You have to write an application and use the module inside the applications tests. This is currently the only way to ensure the functionality of your module.

There's more...

Hopefully, this example gave you a glimpse of how to write a first module, but more importantly explains there is not always a use case for a module.

Think about when to write a module

If you sacrifice readability and structure of your application in favor of reuse, you should think about whether this is the way to go. Make sure your application is kept as simple as possible. It will be done this way the rest of this chapter.

Understanding events

As seen in the last example, events are a nice mechanism for attaching to certain actions without changing any of the existing code. In the last recipe the `RegistrationPlugin` was attached to the event of storing to create an entity. This is only one of many use-cases. This mechanism is quite seldom used in Play. There are some pros and cons about using events. It is always worth to think about whether using events is the right approach.

The source code of the example is available at `examples/chapter5/events`.

How to do it...

Triggering events yourself is absolutely easy. You can put the following anywhere in your code:

```
PlayPlugin.postEvent("foo.bar", "called it");
```

The first argument is an arbitrary string and should by convention consist of two words split by a dot. The first part should resemble some generic identifier. In order to correlate it easily this part is ideally the source of where it is coming from. The second part should be a unique identifier representing the reason for this event to be triggered. The last argument of the method can be any object. You have to take care of correct conversion and usage in the plugin yourself.

Receiving events is just as simple. Just write a plugin and implement the `onEvent` method:

```
    public void onEvent(String event, Object context) {
        if (event.startsWith("foo.")) {
            Logger.info("Some event: %s with content %s", event,
context);
        }
    }
```

How it works...

There are actually surprisingly few events predefined in the framework. Only the JPA plugin emits three different events:

- `JPASupport.objectDeleted` is posted when any object is deleted. You could possibly implement some sort of audit history this way. However, this is better done with hibernate in this case.

- `JPASupport.objectPersisted` is posted when a new object is created.
- `JPASupport.objectUpdated` is posted whenever an existing object is updated.

Currently, the event mechanism is not used that much. At the time of this writing, only the Ebean plugin made use of it.

If you take a look at the `PlayPlugin` class' `postEvent()` method, you will see the following snippet:

```
public static void postEvent(String message, Object context) {
    Play.pluginCollection.onEvent(message, context);
}
```

So, for each emitted event, the framework loops through all plugins and executes the `onEvent()` method. This means that all this is done synchronously. Therefore, you should never put heavy weight operations into this method, as this might block the currently running request.

Possibly, you could also make the business logic inside your `onEvent()` methods run asynchronously in order to speed things up and return to the caller method faster.

There's more...

Events are a nice and handy feature, but you should not use them overly, as they might make your code less readable. Furthermore, you should not mix this mechanism up with real event messaging solutions.

Think about multi-node environments

Events only get emitted inside the JVM. If you want a reliable way to collect and process your events in a multi-node environment, you cannot use the built-in event mechanism of the Play framework. You might be better off with a real messaging solution, which is in the *Integrating with messaging queues* recipe in the next chapter.

Managing module dependencies

As of play 1.2, a new dependency mechanism has been introduced to make the developers life much easier. It is based on Apache Ivy and thus does not require you to have an own repository. You can use all the existing infrastructure and libraries which are provided by the existing maven repositories.

The source code of the example is available at `examples/chapter5/dependencies`.

Getting ready

After a new application has been created, you will find a `conf/dependencies.yml` file in your application. The only dependency is the Play framework itself by default.

How to do it...

In *Chapter 2* the recipe *Writing your own renderRSS method as controller output* showed how to write out RSS feeds with the Rome library. Rome was downloaded back then and put into the `lib/` folder. When searching for Rome on a Maven repository search engine, you will find an entry like `http://mvnrepository.com/artifact/rome/rome/0.9`. All you need to is to extend the existing configuration to:

```
require:
    - play
    - rome 0.9
```

And rerun `play deps`. When checking your `lib/` directory you will see the Rome and JDOM jar files copied into it. JDOM is a dependency of Rome, so it is copied as well. If you remove the line with the Rome dependency again, you should run `play deps --sync` to make sure, both JAR files are removed again.

The next step is to put the actual dependency into your module instead of your application.

So create a `conf/dependencies.yml` in your module:

```
self: play -> depmodule 1.0

require:
    - rome 0.9
    - rome georss 0.1
```

Also note that the version of the module is set explicitly here to 1.0.

Now run `play deps` in the module directory to make sure the JAR files are put into the `lib/` directory.

Put the module dependency in your application `conf/dependencies.yml` file:

```
require:
    - play
    - play -> depmodule 1.0
```

How it works...

As you have probably found out while reading the preceding paragraphs, the dependencies are again defined as a YML file, similar to the fixtures feature of Play. Furthermore, the definition of module dependencies is a little bit different when compared to the normal applications as it has to have a `self:` part defined first.

Generally, the syntax of the name of dependencies is similar to the one of Maven. A Maven package contains a `groupId`, an `artifactId`, and a `version`. A dependency is written in the form of:

```
- $groupId $artifactId $version
```

However, if `groupId` and `artifactId` are the same, one may be omitted, as seen with the preceding Rome example.

There's more...

This recipe is rather short, because you will get used to this mechanism pretty quickly and it is already well documented.

Learn more about dependency management with play

As there are more options available, you should definitely read the dependency documentation at `http://www.playframework.org/documentation/1.2/dependency`. In particular, the parts about versioning and transitive dependencies should be read in order to make sure not to override dependencies from the framework itself. Also, if you are interested in Apache Ivy, you can read more about it at `http://ant.apache.org/ivy/`

Search for jar files in Maven repositories

If you are searching for jar files in public repositories, you should use one of the following sites:

`http://mvnrepository.com/`

`http://mavensearch.net/`

`http://www.jarvana.com/`

`http://search.mvnsearch.org/`

`http://www.mvnbrowser.com/` (also searches many other non-official Maven repositories)

See also

It is also very easy to create your own repositories for Play modules without creating any Maven repositories. See the recipe *Adding private module repositories* in this chapter for more.

Using the same model for different applications

A more frequently occurring problem might be the need for using the same model in different applications. A simple solution is to make the model layer a module itself.

The source code of the example is available at `examples/chapter5/module-model`.

How to do it...

So, create an application and a module:

```
play new app01
play new-module my-module-model
```

Change the `dependencies.yml` to include the module:

```
require:
    - play
    - modelModules -> my-module-model

repositories:
    - playmodelModules:
        type:       local
        artifact:   "/path/to/my-module-model/"
        contains:
            - modelModules -> *
```

If you just created your application like the one that we just saw, do not forget to add a database connection in your application configuration. For testing purposes use the in memory database via `db=mem`.

Put a User model into your module at `my-module-model/app/models/User.java`:

```
@Entity
public class User extends Model {

   public String name;
   public String login;
}
```

Use it like any other model in your controller inside of your application. You can even use all the class reloading features after changing the model.

How it works...

When talking about the directory layout of a module, you have already seen that it also features an `app/` directory. However, this directory is not included when a module is compiled and packaged into a JAR file via `play build-module`. This is intended and should never be changed. This means, a full module not only consists of a compiled jar library, but also of these sources, which are compiled dynamically, when the application is started.

As soon as a Java class is packaged into a JAR file, it cannot be changed anymore on the start of a Play application. The process of bytecode enhancement needs a compiled class file in the filesystem, as this file is modified during the enhancement. Every model class is enhanced, but there are many other classes as well, for example controllers and mailers.

This is also the reason why you cannot reference model classes in your plugin code inside the `src/` directory, although this would have been the quickest solution in the registration plugin example. At the time of module compilation they are not enhanced and if they are included in the JAR file, they will never be.

There's more...

The next step about modules is one of the more complex parts: bytecode enhancement.

Learn more about bytecode enhancers

Bytecode enhancement is quite a tricky process and definitely nothing quick to understand. To learn about integration into Play, which makes writing your own enhancers pretty simple, you should check the `play.classloading.enhancers` package, where several enhancers are already defined.

Check the modules for even more enhancers

If you need more usage example of bytecode enhancers, there are several modules making use of it. It is used mostly in order to resemble the finder methods in the model classes of different persistence layers. Examples for these are the Morphia, Ebean, Siena, and Riak modules.

See also

The next recipe will exclusively deal with two examples of bytecode enhancement in Play and should give you a first impression how to make use of it.

Understanding bytecode enhancement

This recipe should show you why it is important to understand the basic concepts of bytecode enhancement and how it is leveraged in the Play framework. The average developer usually does not get in contact with bytecode enhancement. The usual build cycle is compiling the application, and then using the created class files.

Bytecode enhancement is an additional step between the compilation of the class files and the usage by the application. Basically this enhancement step allows you to change the complete behavior of the application by changing what is written in the class files at a very low level. A common use case for this is aspect oriented programming, is where you add certain features to methods after the class has been compiled. A classical use case for this is the measurement of method runtimes.

If you have already explored the source code of the persistence layer you might have noticed the use of bytecode enhancement. This is primarily to overcome a Java weakness: static methods cannot be inherited with class information about the inherited class, which seems pretty logical, but is a major obstacle. You have already used the `Model` class in most of your entities. It features the nice `findAll()` method, or its simpler companion, the `count()` method. However, if you defined a User entity extending the Model class, all invocations of `User.findAll()` or `User.count()` will always invoke the `Model.findAll()` or `Model.count()`, which would never return any user entity specific data.

This is exactly the place where the bytecode enhancement kicks in. When starting your Play application, every class which inherits from the Model class is enhanced to return entity specific data, when the static methods from `findAll()` or `count()` are executed. As this is not possible with pure java, bytecode enhancement is used to circumvent this problem.

If you take a look at the `play.db.jpa.JPAEnhancer` class, you will see that the code which is inserted by the enhancer is actually written inside quotes and does not represent real code inside the Java class, which is checked during compilation by the compiler. This has the problem of being error prone, as typos are caught only when the actual bytecode enhancement happens or, even worse, when the code is executed. There is currently no good way around this.

The source code of the example is available at `examples/chapter5/bytecode-enhancement`.

Getting ready

The example in this recipe will make use of the search module, which features fulltext search capabilities per entity. Make sure, it is actually installed by adding it to the `conf/dependencies.yml` file. You can get more information about the module at `http://www.playframework.org/modules/search`. This example will enhance every class with a method, which returns whether the entity is actually indexed or not.

This could also be solved with the reflection API of course by checking for annotations at the entity. This should just demonstrate how bytecode enhancement is supposed to work. So create an example application which features an indexed and a not indexed entity.

Write the Test which should be put into the application:

```
public class IndexedModelTest extends UnitTest {

    @Test
    public void testThatUserIsIndexed() {
       assertTrue(User.isIndexed());
       assertTrue(User.getIndexedFields().contains("name"));
       assertTrue(User.getIndexedFields().contains("descr"));
       assertEquals(2, User.getIndexedFields().size());
    }

    @Test
    public void testThatOrderIndexDoesNotExist() {
       assertFalse(Order.isIndexed());
       assertEquals(0, Order.getIndexedFields().size());
    }
}
```

How to do it...

When you have written the preceding test, you see the use of two entities, which need to be modeled. First the `User` entity:

```
@Entity
@Indexed
public class User extends IndexedModel {

   @Field
   public String name;
   @Field
   public String descr;
}
```

In case you are missing the `Indexed` and `Field` annotations, you should now really install the search module, which includes these as described some time back in this chapter. The next step is to create the Order entity:

```
@Entity(name="orders")
public class Order extends IndexedModel {

   public String title;
   public BigDecimal amount;
}
```

Note the change of the order table name as order is a reserved SQL word. As you can see, both entities do not extend from model, but rather extend from `IndexedModel`. This is a helper class which is included in the module we are about to create now. So create a new module named `bytecode-module`. Create the file `bytecode-module/src/play.plugins` with this content:

```
1000:play.modules.searchhelp.SearchHelperPlugin
```

Create the `IndexedModel` class first in `bytecode-module/src/play/modules/searchhelp/IndexedModel.java`:

```
public abstract class IndexedModel extends Model {

   public static Boolean isIndexed() {
      return false;
   }

   public static List<String> getIndexedFields() {
      return Collections.emptyList();
   }
}
```

The next step is to create the bytecode enhancer, which is able to enhance a single entity class. So create `bytecode-module/src/play/modules/searchhelp/SearchHelperEnhancer.java`:

```
public class SearchHelperEnhancer extends Enhancer {

   @Override
   public void enhanceThisClass(ApplicationClass applicationClass)
throws Exception {
      CtClass ctClass = makeClass(applicationClass);

        if (!ctClass.subtypeOf(classPool.get("play.modules.searchhelp.
IndexedModel")) ||
              !hasAnnotation(ctClass, "play.modules.search.Indexed"))
{
            return;
        }

        CtMethod isIndexed = CtMethod.make("public static Boolean
isIndexed() { return Boolean.TRUE; }", ctClass);
        ctClass.addMethod(isIndexed);

        List<String> fields = new ArrayList<string>();
        for (CtField ctField : ctClass.getFields()) {
```

```
            if (hasAnnotation(ctField, "play.modules.search.Field")) {
               fields.add("\"" + ctField.getName() + "\"");
            }
         }

         String method;
         if (fields.size() > 0) {
            String fieldStr = fields.toString().replace("[", "").
replace("]", "");
            method = "public static java.util.List getIndexedFields() {
return java.util.Arrays.asList(new String[]{" + fieldStr + "}); }";
            CtMethod count = CtMethod.make(method, ctClass);
            ctClass.addMethod(count);
         }

         applicationClass.enhancedByteCode = ctClass.toBytecode();
         ctClass.defrost();
      }
   }
```

The last part is to create the plugin, which actually invokes the enhancer on startup of the application. The plugin also features an additional output in the status pages. Put the file into `bytecode-module/src/play/modules/searchhelp/SearchHelperPlugin.java`:

```
   public class SearchHelperPlugin extends PlayPlugin {

      private SearchHelperEnhancer enhancer = new SearchHelperEnhancer();

      @Override
      public void enhance(ApplicationClass applicationClass) throws
   Exception {
         enhancer.enhanceThisClass(applicationClass);
      }

      @Override
      public JsonObject getJsonStatus() {
         JsonObject obj = new JsonObject();
         List<ApplicationClass> classes = Play.classes.getAssignableClass
   es(IndexedModel.class);

         for (ApplicationClass applicationClass : classes) {
            if (isIndexed(applicationClass)) {
               List<String> fieldList = getIndexedFields(applicationCla
   ss);
               JsonArray fields = new JsonArray();
```

```
            for (String field :fieldList) {
               fields.add(new JsonPrimitive(field));
            }

            obj.add(applicationClass.name, fields);
         }
      }

      return obj;
   }


   @Override
   public String getStatus() {
      String output = "";

      List<ApplicationClass> classes = Play.classes.getAssignableClass
es(IndexedModel.class);

      for (ApplicationClass applicationClass : classes) {
         if (isIndexed(applicationClass)) {
            List<String> fieldList = getIndexedFields(applicationCla
ss);
            output += "Entity " + applicationClass.name + ": " +
fieldList +  "\n";
         }
      }

      return output;
   }

   private List<String> getIndexedFields(ApplicationClass
applicationClass) {
      try {
         Class clazz = applicationClass.javaClass;
         List<String> fieldList = (List<String>) clazz.
getMethod("getIndexedFields").invoke(null);
         return fieldList;
      } catch (Exception e) {}

      return Collections.emptyList();
   }
```

```
    private boolean isIndexed(ApplicationClass applicationClass) {
        try {
            Class clazz = applicationClass.javaClass;
            Boolean isIndexed = (Boolean) clazz.getMethod("isIndexed").
invoke(null);
            return isIndexed;
        } catch (Exception e) {}

        return false;
    }
}
```

After this, build the module, add the module dependency to the application which includes the test at the beginning of the recipe, and then go to the test page at `http://localhost:9000/@tests` and check if the test is running successfully.

How it works...

Though the module consists of three classes, only two should need further explanation. The `SearchHelperEnhancer` basically checks whether `@Indexed` and `@Field` annotations exist, and overwrites the statically defined methods of the `IndexedModel` class with methods that actually return true in case of `isIndexed()` and a list of indexed fields in case of `getIndexedFields()`.

As already mentioned in the overview of the modules, the `enhance()` method inside of any Play plugin allows you to include your own enhancers and is basically a one liner in the plugin, as long as you do type checking in the enhancer itself.

As you can see in the source, the code which is actually enhanced, is written as text string inside of the `CtMethod.make()` method in the enhancer. This is error prone, as typos or other mistakes cannot be detected at compile time, but only on runtime. Currently, this is the way to go. You could possibly try out other bytecode enhancers such as JBoss AOP, if this is a big show stopper for you. You can read more about JBoss AOP at `http://www.jboss.org/jbossaop`:

This recipe shows another handy plugin feature. The plugin also implements `getStatus()` and `getJsonStatus()` methods. If you run `play status` in the directory of your application while it is running, you will get the following output at the end:

```
SearchHelperPlugin:
~~~~~~~~~~~~~~~~~~
Entity models.User: [name, descr]
```

There's more...

As writing your own enhancers inside of your own plugins is quite simple, you should check out the different persistence modules, where enhancers are used for the finders. If you want to get more advanced, you should also check out the enhancement of controllers.

Overriding toString() via annotation

Peter Hilton has written a very good article on how one can configure the output of the `toString()` method of an entity with the help of an annotation by using bytecode enhancement. You can check it out at `http://www.lunatech-research.com/archives/2011/01/11/declarative-model-class-enhancement-play`.

See also

In the next chapter the recipe *Adding annotations via bytecode enhancements* will add annotations to classes via bytecode enhancement. This can be used to prevent repeating placing annotations all over your model classes.

Adding private module repositories

With the release of play 1.2 and its new dependency management it is very easy to have private repositories where you can store your own modules.

As the dependency management of Play is based on Apache Ivy, it is theoretically possible to use a Maven repository for this task, but often it is simpler to use a small share on your Intranet web server for such task.

Getting ready

You should have some location, where you can upload your modules. If you just do this for testing purposes, you can start a web server via Python

```
python -m SimpleHTTPServer
```

This will start a web server on port 8000 with the current directory you are in as document root.

How to do it...

When creating a new module, it is not necessary to set a version number in the modules `conf/dependencies.yml` file. However, if you do it, it helps you to keep module versions. So, for test reasons, go into one of your modules, set version to `0.1` and build the module. Then copy the created zip from the `dist/` directory into the directory, where you started the web server or alternatively in the directory of your real web server. Repeat the steps again, but this time create a module using `0.2` as version. Going to `http://localhost:8000` should now provide a listing with two zip files of your module.

Now add another repository to your application specific `conf/dependencies.yml` file:

```
require:
    - play
    - spinscale -> csv-module 0.1

# Custom repository
repositories:
    - spinscaleRepo:
        type:       http
        artifact:   "http://localhost:8000/[module]-[revision].zip"
        contains:
          - spinscale -> *
```

Now you can run `play dependencies` and the output should include this:

```
~ Installing resolved dependencies,
~
~    modules/csv-module-0.1
```

Normal JAR dependencies are put in the `lib/` directory of your application; however modules like the referenced csv-module are unpacked into the `modules/` directory inside of your Play application. Whenever you are starting your application now, all the modules in this directory are loaded automatically.

How it works...

The most important part is the format of the artifact in the dependencies file. It allows you to match arbitrary directory structures on the web server. Furthermore, you can possibly also have local repositories by using an artifact definition like the following:

```
artifact:   "${play.path}/modules/[module]-[revision]"
```

Always keep in mind not to use tabs in the YML files, but spaces when editing them. Furthermore, the artifact path has to be absolute.

If you want to remove a module, delete the entry from the dependencies file, and execute `play dependencies --sync` again to make sure all modules are removed from your application as well.

There's more...

Even though it is possible to have private repositories, you are of course encouraged to open source your modules.

Check the official documentation

There is an own page about dependency management where most of the possible options are written down, as there are still some more possibilities. Please read most of it at `http://www.playframework.org/documentation/1.2/dependency`.

Repositories with older versions

It is not simple to use any of the dependencies here with Play versions older than 1.2. There has been an effort at `http://code.google.com/p/play-repo/` - but it looks like it is not in active development anymore.

Preprocessing content by integrating stylus

If you have taken a look at the available modules on the Play framework site, there are at the time of this writing two modules, which help the developer to write CSS. On one the hand is the SASS module and on the other hand the less module.

Both modules address the same problem: On every change of the source file, either SASS or less follows some recompilation into the destination CSS file. Only the most up to date CSS file may be delivered to the client. In development mode every incoming request should always create the most up to date CSS output, where as in production it is sufficient to create those files seldom and cache them in order to increase performance.

The source code of the example is available at `examples/chapter5/stylus`.

Getting ready

In case you have never heard what SASS, less CSS, or stylus do, now would be the perfect time to learn. Check out the URL of each tool at the end of this recipe. In short, they are simple preprocessors, which parse an arbitrary definition of CSS similar content and create CSS content out of it.

This implies different behaviors in production and development mode. In development mode every incoming request should always create the most up to date CSS output, where as in production it is sufficient to create those files seldom and cache them in order to increase performance.

So, a HTTP request to the URI `/public/test.styl` should result in a pure CSS output, where the original content of the `test.styl` file was compiled and then returned to the client.

As usual, a test is the first point to start with, after installing stylus of course:

```
public class StylusCompilerTest extends UnitTest {

    @Test
    public void checkThatStylusCompilerWorks() throws Exception {
       StylusCompiler compiler = new StylusCompiler();
       File file = Play.getFile("test/test.styl");
       String result = compiler.compile(file);

       File expectedResultFile = Play.getFile("test/test.styl.
result");
       String expectedResult = FileUtils.readFileToString(expectedRes
ultFile);

       assertEquals(expectedResult, result);
    }
}
```

This simple test takes a prepared input file, compiles it, and checks whether the output is similar to an already parsed file. For simplicity, the file used in this test is the example from the stylus readme at `https://github.com/LearnBoost/stylus/blob/master/Readme.md`.

In case you are asking why only the compiler is tested, and not the whole plugin, including the preceding HTTP request: at the time of this writing there was a special handling for non controller resources, which could not be handled in functional tests. If you look at the source code of this recipe, you will see how an example functional test should look.

How to do it...

This plugin is rather short. It consists of two classes, first the `StylusCompiler` doing all the hard work. I skipped the creation of a new module and the `play.plugins` file in this case:

```
public class StylusCompiler {

   public String compile(File realFile) throws Exception {
      if (!realFile.exists() || !realFile.canRead()) {
```

```
            throw new FileNotFoundException(realFile + " not found");
         }
         String stylusPath = Play.configuration.getProperty("stylus.
executable", "/usr/local/share/npm/bin/stylus");

         File stylusFile = new File(stylusPath);
         if (!stylusFile.exists() || !stylusFile.canExecute()) {
            throw new FileNotFoundException(stylusFile + " not found");
         }

         Process p = new ProcessBuilder(stylusPath).start();
         byte data[] = FileUtils.readFileToByteArray(realFile);
         p.getOutputStream().write(data);
         p.getOutputStream().close();

         InputStream is = p.getInputStream();
         String output = IOUtils.toString(is);
         is.close();

         return output;
      }
   }
```

The second class is the plugin itself, which catches every request on files ending with `styl` and hands them over to the compiler:

```
   public class StylusPlugin extends PlayPlugin {

      StylusCompiler compiler = new StylusCompiler();

      @Override
      public void onApplicationStart() {
         Logger.info("Loading stylus plugin");
      }

       @Override
       public boolean serveStatic(VirtualFile file, Request request,
   Response response) {
          String fileEnding = Play.configuration.getProperty("stylus.
   suffix", "styl");
          if(file.getName().endsWith("." + fileEnding)) {
             response.contentType = "text/css";
             response.status = 200;
             try {
                String key = "stylusPlugin-" + file.getName();
```

```
                String css = Cache.get(key, String.class);
                if (css == null) {
                   css = compiler.compile(file.getRealFile());
                }

              // Cache in prod mode
              if(Play.mode == Play.Mode.PROD) {
                 Cache.add(key, css, "1h");
                 response.cacheFor(Play.configuration.getProperty("http.
cacheControl", "3600") + "s");
              }
              response.print(css);
            } catch(Exception e) {
                response.status = 500;
                response.print("Stylus processing failed\n");
                if (Play.mode == Play.Mode.DEV) {
                   e.printStackTrace(new PrintStream(response.out));
                } else {
                   Logger.error(e, "Problem processing stylus file");
                }
            }
            return true;
          }

       return false;
      }
}
```

How it works...

If the requested file ends with a certain configured suffix or `styl`, the response is taken care of in the `serveStatic()` method. If the content is not in the cache, the compiler creates it. If the system is acting in production mode, the created content is put in the cache for an hour, otherwise it is returned. Also, exceptions are only returned to the client in development mode, otherwise they are logged. As the return type of the method is Boolean, it should return true, if the plugin took care of delivering the requested resource.

The most notable thing about the stylus compiler is the way it is invoked. If you use stylus on the command line, you would execute it like `stylus < in.styl` in order to read the input from a file and print the output to the console, or more generally spoken, to `stdout`. As the < operator is a bash feature, we need to supply the input by writing the requested file into the output stream and reading the output from the input stream. This might be vice versa. This is what the `StylusCompiler` class mimics, when using the `ProcessBuilder` to invoke the external process.

Just to give you a feeling, how much a cache is needed: on my MacBook one hundred invocations on a small stylus file takes 16 seconds, including opening a new HTTP connection on each request. In production mode these hundred requests take less than one second.

A possible improvement for this plugin could be better handling of exceptions. The Play framework does a great job of exposing nice looking exceptions with exact error messages. This has not been accounted for in this example.

Stylus also supports CSS compression, so another improvement could be to build it into the compiler. However, this would merely be useful for production mode as it makes debugging more complex.

There's more...

There are lots of CSS preprocessors out there, so take the one you like most and feel comfortable with, as integration will be quite simple with the Play framework. Also you should take a look at the SASS, less, press, and greenscript modules of Play.

More information about CSS preprocessing

If you want to get more information about the CSS preprocessors mentioned in this recipe, go to `http://sass-lang.com/`, `http://lesscss.org/`, and `https://github.com/LearnBoost/stylus`.

Integrating Dojo by adding command line options

There is one last key aspect of modules, which has not been touched yet: the opportunity to add new command line options to modules.

This recipe utilizes the Dojo toolkit to show this feature. Dojo is one of the big JavaScript toolkits, which features tons of widgets, a simple interface, and a very easy start for object oriented developers. If you are using the standard distribution of Dojo together with a lot of widgets, there are many HTTP requests for all the widgets, as every widget is put into its own JavaScript file in a default Dojo installation. Many requests result in very slow application loading. Dojo comes with its own JavaScript optimizer called ShrinkSafe, which handles lots of things, like compressing the JavaScript code needed by your custom application into one single file as well as creating i18n files and compressing CSS code.

In order to use the ShrinkSafe capabilities, you need a developer build of Dojo. A configuration file is created, where you specify exactly which widgets are needed for this custom build. Only these referenced widgets are compressed into the single JavaScript file.

Such a precompilation step implies that there must be some way to download and compile the JavaScript before or during the Play application is running. You can develop without problems with a zero or partly optimized version of Dojo, because missing files are loaded at runtime. However, you should have a simple method to trigger the Dojo build process. A simple method like entering `play dojo:compile` on the command line.

The source code of the example is available at `examples/chapter5/dojo-integration`.

Getting ready

Create a new module called Dojo, and create any example application in which you include this module via your `depdencies.yml` file. Also put a Dojo version in your `application.conf` like this:

```
dojo.version=1.6.0
```

Just put the most up-to-date Dojo version as value in the key.

How to do it...

Your module is actually almost bare. The only file which needs to be touched is the `dojo/commands.py` file:

```
import urllib
import time
import tarfile
import os
import sys
import getopt
import shutil
import play.commands.modulesrepo

MODULE = 'dojo'

COMMANDS = ['dojo:download', 'dojo:compile', 'dojo:copy',
'dojo:clean']

dojoVersion = "1.6.0"
dojoProfile = "play"

def execute(**kargs):
    command = kargs.get("command")
    app = kargs.get("app")
    args = kargs.get("args")
    env = kargs.get("env")
```

```
    global dojoVersion
    global dojoProfile

    dojoVersion = app.readConf("dojo.version")

    try:
        optlist, args = getopt.getopt(args, '', ['version=',
'profile='])
        for o, a in optlist:
            if o in ('--version'):
                dojoVersion = a
            if o in ('--profile'):
                dojoProfile = a
    except getopt.GetoptError, err:
        print "~ %s" % str(err)
        print "~ "
        sys.exit(-1)

    if command == "dojo:download":
        dojoDownload()
    if command == "dojo:compile":
        dojoCompile()
    if command == "dojo:copy":
        dojoCopy()
```

The next three methods are helper methods to construct the standard naming scheme of Dojo directories and filenames including the specified version.

```
def getDirectory():
   global dojoVersion
   return "dojo-release-" + dojoVersion + "-src"

def getFile():
   return getDirectory() + ".zip"

def getUrl():
    global dojoVersion
    return "http://download.dojotoolkit.org/release-" + dojoVersion +
"/" + getFile()
```

All the following defined methods start with Dojo and represent the code executed for each of the command line options. For example, `dojoCompile()` maps to the command line option of `play dojo:compile`:

```
def dojoCompile():
    dir = "dojo/%s/util/buildscripts" % getDirectory()
    os.chdir(dir)
```

```
    os.system("./build.sh profileFile=../../../../conf/dojo-profile-
%s.js action=release" % dojoProfile)

def dojoClean():
    dir = "dojo/%s/util/buildscripts" % getDirectory()
    os.chdir(dir)
    os.chmod("build.sh", 0755)
    os.system("./build.sh action=clean" % dojoProfile)

def dojoCopy():
    src = "dojo/%s/release/dojo/" % getDirectory()
    dst = "public/javascripts/dojo"
    print "Removing current dojo compiled code at %s" % dst
    shutil.rmtree(dst)
    print "Copying dojo %s over to public/ directory" % dojoVersion
    shutil.copytree(src, dst)

def dojoDownload():
    file = getFile()

    if not os.path.exists("dojo"):
        os.mkdir("dojo")

    if not os.path.exists("dojo/" + file):
        Downloader().retrieve(getUrl(), "dojo/" + file)
    else:
        print "Archive already downloaded. Please delete to force new
download or specifiy another version"

    if not os.path.exists("dojo/" + getDirectory()):
        print "Unpacking " + file + " into dojo/"
        modulesrepo.Unzip().extract("dojo/" + file, "dojo/")
    else:
        print "Archive already unpacked. Please delete to force new
extraction"
```

After this is put into your `commands.py`, you can check whether it works by going into your application and running `play dojo:download`.

This will download and unpack the Dojo version you specified in the application configuration file. Now create a custom Dojo configuration in `conf/dojo-profile-play.js`:

```
dependencies = {
  layers:  [
    {
      name: "testdojo.js",
      dependencies: [
        "dijit.Dialog",
```

```
            "dojox.wire.Wire",
            "dojox.wire.XmlWire"
          ]
        }
    ],
    prefixes: [
      [ "dijit", "../dijit" ],
      [ "dojox", "../dojox" ],
    ]
};
```

Now you can run `play dojo:compile` and wait a minute or two. After this you have a customized version of Dojo, which still needs to be copied into the `public/` directory of your application in order to be visible. Just run `play dojo:copy` to copy the files.

The last step is to update the code to load your customized JavaScript. So edit your HTML files appropriately and insert the following snippet:

```
<script src="@{'/public/javascripts/dojo/dojo/dojo.js'}" type="text/
javascript" charset="utf-8"></script>

<script src="@{'/public/javascripts/dojo/dojo/testdojo.js'}"
type="text/javascript" charset="utf-8"></script>

<script type="text/javascript">
  dojo.require("dijit.Dialog")
  dojo.require("dojox.wire.Wire")
</script>
```

How it works...

Before explaining how this all works, you should make sure that an optimized build is actually used. You can connect to the controller, whose template includes the Dojo specific snippet that we saw some time back in this chapter. If you open your browser diagnostics tools, like Firebug for Mozilla Firefox, or the built in network diagnosis in Google Chrome, you should see the loading of only two JavaScript files, one being `dojo.js` itself, and the other `testdojo.js`, or however you named it in the profile configuration. If you require some Dojo module, which was not added in the build profile, you will see another HTTP request asking for it. This is ok for development, but not for production.

Now it is time to explain what actually happens in the Python code. The `commands.py` file does not have too much logic itself. If there is no `dojo.version` parameter specified in the application configuration, it will use 1.6.0, which was the current version at the time of writing. Furthermore, a default profile named Play is defined, which is used for the profile configuration file in the `conf/` directory. You can overwrite the version as well as the profile to be used via the `--version` or the `--profile` command line parameters if you need to.

The `execute()` method parses the optional command line parameters and executes one of the four possible commands.

The `getDirectory()`, `getFile()` and `getUrl()` methods are simple helpers to create the correct named file, directory or download URL of the specified Dojo version.

The `dojoCompile()` method gets executed when entering `play dojo:compile` and switches the working directory to the `util/buildscripts` directory of the specified Dojo version. It then starts a compilation using either the specified or default profile file in the `conf/` directory.

The `dojoClean()` method gets executed when entering `play dojo:clean` and triggers a clean inside of the `dojo/` directory, but will not wipe out any files copied to the `public/` folder. This command is not really necessary, as the code called in `dojoCompile()` already does this as well.

The `dojoCopy()` gets executed when entering `play dojo:copy` and method copies the compiled and optimized version which is inside the releases directory of the Dojo source to the `public/dojo` directory. Before copying the new Dojo build into the public directory, the old data is deleted. So if you stop the execution of this task, you might not have a complete Dojo installation in your public directory.

The `dojoDownload()` method gets executed when entering `play dojo:download` and downloads the specified Dojo version and extracts it into the `dojo/` directory in the current application. It uses the ZIP file, because the unzip class included in Play is operating system independent. As ZIP files do not store permissions, the `build.sh` shell script has to be set as executable again. In order to save bandwidth it only downloads a version if it is not yet in the `dojo/` directory.

There's more...

If you know Python a little bit, you have unlimited possibilities in your plugins, such as generating classes or configuration files automatically or validating data like internationalization data.

More about Dojo

The Dojo JavaScript toolkit is worth a look or two. Go to `http://dojotoolkit.org/` or better yet to a site with tons of examples to view and copy, which is `http://dojocampus.org/`.

Create operating system independent modules

When watching closely, you will see that the module created in this recipe only works on non Unix based operating systems such as MacOS or Linux, as it executes the `build.sh` script directly. In order to be independent of the operating system you should always check the platform you are on and act appropriate. For example, getting the value of `os.name` returns `posix` for both Linux and Mac OS and could be used to differentiate operating systems.

More ideas for command support

Whenever you need to create configuration files from existing data in your application, this is the way to go. An example might be the Solr module in the next chapter. If you could create the needed XML configuration by typing `play solr:xml`, that would prevent some configuration mistakes. You could iterate over all model classes and check whether they have search fields configured and output these in the correct format.

6
Practical Module Examples

In this chapter, we will cover:

- Adding annotations via bytecode enhancement
- Implementing your own persistence layer
- Integrating with messaging queues
- Using Solr for indexing
- Writing your own cache implementation

Introduction

The last chapter introduced you to the basics of writing modules. This chapter will show some examples used in productive applications. It will show an integration of an alternative persistence layer, how to create a Solr module for better search, and how to write an alternative distributed cache implementation among others. You should have read the basic module chapter before this one, if you are not familiar with modules.

Adding annotations via bytecode enhancement

If you remember the JSON and XML plugin written in *Chapter 5*, which was called API plugin, there was a possible improvement at the end of the recipe: adding the XML annotations is cumbersome and a lot of work, so why not add them automatically, so every entity defined in your application does not need the usually required annotations for XML processing with JAXB.

The source code of the example is available at `examples/chapter6/bytecode-enhancement-xml`.

Getting ready

As usual, write a test first, which actually ensures that the annotations are really added to the model. In this case they should not have been added manually to the entity, but with the help of bytecode enhancement:

```
public class XmlEnhancerTest extends UnitTest {

    @Test
    public void testThingEntity() {
       XmlRootElement xmlRootElem = Thing.class.
getAnnotation(XmlRootElement.class);
       assertNotNull(xmlRootElem);
       assertEquals("thing", xmlRootElem.name());

       XmlAccessorType anno = Thing.class.
getAnnotation(XmlAccessorType.class);
       assertNotNull(anno);
       assertEquals(XmlAccessType.FIELD, anno.value());
    }
}
```

All this test does is to check for the `XmlAccessorType` and the `XmlRootElement` annotations inside the `Thing` entity. The `Thing` class looks like any normal entity in the following example:

```
@Entity
public class Thing extends Model {
   public String foo;
   public String bar;

   public String toString() {
      return "foo " + foo + " / bar " + bar;
   }
}
```

How to do it...

As most of the work has already been done in the recipe about JSON and XML, we will only line out the differences in this case and create our own module. So create a module, copy the `play.plugins` file, the `ApiPlugin`, and `RenderXml` class into the `src/` folder, and create a new `XmlEnhancer` as shown in the following code:

```
public class XmlEnhancer extends Enhancer {

   @Override
   public void enhanceThisClass(ApplicationClass applicationClass)
throws Exception {
        CtClass ctClass = makeClass(applicationClass);

        if (!ctClass.subtypeOf(classPool.get("play.db.jpa.JPABase")))
{
            return;
        }

        if (!hasAnnotation(ctClass, "javax.persistence.Entity")) {
            return;
        }

        ConstPool constpool = ctClass.getClassFile().getConstPool();
        AnnotationsAttribute attr = new AnnotationsAttribute(constpo
ol, AnnotationsAttribute.visibleTag);

        if (!hasAnnotation(ctClass, "javax.xml.bind.annotation.
XmlAccessorType")) {
           Annotation annot = new Annotation("javax.xml.bind.
annotation.XmlAccessorType", constpool);
           EnumMemberValue enumValue = new EnumMemberValue(constpool);
           enumValue.setType("javax.xml.bind.annotation.
XmlAccessType");
           enumValue.setValue("FIELD");
           annot.addMemberValue("value", enumValue);
           attr.addAnnotation(annot);
           ctClass.getClassFile().addAttribute(attr);
        }

        if (!hasAnnotation(ctClass, "javax.xml.bind.annotation.
XmlRootElement")) {
           Annotation annot = new Annotation("javax.xml.bind.
annotation.XmlRootElement", constpool);
           String entityName = ctClass.getName();
           String entity = entityName.substring(entityName.
lastIndexOf('.') + 1).toLowerCase();
           annot.addMemberValue("name", new StringMemberValue(entity,
constpool));
           attr.addAnnotation(annot);
           ctClass.getClassFile().addAttribute(attr);
        }

        applicationClass.enhancedByteCode = ctClass.toBytecode();
        ctClass.defrost();
   }
}
```

Finally, add loading of the enhancer to your plugin:

```
public class ApiPlugin extends PlayPlugin {
   ...
   private XmlEnhancer enhancer = new XmlEnhancer();
   ...

   public void enhance(ApplicationClass applicationClass) throws
Exception {
       enhancer.enhanceThisClass(applicationClass);
    }
   ...
}
```

From now on whenever the application starts, all the models used are enhanced automatically with the two annotations.

How it works...

In order to check whether the module actually works, you can fire up the preceding unit test written.

As most of the code has already been written, the only part which needs a closer look is actually the `XmlEnhancer`. The `XmlEnhancer` first checks whether the application class is an entity, otherwise it returns without doing any enhancement.

The next step is to check for the `@XmlAccessorType` annotation, which must be set to field access. As the `XmlAccessType` is actually an enum, you have to create an `EnumMemberValue` object, which then gets added to the annotation.

The last step is to add the `@XmlRootElement` annotation, which marks the class for the JAXB marshaller to parse it. The name of the entity in lowercase is used as the root element name. If you want to change it, you can always use the annotation at the model and overwrite it. Here a `StringMemberValue` object is used, as the annotation takes a string as argument.

There's more...

Bytecode enhancement always helps to make your application easier, by possibly shortening error prone tasks such as adding annotations, or doing things which are not possible with standard Java, such as changing behavior of static methods. However, using bytecode enhancement means that you are doing something not visible at source code level. You should use this powerful feature only, if really needed, as it reduces the readability of your project. Do not forget to document these features clearly.

Javassist documentation

The Javassist documentation is actually not the easiest to read; however, it often helps because there are not too many examples floating around on the Internet. You can check most of it at `http://www.javassist.org` as well as some more links to introductions at `http://www.jboss.org/javassist`.

Implementing your own persistence layer

Usually as a developer you are choosing a framework (be it a web framework or for any type of application development) because it already delivers most of the part's you need. Typically, a standard requirement is the support of a persistence layer or technology you are going to use for your project. However, there might be cases where you have to create your own persistence layer, for example, if you are using a proprietary or in-house developed solution or if you need access for a new technology like one of the many available NoSQL databases. There are several steps which need to be completed. All of these are optional; however, it makes sense to develop your own persistence layer as similar to other persistence layers in Play, so you actually make sure it fits best in the concept of Play and is easily understood by most of the users. These steps are:

- Active record pattern for your models including bytecode enhancement for finders
- Range queries, which are absolutely needed for paging
- Having support for fixtures, so it is easy to write tests
- Supporting the CRUD module when possible
- Writing a module which keeps all this stuff together

In this recipe, simple CSV files will be used as persistence layer. Only `Strings` are supported along with `Referencing` between two entities. This is what an entity file like `Cars.csv` might look like:

```
"1"   "BMW"   "320"
"2"   "VW"   "Passat"
"3"   "VW"   "Golf"
```

The first column is always the unique ID, whereas the others are arbitrary fields of the class. Referencing works like the example in this `User.csv` file:

```
"1"   "Paul"   "#Car#1"
```

The connection to the BMW car above is done via the type and the ID by using additional hashes. As you might have already seen, the shown CSV files do not have any fieldnames. Currently, this persistence layer relies on the order read out of the class fields, which of course does not work when you change the order of fields add or remove other fields. This system is not ready for changes, but it works if you do not need such features as in this example.

The source code of the example is available at `examples/chapter6/persistence`.

Getting ready

In order to have a running example application, you should create a new application, include the module you are going to write in your configuration, and create two example classes along with their CRUD classes. So you should also include the CRUD module. The example entities used here are a user and Car entity. In case you are wondering about the `no args` constructor, it is needed in order to support fixtures:

```
public class Car extends CsvModel {

   public Car() {}
   public Car(String brand, String type) {
      this.brand = brand;
      this.type = type;
   }

   public String brand;
   public String type;

   public String toString() {
      return brand + " " + type;
   }
}
```

Now the user:

```
public class User extends CsvModel {

   public String name;
   public Car currentCar;

   public String toString() {
      return name + "/" + getId();
   }
}
```

In order to show the support of fixtures, it is always good to show some tests using it:

```
public class CsvTest extends UnitTest {

   private Car c;

   @Before
   public void cleanUp() {
      Fixtures.deleteAllModels();
      CsvHelper.clean();
```

```
        Fixtures.loadModels("car-data.yml");
        c = Car.findById(1L);
    }

    // Many other tests

    @Test
    public void readSimpleEntityById() {
        Car car = Car.findById(1L);
        assertValidCar(car, "BMW", "320");
    }

    @Test
    public void readComplexEntityWithOtherEntites() {
        User u = new User();
        u.name = "alex";
        u.currentCar = c;
        u.save();

        u = User.findById(1L);
        assertNotNull(u);
        assertEquals("alex", u.name);
        assertValidCar(u.currentCar, "BMW", "320");
    }

    // Many other tests not put in here

    private void assertValidCar(Car car, String expectedBrand, String
expectedType) {
        assertNotNull(car);
        assertEquals(expectedBrand, car.brand);
        assertEquals(expectedType, car.type);
    }
}
```

The YAML file referenced in the test should be put in `conf/car-data.yml`. It includes the data of a single car:

```
Car(c1):
  brand: BMW
  type: 320
```

As you can see in the tests, referencing is supposed to work. Now let's check how the plugin is built together.

How to do it...

You should already have created a csv module, adapted the `play.plugins` file to specify the `CsvPlugin` to load, and started to implement the `play.modules.csv.CsvPlugin` class by now:

```
public class CsvPlugin extends PlayPlugin {

   private CsvEnhancer enhancer = new CsvEnhancer();

  public void enhance(ApplicationClass applicationClass) throws
Exception {
       enhancer.enhanceThisClass(applicationClass);
    }

   public void onApplicationStart() {
      CsvHelper.clean();
   }

    public Model.Factory modelFactory(Class<? extends Model>
modelClass) {
      if (CsvModel.class.isAssignableFrom(modelClass)) {
         return new CsvModelFactory(modelClass);
      }
     return null;
    }
}
```

Also, this example heavily relies on OpenCSV, which can be added to the `conf/dependencies.yml` file of the module (do not forget to run `play deps`):

```
self: play -> csv 0.1

require:
    - net.sf.opencsv -> opencsv 2.0
```

The enhancer is pretty simple because it only enhances two methods, `find()` and `findById()`. It should be put into your module at `src/play/modules/csv/CsvEnhancer.java`:

```
public class CsvEnhancer extends Enhancer {

   public void enhanceThisClass(ApplicationClass applicationClass)
throws Exception {
        CtClass ctClass = makeClass(applicationClass);
```

```
        if (!ctClass.subtypeOf(classPool.get("play.modules.csv.
CsvModel"))) {
            return;
        }

        CtMethod findById = CtMethod.make("public static play.modules.
csv.CsvModel findById(Long id) { return findById(" + applicationClass.
name + ".class, id); }", ctClass);
        ctClass.addMethod(findById);

        CtMethod find = CtMethod.make("public static play.modules.
csv.CsvQuery find(String query, Object[] fields) { return find(" +
applicationClass.name + ".class, query, fields); }", ctClass);
        ctClass.addMethod(find);

        applicationClass.enhancedByteCode = ctClass.toBytecode();
        ctClass.defrost();
    }
}
```

The enhancer checks whether a `CsvModel` class is handed over and enhances the `find()` and `findById()` methods to execute the already defined methods, which take the class as argument. The `CsvModel` class should be put into the module at `src/play/modules/csv/` and should look like the following:

```
public abstract class CsvModel implements Model {

   public Long id;

  // Getter and setter for id omitted
  …

   public Object _key() {
      return getId();
   }

   public void _save() {
      save();
   }

   public void _delete() {
      delete();
   }
```

```
    public void delete() {
        CsvHelper helper = CsvHelper.getCsvHelper(this.getClass());
        helper.delete(this);
    }

    public <T extends CsvModel> T save() {
        CsvHelper helper = CsvHelper.getCsvHelper(this.getClass());
        return (T) helper.save(this);
    }

    public static <T extends CsvModel> T findById(Long id) {
        throw new UnsupportedOperationException("No bytecode
enhancement?");
    }

    public static <T extends CsvModel> CsvQuery find(String query,
Object ... fields) {
        throw new UnsupportedOperationException("No bytecode
enhancement?");
    }

    protected static <T extends CsvModel> CsvQuery find(Class<T> clazz,
String query, Object ... fields) {
      // Implementation omitted
    }

    protected static <T extends CsvModel> T findById(Class<T> clazz,
Long id) {
        CsvHelper helper = CsvHelper.getCsvHelper(clazz);
        return (T) helper.findById(id);
    }
}
```

The most important part of the `CsvModel` is to implement the `Model` interface and its methods `save()`, `delete()`, and `_key()`—this is needed for CRUD and fixtures. One of the preceding find methods returns a query class, which allows restricting of the query even further, for example with a limit and an offset. This query class should be put into the module at `src/play/modules/csv/CsvQuery.java` and looks like this:

```
public class CsvQuery {

    private int limit = 0;
    private int offset = 0;
    private CsvHelper helper;
    private Map<String, String> fieldMap;
```

```
    public CsvQuery(Class clazz, Map<String, String> fieldMap) {
       this.helper = CsvHelper.getCsvHelper(clazz);
       this.fieldMap = fieldMap;
    }

    public CsvQuery limit (int limit) {
       this.limit = limit;
       return this;
    }

    public CsvQuery offset (int offset) {
       this.offset = offset;
       return this;
    }

    public <T extends CsvModel> T first() {
       List<T> results = fetch(1,0);
       if (results.size() > 0) {
          return (T) results.get(0);
       }
       return null;
    }

    public <T> List<T> fetch() {
       return fetch(limit, offset);
    }

    public <T> List<T> fetch(int limit, int offset) {
       return helper.findByExample(fieldMap, limit, offset);
    }
}
```

If you have already used the JPA classes from Play, most of you will be familiar from a user point of view. As in the `CsvModel` class, most of the functionality boils down to the `CsvHelper` class, which is the core of this module. It should be put into the module at `src/play/modules/csv/CsvHelper.java`:

```
public class CsvHelper {

   private static ConcurrentHashMap<Class, AtomicLong> ids = new
ConcurrentHashMap<Class, AtomicLong>();
   private static ConcurrentHashMap<Class, ReentrantLock> locks = new
ConcurrentHashMap<Class, ReentrantLock>();
   private static ConcurrentHashMap<Class, CsvHelper> helpers = new
ConcurrentHashMap<Class, CsvHelper>();
```

```
    private static final char separator = '\t';
    private Class clazz;
    private File dataFile;

    private CsvHelper(Class clazz) {
       this.clazz = clazz;
       File dir = new File(Play.configuration.getProperty("csv.path",
"/tmp"));
       this.dataFile = new File(dir, clazz.getSimpleName() + ".csv");

       locks.put(clazz, new ReentrantLock());
       ids.put(clazz, getMaxId());
    }

    public static CsvHelper getCsvHelper(Class clazz) {
       if (!helpers.containsKey(clazz)) {
          helpers.put(clazz, new CsvHelper(clazz));
       }
       return helpers.get(clazz);
    }

    public static void clean() {
       helpers.clear();
       locks.clear();
       ids.clear();
    }
```

The next method definitions include the data specific functions to find, delete, and save arbitrary model entities:

```
    public <T> List<T> findByExample(Map<String, String> fieldMap, int
limit, int offset) {
       List<T> results = new ArrayList<T>();

      // Implementation removed to save space

       return results;
    }

    public <T extends CsvModel> void delete(T model) {
      // Iterates through csv, writes every line except
      // the one matching the id of the entity
    }
```

```
    public void deleteAll() {
      // Delete the CSV file
    }

    public <T extends CsvModel> T findById(Long id) {
       Map<String, String> fieldMap = new HashMap<String, String>();
       fieldMap.put("id", id.toString());      List<T> results =
findByExample(fieldMap, 1, 0);
       if (results.size() > 0) {
          return results.get(0);
       }
       return null;
    }

    public synchronized <T extends CsvModel> T save(T model) {

     // Writes the entity into the file
     // Handles case one: Creation of new entity
     // Handles case two: Update of existing entity

       return model;
    }
```

The next methods are private and needed as helper methods. Methods for reading entity files, for creating an object from a line of the CSV file, as well as the reversed operation, which creates a data array from an object, are defined here. Furthermore, file locking functions and methods to find out a the next free id on entity creation are defined here:

```
    private List<String[]> getEntriesFromFile() throws IOException {
       // Reads csv file to string array
    }

    private <T extends CsvModel> String[] createArrayFromObject(T
model, String id) throws IllegalArgumentException,
IllegalAccessException {
     // Takes and object and creates and array from it
     // Dynamically converts other CsvModels to something like
     // #Car#19
    }

    private <T extends CsvModel> T createObjectFromArray(String[] obj)
throws InstantiationException, IllegalAccessException {
     // Takes an array and creates an object from it
     // Dynamically loads #Car#19 to the real object
```

```
        return model;
    }

    private void getLock() {
     // helper method to create a lock for the csv file
    }

    private void releaseLock() {
     // helper method to remove a lock for the csv file
    }

    private synchronized void moveDataFile(File file) {
     // Replaces the current csv file with the new one
    }

    private Long getNewId() {
        return ids.get(clazz).addAndGet(1);
    }

    private AtomicLong getMaxId() {
      // Finds the maximum id used in the csv file
        return result;
    }

}
```

The last class file needed is the `CsvModelFactory`, which has been referenced in the `CsvPlugin` already. This is needed to support the CRUD plugin and have correct lookups of entities when referencing them in the GUI, for example in select boxes. As this class is almost similar to `JpaModelLoader` used by JPA, you can directly look into its source as well. The interface will be explained in some time.

How it works...

In order to understand what is happening in the source, you should build the module, include it together with the crud module in an example application, create some test classes (such as the `User` and `Car` entities as we did some time back), and click around in the CRUD module.

The `CsvHelper` is the core of the module. It uses HashMaps internally to hold some state per class, like the current ID to use when creating a new entity. Furthermore, it uses a lock in order to write any CSV file and holds `CsvHelper` instances per model entity. The main method in order to find entities is the `findByExample()` method, which allows the developer to hand over a map with fields containing specific values. This method loops through all entries and checks whether there are any entries matching the fields and their corresponding value. The `findById()` method for example is just a `findByExample()` invocation with a search for a specific ID value. Saving objects is either appending them to the file or alternatively replacing them in the CSV file.

Some persistence layer APIs feature functions to map Java objects to the persistence layer. Naturally these functions heavily differ from the library you are using. The Morphia/MongoDB module for example uses Morphia, which already includes much of this mapping functionality and helps to keep the glue code between Play and the external library as small as possible. Whenever you implement a persistence layer make sure you are choosing a library, where in the best case you only need to extend it with Play functionality instead of writing it from scratch like this one. It gets very complex to support, as the following things can pose serious problems, and do in this concrete example:

- Multiple accesses from different threads: When you take a closer look at the implementation, it is not thread safe at all. If two threads add a new entity each at the same time, then the added entity of the first thread will be lost after the second thread has stored its entity. This is because both threads take a backup from the data when both entities were not yet added.
- No referential integrity: This example has no referential integrity. If you remove a car, which is referenced in another CSV file, no one will notice.
- Circular dependencies: It is possible to bring down this application in an endless loop by referencing entity a in entity b and vice versa.
- Single node: All the data is stored on a single node. This is not scalable.
- Complex object support: Currently everything is stored as a string. This is not really a solution for number focused applications. However, many web applications are number specific, for example e-commerce or statistical ones.
- Collection support: This is a major problem. Of course you want to be able to store collections, such as `@OneToMany` and `@ManyToOne` annotations in JPA, with your persistence layer. A basic implementation in this example could be a string such as `#Car#5,#Car#6` in the string representation of another entity. However, there is no implementation done for this here.

In most cases most of the preceding problems should be solved by your database or the driver you are using. Otherwise, you should split this code away from your Play framework support and provide it as an own library.

As a last step it makes a lot of sense to take a more exact look at the interfaces provided by the Play framework in order to have support for CRUD and fixtures. Take a look at the `play.db.Model` interface, which incorporates all the other interfaces needed.

Every model class, like the `CsvModel`, should implement the `Model` interface itself. The interface makes sure there are methods provided for saving, deleting entities, and as getting a unique key.

The `Factory` interface referenced at the `CsvModelFactory` provides methods needed for finding, counting, and searching entities. The `keyName()`, `keyType()`, and `keyValue()` methods return different information about the unique key of a model class. In this case the name of the unique key field is `id`, the type is `Long`, and the value can be an arbitrary long out of the CSV file. The `findById()` method provides a generic way to find an entity with any unique key. The `fetch()` method is the most complex to be implemented. Fetching allows you to page through results by setting a limit and offset parameter. You can optionally define only to return certain fields, add ordering (direction and fields to order by), and add additional filtering by a `where` clause. This looks a little bit SQL centric, but can be implemented for any persistence layer. In order to support search you can additionally provide a list of fields to search in along with a keyword to search for. This allows you to do a search over on more than one field. Make sure that your database system does not suffer severe performance problems because of this. The `count()` method basically does the same as `fetch()`, but it does not need limits, offsets, or order information in order to run. The `deleteAll()` method deletes all data of all entities. This is especially important for testing in order to make sure to start with zero entries in your persistence layer. The last method to be implemented is `listProperties()`, which returns a representation of all properties of an entity. The representation includes a lot of metadata, which helps the CRUD controllers to display data.

This `Property` class also consists of several important fields. The name, type, and field properties resemble data of the field in the entity and are mostly read out of it. You could possibly read another name out of an annotation if you wanted. The `isSearchable` Boolean marks the field as being searchable. The JPA plugin uses this for only making fields searchable, which have this attribute set. The `isMultiple` Boolean marks a field as a collection. The `isRelation` Boolean should be used when the field is also a model entity. The last Boolean is the `isGenerated` Boolean, which marks a field you cannot set manually. This is often the unique key of an entity because it is being generated by the persistence layer, but could possibly also have fields like the creation date or the last modified date. The `relationType` class holds the class of related fields. If you have a one-to-one connection, it would be the entity type of the other class. If you have a one-to-many collection, it would be the entity type of the lists contents and not the list itself. This is important for select boxes filled with options of other entities in CRUD controllers. The last field to be explained is the `choices` field. It is responsible for the content in select boxes referencing entities. You can put an appropriate query in the `list()` method of the `Choices` interface and the returned results will then occur in the select boxes. For the sake of simplicity you can return all entities here. This is also done in the example code.

There is a last interface defined in the `Model`. The `BinaryField` interface allows you to treat binaries special. The `play.db.jpa.Blob` class is actually doing this. It stores files on the hard disk as they cannot be stored inside any database supporting JPA.

The factory itself is always returned in the plugin, where the developer may decide whether the plugin is responsible for the class asked for.

There's more...

As documentation about support for the CRUD module is sparse currently, the quickest thing to get up and running is to look at existing plugins.

Check the JPA and Morphia plugins

If you want to know more, these two plugins are the place to start in order to get some more information about implementing own persistence layers. Also, the crudsiena plugin is worth taking a look at.

Integrating with messaging queues

As soon as your system and application landscape widens, you try to create components which are independent of each other. One of the first steps is to create an asynchronous messaging architecture. This allows decoupling of components, but more importantly, you can also be implementation-independent by simply using standards of data exchange. There exist messaging standards such as AMQP or STOMP, which can be read by any application. The most well known, defined API for messaging in the Java world is **JMS**, the **Java Message Service**.

In this example Apache ActiveMQ will be used to let an arbitrary amount of Play instances communicate with each other. The communication will be done via a publish-subscribe mechanism which is called a topic in JMS terms. In order to get a short overview, there are two mechanisms, which could be chosen for this task. First a producer-consumer pattern, second a publish-subscribe pattern. The producer-consumer pattern however requires not more than one listener which is an often needed use-case, but not feasible in this example. Think a little bit, as to which of these patterns you really need for your application.

In case you are asking why another technology is needed for this, it is because a distributed cache can possibly be accessed already by all Play nodes. This is right, but you would have to implement a reliable publish-subscribe infrastructure yourself using memcached. This is quite a lot of work and actually the wrong tool for this task.

The source code of the example is available at `examples/chapter6/messaging`.

Getting ready

In order to understand what happens in this example you should have taken a look at the chat example application in the `samples-and-tests` directory of your Play installation. This example will change the functionality of the websocket implementation of the chat application. Websockets are a new standard defined by W3C in order to support long running connections and to be able to communicate in both directions, from client to server and from server to client, which is not possible with pure HTTP. This is one of the biggest drawbacks in regular HTTP communication. Basically, the chat application takes a message from one user and directs it to all others currently connected.

This works as long as only one application is running. As soon as more than one Play node is running, you need communication between these nodes. Every node has to:

- Publish a message, whenever an action at this node happens
- Receive messages at any time and feed them to their connected users

This sounds pretty simple with a publish-subscribe architecture. Every node subscribes and publishes to the same queue and whenever a received message on this queue has not been sent by the node itself, it has to be sent to all connected users on the node.

You need a running installation of Apache ActiveMQ. The assumption in this example is always a running ActiveMQ system on localhost. Installation of ActiveMQ is straightforward; you just grab a copy at `http://activemq.apache.org/` and run `bin/activemq start`. Either an error message is written out on the console including the command you should execute next, or a process ID along with the message that ActiveMQ was started. You can check at `http://localhost:8161` whether your ActiveMQ instance is running.

The next step is to write a unit test, which emulates the behavior of a second node. The unit test checks whether the plugin sends messages to ActiveMQ and also whether it correctly receives messages, when other nodes are sending messages:

```
public class ActiveMqTest extends UnitTest {

   private TopicConnection receiveConnection;
   private TopicSession receiveSession;
   private Topic receiveTopic;
   private TopicSubscriber receiveSubscriber;
   private TopicConnection sendingConnection;
   private TopicSession sendingSession;
   private Topic sendingTopic;
   private TopicPublisher sendingPublisher;

   @Before
   public void initialize() throws Exception {
      ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(
            ActiveMQConnection.DEFAULT_USER,
            ActiveMQConnection.DEFAULT_PASSWORD,
            ActiveMQConnection.DEFAULT_BROKER_URL);

      // removed code to create a connection for the subscriber and
the publisher
      ...
      receiveSubscriber = receiveSession.
createSubscriber(receiveTopic);
      sendingPublisher = sendingSession.createPublisher(sendingTopic);
```

```
        ChatRoom.clean();
    }

    @After
    public void shutdown() throws Exception {
        // removed: closing all connections
        ...
    }

    @Test
    public void assertThatPluginSendsMessages() throws Exception {
        assertEquals(0, ChatRoom.get().archive().size());
        Event event = new Join("user");
        ChatRoom.get().publish(event);

        // Check chatroom
        int currentEventCount = ChatRoom.get().archive().size();
        assertEquals(1, currentEventCount);

        // Check for messages
        Message msg = receiveSubscriber.receive(2000);
        Event evt = (ChatRoom.Event) ((ObjectMessage) msg).getObject();
        assertEquals("join", evt.type);
    }

    @Test
    public void assertThatPluginReceivesMessages() throws Exception {
        assertEquals(0, ChatRoom.get().archive().size());

        // Send event via JMS
        Event event = new ChatRoom.Message("alex", "cool here");
        ObjectMessage objMsg = sendingSession.
createObjectMessage(event);
        sendingPublisher.publish(objMsg);

        Thread.sleep(1000); // short sleep to make sure the content
arrives
        assertEquals(1, ChatRoom.get().archive().size());
    }

}
```

As you can see, there are two tests. The first one publishes a local event like a web request would do via `ChatRoom.get().publish(event)`. It then waits for a subscriber a maximum of two seconds to check whether the message has been published via ActiveMQ.

The second test fakes the existence of another node by sending a message directly via the publisher and checks whether the `ChatRoom` archive field now contains any content. In order to make sure the local archive of events is empty, `ChatRoom.clean()` is invoked in `initialize()` method.

How to do it...

The ActiveMQ plugin luckily only consists of one class and one interface. The first class is the plugin itself, which starts the JMS connection on application start and frees all resources when the application is stopped:

```
public class ActiveMqPlugin extends PlayPlugin {

   private ActiveMQConnectionFactory connectionFactory;

   private TopicConnection sendingConnection;
   private TopicSession sendingSession;
   private Topic sendingTopic;
   private TopicPublisher sendingPublisher;

   private TopicConnection receiveConnection;
   private TopicSession receiveSession;
   private Topic receiveTopic;
   private TopicSubscriber receiveSubscriber;

   private static final String uuid = UUID.randomUUID().toString();

   public void onApplicationStart() {
      Logger.info("ActiveMQ Plugin started");
      try {
         List<Class> jobClasses = new ArrayList<Class>();
         for (ApplicationClass applicationClass : Play.classes.
getAssignableClasses(ActiveMqJob.class)) {
            if (Job.class.isAssignableFrom(applicationClass.
javaClass)) {
               jobClasses.add(applicationClass.javaClass);
            }
         }
         MessageListener listener = new ActiveMqConsumer(jobClasses);
```

```
            // setting up all the topic specific variables
            ... // removed to save space
         } catch (Exception e) {
            Logger.error(e, "Could not start activemq broker");
         }
      }

      public void onApplicationStop() {
         Logger.info("Stopping activemq connections");
         try {
            // closing all activemq specific connections
            ... // removed to save space
         } catch (JMSException e) {
            Logger.error(e, "Problem closing connection");
         }
      }
```

The `onEvent()` method listens for all messages beginning with `chatEvent` and sends them via ActiveMQ if possible. The inner `ActiveMQConsumer` class handles incoming messages, extracts data out of the message, and executes a job which sets the received data in the application:

```
      @Override
      public void onEvent(String message, Object context) {
         if ("chatEvent".equals(message) && context instanceof
   Serializable) {
            Serializable event = (Serializable) context;
            try {
               ObjectMessage objMsg = sendingSession.
   createObjectMessage(event);
               objMsg.setStringProperty("hostId", uuid);
               sendingPublisher.publish(objMsg);
               Logger.info("Sent event to queue: %s", ToStringBuilder.
   reflectionToString(event));
            } catch (Exception e) {
               Logger.error(e, "Could not publish message");
            }
         }
      }

      public static class ActiveMqConsumer implements MessageListener {

         private List<Class> jobs;
```

```
        public ActiveMqConsumer(List<Class> jobs) {
           this.jobs = jobs;
        }

        @Override
        public void onMessage(Message message) {
           try {
              if (message instanceof ObjectMessage) {
                 ObjectMessage objectMessage = (ObjectMessage) message;
                 if (uuid.equals(objectMessage.
   getStringProperty("hostId"))) {
                    Logger.debug("Ignoring activemq event because it
   came from own host");
                    return;
                 }
                 Serializable event = (Serializable) objectMessage.
   getObject();
                 Logger.debug("Received activemq event to plugin: %s",
   ToStringBuilder.reflectionToString(event));
                 for (Class jobClass : jobs) {
                    Job job = (Job) jobClass.newInstance();
                    job.getClass().getField("serializable").set(job,
   event);
                    job.now().get(); // run in sync

                 }
              }
           } catch (Exception e) {
              Logger.error(e, "Error getting message");
           }
        }
     }
   }
```

A small interface definition is needed to mark jobs, which should be executed upon incoming messages:

```
public interface ActiveMqJob {
   public void setSerializable(Serializable serializable);
   public Serializable getSerializable();
}
```

After adding a `play.plugins` file for the ActiveMQ plugin the module can now be built and added to your copy of the chat application.

The next change is regarding the `ChatRoom` class in the chat application. You need to add three methods:

```
public void publish(Serializable event) {
       PlayPlugin.postEvent("chatEvent", event);
       chatEvents.publish((Event) event);
}

public void publishWithoutPluginNotification(Serializable event) {
       chatEvents.publish((Event) event);
}

public static void clean() {
       instance = null;
}
```

Apart from that you need to make the `Chatroom.Event` class serializable so that it can be transmitted over the network. The last step is to change the three methods, which indicate a chat room join or leave or a text message sent by a user. You have to use the publish method inside each of these methods:

```
public EventStream<ChatRoom.Event> join(String user) {
      publish(new Join(user));
    return chatEvents.eventStream();
}

public void leave(String user) {
    publish(new Leave(user));
}

public void say(String user, String text) {
    if(text == null || text.trim().equals("")) {
        return;
    }
    publish(new Message(user, text));
}
```

The application needs to implement a special job, which implements the interface that we defined some time back:

```
public class UpdateChatroomJob extends Job implements ActiveMqJob {

   private Serializable serializable;

   public Serializable getSerializable() {
      return serializable;
```

```
    }

    public void setSerializable(Serializable serializable) {
       this.serializable = serializable;
    }

    public void doJob() {
       if (serializable != null) {   ChatRoom.get().publishWithoutPlugi
nNotification(serializable);
       }
    }

}
```

How it works...

After making your changes you have two possibilities to test your code. First, start up your application and run the tests. Second, copy the application directory to a second directory, change the `http.port` variable to some arbitrary port, and remove the `%test.db=mem` entry in case you are running in test mode. Otherwise, the H2 databases will conflict when trying to bind to port 8082. After both applications are started, connect to both in a browser, choose websocket for chat connection, and check whether you can see the messages of the other user, even though both are connected to two different web applications.

So, after checking if it works, it is time to explain it in a little detail. The simplest part in the plugin is the interface definition. You are expected to have an arbitrary amount of jobs implementing this interface. The job implementation makes sure that the serialized object representing the content of the message received via the subscriber gets to its endpoint, which is the `ChatRoom` in the implementation. In order to make sure it does not create any loop or again sends this message to the ActiveMQ topic, the implementation invokes the `publishWithoutPluginNotification()` method. Otherwise content might be duplicated.

The plugin itself has a lot of code to create and initialize the connection to the broker. Broker is an alternative name for the messaging endpoint. The `onApplicationStart()` method collects all classes which implement the `ActiveMqJob` interface and are assigned from a job. The list of these classes is handed over to the internally defined `ActiveMqConsumer` class, which represents the code executed on each incoming message. This class checks if the incoming message was not sent from this host, as in this case the message has already been spread locally. Then it loops through the list of jobs and executes each of them, after setting the `serializable` field in order to hand over the incoming message to the job. This is not really a better solution, but it works in this case.

The remaining part is the sending of messages, so that all other nodes can process them. As the `ChatRoom` class has been changed to always post a plugin event upon receiving a new chatroom specific event, the plugin checks for such events and sends chatroom specific events over the wire. Before sending an event a UUID gets attached as a property, which allows checking of whether an incoming message has been sent from this host.

As you can see, integrating messaging in your application is not too complex. However, keeping it fast and closely adapted to your needs might be complex. A message based architecture helps you to scale your systems, as you can decouple presentation from backend logic and scale your systems differently.

There's more...

Messaging is quite a complex topic. There are many more implementations and protocol standards other than those provided by ActiveMQ, so it's highly possible that you are already using a similar software.

More info about AMQP

In order to be more independent of an API you should try AMQP implementing services such as RabbitMQ or ZeroMQ. They provide a standard protocol, which can be used in any programming language. If you need speed in delivery, these might be faster than old fashioned JMS services. Find more information about AMQP at `http://www.amqp.org/`, about RabbitMQ at `http://www.rabbitmq.com/`, and about ZeroMQ at `http://www.zeromq.org/`.

Using Solr for indexing

As already mentioned several times in this book, there already exists a very nice plugin for searching, which is based on Apache Lucene. The primary drawback of this plugin is that it does not scale due to the fact that it searches on indexes stored on the local file system. If you had a multi-node setup, you would have to make sure that changes of objects are propagated to all nodes and all indexes are updated. This would be very cumbersome and there is already a solution for this problem. Its name is Apache Solr and basically it is a HTTP interface around Lucene. In fact, it has more functionality than that, but for this example, this fact is sufficient. This means using Apache Solr basically moves the index from one Play instance to a real search engine, which then is queried and updated by any Play instance.

This recipe will create a completely new plugin, which will use Solr for searching. This means that every change of an entity has to be forwarded to the search index. The search itself will also always be handled by Solr.

The source code of the example is available at `examples/chapter6/search-solr`.

Getting ready

As usual we will write a test to show off what the Solr support should look like. This test should be put inside an application, not inside of the plugin which is created throughout this recipe:

```
public class SolrSearchTest extends UnitTest {

   CommonsHttpSolrServer server;

   @Before
   public void setup() throws Exception {
      Fixtures.deleteAllModels();

      server = new CommonsHttpSolrServer("http://localhost:8983/
solr");
      server.setRequestWriter(new BinaryRequestWriter());
      clearSolrServerIndex();

      Fixtures.loadModels("test-data.yml");
   }

   private void clearSolrServerIndex() throws Exception {
      server.deleteByQuery( "*:*" );
      server.commit();
   }

    @Test
    public void testEnhancedSearchCapability() {
       assertEquals(1, Car.search("byBrandAndType", "BMW", "320").
fetchIds().size());

       List<User> users = User.search("byNameAndTwitter", "a*ex",
"spinscale*").fetch();
       User user = users.get(0);
       User u1 = User.find("byName", "alex").first();
       assertEquals(user.id, u1.id);
    }
}
```

If you ignore the setup of the test and take a look at the test, you will see the `Model.search()` method, which looks pretty similar to the `Model.find()` method. Its basic function is the same, with the exception that it always queries the Solr index instead of the database. Also, there are two further methods to access the data returned by the Solr query. The first is `Model.search().fetchIds()` and the second is `Model.search().fetch()`. The first method fetches the IDs from the index, allowing you to get the data out of the database manually, whereas the second method issues the database query automatically.

You also should have a working Solr instance in order to follow this example. You can download the most up to date release of Solr at `http://lucene.apache.org/solr/` and start the example application. It is actually sufficient. Any changes in the configuration file of Solr will be documented in the next section. After downloading and extracting the archive, you can start Solr by:

```
cd apache-solr-3.1.0/example
java –jar start.jar
```

You should now be able to go to `http://localhost:8983/solr/` and see a Welcome to Solr message.

How to do it...

After creating a new module via `play new-module solr` for a Solr plugin you need to get the dependencies right. As there are several dependencies overlapping with the versions in Play, you have to carefully define what files should be downloaded and what files should not. Create the following `dependencies.yml` file and run `play dependencies`:

```
self: play -> solr 0.1

require:
    - org.apache.solr -> solr-solrj 3.1.0:
       transitive: false
    - commons-httpclient -> commons-httpclient 3.1:
        transitive: false
    - org.codehaus.woodstox -> wstx-asl 3.2.7
    - stax -> stax-api 1.0.1
```

The next step is to put the module together. Do not forget to create an appropriate `play.plugins` file. The plugin actually needs only four classes. The first is the `SearchableModel`, which extends the standard `Model` class. Put all classes into the `play.modules.solr` package:

```
public class SearchModel extends Model {

   public static Query search(String query, String ... values) {
        throw new UnsupportedOperationException("Check your
configuration. Bytecode enhancement did not happen");
   }

   protected static Query search(Class clazz, String query, String ...
values) {
      StringBuilder sb = new StringBuilder();
      if (query.startsWith("by")) {
         query = query.replaceAll("^by", "");
```

```
        }
        String fieldNames[] = query.split("And");

        for (int i = 0 ; i < fieldNames.length; i++) {
           String fieldStr = fieldNames[i];
           String value = values[i];

           String fieldName = StringUtils.uncapitalize(fieldStr);
           String solrFieldName = getSolrFieldName(fieldName, clazz);

           sb.append(solrFieldName);
           sb.append(":");
           sb.append(value);

           if (i < fieldNames.length-1) {
              sb.append(" AND ");
           }
        }

        return new Query(sb.toString(), clazz);
     }

     private static String getSolrFieldName(String fieldName, Class
  clazz) {
        try {
           java.lang.reflect.Field field = clazz.getField(fieldName);
           Field annot = field.getAnnotation(Field.class);
           if (annot != null && !annot.value().equals("#default")) {
              return annot.value();
           }
        } catch (Exception e) {
           e.printStackTrace();
        }
        return fieldName;
     }
  }
```

The main part of this class is to define the static `search()` method, which was already used in the test. This method is filled by using bytecode enhancement, so a bytecode enhancer is needed next:

```
  public class SolrEnhancer extends Enhancer {

     public void enhanceThisClass(ApplicationClass applicationClass)
  throws Exception {
```

```
        CtClass ctClass = makeClass(applicationClass);

        if (!ctClass.subtypeOf(classPool.get("play.modules.solr.
SearchModel"))) {
            return;
        }

        String method = "public static play.modules.solr.
Query search(String query, String[] values) { return
search("+applicationClass.name+".class, query, values); }";
        CtMethod count = CtMethod.make(method, ctClass);
        ctClass.addMethod(count);

        // Done.
        applicationClass.enhancedByteCode = ctClass.toBytecode();
        ctClass.defrost();

        Logger.info("Enhanced search of %s", applicationClass.name);
    }
}
```

The enhancer replaces the empty static `search()` method by invoking the second defined `search()` method with the class parameter and supplies it during enhancement with this information. As the method does not return entity objects but an object being a `Query` class, this result class has to be defined as well. The `Query` class issues the actual query to the Solr server and handles the response:

```
public class Query {

   private SolrQuery query;
   private SolrServer server;
   private Class clazz;

   public <T extends Model> Query(String queryString, Class<T> clazz)
{
      query = new SolrQuery();
      query.setFilterQueries("searchClass:" + clazz.getName());
      query.setQuery(queryString);
      this.server = SolrPlugin.getSearchServer();
      this.clazz = clazz;
   }

   public Query limit(int limit) {
      query.setRows(limit);
      return this;
```

```
    }

    public Query start(int start) {
       query.setStart(start);
       return this;
    }

    public List<String> fetchIds() {
       query.setFields("id");
       SolrDocumentList results = getResponse();
       List<String> ids = new ArrayList(results.size());
       for (SolrDocument doc : results) {
          String entityId = doc.getFieldValue("id").toString().
split(":")[1];
          ids.add(entityId);
       }

       return ids;
    }

    public <T extends Model> List<T> fetch() {
       List<T> result = new ArrayList<T>();

       List<String> ids = fetchIds();
       for (String id : ids) {
          Object objectId = getIdValueFromIndex(clazz, id);
          result.add((T) JPA.em().find(clazz, objectId));
       }

       return result;
    }

    private SolrDocumentList getResponse() {
       try {
          QueryResponse rp = server.query(query);
          return rp.getResults();
       } catch (SolrServerException e) {
          Logger.error(e, "Error on solr query: %s", e.getMessage());
       }

       return new SolrDocumentList();
    }
```

```
    private Object getIdValueFromIndex(Class<?> clazz, String
indexValue) {
        java.lang.reflect.Field field = getIdField(clazz);
        Class<?> parameter = field.getType();
        try {
            return Binder.directBind(indexValue, parameter);
        } catch (Exception e) {
            throw new UnexpectedException("Could not convert the ID
from index to corresponding type", e);
        }
    }

    private java.lang.reflect.Field getIdField(Class<?> clazz) {
        for (java.lang.reflect.Field field : clazz.getFields()) {
            if (field.getAnnotation(Id.class) != null) {
                return field;
            }
        }
        throw new RuntimeException("Your class " + clazz.getName()
+ " is annotated with javax.persistence.Id but the field Id was not
found");
    }
}
```

This class issues the query, gets the result, and performs database lookups for the returned IDs, if necessary.

The plugin class which invokes the bytecode enhancer on startup is the central piece of the following plugin:

```
public class SolrPlugin extends PlayPlugin {

   private SolrEnhancer enhancer = new SolrEnhancer();

    public void enhance(ApplicationClass applicationClass) throws
Exception {
      enhancer.enhanceThisClass(applicationClass);
    }

   public void onEvent(String message, Object context) {
      if (!StringUtils.startsWith(message, "JPASupport.")) {
         return;
      }

      try {
```

```
            Model model = (Model) context;
            String entityId = model.getClass().getName() + ":" + model.
getId().toString();

            SolrServer server = getSearchServer();
            server.deleteById(entityId);

            if ("JPASupport.objectUpdated".equals(message)) {
               SolrInputDocument doc = new SolrInputDocument();
               doc.addField("id", entityId);
               doc.addField("searchClass", model.getClass().getName());

               for (java.lang.reflect.Field field : context.getClass().
getFields()) {
                  String fieldName = field.getName();
                  Field annot = field.getAnnotation(Field.class);
                  if (annot == null) {
                     continue;
                  }

                  String annotationValue = annot.value();
                  if (annotationValue != null && !"#default".
equals(annotationValue)) {
                     fieldName = annotationValue;
                  }

                  doc.addField(fieldName, field.get(context));
               }
               server.add(doc);
            }
            server.commit();
         } catch (Exception e) {
            Logger.error(e, "Problem updating entity %s on event %s with
error %s", context, message, e.getMessage());
         }
      }

      public static SolrServer getSearchServer() {
         String url = Play.configuration.getProperty("solr.server",
"http://localhost:8983/solr");
         CommonsHttpSolrServer server = null;
         try {
            server = new CommonsHttpSolrServer( url );
            server.setRequestWriter(new BinaryRequestWriter());
         } catch (MalformedURLException e) {
```

```
            Logger.error(e, "Problem creating solr server object: %s",
e.getMessage());
        }
        return server;
    }
}
```

The last step is to configure your entities inside your application appropriately like the following `User` entity:

```
package models;

import javax.persistence.Entity;
import org.apache.solr.client.solrj.beans.Field
import play.modules.solr.SearchModel;

@Entity
public class User extends SearchModel {

    @Field
    public String name;
    @Field
    public String shortDescription;
    @Field("tw_s")
    public String twitter;

    public Car car;
}
```

As you can see, you can annotate your fields to be indexed with the `@Field` annotation from the `solrj` client. As SolrJ is also used when indexing the entities the plugin does not have to define its own annotations.

How it works...

A lot of source code is needed, for a lot of functionality. There are many pieces needing an explanation. First, a little bit of information about how Solr querying works. The plugin does not use XML to import data into Solr, but rather binary data, as it is way faster than creating an XML response and sending it to the server. You explicitly have to configure Solr to support this binary format, which is described in the SolrJ wiki page at `http://wiki.apache.org/solr/Solrj`.

Furthermore, SolrJ supports indexing beans directly instead of creating a `SolrInputDocument` out of it. This does work for entities, as they are complex objects with more than simple types. The problem is the entity ID field. This field is included in the model class, where it also would need to be annotated. However, this would mean that there could be only one class stored, as entity IDs are similar on different models. So the creation of the ID has to be performed manually. The adding of entities to the index is again done via the event mechanism of Play. Whenever an entity is updated or deleted it is propagated to the index. This is done in the `SolrPlugin`. When you take a look at the `onEvent()` method, you will see that when the right event occurs the following happens:

- The old document gets deleted from the index
- A new document is created, with the ID of the entity
- A `searchClass` property is added, which is needed for querying entities of a certain type
- Every field of an entity with a `@Field` annotation is added to the document, possibly under a different name than the name of the field in the entity, if specified in the annotation

Be aware that you cannot store object graphs with Solr. This is basically all the code needed to store data into Solr.

The `SearchModel` class serves two purposes. First, it enables you to have IDE support and auto completion for the static `search()` method, second it translates the already known `find("byFooAndBar")` syntax to a real search query.

The last part is the `Query` class, which issues actual search queries and parses the results. Similar to the `JPAQuery`, which is returned when calling the `find()` method of an entity, it features a limit on the result size as well as starting from an offset. It also features the `fetch()` method. This method fetches all the IDs from the Solr search and then queries the database to get the complete entity. Be aware that this again adds some load to your database. The `fetchIds()` method returns the IDs from the search, but will not query your database.

Until now it has looked like as if it was not necessary at all to configure Apache Solr. This is not completely true. When you take a look at your `schema.xml` file, you will find the case of defining a schema for indexing and the support for dynamic fields. Dynamic fields often end with "_s" to represent a dynamic string field. This is used in the `User` entity above for the Twitter handle. However, as the `name` and `shortDescription` fields of this entity do not have such an alias set, they have to be defined like this:

```
    <field name="name" type="string" indexed="true" stored="true"/>
    <field name="shortDescription" type="text" indexed="true"
stored="true"/>
    <field name="searchClass" type="textgen" indexed="true"
stored="true" required="true" />
    <field name="id" type="string" indexed="true" stored="true"
required="true" />
```

The name and `shortDescription` fields are needed for the `User` class, while the `searchClass` and `id` properties are used for any entity which gets indexed.

When writing real applications you should also use a better index value than the one used in this case like `models.User.1`, which consists of a long string instead of a numerical hash, though of course still being unique from an application point of view. A murmur hash function or similar might be a good choice. Solr also has support for UUID based IDs. Check `http://wiki.apache.org/solr/UniqueKey` for more information.

This is quite a good example on how to integrate an existing API into Play framework, as it closely resembles the existing methods like `find()`. It adds up the `search()` methods, which makes it pretty easy for the developer to adapt to this new functionality. You should always think about the possibilities of integration. For example, the `SolrQuery` class used in this example supports facetted searches. However, integrating this neatly into the Play framework is a completely different issue. If it does not work in a nice way, you should use the native API instead of just adding another layer of complexity, which needs to be maintained.

If you want to optimize this module, the first possible spot to look for would be the `fetch()` method in the `Query` class. It is very inefficient to execute `JPA.em().find()` on every ID returned by the search query. You could easily group this query into one statement using a JPA query and the IN operator of JPQL.

There's more...

Search is a really complex topic. Make sure you know the search engine you are using before you tie it to your application.

More information about SolrJ

SolrJ is an incredibly useful library for connecting Solr with Java. You should read more about it at `http://wiki.apache.org/solr/Solrj`.

More complex queries

As you can access the Solr query inside the query class, you can define arbitrary complex queries if you change the query class as needed. This would include stuff such as facetting, range queries, and many more. Solr is actually very versatile and here we use only a very small number of features.

Support for other search engines

There is a very nice module for the elasticsearch engine, which is a new kid on the block of Lucene-based search engines. It scales transparently and you can even use it as a complete backend data store. More information about elasticsearch is available at `http://www.elasticsearch.org`

Writing your own cache implementation

Play already comes with an easy to scale cache solution included, namely memcached. However, you might not want to use memcached at all. It might not fit your needs, your admin may not want to install another application on a system, or anything else. Instead of not using a cache at all, this recipe shows you how to write your own implementation of a cache. This example will use Hazelcast, a pure Java distributed system as implementation.

You can find the source code of this example in the `examples/chapter6/caching-implementation` directory.

Getting ready

It requires a little bit of work to get to the point to start programming. We need to get Hazelcast instance up and running and setup an application for development. Hazelcast is basically a distributed data platform for the JVM. Hazelcast has tons of more features that are not needed in this little example—basically only a distributed map is featured here. Hazelcast scales very well, there are live examples of even 100 parallel running instances.

So, first download Hazelcast. Go to `http://www.hazelcast.com`, click on the **Downloads** menu and get the latest version, it usually comes in a ZIP file. Unzip it, and run the following commands:

```
cd hazelcast-${VERSION}
cd bin
sh run.sh
```

That's it. You should now have a running Hazelcast instance on localhost.

Furthermore, you need to copy two jars into the `lib/` directory of your Play application. These are:

```
hazelcast-${VERSION}.jar
hazelcast-client-${VERSION}.jar
```

The files are in the `lib/` directory of your Hazelcast installation.

As there is already a caching test included in the samples-and-tests directory of every Play framework distribution, we can reuse this test to make sure the hazelcast-based cache implementation works. You should copy the following files to the corresponding directories in your created application:

```
samples-and-tests/just-test-case/app/models/User.java

samples-and-tests/just-test-case/app/models/MyAddress.java
```

`samples-and-tests/just-test-case/test/CacheTest.java`

`samples-and-tests/just-test-case/test/users.yml`

After copying these files and running the test in your web browser, you should get a working test because no code to change the cache has been built yet. If you are using a Play version other than 1.2, please check the Java files, to see whether you might need to copy other dependant classes as well in order to be able to compile your application.

How to do it...

You should create a new module for your application, for example, via `play new-module hazelcast`. You should put the following into the `conf/dependencies.yml` file of your module and call `play deps` afterwards:

```
self: play -> hazelcast 0.1

require:
    - play
    - com.hazelcast -> hazelcast-client 1.9.3.1:
        transitive: false
    - com.hazelcast -> hazelcast 1.9.3.1:
        transitive: false
```

Now create a `play.plugins` file like this, which references the plugin, which is going to be created:

```
1000:play.modules.hazelcast.HazelcastPlugin
```

The next step is to create the plugin itself, which checks whether the Hazelcast cache should be enabled and replaces the cache appropriately:

```
public class HazelcastPlugin extends PlayPlugin {

  public void onApplicationStart() {
    Boolean isEnabled = new Boolean(Play.configuration.
getProperty("hazelcast.enabled"));
    if (isEnabled) {
     Logger.info("Setting cache to hazelcast implementation");
     Cache.forcedCacheImpl = HazelCastCacheImpl.getInstance();
     Cache.init();
    }
  }
}
```

Now the only missing part is the cache implementation itself. All this code belongs to one class, but to help you understand the code better, some comments follow between some methods. The first step is to create a private constructor and a `getInstance()` method in order to implement the singleton pattern for this class. The constructor also loads some default parameters from the configuration file. Put this code into your module at `src/play/modules/hazelcast/HazelCastCacheImpl.java`:

```
public class HazelCastCacheImpl implements CacheImpl {

   private HazelcastClient client;
   private static HazelCastCacheImpl instance;

   public static HazelCastCacheImpl getInstance() {
      if (instance == null) {
         instance = new HazelCastCacheImpl();
      }
      return instance;
   }

   private HazelCastCacheImpl() {
      String groupName = Play.configuration.getProperty("hazelcast.
groupname", "dev");
      String groupPass = Play.configuration.getProperty("hazelcast.
grouppass", "dev-pass");
      String[] addresses = Play.configuration.getProperty("hazelcast.
addresses", "127.0.0.1:5701").replaceAll(" ", "").split(",");
      client = HazelcastClient.newHazelcastClient(groupName,
groupPass, addresses);
   }

   private IMap getMap() {
      return client.getMap("default");
   }
```

The `add()` and `safeAdd()` methods only put objects in the cache, if they do not exist in the cache already. On the other hand the `set()` and `safeSet()` methods overwrite existing objects. Note that any method with a `safe` prefix in its name has the contract of executing the action (add, set, replace, or delete) synchronous and must make sure the cache has finished the operation before returning to the caller:

```
   @Override
   public void add(String key, Object value, int expiration) {
      if (!getMap().containsKey(key)) {
         getMap().put(key, value, expiration, TimeUnit.SECONDS);
      }
   }
```

```
    @Override
    public boolean safeAdd(String key, Object value, int expiration) {
       getMap().putIfAbsent(key, value, expiration, TimeUnit.SECONDS);
       return getMap().get(key).equals(value);
    }

    @Override
    public void set(String key, Object value, int expiration) {
       getMap().put(key, value, expiration, TimeUnit.SECONDS);
    }

    @Override
    public boolean safeSet(String key, Object value, int expiration) {
       try {
           set(key, value, expiration);
           return true;
       } catch (Exception e) {}
       return false;
    }
```

The `replace()` and `safeReplace()` methods only place objects in the cache if other objects already exist under the referenced key. The `get()` method returns a not casted object from the cache, which also might be null. The `get()` method taking the `keys` parameter as varargs allows the developer to get several values at once. In case you are in need of easily incrementing or decrementing numbers, you also should implement the `incr()` and `decr()` methods:

```
    @Override
    public void replace(String key, Object value, int expiration) {
       if (getMap().containsKey(key)) {
          getMap().replace(key, value);
       }
    }

    @Override
    public boolean safeReplace(String key, Object value, int
expiration) {
       if (getMap().containsKey(key)) {
          getMap().replace(key, value);
          return true;
       }
       return false;
    }
```

```
@Override
public Object get(String key) {
   return getMap().get(key);
}

@Override
public Map<String, Object> get(String[] keys) {
   Map<String, Object> map = new HashMap(keys.length);
   for (String key : keys) {
      map.put(key, getMap().get(key));
   }
   return map;
}

@Override
public long incr(String key, int by) {
   if (getMap().containsKey(key)) {
      getMap().lock(key);
      Object obj = getMap().get(key);
      if (obj instanceof Long) {
         Long number = (Long) obj;
         number += by;
         getMap().put(key, number);
      }
      getMap().unlock(key);
      return (Long) getMap().get(key);
   }
   return 0;
}

@Override
public long decr(String key, int by) {
   return incr(key, -by);
}
```

The last methods are also quickly implemented. The `clear()` method should implement a complete cache clear. The `delete()` and `deleteSafe()` methods remove a single element from the cache. An important method to implement is the `stop()` method, which allows the implementation to correctly shutdown before it stops working. This is executed if your Play application is stopped, for example:

```
@Override
public void clear() {
   getMap().clear();
}
```

```
    @Override
    public void delete(String key) {
       getMap().remove(key);
    }

    @Override
    public boolean safeDelete(String key) {
       if (getMap().containsKey(key)) {
          getMap().remove(key);
          return true;
       }
       return false;
    }

    @Override
    public void stop() {
       client.shutdown();
    }
}
```

You can now build the module via `play build-module` and then switch to the application, which includes the `CacheTest` class.

There is only one thing left to do to get the new cache implementation working in your application. Enable your newly built module in your application configuration. You need to edit `conf/application.conf` and add Hazelcast-specific configuration parameters:

```
hazelcast.enabled=true
hazelcast.groupname=dev
hazelcast.grouppass=dev-pass
hazelcast.addresses=127.0.0.1:5701
```

Then start the application and you should be able to run the preceding `CacheTest` that is referenced.

How it works...

The configuration is pretty self explanatory. Every Hazelcast client needs a username, a password, and an IP to connect to. The `HazelcastPlugin` sets the field `forcedCacheImpl` of the Cache class to the hazelcast implementation and reruns the `init()` method. From now on the Hazelcast cache is used.

I will not explain every method of the cache, as most of the methods are pretty straightforward. The `getMap()` method fetches the default map of the Hazelcast cluster which basically works like the map interface, but has a few more features such as the expiration.

Primarily, the class implements the singleton pattern by not providing a public constructor, but a `getInstance()` method in order to make sure that always the same instance is used.

There is also a client variable of type `HazelcastClient`, which does the communication with the cluster. The private constructor parses the Hazelcast specific setup variables and tries to connect to the cluster then.

The `getMap()` method returns a Hazelcast-specific IMAP, which basically is a map, but also has support for expiry. In case you are wondering where the `client.getMap("default")` is coming from, it is the default name of the distributed map, configured in Hazelcast. There is a `hazelcast.xml` file in the same directory, where the `run.sh` file resides.

The `IMap` structure gives us basically everything needed to implement the add, set, replace, and delete and their safe companion methods used in the cache. As opposed to other cache implementations, often there is no difference between the regular and the safe implementation due to the behavior of Hazelcast.

A little trick had to be done for the `incr()` method. First, the `CacheTest` enforces that incrementing in the cache only works if the key has already been created before. Second, it is the only code where a lock is put on the key of the map (not on the whole map), though no increments are actually missing out. If another process is incrementing, the method blocks until the key is unlocked again. Third, we only add increment counters if the values are of type Long and thus can be incremented.

It is also useful to implement the `stop()` method of the cache interface, which allows a graceful shutdown of the client.

There's more...

When you start the Hazelcast server, you get into a console where you can issue some commands. For example, you can issue `m.size()` there to get the current size of the distributed map. This could help you to debug some issues you might face.

More about Hazelcast

You should read more about Hazelcast as it is a really interesting technology. Go to `http://www.hazelcast.com/` or `http://code.google.com/p/hazelcast/` for more information.

Try building a Redis cache

As you have seen, it is pretty simple to build your own cache implementation. You could try to build a Redis cache as the next step. There are pretty simple libraries out there for Redis, like JRedis, which would make such an implementation quite easy.

7
Running in Production

In this chapter, we will cover:

- Test automation with Jenkins
- Test automation with Calimoucho
- Creating a distributed configuration service
- Running jobs in a distributed environment
- Running one play instance for several hosts
- Forcing SSL for chosen controllers
- Implementing your own monitoring points
- Configuring log4j for log rotation
- Integrating with Icinga
- Integrating with Munin
- Setting up the Apache web server with Play
- Setting up the Nginx web server with Play
- Setting up the Lighttpd web server with Play
- Multi-node deployment introduction

Introduction

Until now the whole book has been about development. However, this is only half the story. The real complexity begins after that, when your application goes live, and even before that during the testing phase.

Usually, this process is split apart from the developer, as another department or at least other people than the developers perform the deployment. However, this chapter is targeted at both groups, developers as well as system administrators. It is generally a good idea to try and merge the groups a little closer in order to be able to cooperate better and understand, why each groups prefers doing things differently.

Testing is another important issue. Test-driven development in small and agile teams is getting more common. This requires continuous integration systems in order to have constant information about the state of your code. These continuous integration systems help you to determine whether your code is ready to be deployed, and whether you have written enough tests. Two recipes are about making Play adapt to continuous integration, either using Jenkins (formerly Hudson) or using the Calimoucho application, a Play application itself which runs all tests of another Play application.

This chapter does not only include information on how to roll out a Play application, it also includes information on what the developer has to think about when an application goes from desktop development to a complex multi-node production setup. The complexity of switching to a complex production setup should be known by the developers as well as the system engineers.

Test automation with Jenkins

If you are already having a continuous integration system, it might already be Jenkins as it is quite common in the Java world. If you do not want to set up another system, you should try to use Jenkins as well to run automated tests of your play applications. This recipe will help you to do a quick setup of Jenkins and make it ready to deploy on a Play application.

You can find the source code of this example in the `examples/chapter7/test-jenkins` directory.

Getting ready

A running Jenkins is essential of course. Either use the current running one, or get a fresh copy from `http://www.jenkins-ci.org`, download the WAR file and just start it with the following:

```
java -jar jenkins.war
```

Again, we will also set up a repository using mercurial. This time we will also access the repository via HTTP instead of file system, as it would be done in a real world setup as well. First, you should install the mercurial plugin of Jenkins. Go to `http://localhost:8080`, click on **Manage Jenkins** on the left, then click on **Manage Plugins**. Just select the mercurial plugin there and click on **Install** at the bottom of the page. After the installation is finished, you need to restart Jenkins. Create a Play application, a mercurial repo inside of it, and start serving the repository in order for it to be accessible via HTTP:

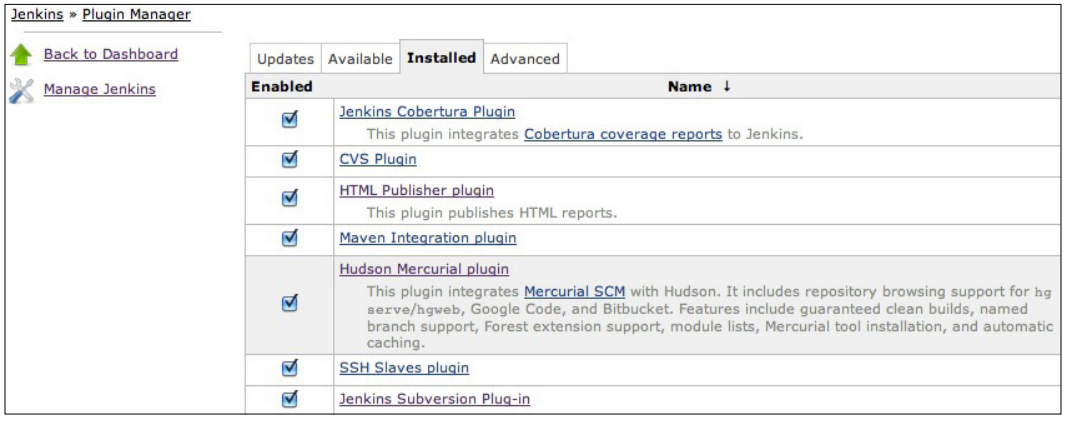
```
play new test-jenkins
cd test-jenkins
hg init
hg add
hg commit -m "initial import"
hg serve
```

You can go to `http://localhost:8000/` now and check the first commit. The preceding `hg serve` command started a web server and can be used to check out the Play application using the `hg clone` command. We will use this URL in Jenkins later.

In order to have support for Cobertura, you should install the Cobertura plugin for Jenkins the same way you installed the mercurial plugin. Furthermore, you also need to install the Cobertura plugin for Play. Add Cobertura to your `conf/dependencies.yml` file and commit it via `hg commit –m "added cobertura dependency"`:

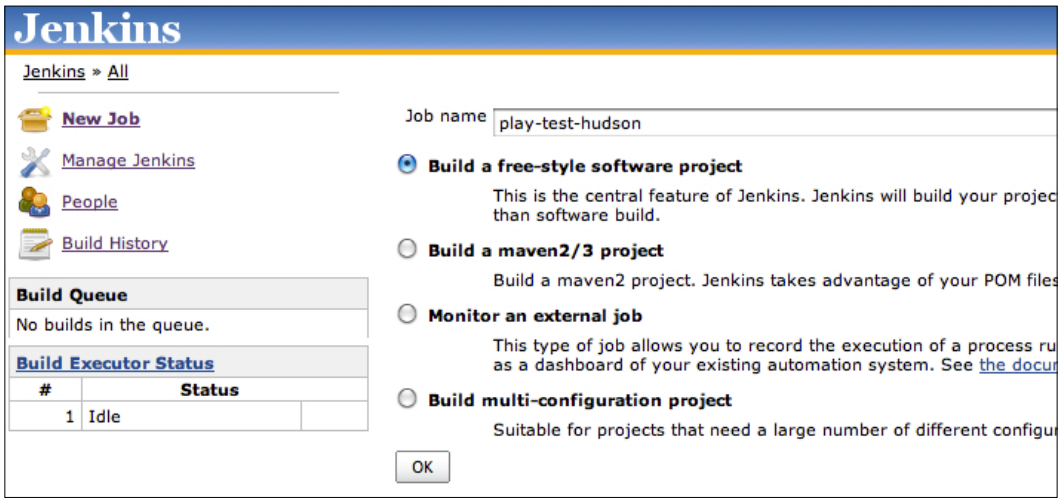
```
require
    play -> cobertura 2.1
```

There is a final plugin you should install for Jenkins. This is the HTML publisher plugin, which allows you to link special pages for each build. This makes it easy to link to the HTML results of the tests—the same you usually see in the browser test runner under the `/@tests/` URL. When checking your Jenkins plugin page it should now look like the following. Note that all plugins except the mercurial, Cobertura, and HTML publisher plugin are a part of core Jenkins:



How to do it...

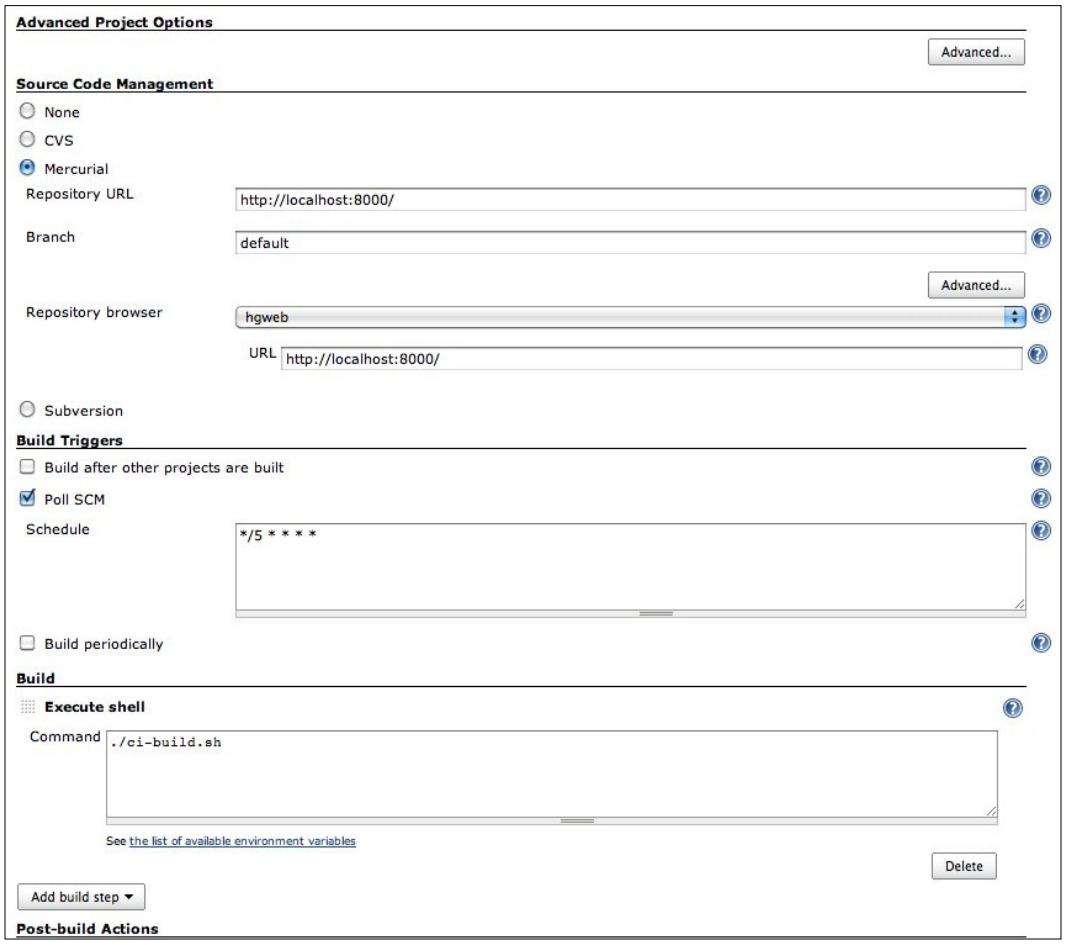
Configuring the job in Jenkins is relatively straightforward. Here are three screenshots. First you should create a free style project for your Play project:



After creating the project, you can add a small description in the next screen, where you configure the complete build process of the project:

The next step is to specify the source code repository to get the application from for each build. You should add the repository URL, optionally a branch, but also a cron-like time expression, how often the repository should be checked—every 5 minutes in this example by using ***/5 * * * ***, which is a cron-like expression. For more information about timing, read the crontab(5) man page on most Unix systems.

The **Poll SCM** option ensures that a new build is only triggered on changes:



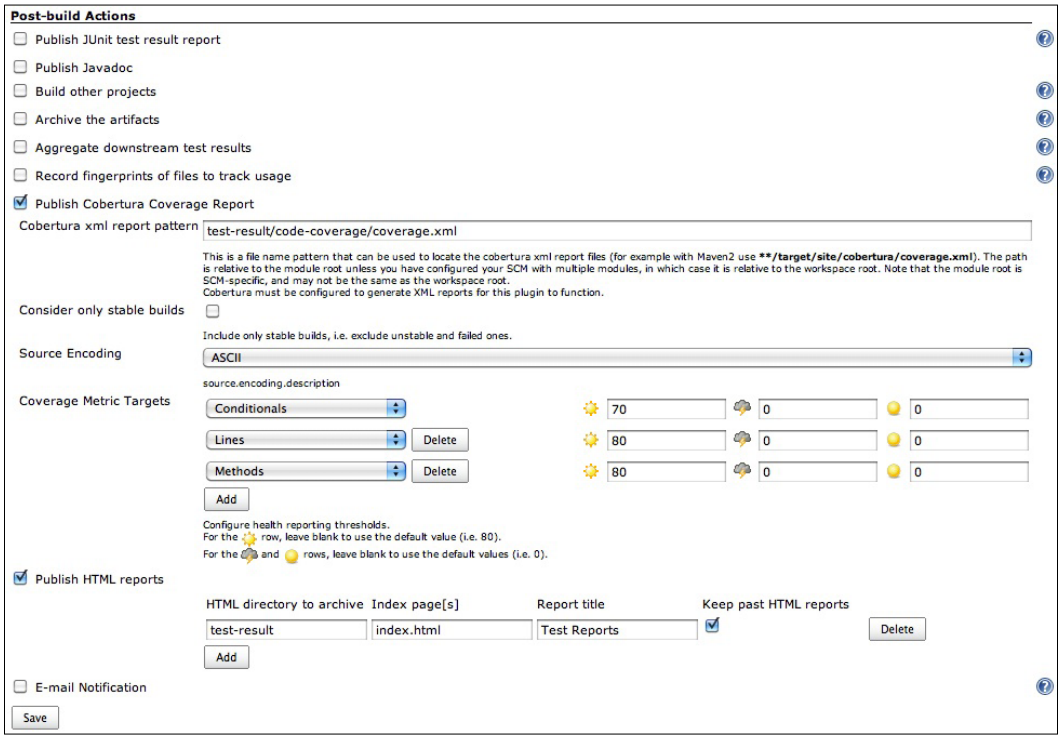
The next step is to care for the code execution itself. You need to configure the start of a shell script there, which should be put in the repository as well. This makes it easy to track changes for it. You can see the configuration for it in the preceding screenshot. The `ci-build.sh` script is called, which looks like the following:

```
#!/bin/bash

set -xe
PLAY_PATH="/path/to/play/installation/"

$PLAY_PATH/play deps
```

```
$PLAY_PATH/play auto-test

cd test-result
(
echo "<html><head><title>Test results</title></head><body><ul>"

for i in *.failed.html ; do
        if [ -e $i ] ; then
                echo "<li><a href=\"$i\">$i</li>"
        fi
done

echo "</ul><p><ul>"

for i in *.passed.html ; do
        if [ -e $i ] ; then
                echo "<li><a href=\"$i\">$i</li>"
        fi
done

echo "</ul></body></html>"
) > index.html

if [ -e result.failed ] ; then
  return -1
fi

cd -
```

The shell script runs the headless test runner of Play and creates an additional `index.html` which summarizes and links to all failed and passed tests. As a last feature it returns a non-zero code if failed tests occur, in order to notify the developers about unsuccessful test runs.

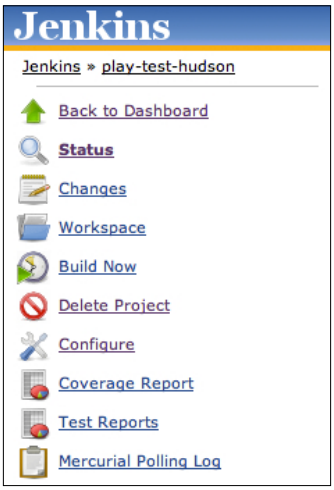
The last step is to enable Cobertura reports and to link to the HTML files of the test reports in order to examine the reasons for failed tests.



You need to configure the path to the Cobertura coverage XML file and can optionally set some thresholds to define the build as successful, if these have been passed. The Cobertura is at `test-result/code-coverage/coverage.xml`.

Below the Cobertura settings is the setup of the HTML publish plugin, which links to the `index.html` file in the `test-result` directory. This is the reason why the `ci-build.sh` script created this file. It has a nice overview now.

After setting up everything as needed and executing a successful build, you should have the following menu for every build record on the left, as shown in the following screenshot:



Now you have a Jenkins configuration, which builds your Play project on changes and creates Cobertura reports, as well as showing the test messages from the Play compilation.

How it works...

As all functionality is already explained, it is noteworthy here to mention the shortcomings. And there is a big one, depending on what your test and build philosophy is. Usually you have three states in Jenkins: success, build failure, and failing tests.

With this setup there are only two states: success and failure. There is no exact display of the amount of failing tests on the front page, only notification that something has gone wrong.

Usually a deployment only happens if all tests are successful, so it does not matter if either tests fail or the complete build fails.

There's more...

Jenkins has a very nice plugin system, which makes it pretty easy to write custom plugins and enhance them for a specific use case. If you need a better and deeper integration, you should check the plugins or even consider writing your own, if you have special requirements.

Other plugins might be useful

You should definitely check out the plugin page of Jenkins. There are lots of plugins and perhaps by the time of this writing, there is a plugin that allows you to count tests with the help of a shell script. This would allow some extension. Or the Play framework itself writes out the xUnit XML file like the Maven surefire plugin. The Jenkins plugin page is available under `http://wiki.jenkins-ci.org/display/JENKINS/Plugins`.

Check out the new play Jenkins plugin

Play plugin for Jenkins has recently been added. You can install it by selecting it from the plugin list and configuring the path of your Play installation in the system-wide configuration. It features a similar behavior as the shellscript in this recipe in order to supply the HTML test results to the client.

Test automation with Calimoucho

If you have not followed the Play framework mailing list, it is likely you have never heard of Calimoucho as software. Basically, Calimoucho itself is a Play application, which checks out other applications from a VCS, runs their tests, and reports this in a simple GUI. If you think Setting up a Jenkins instance and configuring it appropriately usually takes quite some time and might not be needed for a small play project. In such a case using Calimoucho makes sense.

In case you are wondering where the name is from: It is a drink, which consists of cola and red wine. The tool is way better than the drink.

You can find the source code of this example in the `examples/chapter7/test-calimoucho` directory.

Getting ready

First you should check out Calimoucho from `github`:

```
git clone https://github.com/guillaumebort/calimoucho.git
```

Now you can check the `calimoucho/conf/projects.yaml` file, where the current projects are placed.

When following the examples of this recipe, you should create an application for each version control system, as each system stores its version data differently. This might lead to clashes.

How to do it...

We will try to add support for some version control systems. The starting point is bazaar:

```
play new test-bazaar
cd test-bazaar
bzr init
bzr whoami "Your name <me@example.com>"
bzr add
bzr commit -m "Initial commit"
```

Now edit the `conf/projects.yml` file if the Calimoucho application in order to set up configuration paths:

```
models.Project(bazaar):
  name:                   Bazaar test project
  path:                   checkout/test-bazaar
  framework:              /path/to/your/play/installation/
  updateCommand:          bzr update
  versionCommand:         bzr revno %path
  versionPattern:         1.0r%version
  revisionDetailPattern:  http://bzr/test/browse/%version
  notifications:
                          - alexander@reelsen.net
```

Now put a copy of the repository into the `checkout/test-bazaar` (make sure the checkout directory actually exists) directory via:

```
bzr branch ../../test-bazaar
```

It is time to start Calimoucho now and see it working. After five minutes you should check the main page of the application, which runs on port 8084—also do not forget to visit the main page after starting the application in order to make sure it is running. The above mentioned 5 minute waiting time is the default time to wait before checking a repository for changes. You can configure this time via the `cron.checkInterval` parameter in your `application conf`.

To show you that Calimoucho actually works and updates, add a commit to the main repository (not the one in the checkout directory):

```
echo Foo > README
bzr add README
bzr commit -m "Added Readme"
```

Now you should see the second successful test run.

How it works...

As you can see, Calimoucho automatically updates the repository with the command provided in the **projects YAML** file. However, what it does not do is check out the repository initially. That would be a nice feature indeed. Basically, you need to configure five different aspects per build job:

- **framework**: The path to your Play framework installation. Changing this makes it easy to perform a build for a new Play framework version.
- **UpdateCommand**: The command to update the local repository to the most up-to-date revision.
- **versionCommand**: The command to get the current revision—may return anything unique.
- **revisionDetailPattern**: A URL where you can link to the direct source code repository.
- **notifications**: A list of e-mail addresses to send the results to. You have to fill the `app/views/Notifier/sendResult.txt` template file with some content in order to have this feature working.

After configuring it, all you have to do is make an initial checkout and from then on updating works automatically.

As there are more versioning systems available, you might want to check how you can use Git, mercurial, and SVN repositories as well.

There's more...

As there are plenty of version control systems available, the following paragraphs provide help to configure Calimoucho to also support Git, mercurial, and subversion.

Using git with Calimoucho

As Git is one of the most used versioning systems, so it should be listed here as well. Just change your project configuration to this one.

```
play new test-git
cd test-git
git init
git add .
git commit -m "initial import"
```

Go to the `calimoucho/checkout` directory and check out the freshly created repo:

```
git clone /path/to/absolute-repo
```

Add this to your `conf/projects.yml`:

```
models.Project(git):
  name:                 Git test project
  path:                 checkout/test-git
  framework:            /path/to/play-1.1/
  updateCommand:        git --git-dir=%path/.git pull
  versionCommand:       git --git-dir=%path/.git log --format=%h -1
  versionPattern:       g%version
  revisionDetailPattern: http://git/test/browse/%version
```

Using mercurial with Calimoucho

Mercurial is another distributed versioning control system just like Git, and is also used in many cases:

```
play new test-mercurial
cd test-mercurial
hg init
hg add
hg commit -m "initial commit"
```

Now go to the `calimoucho/checkout` directory and check out the project:

```
hg clone file:///path/to/repo
```

Add this to your `conf/projects.yml`:

```
models.Project(mercurial):
  name:                 Mercurial test project
  path:                 checkout/test-mercurial
  framework:            /path/to/play-1.1/
  updateCommand:        hg pull -u -R %path
  versionCommand:       hg log -l 1 --template {rev} %path
  versionPattern:       r%version
  revisionDetailPattern: http://hg/test/browse/%version
  notifications:
                        - alexander@reelsen.net
```

Using subversion with Calimoucho

Though subversion has already been around for a while, it is still used a lot. You can configure Calimoucho to check out any subversion based repository as well:

```
svnadmin create svn-repo
play new test-svn
svn import test-svn/ file:///absolute/path/to/svn-repo/test-svn/trunk
-m "initial import"
rm -fr test-svn
```

Now go to the `calimoucho/checkout` directory and check out the project:

```
svn co file:///path/to/svn-repo/test-svn/trunk test-svn
```

Add this to your `conf/projects.yml`:

```
models.Project(svn):
  name:                   SVN test project
  path:                   checkout/test-svn
  framework:              /path/to/play-1.1/
  updateCommand:          svn update %path
  versionCommand:         svnversion %path
  versionPattern:         r%version
  revisionDetailPattern:  http://svn/test/browse/%version
  notifications:
                          - your@email
```

Creating a distributed configuration service

As soon as your application and especially your application load gets bigger, you might want to share the load between several hosts. So you would setup several Play application nodes, all having the same configuration file, which is easily possible with Play. However, you would have to restart all application nodes, when you change one parameter. This is useful when you change something fundamental, like your database configuration. However there are cases where it is not useful to have a downtime of your application. This is where the principle of a centralised configuration service becomes useful. This recipe shows a simple example of how to implement such a service.

You can find the source code of this example in the `examples/chapter7/configuration-service` directory.

Getting ready

You should set up memcached and the example application written here should run on two different ports. So, after installing memcached, it might have been started by your distribution already, if you are using Linux. Otherwise you can start it by typing: the following

```
memcached
```

As you need more than one instance of a running Play application, you could install it on two nodes. And you need to enable memcached as the caching server. Both can be done in the `application.conf`:

```
memcached=enabled
memcached.host=127.0.0.1:11211

%app01.http.port=9001
%app02.http.port=9002
```

The first two lines enable memcached, and the last two lines make use of the so-called application IDs. You can now start the application with a special ID via this command:

```
play run --%app01
```

This instance will bind to port 9001. Now open another terminal, copy the directory of the application to another directory, change to the copied directory, and enter the following:

```
play run --%app02
```

Now you will have one memcached and two Play instances of your application running. Do not forget to keep the directories in sync, when making changes.

The example itself will have two features. First, it allows you to set parameters on any host, which are then cached and can be retrieved from any other host. Second, it implements a good heartbeat function, which allows the admin to check quickly which hosts are alive—this would be interesting for monitoring and alerting purposes.

How to do it...

Three routes are needed for this example:

```
GET     /nodes           Application.nodes
GET     /config/{key}    Application.getConfig
POST    /config/{key}    Application.setConfig
```

One job, which registers itself on application startup in the cache, is needed:

```
@OnApplicationStart
public class CacheRegisterNodeJob extends Job {

   public static final key = "CS:nodes";

   public void doJob() {
          Set<String> nodes = Cache.get(key, Set.class);
          if (nodes == null) {
                Logger.info("%s: Creating new node list in cache",
this.getClass().getSimpleName());
```

```
                nodes = new HashSet();
            }
            nodes.add(Play.id);
            Cache.set(key, nodes);
        }
    }
```

Another job, which features a heartbeat operation, is needed:

```
@Every("10s")
public class CacheAliveJob extends Job {

    private static final String key = "CS:alive:" + Play.id;

    public void doJob() {
            Date now = new Date();
            Logger.info("CacheAliveJob: %s", now);
            Cache.set(key, now.getTime(), "15s");
    }
}
```

The last part is the controller code:

```
public class Application extends Controller {

    public static void getConfig(String key) {
            String value = (String) Cache.get(key);
            String message = String.format("{ %s: \"%s\" }", key,
value);
            response.setContentTypeIfNotSet("application/json;
charset=utf-8");
            renderText(message);
    }

    public static void setConfig(String key) {
            try {
                    Cache.set(key, IOUtils.toString(request.body));
            } catch (IOException e) {
                    Logger.error(e, "Fishy request");
            }
            throw new Status(204);
    }

     public static void nodes() {
            Set<String> nodes = Cache.get("CS:nodes", Set.class);
            Map<String, List> nodeMap = new HashMap();
```

```
            nodeMap.put("alive", new ArrayList<String>());
            nodeMap.put("dead", new ArrayList<String>());

            for (String node : nodes) {
                  if (Cache.get("CS:alive:" + node) == null) {
                        nodeMap.get("dead").add(node);
                  } else {
                        nodeMap.get("alive").add(node);
                  }
            }

            renderJSON(nodeMap);
      }
}
```

How it works...

The simple part is—as you can see—using the cache as a node wide configuration service. In order to test this functionality, you should start two instances of this application and set the configuration parameter on the first node, while receiving it on the second:

```
curl -v -X POST --data "This is my cool configuration content"
localhost:9001/config/myParameter
* About to connect() to localhost port 9001 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 9001 (#0)
> POST /config/myParameter HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7
OpenSSL/0.9.8l zlib/1.2.3
> Host: localhost:9001
> Accept: */*
> Content-Length: 37
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 204 No Content
< Server: Play! Framework;1.1;dev
< Content-Type: text/plain; charset=utf-8
< Set-Cookie: PLAY_FLASH=;Path=/
< Set-Cookie: PLAY_ERRORS=;Path=/
< Set-Cookie: PLAY_SESSION=83af29131d389f83adc89cea9ba65ab49347e5
8a-%00___ID%3A138cc7e0-5e11-4a52-a725-4109f7495c8f%00;Path=/
< Cache-Control: no-cache
< Content-Length: 0
<
* Connection #0 to host localhost left intact
* Closing connection #0
```

As you can see, there is no data sent back to the client in the response body, and therefore a HTTP 204 can be sent back. This means that the operation was successful, but nothing was returned to the client. Now, checking the data can be easily done on the other host:

```
curl -v localhost:9002/config/myParameter
* About to connect() to localhost port 9002 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 9002 (#0)
> GET /config/myParameter HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7
OpenSSL/0.9.8l zlib/1.2.3
> Host: localhost:9002
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: Play! Framework;1.1;dev
< Content-Type: application/json; charset=utf-8
< Set-Cookie: PLAY_FLASH=;Path=/
< Set-Cookie: PLAY_ERRORS=;Path=/
< Set-Cookie: PLAY_SESSION=ad8a75f8a74cea83283eaa6c695aadb1810f1
5c3-%00___ID%3A256a3510-a000-45cb-a0a0-e4b98a2abf53%00;Path=/
< Cache-Control: no-cache
< Content-Length: 56
<
* Connection #0 to host localhost left intact
* Closing connection #0
{ myParameter: "This is my cool configuration content" }
```

By connecting to port 9002 with cURL the second Play instance is queried for the content of the cache variable. A (hand-crafted) JSON reply containing the configuration content is returned.

The second part of this recipe is the heartbeat function. It is pretty cool to play with. When checking the `CacheAliveJob`, you will see that it runs every 10 seconds, and puts some data in the cache with an expiry time of 15 seconds. This means, if after 15 seconds no new entry is inserted, it will vanish from the cache.

However, in order to be able to check whether a node is active, one must know which nodes were online before. This is what the `CacheRegisterNodeJob` does. It gets a set with nodes from the cache, adds its own name to this set, and stores it back in the cache.

So there is now one list with nodes which have been added upon start up to this list. Using this list with node names, and then checking whether there are any cache entries which show that these nodes are up, enables us to create some sort of heartbeat service.

If you did not stop any of your two nodes, just check whether they are up with cURL:

```
curl localhost:9002/nodes
```

The above should return this:

```
{"dead":[],"alive":["app02","app01"]}
```

Now it is time to stop one of the instances. Stop the one running on port 9002, wait a little more than 15 seconds and call cURL again on it:

```
curl localhost:9001/nodes
```

This time it should return this:

```
{"dead":["app02"],"alive":["app01"]}
```

To show you that the cache works as expected, you can even stop the other instance as well; start the second one again and you should be able to call cURL again (of course, always be aware of the 15 second timeout):

```
curl localhost:9002/nodes
```

The above will get you this:

```
{"dead":["app01"],"alive":["app02"]}
```

Be aware that if you are in DEV mode, the first reply of the nodes query in this case will return both hosts as dead. This is because the whole application only starts when you issue the request above, which in turn means that the first run of the `CacheAliveJob` has not been run yet, so you have to wait another ten seconds. This is a problem you will not meet in production.

Now put up the first instance again and you should get the response that both nodes are up. Again pay attention here, if you are in DEV mode and wondering why the node you are querying is still marked as dead.

There's more...

If you have taken a closer look at the jobs you might have spotted the flaw in the design. If you bring up two nodes in parallel, they might get the set of nodes of the cache, add their own node, and store it back. This of course might imply losing one of the two nodes. Unfortunately, the Play cache implementation does not help here in any way.

More on memcached

Memcached is pretty powerful. Play supports multi memcached hosts as cache per default, by setting more hosts in the configuration:

```
memcached.1.host=127.0.0.1:11211
memcached.2.host=127.0.0.1:11212
```

In order to understand this configuration a little background know-how on memcached is needed. Every memcached client (in this case the library included with Play) can have its own hashing function on how to distribute data to a cluster of memcache hosts. This means, both servers are completely independent from each other and no data is stored twice. This has two implications. If you use such a multi-node memcached setup in play, you have to make sure that every application node includes a configuration for all memcached hosts. Furthermore, you cannot be sure that any other library you are using for accessing memcached use the same hashing algorithm.

See also

If you are iclined to want to know more about cache, check out the recipe *Writing your own cache implementation* in *Chapter 6*, where you can write your own cache implementation.

Running jobs in a distributed environment

As you should be more than familiar with jobs by now, you might have implemented them in your own application. For example, take clean up jobs, which are weekly or hourly jobs that clear leftover or unneeded data that has accumulated over time out of the database. You've developed your application as usual, added this neat clean up job, and deployed it to a multi-node system. Now your job runs on every node at the same time, which is most likely not what you wanted. This wastes application resources or might even lead to lock problems, as many systems try to access the same database resources at the same time.

In this example the standard cache functionality will be used to overcome this problem.

The source code of the example is available at `examples/chapter7/distributed-jobs`.

Getting ready

All you need is an application running on two nodes, in which both use the same memcached instance for caching. The configuration for memcached setup is already commented out in the standard `application.conf` file.

How to do it...

An easy method is to define an abstract job, which makes sure that it only runs on one node, but leaves the logic to the concrete implementation of the job:

```
public abstract class MultiNodeJob extends Job {

   protected final String cacheName =  this.getClass().
getSimpleName();

   public abstract void execute();
```

```
    public void doJob() {
        boolean isRunning = Cache.get(cacheName) != null;
        if (isRunning) {
            Logger.debug("Job %s is already running", cacheName);
            return;
        }
        // Possible race condition here
        Cache.safeSet(cacheName, Play.id, "10min");

        try {
            execute();
        } finally {
            if (!isRunning) {
                Cache.safeDelete(cacheName);
            }
        }
    }
}
```

The concrete implementation boils down to a standard job now:

```
@Every("10s")
public class CleanUpJob extends MultiNodeJob {

    @Override
    public void execute() {
        Logger.debug("Doing my job of all greatness");
        // do something long running here
    }
}
```

How it works...

In order to create arbitrary jobs, which run only on one host out of an arbitrary number of applications, a helper class is needed. The `MultiNodeJob` class checks in the cache whether an element with the same name as the short class name exists. If this is the case, no execution will happen. If not, then the element will be set so no other nodes will execute this job.

The concrete implementation of any job consists only of the code to execute and the definition of how often it should run. It does not include any specific checks whether another node already runs this job. It needs to inherit from the `MultiNodeJob`.

As seen in the preceding code, the method `Cache.safeSet()` was used because it waits until the set has been executed, before going on. You could possibly change this to `set()`. What is far more important is the expiration date of the set key. If any node setting the key crashes while running this job, it might never be executed again because no one explicitly removes the set key from the cache.

As already written in the preceding comment, this solution is not one hundred percent correct, if you really need to make sure that the job runs exactly once at one time. If two nodes try to get the value of an non-existing element in the cache at the same time, both nodes will execute the job. This is a classical locking problem, which is hard to solve in multi node systems and cannot be solved with the current setup.

This problem gets more complex if you run jobs at a certain time via the `@On` annotation. Most likely the clocks of your servers are synced with an NTP time server, which means they will start pretty much at the same time.

In case you are using the `@Every` annotation, this problem will blur a little bit because the time between two jobs is actually the waiting time you defined in the annotation plus the execution time. So, if one node executes the job, and the other one does not, the difference in repeating times will start to differ a lot. If you do not intend this behavior at all, just use the `@On` annotation.

There's more...

Though this is a nice solution, it has already proven not to be perfect. Here are some possible areas for improvement.

Solving the locking problem

You cannot solve the above explained locking problem with the current cache implementation. You can however work around this problem. The easiest solution is to only allow one node to execute jobs. By setting the configuration property `play.jobs.pool=0`, no thread pool for jobs is created and no jobs except the synchronous ones on application startup are executed. You should however not forget to configure one node to execute jobs. This can easily be done with a specialized framework ID on one node. Another possible, but complex solution would be to go with events where a job triggers some event-based software such as ActiveMQ, which in turn decides who should execute this job. This is quite some engineering overhead however; it might be possible to live with this problem, or go the complex way of integrating a message queue if really needed. You might also create some better cache implementation (or a special implementation for locking), which acts as a single locking solution.

Changing cache clear times

It might not be a good idea to hard code the cache expiry time in the `MultiNodeJob`. If a job gets executed every five seconds and then stops for two hours because of one node crashing and not clearing the cache entry, you might wonder about unexpected results. Instead, you could use the time duration used in the `@Every` annotation and double or triple it. This would make sure the job is not run for a short period of time.

Running one Play instance for several hosts

As soon as you need multi tenancy support for your application, chances are high that every tenant will also run on its own host, such as `http://firsttenant.yourapp.com` and `http://secondtenant.yourapp.com`. For example, this way you can customize the application a little bit and the user who wants to log in does not have to supply any tenant information.

The source code of the example is available at `examples/chapter7/virtualhost`.

How to do it...

Put the following in your `routes` file:

```
GET    {tenant}.mysoftware.com/          Application.index
```

From then on you can use the client variable in your controller methods:

```
public static void index(String tenant) {
    renderText(tenant);
}
```

You can even configure a different handling of CDN-based content for development and production right, use mode in your routes file. Write the following in your `routes` file:

```
#{if play.Play.mode.isDev()}
   GET /public/ staticDir:public
#{/}
#{else}
   GET assets.myapp.com/ staticDir:public
#{/}
```

In order to access the content, you need to use the `@@` controller call to resolve to the absolute URL when templates are rendered:

```
<img src="@@{'/js/foo.js'}">
```

How it works...

If you want to test the first explained feature, you can use cURL like this and add your own host header, which is used to set the virtual host being used in HTTP 1.1:

```
> curl --header "Host: test.example.com" 127.0.0.1:9000/

test
```

There's more...

Hosting an application supporting multiple tenants on the same host for different tenants is generally not a good idea except those low in traffic and you do not have strict security constraints on the data your are storing. Think about whether it is a good idea to add such a complexity to your application, or if splitting systems would just make more sense. Maintenance of course might be significantly reduced, if you only need to handle one instance.

Implementing multi tenant applications using hibernate

Erik Bakker from Lunatech Research has written an excellent write up on how to use hibernate filters to implement multi tenancy with Play. Check it out at `http://www.lunatech-research.com/archives/2011/03/04/play-framework-writing-multitenancy-application-hibernate-filters`.

A virtual hosting module

There is a vhost module included in the Play framework modules list. However, it suffers from the lack of documentation as it includes JNI based libnotify code, which is not guaranteed to run on every operating system. This module allows you to have a directory with subdirectories, which resembles virtual hosts including static files, but also allows each domain to have its own datasource.

Using properties to be more flexible

As you can see in the preceding `routes` file, a hardcoded hostname might not be a problem, but still is somewhat inflexible. The great thing is that the `conf/routes` file is preprocessed by the template engine, so there is absolutely no problem to use an expression like:

```
GET    {tenant}.${play.configuration['host']}/         Application.
index
```

And adding this to your `application.conf`:

```
host=example.com
```

Just retry cURL call in the last section and check whether it works.

Forcing SSL for chosen controllers

Sooner or later you will have to include encrypted communication in your software. Today most sites merely use it for login purposes in order to submit the username and password in a secure way. This prevents eavesdroppers from sniffing them, even in a public wireless LAN. However, it is still possible to take over a session in a public WLAN, after the login. In order to prevent this, the full application must run with an SSL encryption. However, this will increase your system load.

The source code of the example is available at `examples/chapter7/ssl/example-app`.

Getting ready

You should have a configured Play application and followed the steps at: `http://www.playframework.org/documentation/1.1.1/releasenotes-1.1#https`.

Ensure that you can access your application at `https://localhost:9443/` as well as `http://localhost:9000`. The next step is to force a login controller to be always SSL.

How to do it...

Create a controller with a `@Before` annotation, which checks whether the request is actually SSL encrypted:

```
public class EnsureSSL extends Controller {
    @Before
    static void sslOrRedirect() {
        if(!request.secure) {
            redirect("https://" + request.host + request.url);
        }
    }
}
```

Now you can add this controller via the `@With` annotation to any other controller and make sure cleartext requests are redirected to SSL:

```
@With(EnsureSSL.class)
public class Registration extends Controller {
   ...
}
```

How it works...

As you can see there is almost no logic in the preceding code. You can check if it works by trying to access your controller via HTTP:

```
> curl -v www.test.local/login
* Connected to www.test.local (192.168.0.7) port 80 (#0)
> GET /login HTTP/1.1
>
< HTTP/1.1 302 Found
< Location: https://www.test.local/login
```

You should however always keep in mind, that it is useless to redirect a POST request from a cleartext to an encrypted connection, which already includes all the sensitive data. So just make sure that the redirection happens before the transmission of data. Use a sniffer like Wireshark or TCPDUMP if you are not sure about your current implementation. However, you should be able to understand what happens in your application anyway.

Implementing own monitoring points

As soon as you experience the first performance bottleneck in your application, there are two steps to carry out: analyze the problem and fix it. If you application design is not completely flawed, fixing usually takes a fraction of the time needed for analyzing. The analysis is split in two parts. Finding the occurrence of the problem in your platform and reproducing the problem reliably in order to fix it. Most likely problems will occur on production because real live problems will never be completely found by lab testing.

After you have fixed the problem, hopefully the operations department or even you would raise the rhetorical question about making sure this does not happen again. The answer is simple: Monitoring. You should always be able to return reliable statistics from the core of the application instead of relying on external measuring points such as database query times or HTTP request and response times.

There should be data about cache hit and miss times of your custom caches. There should be very application specific data like the amount of API login calls, or the count and top 10 of your daily search terms or your daily revenue. Anything remotely helping to gather problems should be easy to monitor.

The Play framework ships with **JAMon**, which is short for **Java Application Monitor** and provides exactly the infrastructure needed in order to integrate such monitoring points as easy as possible.

The source code of the example is available at `examples/chapter7/monitoring`.

Getting ready

You should be familiar with the output of `play status` before starting this recipe. Play already offers the minimum, maximum, and average run times as well as hit count of all of your controllers, error pages, jobs, and even all your custom tags. An already existing bottleneck is not needed, but might help with your concrete problem.

The example supplied with this recipe is a query to another system, which should help us to make sure that there are no performance issues with accessing an external API.

How to do it...

Imagine the following job, which gets remote data, transforms it from JSON to an object, and stores it in the cache. Now the task is to find out whether the complex JSON parsing lasts so long or retrieving the content from the remote system. This job shows how to do it:

```
@Every("5s")
public class RemoteApiJob extends Job {

   public void doJob() {
      Logger.info("Running job");
      Monitor monitor = MonitorFactory.start("RemoteApiCall");
      WSRequest req = WS.url("http://some.URL");
      HttpResponse resp = req.get();
      monitor.stop();

      // some other stuff like parsing JSON/XML into entities
      // ...
   }
}
```

This job starts and stops a monitor, which sets this monitor to automatically account for min, max, and average content. If you want to count for cache hits or misses, this functionality is not needed and you could go for the following in your code:

```
String foo = Cache.get("anyKey", String.class);

String monitorName = foo == null ? "cacheMiss" : "cachehHit";
MonitorFactory.add(monitorName, "cnt", 1);

Cache.set("anyKey", "anyContent", "5mn");
```

Instead of string names for the monitors you could also use the `MonKey` and `MonKeyItem` interfaces, if you are used to this from other applications using JAMon. However, using simple strings such as `cacheMiss` and `cacheHit` to identify your monitoring names should be sufficient in most cases.

How it works...

As you can see due to the rather short examples, integration of new monitors is only a few lines of code. As already mentioned, there will mostly be two different kinds of monitors in your code. There are simple monitors, which are counters, like cache hits or misses or the amount of currently open orders in your e-commerce shop system. The other kind of counter is the one who needs a data set to provide a real value to the reader of the monitor. If your system suffers from a peak one or two minutes a day and you are only taking the request

response values from this moment, you will not have a good statistical average, which might help you in finding bottlenecks. But when you access statistic values from a complete time duration, you might be able to analyze a peak load in a certain period which gives you an idea to only analyze this specific time period, instead of not being able to pinpoint the problem. Also, this allows you to measure performance decreases over time in a better way.

Both monitor types differ slightly in implementation. One can be called with `monitor.add()`, or you can directly use the `MonitorFactory` as done in the example. The other one needs a `start()` and a `stop()` method in order to calculate time differences correctly.

One of the bigger disadvantages of this example remains the same. Whenever you try to measure your code, you will clutter it when putting information such as monitors into it. You might be able to create some nice workaround via AOP and/or bytecode enhancement, but this will be a lot more work, for not too much change. Think about if this is really needed in your case.

You can get further statistics via the `Monitor` class. For example, the standard deviation, the last values for minimum, maximum, and average, or the date of the last access.

There's more...

Monitoring is often a crucially undervalued pinpoint in application development, as this is never a functional requirement. Make sure you know your application well enough in order to find out upcoming operations problems as soon as possible.

More about JAMon

Although, JAMon is currently not in active development (the last release is from 2007), it serves its purpose well. You can find the complete documentation of JAMon at `http://jamonapi.sourceforge.net/`.

See also

After adding new monitors, the next logical step is of course to really monitor such data. Read on for how to monitor this in Icinga and Munin.

Configuring log4j for log rotation

A big difference in production environments is the logging aspect. When testing your application locally, you start it via `play start` or `play test` and check the console output every now and then. On production system the application is started with `play run` and forked into the background from then on. Log output is written to the logs directory inside the application.

As the default log4j configuration in Play logs to the console, you should always go with a custom log4j file in production deployments.

How to do it...

There are several possibilities to do log rotation because it is being built into log4j. However, the most common ones are either rotating if the file has reached a certain limit, or does daily log rotation. This example works for file size based rotation:

```
log4j.rootLogger=ERROR, Rolling
log4j.logger.play=INFO # default log level

log4j.appender.Rolling=org.apache.log4j.RollingFileAppender
log4j.appender.Rolling.File=logs/application.log
log4j.appender.Rolling.MaxFileSize=1MB
log4j.appender.Rolling.MaxBackupIndex=100
log4j.appender.Rolling.layout=org.apache.log4j.PatternLayout
log4j.appender.Rolling.layout.ConversionPattern=%d{ABSOLUTE} %-5p ~
%m%n
```

You can change it to daily (or even hourly rotation) by writing the following:

```
log4j.appender.Rolling=org.apache.log4j.DailyRollingFileAppender
log4j.appender.Rolling.DatePattern=yyyy-MM-dd
```

If you need hourly rotation due to heavy log writing, use `.yyyy-MM-dd.HH` as `DatePattern`.

In case you need class based logging, you may not use the `play.logger.Logger` class, but you have to implement your own logger in your class like the following code:

```
private static final Logger log = LoggerFactory.getLogger(YourClass.
class);
```

Use the logger now as every normal log4j logger in your source code. This means you will lose the varargs style logging, which comes with the nice Play logger, as it is wrapped around log4j.

Then add this to your `log4j.properties`:

```
log4j.logger.yourPackage.YourClass=DEBUG
```

How it works...

Logging is simple, as long as you keep it simple. You can possibly have arbitrary complex log4j configurations with different rotation and log levels per class. You should never have more than a few log files. Otherwise, you will lose overview pretty quick. A good rule of thumb is to talk to the system engineers, what kind of logging they want and need, as they are more likely to wade through these files instead of the developers.

If you really wish to (as you might be used to or you have old configurations lying around) you can also put a `log4j.xml` file into the `conf/` directory of your application. It will work as well as the `log4j.properties` file.

It is important to mention that the log4j configuration does not get reloaded on each change, especially if you are in production mode. You might be used to that, if you are using JBoss or other application servers. If you really need to change this, you can recompile the Play framework and change the `play.logger.Logger` class from:

```
PropertyConfigurator.configure(log4jConf);
```

To:

```
PropertyConfigurator.configureAndWatch(log4jPath);
```

This checks every 60 seconds whether the log4j configuration file has changed and reloads it. Not all changes are reflected though. You should check `play.logger.Logger.setUp()` for more information.

There's more...

As there is almost no Play specific part in logging, there is not much extra information to be listed.

More about log4j

You can read a whole lot more about log4j at `http://logging.apache.org/log4j/`.

Default log levels of others frameworks

Inside of the Play source a log4j configuration has already been defined. It defines several log levels by default, for example for Hibernate, Spring, Quartz, DataNucleus, Apache HttpClient, or OVAL. If you experience problems with one of these in logging, you should check this exact log4j configuration in the source code. You can grab the configuration file from `framework/src/log4j.properties` in your Play framework folder.

Integrating with Icinga

External monitoring is one of the key tasks of the operations department. System engineers have to be able to measure performance degradation in the live setup in order to inform the developers, who in turn have to seek for solutions for these problems. Without monitoring no measurement is possible. Without measurement no debugging will happen. Icinga provides a pretty simple kind of monitoring. You can check whether your service, which is in our case the Play application, responds to a simple request or has to be marked as down.

One of the tools used for such tasks is Icinga, which is used for monitoring of whole networks and service landscapes. Icinga is a fork of the common Nagios monitoring system. It has been forked due to problems in the open source development process.

The source code is available at `examples/chapter7/ssl/example-app` for the example application used and in `examples/chapter7/icinga` for the Icinga specific files.

Getting ready

You need an up and running instance of Icinga. No additional tools are needed, as Icinga is shipped with everything needed.

The example listed here checks the output of the / URI. It checks for the occurrence of the string `Is secure: true` or `Is secure: false`. The string is created in the template by printing out the `${request.secure}` variable. Using this it is simple to create a HTTP as well as a HTTPS check. Therefore, you should make sure that your Play application is configured with SSL support as well.

How to do it...

Create a Play specific Icinga configuration in the `objects/` directory. On Ubuntu Linux you could create the following file in `/etc/icinga/objects/play_icinga.cfg`:

```
define host {
  use                  generic-host
  host_name            puck
  address              192.168.0.2
}

define service {
  use                  generic-service
  host_name            puck
  service_description  HTTP/9000
  check_command      check_http!-p 9000 -s "Is secure: false"
}

define service {
  use                  generic-service
  host_name            puck
  service_description  HTTPS/9443
  check_command    check_http!-p 9443 -S -s "Is secure: true"
}
```

How it works...

As you can see, the configuration is pretty short. Basically, there are two service checks defined for the same target host, which has the name `puck` and is reachable at `192.168.0.2`. The `check_command` parameter defines what kind of check to execute. The first check connects to `http://192.168.0.2:9000/` and ensures the defined string, which marks the request as cleartext HTTP, occurs in the response. If this is not the case, the test will not succeed. The same happens to `https://192.168.0.2:9443/`, however the `-S` parameter tells Icinga to handle this as an SSL connection and the string searched in this case marks the request as secure.

As you can see, this check is pretty simple, you could possibly go further and check the response times to define a maximum allowed latency, before the service is marked as `CRITICAL` or `WARN` in the Icinga system. This helps you to keep track of service level agreements.

There's more...

Icinga is far more complex and used for complete network monitoring, so you should read about it.

Find out more about Icinga

Icinga is a pretty feature complete solution. Check more and especially its documentation at `http://www.icinga.org/`.

See also

If you need to monitor more information than a simple up/down state, you should read on and check the integration with Munin, where it is used to monitor controller runtimes.

Integrating with Munin

Munin is an ideal companion to Icinga, as it goes further than it. Munin is merely a visualizer of metrics, rather than a simple checker. As soon as your service is up and running you might want more information about it. And you want it to be OSI layer 8 compatible. Management should also know what is going on.

Munin with its default install is able to monitor stuff like disk usage, IO, CPU load, and memory usage, but it does not ship with any Play specific configuration. From a Play developer and administrator point of view, you want to see the output of `play status` graphed over time. This is exactly what Munin is made for. This example will graph the minimum, maximum, and average response times of controllers.

The source code is available at `examples/chapter7/ssl/example-app` for the example application used and in `examples/chapter7/munin` for the Munin specific files.

Getting ready

You should have an up and running Munin system somewhere, with access to added plugins. Due to Munins distributed architecture, there are two possibilities of setting Munin up to gather data. First, you could install an agent on the system Play is running on, and sending the data over to the Munin server. Second, you could query the `/@status.json` URI of the Play application. This example opts for the second way.

Again we will go through this recipe by using Ubuntu Linux as the monitoring system. Therefore, the paths of configuration files on other Linux distributions and server systems may vary.

You also need some Perl modules installed in order to make this work. In the case of Ubuntu Linux the packages needed are `libjson-perl`, `libdigest-hmac-perl`, and `libwww-perl`.

How to do it...

The first step is to add Play to your Munin node configuration, and append this to the munin-node file, which is placed under `/etc/munin/plugin-conf.d/munin-node`:

```
[play]
user munin
env.secret YOURSECRETHERE
```

Copy the `secret` from your `application.conf` file directly into this configuration, as the secret is needed to access the status page from remote.

Now create the check itself. Put it into the directory where all other checks reside or into `/etc/munin/plugins/play` and make sure it is flagged executable. If this is not the case, do this by entering `chmod 755 /etc/munin/plugins/play`:

```
#!/usr/bin/perl

use strict;
use Munin::Plugin;
use LWP::UserAgent;
use Digest::HMAC_SHA1;
use JSON;

my $secret = $ENV{'secret'};

my $hmac = Digest::HMAC_SHA1->new($secret);
$hmac->add('@status');
my $digest = $hmac->hexdigest;
```

```
my $ua = LWP::UserAgent->new;
$ua->timeout(10);
$ua->default_header( 'Authorization' => "$digest" );

my $r = $ua->get('http://192.168.0.2:9000/@status.json');

my $json = new JSON->decode($r->content);

if ( defined($ARGV[0]) and $ARGV[0] eq "config" ) {

    foreach my $controller(@{$json->{"play.CorePlugin"}-
>{"monitors"}}) {
        my $name = $controller->{"name"};
        if ($name !~ /\(\)/) {
            next;
        }
        $name =~ s/^(.*)\(\).*/$1/;
        my $classMethod = $name;
        $name =~ s/\./_/;

        print "multigraph runtimes_$name\n";
        print "graph_title Play Controller runtimes $classMethod\n";
        print "graph_vlabel ms\n";
        print "graph_scale no\n";
        print "graph_category play\n";
        print "graph_info Statistics about min/max/avg controller
rates\n";
        print "graph_args -l 0\n";

        print $name."_min.label $name min\n";
        print $name."_min.draw LINE2\n";
        print $name."_max.label $name max\n";
        print $name."_max.draw LINE2\n";
        print $name."_avg.label $name avg\n";
        print $name."_avg.draw LINE2\n";
    }

} else {

    foreach my $controller(@{$json->{"play.CorePlugin"}-
>{"monitors"}}) {
        my $name = $controller->{"name"};
        if ($name !~ /\(\)/) {
            next;
```

```
            }
            $name =~ s/^(.*)\(\).*/$1/;
            $name =~ s/\./_/;

            my $avg = $controller->{"avg"};
            my $min = $controller->{"min"};
            my $max = $controller->{"max"};

            print "multigraph runtimes_$name\n";
            print $name."_min.value %d\n", $min;
            print $name."_max.value %d\n", $max;
            print $name."_avg.value %d\n", $avg;
        }
    }
```

Restart Munin-node via `/etc/init.d/munin-node restart` or alternatively use upstart after storing the Perl script.

How it works...

Before explaining what happens here, you can check whether your configuration is working at all. Use Telnet to connect to your Munin server on port 4949 and check for configuration and returning data:

```
> telnet localhost 4949
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
# munin node at yourServer
config play
multigraph runtimes_Application_index
...

fetch play
multigraph runtimes_Application_index
Application_index_min.value 3
Application_index_max.value 253
Application_index_avg.value 132
...
```

The bold marked lines are those typed in. The response of entering `config play` shows the configuration of the Play service, while the response of typing `fetch play` parses the data specific output of `play status` and returns it as needed for graphing. If you get similar results, just wait a few minutes for the next updates of your Munin server and you should see a nice graph for each controller, which yields the minimum, maximum, and average response times.

Back to the question—how is this all working? A Munin plugin has to implement two parts. First, a configuration setup needs to be returned if the plugin is called with a `config` parameter. The other part is to return the data needed to graph. In both cases the plugin code has to connect to the Play application in order to get a list of the currently active controllers. This means you can add or remove controllers, and the plugin will always cope with this. There is no need to add new controller methods somewhere manually in the Munin setup.

If you have read the configuration specific code of the plugin, you will see it uses another feature of Munin called multigraph. Usually one plugin can only emit one graph. However, it does not make too much sense to have all of your controllers packed into one graph. As soon as one controller is way slower than all the others, which might be an event intended in case of a big calculation or PDF creation, all your other graphs will not be readable at all when looking at the Munin GUI. The multigraph plugin allows having an arbitrary amount of graphs for one plugin, exactly what is needed.

I will not explain the Perl code too much. It parses the needed parts of the JSON response, checks whether the monitor name includes `()` which marks it as a controller and includes it then. It uses some regular expressions to beautify it for human reading. The only unknown part may be the use of the HMAC digest for the secret set in the configuration file. The digest is created by using the Play application secret as key for the message `@status`. The result is used in the `Authorization` HTTP header in order to authorize and get all needed data from the Play application. This is exactly how the command `play status` works as well.

There's more...

Munin is an incredibly flexible tool and should be used by you or your operations department to find out a great deal about your application. After you have added your monitoring points, as shown earlier in this chapter, this is one of the best ways to visualize such data.

Find out more about Munin and its plugins

Munin has tons of plugins, which might help you to graph some of your Play specific data further or to sharpen this example. Check Munin at `http://munin-monitoring.org/`.

Add and graph hit counts, tag runtimes, running jobs, and so on

As you might have noticed, only the most needed data has been used here to graph. The `play status` command includes lots of interesting data, waiting to be graphed. For example, each controller also includes the total hit count since starting the application, or the runtime of each tag, which might help you to decide whether to create a fast tag out of it. Also the status output shows scheduling data about your jobs. All this might be very useful to graph. All you have to do is to extend this or create a new Munin plugin based on the preceding one.

Setting up the Apache web server with Play

Apache is the most widely used web server. If you are already using it, you can set up Apache httpd as a proxy for your Play application, instead of exposing it to the Internet directly.

This recipe shows you how to configure SSL and non-SSL Play applications together with the Apache httpd. This chapter features a couple of recipes on how to use a web server as a proxy for Play. In any configuration example one assumption is made: You want to use SSL, but you do not want to use the built-in SSL support of the Play framework, because you are using a proxy in front of your Play application. The forwarding from the external SSL port is done to the internal Play application. However, the internal application does not use SSL. This prevents doing encryption twice or redirecting SSL directly to the Play application. However, you still have to make sure the Play application handles the request as if it came via HTTPS.

The source code of the example application is available at `examples/chapter7/ssl/example-app` as well as for `examples/chapter7/ssl/example-app-2` for the second application instance, which is needed for clustering. The configuration files for the web server are put at `examples/chapter7/apache`.

Getting ready

You should have an Apache 2 web server up and running. Furthermore, you should have the following modules enabled: `mod_ssl`, `mod_headers`, `mod_proxy`, `mod_proxy_http`, `mod_proxy_balancer`. The last one is only needed in case you want to do load balancing. When using a Debian derived distribution like Ubuntu, you can use the `a2enmod` command, for example `a2enmod ssl` to enable the SSL module.

How to do it...

The only thing you need to change on your Play `application.conf`, is the configuration about proxies, which redirect to Play. The IP referenced here is the one from the host Apache runs on:

```
XForwardedSupport=192.168.0.1
```

This is a simple Apache configuration for redirecting from external port 80 to internal IP and port 9000 of your Play application.

Be aware that the configuration snippets are highly specific and might not work with your Apache configuration. If you have a web server with several virtual domains you have to specify a name in the `<VirtualHost>` directive. The snippets here have been tried without any other virtual hosts running on the apache instance. Please refer to the Apache documentation when having a more complex setup.

```
<VirtualHost *:80>
      ProxyPreserveHost on
      ProxyPass / http://192.168.0.2:9000/
      ProxyPassReverse / http://192.168.0.2:9000/
</VirtualHost>
```

Now a configuration for redirecting an external SSL connection to internal HTTP, but still making sure it is handled as SSL by the Play application:

```
<VirtualHost *:443>
        SSLEngine on
        SSLCertificateFile    /etc/ssl/host.crt
        SSLCertificateKeyFile /etc/ssl/host.key

        ProxyPreserveHost on
        ProxyPass / http://192.168.0.2:9000/
        ProxyPassReverse / http://192.168.0.2:9000/
        RequestHeader set "X-Forwarded-SSL" "on"
        RequestHeader set "X-Forwarded-Proto" "https"
</VirtualHost>
```

The main difference here is the enabling of SSL and adding a special header, which will tell Play to mark this request as secure.

The next possible step is to cluster your application by running it on several nodes:

```
<Proxy balancer://playcluster>
        BalancerMember http://192.168.0.2:9000
        BalancerMember http://192.168.0.3:9000
</Proxy>

<VirtualHost *:80>
       ProxyPreserveHost on
       ProxyPass / balancer://playcluster/
       ProxyPassReverse / balancer://playcluster/
</VirtualHost>
```

How it works...

The Play configuration itself is merely a security feature. The support for proxy forwarding should be done explicitly. Instead of a single IP address you can also supply a comma separated list of IP addresses.

Depending on your setup, for example either by using localhost or only IP addresses, you also might have to use the `ProxyPassReverseCookieDomain` to make sure session cookies are set and handled correctly. Furthermore, it is not the best idea to use a wild card virtual host definition, but rather use name or IP-based virtual hosts. This has just been done out of convenience in this case. Also you should make sure to really use the `RequestHeader` instead of the `Header` directive to add the needed headers, as this directive ensures the headers are set before the request is forwarded to Play.

If you are seeking an answer about why the SSL code works and is then set up correctly, the answer is simple. This is a framework convention. You can read about it directly in the source at `play.mvc.Http.Request.parseXForwarded()`.

Regarding load balancing, you need to replace the real hostname in the proxy-specific directive with the balancer configuration. As the Play framework has been designed as a real share nothing system, there is no need to add complex configurations like sticky session IDs, where load balancing is done by keeping the session ID of a node mapped to the application server instance, which always means to keep state on the redirect host. The configuration for SSL has to be done accordingly.

Also be aware that the trailing slash in the `ProxyPass` and `ProxyPassReverse` directives is absolutely essential, regardless of whether you are doing balanced or single forwarding, because you have to match to a URL in this case.

There's more...

As Apache is widely used, there is not much point in showing off other features that aren't needed. However, you can even do transparent upgrades of your applications with zero downtime!

Transparent upgrade of your application

As this feature of upgrading your application without any downtime is already written in the documentation, it does not make any sense to talk about it in this book, so just check out `http://www.playframework.org/documentation/1.1.1/production#Apacheasafrontproxytoallowtransparentupgradeofyourapplication`.

Setting up the Nginx web server with Play

Nginx is a high performance web server and proxy, which is used by many big sites. If you need Nginx, you either have a high performance site, or you like the asynchronous event driven approach of it, which ensures a consistent usage of memory almost regardless of the connection amount. Nginx comes with a ton of features, but most likely, only the proxying and SSL features are needed in this recipe.

The source code of the example application is available at `examples/chapter7/ssl/example-app` as well as for `examples/chapter7/ssl/example-app-2` for the second application instance. The configuration files for the web server are put in `examples/chapter7/nginx`.

Getting ready

Nginx should be installed, and up and running. Execute `apt-get install nginx` on any Debian derived distribution.

How to do it...

Let us start with a single external port 80 to internal port 9000 redirection:

```
server {
  listen       80;

  location / {
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Server $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_pass http://192.168.0.2:9000;
    proxy_redirect http://192.168.0.2:9000/ http://your.external.
domain/;
}
```

The next step is to get SSL up and running:

```
server {
  listen               443;
  server_name          your.server.name;
  ssl                  on;
  ssl_certificate      /etc/nginx/host.crt;
  ssl_certificate_key  /etc/nginx/host.key;
  ssl_protocols        SSLv3 TLSv1;
  ssl_ciphers          HIGH:!ADH:!MD5;
```

```
    location / {
      proxy_set_header Host $host;
      proxy_set_header X-Forwarded-Host $host;
      proxy_set_header X-Forwarded-Server $host;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Ssl on;
      proxy_set_header X-Forwarded-Proto https;
      proxy_pass http://192.168.0.2:9000;
    }
  }
```

As usual, load balancing is the last part of the setup:

```
  upstream www.test.local {
    server 192.168.0.2:9000;
    server 192.168.0.3:9000;
  }

  server {
    listen       80;

    location / {
      proxy_set_header Host $host;
      proxy_set_header X-Forwarded-Host $host;
      proxy_set_header X-Forwarded-Server $host;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_pass http://www.test.local;
      proxy_redirect / http://www.test.local/;
    }
  }
```

How it works...

While taking a closer look at the setup, there are almost no differences to the Apache setup except for the header rewriting. A closer look has to be taken at the `proxy_redirect` directive, as this is changing depending whether you are load balancing or not.

If you need load balancing support, you should make sure that you name the cluster the same as your site. This allows a far more readable configuration, when using the `proxy_pass` directive.

The SSL configuration also sets all needed headers, where in most cases, some predefined variables are used, like the `$host` one. For a complete reference go to `http://wiki.nginx.org/HttpCoreModule#Variables`. Furthermore, you should make sure that the `server_name` directive in the SSL configuration matches with the SSL certificate host name.

There's more...

You could possibly use Nginx for many more things because it has support for memcached, MPEG, and flash streaming or WebDAV.

More information about Nginx

Nginx had the problem of not having too much documentation. However, today the website at `http://www.nginx.org` as well as the wiki at `http://wiki.nginx.org/` show you most of the configuration options. Furthermore, there is also a good book named Nginx HTTP Server about Nginx, which is also released by Packt Publishing.

Better load balancing

Nginx has a nice module called fair balancer that keeps in mind which node has the lowest load by counting currently processed requests, and thus balances all requests in an even manner instead of using pure round-robin based load balancing.

Transparent upgrade of your application

You can remove the node to be upgraded out of your load balancing configuration and call `nginx -s reload` to force it to reread its configuration. Then update your application and re-enable it in the configuration file. Reload the configuration again and you are done.

Setting up the Lighttpd web server with Play

Lighttpd (pronounced lighty) is another lightweight web server, similar to Nginx, as it also handles requests asynchronously. It is a matter of taste, and which web server you prefer, so this recipe is here for the sake of completeness.

The source code of the example application is available at `examples/chapter7/ssl/example-app` as well as for `examples/chapter7/ssl/example-app-2` for the second application instance. The configuration files for the web server are put in `examples/chapter7/lighttpd`.

Getting ready

You should have a lighttpd installed on your system and have the the `accesslog`, `proxy`, and `ssl` modules enabled. When using a Debian derived operating system, you can use the `lighttpd-enable-mod` command, for example: `lighttpd-enable-mod ssl`.

How to do it...

Simple redirection works like this—put it into `/etc/lighttpd/conf-enabled/10-proxy.conf`:

```
$HTTP["host"] == "www.test.local" {
        proxy.server  = ( "" => ((
                 "host" => "192.168.0.2",
                 "port" => 9000
        )))
}
```

SSL is also set up pretty quickly; however, there is one major difference. Instead of having two different files for certificate and key file, you have to merge both files into one. Concatenate them together like this:

```
cat host.crt host.key > host.pem
```

Put your SSL specific configuration into `/etc/lighttpd/conf-enabled/10-ssl.conf`:

```
$SERVER["socket"] == "0.0.0.0:443" {
    ssl.engine     = "enable"
    ssl.pemfile    = "/etc/nginx/host.pem"

    $HTTP["host"] == "www.test.local" {
        proxy.server  = ( "" => (
            (
                "host" => "192.168.0.2",
                "port" => 9000
            )
        ))
    }
}
```

As you might have already noticed, the proxy configuration looks as if you could put more than one host into it. Exactly this is the case, which makes configuration very simple. You can just enhance your current configuration to enable load balancing:

```
$HTTP["host"] == "www.test.local" {
        proxy.balance = "round-robin"
        proxy.server  = ( "" => (
                (
                 "host" => "192.168.0.2",
                 "port" => 9000
                ),
                (
```

```
                    "host" => "192.168.0.2",
                    "port" => 9001
                  )
          ) )
    }
```

How it works...

This configuration is by far the shortest, as it does most of the things which need to be done manually in Nginx and Apache configuration, automatically. There is no need to specify special redirect headers, and there is no need to specify special headers for SSL rewriting, as it's all implicit, which makes it by far the laziest solution.

If you are wondering about the somewhat strange configuration of proxy servers with the empty key for the servers, be aware that you can configure redirection for certain file extensions. This is actually pretty useless with a Play application. If you have designed nice looking URLs in your `conf/routes` file you will not have any file endings.

There's more...

Though lighttpd is not used as much as Apache or Nginx, you might want to consider it as a viable alternative in your web server landscape.

Learn more about lighttpd

You can find more information about lighttpd at `http://www.lighttpd.net/`, together with an impressive list of high performance sites currently using it.

Multi-node deployment introduction

Deployment is always a hot topic. There are developers, most of the time including myself, who do not worry about deployment, until it becomes a problem. Updating one node is never too much of a problem. The average developer can do it in a couple of minutes. But can you still do this with a multi node installation. Do you suddenly need half a day?

As this is the last recipe, I will not provide you with a solution, because it does not exist. There are a dozen ways of deploying nodes:

- Check out the application from VCS and run it. Pretty simple. However, do you also keep Play itself in the VCS? How do you keep that updated?
- Start some shell script from one main node, which copies everything via `scp` or `rsync`. Great, but you still need to restart. How do you handle database upgrades?

- You can use BitTorrent. Twitter does this to deploy tens of thousands of nodes in well under a minute. They use a self-written system called murder, which is available at `https://github.com/lg/murder`.
- You might want to create packages before distributing your application or the framework itself across your nodes. This makes rollbacks pretty simple, while still making sure that only one installation is on your server. Lunatech Research has a nice module for this for Debian Linux, which can be downloaded at `http://www.lunatech-labs.com/open-source/debian-play-module`.

How to do it...

You should definitely use an init script, which is executed when the system starts. Possibly a script like this:

```
#!/bin/sh

PLAY=/usr/local/bin/play
APP=/usr/local/play/myapp
NAME=myapp
DESC="play application"

set -e

case "$1" in
    start)
        echo -n "Starting $DESC: "
         $PLAY start $APP
         echo "$NAME."
        ;;
    stop)
        echo -n "Stopping $DESC: "
         $PLAY stop $APP
         echo "$NAME."
        ;;
    *)
        echo "Usage: $0 { start | stop }" >&2
        exit 1
        ;;
esac

exit 0
```

This script requires the correct path of the Play binary and the correct path of the application. If both are symbolical links, you will always be able to point to the most up-to-date application as well as your current Play installation.

This way you could put all your Play installations into `/usr/local/play-inst/` and have directories like play-1.1 or play-1.2 Similar to this setup you could have `/usr/local/play` directory, where `/usr/local/play/myapp` is a symlink pointing to the application to run in the directory. A sample application directory name could include a timestamp like `/usr/local/play/myapp-20110506`.

As a preparation for every deployment, you could trigger the checkout of the VCS remotely, so that the application is already put into such a directory, when you switch to your updated application. You should also make sure, that your versioning system always grabs only the latest copy of your repository and not its complete history. `Git` has a so-called `depth` option, and bazaar features a lightweight checkout. Mercurial features this with some hacking around the convert extension.

If you do not want to use VCS on your application nodes, you could alternatively create an archive and put it up there.

How it works...

So let's sum up the preparation for each deployment here, in order to give you a guideline:

1. Optional: Copy a new framework version to `/usr/local/play-inst/` (out of your VCS, as a tarball via `rsync/scp`).
2. Copy a new version of your application to `/usr/local/play/myapp-$timestamp`—set your timestamp correctly, so none of your old installations are overwritten.
3. Stop your application.
4. Change the symlink of the Play framework, if you did an upgrade.
5. Change the symlink of the Play installation.
6. Execute scripts to be executed. Tasks like database migration should be executed only once and not for all nodes.
7. Start your application, make sure it starts, check the logs, and make sure the web server in front marks the application as up again.

There's more...

Multi node scalability in production is quite a complex topic, even with a shared nothing system like Play. You will run into dozens of obstacles, so always think ahead of problems instead of watching them flying in after deployment.

Add centralized logging to your multi node installation

Whenever you run multiple application nodes, it is pretty hard to track exceptions or even web sessions, as requests switch arbitrarily between hosts. A possible solution to this is a centralized log system, like Splunk. For more information on Splunk, check out `http://www.splunk.com/`. A possible problem with Splunk is, it not being a free software. The free version has only a limited amount of features, like only logging 500 megabytes of data a day.

Distributing configuration files in a different way

If you want to keep your configuration files away from your version control repository, you need to have another way to distribute them. You might want to check out some configuration management tools like puppet, which is available at `http://www.puppetlabs.com/`

Another more recent alternative is chef, which is newer than puppet, but does not yet feature as many plugins. Chef is available at `http://wiki.opscode.com/display/chef/Home`.

If you want to keep your data protected, like database passwords, this strategy might make sense, it also introduces quite some overhead because you have to keep two sources, namely the application and its configuration, in sync.

Further Information About the Play Framework

Further information

Before leaving you alone to try, copy, and test all the sources and samples included, you might be interested in some further information.

As already mentioned in the foreword, I would like to repeat that you are encouraged to use, copy, paste, change, improve, and of course extend all of the sources here. If you plan to create any module of the examples provided here, feel free to do so. Contributing is one of the core aspects to a viable open source framework, so do not hesitate in doing it. The worst thing would be no one using your module, which would have been the same if you never released it.

As a last hint I would like to point to some links you should know about, if you plan to use the Play framework in production.

Links

The following links feature some additional information about the Play framework. It is also a list of interesting articles and resources, which were not mentioned throughout this book, but should help you to get to know the framework better.

`http://www.playframework.org`

The Play framework homepage features a short introductory video, documentation, code snippets, binary downloads, modules, and many more.

`http://www.playframework.org/community/planet`

This link is an aggregator for community created content. Such content varies greatly, ranging from small how to's about a specific functionality to bigger articles explaining the usage of modules.

`https://groups.google.com/forum/#!forum/play-framework`

The Play framework's Google group is the most viable source of information. Be aware that this is a high traffic list, featuring far more than 1000 mails a month. Do not hesitate to ask questions as there are lots of active actors on the list.

`http://stackoverflow.com/tags/playframework`

Stack overflow features in a lot of questions about the Play framework, more than 250 currently.

`https://github.com/playframework/play`

If you need to take a look at the source code, check out Play framework on GitHub. There used to be a bazaar repository on Launchpad, but GitHub is the primary resource since play 1.1.

`https://github.com/alexanderstrebkov/auditlog/wiki/`

`http://blog.matthieuguillermin.fr/2011/02/utiliser-envers-avec-play-framework/`

These two links show different solutions on how to add auditing to your application by using Hibernate Envers. You simply add an annotation to your entities and configure specific entity listeners. Note, that the second link is actually in French, but there is no problem using Google Translate for this. The second URL also does not require an own module for these features.

`https://github.com/sebcreme/strike`

Strike is a mobile framework on top of SASS, which resembles iPhone behavior. Going to the `/@emulator` URL shows you basically how your application would look, when rendered on a phone. It seems to not work with the current SASS version however.

`https://github.com/hz/play-groovy/`

This is quite an interesting module. It allows you to write model and controller code in pure groovy. It is simply a plugin, which uses a groovy compiler for all kinds of classes. It also includes samples for groovy and GPP, especially a (not completely groovy-fied) converted YABE example.

`https://github.com/GuyMograbi/play_test_module`

`http://blog.mograbi.info/p/test-module-for-playframework.html`

This test module offers a different approach as the standard Play framework in order to test controllers. The module resembles controller method calls as close as possible. This means, you do not need to call a specific URL, but rather a direct controller method. It is merely a matter of taste, and which approach is preferred. If you like testing more on the Java and method level instead of testing via URLs, you should take a look at this module.

`https://github.com/fabienamico/sshdeploy`

This little module is a helper for automatic deployment of single nodes via SSH – the secure shell. You can add your node to a configuration file and simply call `play sshdeploy:nodeName`, which copies and unzips your application to a remote host.

`http://www.lunatech-research.com/editorials/tags/playframework`

The blog from Lunatech (one of the companies that support Play commercially besides Zenexity) features high quality articles about many aspects of the Play framework. You should definitely get the RSS feed.

`http://blog.aheron.de/`

A weblog featuring some Play framework specific articles. However, most articles are written in German.

`http://agaoglu.tumblr.com/post/2501419423/a-prototype-play-plugin-for-activiti-integration`

`http://agaoglu.tumblr.com/post/2700789235/activiti-hello-world-on-play`

These articles feature a nice integration with Activiti, a business process engine. The second link includes a good step-by-step example.

`http://splink.posterous.com/an-amf-module-for-the-amazing-playframework`

`https://bitbucket.org/maxmc/cinnamon-play/overview`

This module allows data exchange with flex and flash clients via AMF3 – the action message format.

Twitter

Here is a list of twitter nicks you might want to follow, as they sometimes post about the Play framework:

@playframework	Guillaume Bort, founder of Play, founded Zenexity (also offering commercial support for Play)
@nicolasleroux	Nicolas Leroux, committer, regular speaker about Play
@PeterHilton	Peter Hilton, committer, works at Lunatech (commercial support for Play)
@erwan	Erwan Loisant, committer, works at Zenexity
@nmartignole	Nicolas Martignole, committer
@mbknor	Morten Kjetland, committer
@pk11	Peter Hausel, committer, focuses on scala
@codemwnci	Wayne Ellis, author of an introductory Play framework book
@_felipera	Felipe Oliveira, Play and web hacker, author of elasticsearch module besides several other modules
@steve_objectify	Steve Chaloner, author of several modules
@eamelink	Eric Bakker, module hacker, works at Lunatech
@ikeike443	Ikeda Takafumi, author of the Jenkins/Hudson plugin
@sadache	Sadik Drobi, scala hacker, Play scala advocate
@sebcreme	Sebastien Crème, Play framework developer at Zenexity
@spinscale	Alexander Reelsen, software engineer, author of this book

In case you are using IRC (Internet Relay Chat), there is also an IRC channel #playframework on Freenode. You can use the server `chat.freenode.net` in order to connect to the freenode IRC network.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Thank you for buying
Play Framework Cookbook

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`.

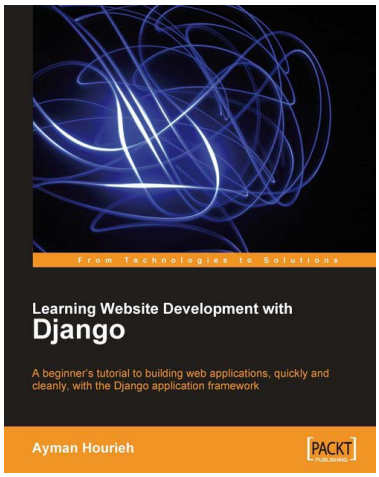
About Packt Open Source

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 licences, 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.

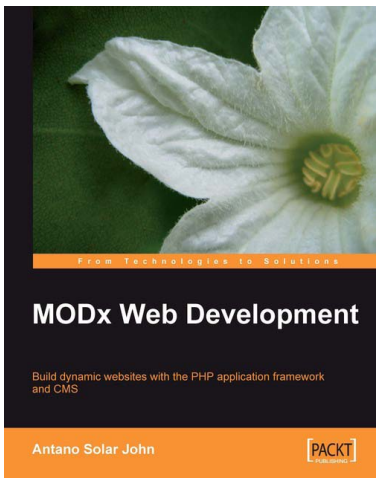


Learning Website Development with Django

ISBN: 978-1-847193-35-3 Paperback: 264 pages

A beginner's tutorial to building web applications, quickly and cleanly, with the Django application framework

1. Create a complete Web 2.0-style web application with Django
2. Learn rapid development and clean, pragmatic design
3. Build a social bookmarking application
4. No knowledge of Django required

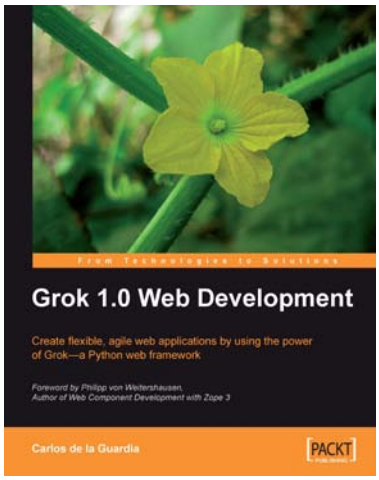


MODx Web Development

ISBN: 978-1-847194-90-9 Paperback: 276 pages

Building dynamic websites with the PHP application framework and CMS

1. Simple, step-by-step instructions detailing how to install, configure, and customize MODx
2. Covers detailed theory from the basics, to practical implementation
3. Learn the most common web requirements and solutions, and build a site in the process



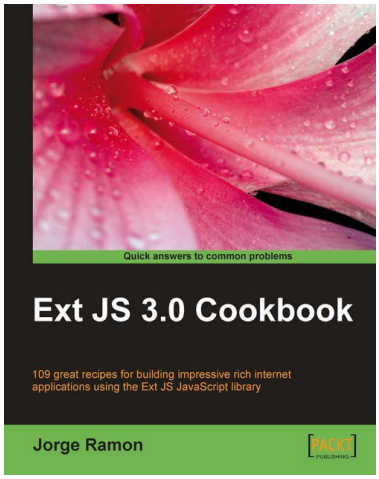
Grok 1.0 Web Development

ISBN: 978-1-847197-48-1 Paperback: 308 pages

Create flexible, agile web applications using the power of Grok—a Python web framework

1. Develop efficient and powerful web applications and web sites from start to finish using Grok, which is based on Zope 3
2. Integrate your applications or web sites with relational databases easily
3. Extend your applications using the power of the Zope Toolkit
4. Easy-to-follow and packed with practical, working code with clear explanations

---



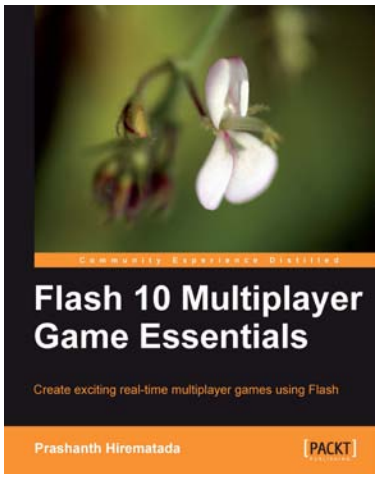
Ext JS 3.0 Cookbook

ISBN: 978-1-847198-70-9 Paperback: 376 pages

Clear step-by-step recipes for building impressive rich internet applications using the Ext JS JavaScript library

1. Master the Ext JS widgets and learn to create custom components to suit your needs
2. Build striking native and custom layouts, forms, grids, listviews, treeviews, charts, tab panels, menus, toolbars and much more for your real-world user interfaces
3. Packed with easy-to-follow examples to exercise all of the features of the Ext JS library
4. Part of Packt's Cookbook series: Each recipe is a carefully organized sequence of instructions to complete the task as efficiently as possible

[PACKT] open source
community experience distilled
PUBLISHING



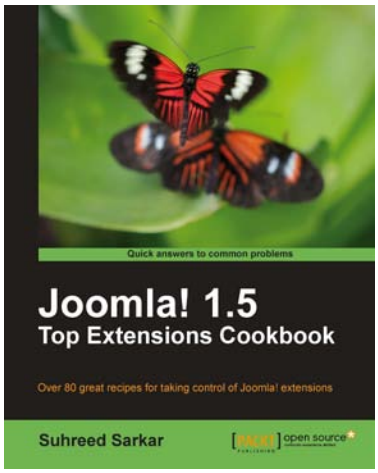
Flash 10 Multiplayer Game Essentials

ISBN: 978-1-847196-60-6 Paperback: 336 pages

Flash 10 Multiplayer Game Essentials

1. A complete end-to-end guide for creating fully featured multiplayer games
2. The author's experience in the gaming industry enables him to share insights on multiplayer game development
3. Walk-though several real-time multiplayer game implementations
4. Packed with illustrations and code snippets with supporting explanations for ease of understanding

---

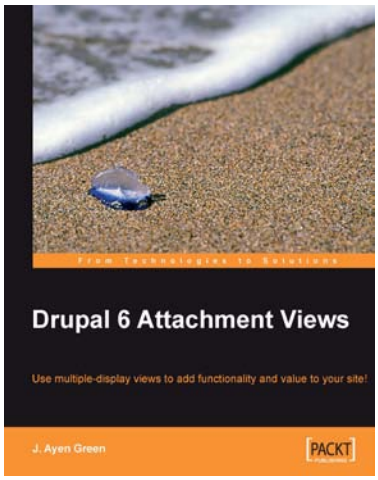


Joomla! 1.5 Top Extensions Cookbook

ISBN: 978-1-84951-180-3 Paperback: 460 pages

Over 80 great recipes for taking control of Joomla! Extensions

1. Set up and use the best extensions available for Joomla!
2. Covers extensions for just about every use of Joomla!
3. Packed with recipes to help you get the most of the Joomla! extensions



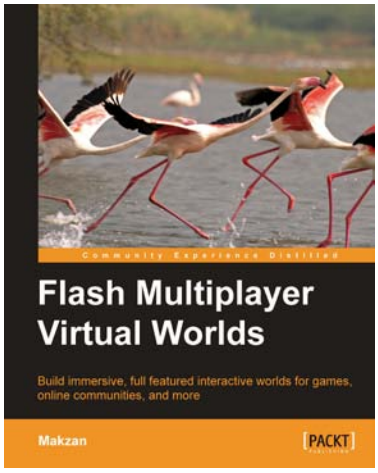
Drupal 6 Attachment Views

ISBN: 978-1-849510-80-6 Paperback: 300 pages

Use multiple-display views to add functionality and value to your site!

1. Quickly learn about painlessly increasing the functionality of your Drupal 6 web site
2. Get more from your Views than you thought possible
3. Topics provide rapid instruction and results
4. Concise, targeted information rather than voluminous reference material

---



Flash Multiplayer Virtual Worlds

ISBN: 978-1-849690-36-2 Paperback: 412 pages

Build immersive, full-featured interactive worlds for games, online communities, and more

1. Build virtual worlds in Flash and enhance them with avatars, non player characters, quests, and by adding social network community
2. Design, present, and integrate the quests to the virtual worlds
3. Create a whiteboard that every connected user can draw on
4. A practical guide filled with real-world examples of building virtual worlds